ÉTUDES & DÉVELOPPEMENT

ARCHITECTURE LOGICIELLE

Concevoir des applications simples, sûres et adaptables

Jacques Printz

Préface de Yves Caseau

DUNOD

ARCHITECTURE LOGICIELLE

Concevoir des applications simples, sûres et adaptables

Jacques Printz

Professeur au Conservatoire national des arts et métiers
Titulaire de la Chaire de génie logiciel

Préface de
Yves Caseau

DUNOD

Illustration de couverture :
Sugarloaf Mountain, Rio de Janeiro, Brazil

Source : digitalvision®

ISBN 2 10 049910 6

Préface

Ce livre parle, de façon centrale, de l'architecture des logiciels et des systèmes d'information, sous toutes ses facettes. L'architecture logicielle est un sujet scientifique et académique, qui a une histoire, des spécialistes et des références. Il existe d'excellents livres, cités au demeurant par Jacques Printz, et c'est une spécialité qui est enseignée dans la plupart des formations informatiques. L'architecture du système d'information est un sujet plus flou, dont l'importance pratique est évidente, mais dont les fondements sont moins théorisés. Les documents de référence ne sont plus des articles ou des livres, ce sont le plus souvent des schémas. Pour caricaturer, le dessin remplace l'équation et l'expérience remplace le raisonnement. Le lecteur qui s'aventure dans une entreprise découvre des merveilles d'ingéniosité, mais aussi une tradition orale, beaucoup de re-découvertes et de ré-inventions, et une difficulté à capitaliser et à faire émerger une véritable discipline scientifique. Même s'il existe des cours, dans les bons établissements, ou des livres, ils n'ont ni la maturité ni la rigueur de ce qui est consacré à l'architecture logicielle.

Cet état de fait n'est pas un hasard, ni une simple conséquence historique (on a commencé à faire des logiciels avant de faire des systèmes d'information). L'architecture du système d'information est un sujet « en tension ». Cette tension s'exprime entre la vision globale et les propriétés locales, entre les dimensions techniques et les dimensions métiers (ce qui est superbement illustré dans ce livre), entre une échelle de temps courte pour comprendre le fonctionnement et une échelle beaucoup plus longue pour comprendre l'écosystème des acteurs du système d'information. Comprendre l'architecture du système d'information est une gageure, il faut des compétences multiples, une vaste culture scientifique mais également une véritable expérience métier et industrielle. C'est un chemin semé d'embûches, qui est forcément long et demande, le lecteur va s'en apercevoir, une certaine patience.

Il se trouve que Jacques Printz est le guide idéal pour nous conduire sur ce chemin. Il combine une expérience professionnelle et industrielle remarquable et un talent d'enseignant dont témoignent ses précédents ouvrages et les cours qu'il propose au CNAM. Sa culture scientifique est exceptionnelle, et couvre un spectre très large. Le lecteur va s'apercevoir qu'il ne s'agit pas d'agrémenter son exposé par quel-

ques références brillantes, mais bien de donner du sens et de la vie à des principes qui pourraient, autrement, sembler abstraits, voire arbitraires. Jacques Printz a un regard unique sur quarante ans d'histoire de l'informatique et des systèmes d'information, dans lequel la pratique et la théorie sont mêlées de façon indissociable.

Ce livre propose au lecteur une promenade érudite et passionnante. Je parle de promenade car la lecture en continu sera indigeste, sauf pour des lecteurs experts. Ce livre est d'une très grande richesse, il s'agit clairement d'une « somme », issue d'une grande expérience et d'un long travail. Le terme de promenade fait également référence au plaisir qu'il y a de parcourir, de découvrir et de comprendre, de se former un jugement « esthétique » sur les choses. Le lecteur trouvera de nombreuses références à l'élégance et à l'esthétique des constructions.

Il ne s'agit pas, en revanche, d'un ouvrage théorique. Ce livre contient des données chiffrées, de nombreux exemples, des références aux différentes expériences pratiques de son auteur. Jacques Printz connaît bien le monde de l'entreprise, ce qui est essentiel pour bien comprendre le système d'information. La construction de ce système est une aventure humaine, avec de nombreux acteurs et de nombreux rôles. Cette répartition des pouvoirs et des responsabilités est expliquée de façon claire et intuitive, avec l'aide de nombreux schémas.

Il s'agit clairement d'un livre de référence, de ceux qu'on a plaisir à posséder dans sa bibliothèque pour pouvoir trouver la réponse à telle ou telle question. Il va au fond des choses, et possède une bibliographie complète, commentée et très pertinente. C'est la première porte, le lecteur est invité à en pousser d'autres (une évidence compte tenu de l'immensité du sujet). Ce livre est une « bible » de l'histoire des idées en informatique, ce qui en fait un ouvrage fondamental pour les enseignants et les chercheurs.

Il serait vain, en conséquence, de vouloir aborder les « principaux thèmes » de ce livre. Pour n'en citer que quelques-unes, je vais m'arrêter sur ceux qui me semblent les plus importants et qui sont encore trop ignorés dans les formations sur le système d'information. Par exemple, ce livre fait la part belle au thème de l'architecture de données, pris sous une forme extrêmement vaste, qui va de la modélisation jusqu'aux transactions distribuées. Les explications fournies par Jacques Printz sont tout à la fois claires et détaillées, et font de ce livre une contribution significative au domaine. Il est rare, en effet, de voir les sujets de distribution de données traités de « bout en bout », depuis le point de vue sémantique et conceptuel jusqu'à l'architecture interne du moniteur transactionnel.

De la même façon, la troisième partie sur l'architecture fonctionnelle propose une « reconstruction complète du génie logiciel » en partant de ses finalités (dans le cadre du système d'information), c'est-à-dire les processus de l'entreprise. Le lecteur qui suit ce chemin va redécouvrir, dans une vision globale et cohérente, tous les principes de l'urbanisation du système d'information. Cette partie permet d'ailleurs de souligner la continuité remarquable entre la vision métier et la vision technique qui est présentée dans ce livre.

Une autre contribution formidable de ce livre est la vulgarisation des « patrons de conceptions » du système d'information. C'est la bonne réponse à cette tendance néfaste à la réinvention et la répétition des mêmes erreurs qui trahit parfois l'immaturité de la discipline. Jacques Printz n'est pas le premier à faire émerger des « *patterns* » de l'architecture du système d'information, mais sa présentation est lumineuse. On y trouvera, en particulier, les descriptions « intelligentes » des concepts tels que le client-serveur, l'architecture en couches (ou en tiers) ou l'utilisation de bus logiciels. L'intelligence ici est ce qui permet de simplifier, de limiter la complexité, de faire les généralisations qui s'imposent, ou de replacer l'évolution des idées dans un contexte historique. La maîtrise de la complexité est probablement le sujet central de l'architecture du système d'information, et il reçoit cette place dans le livre de Jacques Printz. Il est l'occasion, une fois de plus, de profiter de l'érudition de l'auteur et de replacer certaines questions informatiques dans un cadre philosophique plus large.

Pour conclure, je noterai que ce livre contient l'essentiel de ce qu'il faut savoir pour comprendre le fonctionnement d'un système d'information complexe d'une grande entreprise, ce qui en fait un ouvrage unique. En tant que DSI, je pourrais ajouter, à titre de clin d'œil, que c'est le livre que je souhaiterais que tous mes fournisseurs de logiciel aient lu (et compris). Il s'agit sans nul doute d'un livre qui comble un vide, qui va jouer un rôle de référence pour de nombreuses années. Il fait partie de ces livres qu'on regrette de ne pas avoir lu plus tôt lorsqu'on les referme.

Yves Caseau
DSI de Bouygues Telecom,
Membre de l'Académie des Technologies

Table des matières

Partie 2 – Analyse de deux chefs-d'œuvre d'architecture

Partie 3 – Architecture fonctionnelle logique

Partie 4 – Propriétés d'une bonne architecture

Avant-propos

Jusqu'à une date récente l'architecture logicielle était une notion qui ne dépassait pas le cercle très restreint des concepteurs de systèmes d'exploitation, de progiciels système comme des SGBD (Système de Gestion de Base de Données), ou des logiciels de télécommunication ou celui des concepteurs de systèmes temps réel pour la défense, le spatial ou le contrôle aérien. C'est ce domaine technologique qui a donné naissance, dès les années 50-60 à l'ingénierie système[1].

Dans un système d'information « traditionnel », disons dans les années 70-80, l'architecture de ce qu'on appelle encore « application de gestion » se résume à l'organisation du modèle de données, avec des méthodologies comme MERISE ou Warnier-Orr, très populaires en France, mais quasiment ignorées des pays anglo-saxons, méthodologies fondées sur le langage logique du modèle ENTITÉS – RELATIONS – ATTRIBUTS.

Sur une machine centralisée, le « *mainframe* » des années 70-80, le système d'exploitation prend complètement à sa charge tout le contrôle de ce qui se passe dans la machine ainsi que les diverses régulations nécessaires à une répartition équitable des ressources disponibles pour les applications que la machine héberge. Le concepteur d'application est donc déchargé d'un travail difficile de gestion de ressources, qui met en œuvre des algorithmes dont certains ont mis 10 ou 15 ans à trouver leur point d'équilibre, comme par exemple les algorithmes de gestion de la mémoire virtuelle qui donnent au programmeur l'apparence d'une mémoire sans limite, de son point de vue.

Avec l'arrivée des stations de travail, des micro-ordinateurs et des architectures clients-serveurs distribuées ce cadre relativement simple avec un point de contrôle unique vole en éclats. Les architectes-concepteurs de systèmes d'information se retrouvent confrontés aux mêmes problèmes que leurs collègues des systèmes technologiques, mais bien souvent sans avoir un « *back ground* » scientifique suffisant, car limité aux données. Toutefois, avec des systèmes d'information comme les systèmes

1. Sur ce sujet, voir les deux ouvrages de JP. Meinadier : *Ingénierie et intégration des systèmes* et *Le métier d'intégration de systèmes*, chez Hermès-Lavoisier.

de réservation des compagnies aériennes (par exemple SABRE pour American Airlines ou AMADEUS pour Air France), l'évolution vers les architectures distribuées complexes, et les problèmes afférents, se dessine déjà clairement (NB : ce type de systèmes gèrent des dizaines de milliers d'équipements sur toute la planète, et absorbent des charges de traitements dont le pic peut dépasser 1 000 transactions à la seconde, avec des contraintes importantes de sûreté de fonctionnement). Le transactionnel[1] naît dans cet environnement

C'est grâce à la présence d'un moniteur transactionnel que le programmeur COBOL a pu passer sans trop de difficulté du monde de la programmation batch à celui de la programmation dite transactionnelle qui est celle de tous les systèmes d'information. Faute de ce moniteur, la gestion des files d'attente de messages, de la mémoire contextuelle spécifique à une transaction, de la gestion des priorités et des ressources entre les transactions, etc., auraient été à la charge du programmeur d'application. Le moniteur transactionnel gère la partie la plus délicate, les interactions et les événements avec l'environnement de la transaction. Ne reste à la charge du programmeur, que la partie transformationnelle concernant les données de l'application.

La même remarque vaut pour la gestion des données faite par les SGBD qui prennent en charge la gestion de l'adressage interne et de l'intégrité des données stockées dans les mémoires de masse, dont le nombre d'occurrences peut aller de quelques milliers à plusieurs dizaines de milliards dans les « magasins de données ».

Dans un système distribué, où les ressources sont réparties entre différents ordinateurs et équipements reliés entre eux par des réseaux, toute cette belle mécanique quasiment invisible au programmeur d'application remonte dans les couches applicatives. Les programmes d'applications doivent alors prendre en charge certains états du réseau qui affectent leurs comportements ; ils doivent se faire notifier des débuts ou des fins de travaux effectués sur d'autres équipements pour pouvoir eux-mêmes se déclarer en état de terminaison. S'il y a des incidents ou des pannes sur le réseau, le programmeur devra s'assurer que l'intégrité des travaux en cours n'est pas affectée, ou que l'on sait arrêter proprement les applications et/ou les systèmes, d'où une interface beaucoup plus complexe avec les équipes d'exploitation. Ce qui était intégré se désintègre, avec comme conséquence immédiate une explosion du coût d'intégration système, quasiment inexistant en centralisé, qui est un nouveau coût à la charge de l'équipe d'ingénierie. L'évolution d'un programme devient l'évolution d'un système. Le concepteur d'une application distribuée doit savoir explicitement qu'elles sont les ressources logiques dont l'application a besoin, de façon à contrôler le mapping de ces ressources par rapport aux ressources physiques effectivement disponibles sur la plate-forme d'exécution.

L'émergence désormais irréversible de l'informatique distribuée et des architectures clients-serveurs, ainsi que la nécessité de la décentralisation géographique des

1. Cf. le système de réservation SABRE définit conjointement par IBM et American Airlines, qui a servi de modèle, beaucoup plus tard, au système SOCRATE de la SNCF dont on se rappelle les déboires.

traitements, transforme le concepteur d'application – c'est déjà une tâche fort difficile – en architecte de système qui doit s'assurer que les aléas qui affectent à l'instant t tel ou tel élément du réseau, ne compromettent pas la cohérence globale du système. Le bon fonctionnement de cette nouvelle couche d'infrastructures, appelé « *middleware* » dans le jargon informatique, est le prix à payer des facilités et de la souplesse d'emploi incontestable qu'apporte l'informatique distribuée.

Contrairement à ce qu'un marketing parfois racoleur essaye de faire croire, les progiciels systèmes « *middleware* » sont intrinsèquement complexes car ce sont des programmes réactifs, asynchrones et globalement non déterministes à cause des réseaux ; ils sont loin d'offrir, pour le moment, la même sûreté de fonctionnement que celle que l'on trouvait dans les bons vieux mainframes – la communication entre les sous-systèmes du mainframe se fait via la mémoire commune – pour des raisons techniques inhérentes à la complexité des architectures distribuées. Ce sont des logiciels très coûteux à développer, ce qui exclut de les trouver sur des plates-formes à faible diffusion où alors avec une performance et une fiabilité mauvaise, sans parler de leur maintenabilité ou de leur pérennité aléatoire. La conséquence de ce nouvel état de fait est que le concepteur d'application se voit obligé de développer certains mécanismes qui lui étaient jusqu'alors cachés, ou dont il n'avait pas à se soucier, de façon à pouvoir utiliser de façon sûre les parties les plus solides des middlewares ou des environnements d'intégration qui sont à sa disposition. De vraies compétences d'architecte, au sens défini dans cet ouvrage, deviennent indispensables. Sans se transformer complètement en concepteur système, il doit cependant être à même de comprendre ce qui se passe, ne serait-ce que pour gérer correctement tous les évènements nouveaux dont il a désormais la charge : administration automatisée et autorégulée, surveillance des ressources, modes de fonctionnement dégradés et continuité du service, contrat de service (*Service Level Agreement*), etc.

Le but de ce livre est de mieux faire comprendre, par un certain nombre d'exemples et d'analyse, ce qu'est l'architecture logicielle et le métier d'architecte de façon à permettre aux DSI[1], aux chefs de projet MOA/MOE, aux responsables d'équipes de développement et de maintenance, ainsi qu'à la grande masse des programmeurs, de pouvoir continuer à travailler correctement avec une bonne productivité, et de bien comprendre le contexte système dans lequel ils opèrent. C'est une nouvelle responsabilité pour le concepteur d'application et/ou le chef de projet qui doivent en mesurer la difficulté afin d'éviter de se lancer dans des développements dont les fondements ne sont pas solidement établis et qui les conduiraient droit à l'échec[2]. Un nouveau métier apparaît de façon explicite, celui d'architecte de systèmes, au sens large, dont le rôle est d'assurer l'intermédiation entre les métiers déjà connus comme le développement, la maintenance, l'exploitation et les usagers du système généralement représenté par une maîtrise d'ouvrage. C'est un métier très technique qui requiert de grande capacité d'abstraction et de communication. L'architecte doit se préoccuper de l'acceptabilité

1. Cf. le livre de Y.Caseau, *Urbanisation et BPM – Le point de vue d'un DSI*, Dunod, 2005, qui explique ce qu'un DSI doit comprendre de l'architecture pour effectivement manager le SI de l'entreprise.
2. Cf. L'étude du Standish Group : *Chaos chronicles*, version 3.0 ; édition 2003 du rapport.

sociale du système du point de vue de la communauté des exploitants et des usagers métiers, ainsi que de l'intégration globale dans le système d'information de l'entreprise, rôle assimilé métaphoriquement à de l'urbanisme.

Nous aborderons brièvement les aspects économiques (CQFD des projets, TCO et ROI des systèmes) dans la mesure où ces différents coûts sont largement dépendants de la qualité du travail de l'architecte. Les DSI, les responsables de projets de MOA, sans être eux-mêmes nécessairement des architectes, doivent comprendre les enjeux de l'architecture en sachant que le TCO d'un système est engagé dès le début de la vie du système, lorsque l'architecture est fixée dans ses grandes lignes.

L'ouvrage est organisé en quatre parties : la première partie présente la problématique d'architecture dans ses différents aspects ; la seconde partie est un témoignage vécu par l'auteur sur deux aspects fondamentaux de l'architecture des systèmes, les langages et compilateurs d'une part, les processus et leurs interactions d'autre part ; la troisième partie présente un ensemble de modèles d'architecture de haut niveau basé sur les deux domaines les mieux connus de la technologie informatique, les traducteurs/compilateurs de langages et les moniteurs, qui s'intègrent tous deux dans une problématique plus générale de machines abstraites ; la quatrième partie explique pourquoi il est fondamental pour l'architecte de se préoccuper le plus tôt possible des caractéristiques qualité de simplicité, disponibilité, évolutivité ainsi que de la définition des interfaces.

Remerciements

La matière de ce livre n'aurait pas pu être assemblée sans les deux expériences qui ont le plus marqué mon parcours professionnel.

Bull, dans les années 70-80 a su réunir autour du projet DPS7/GCOS7 une somme exceptionnelle de talents, et à vrai dire, unique dans le paysage informatique français et européen. Ces talents ont irrigué toute la profession. Le lien direct avec le MIT, avec les concepteurs de MULTICS, avec les architectes de GE et de Honeywell, etc., nous garantissaient l'accès aux créateurs de cette technologie nouvelle révolutionnaire. Ce fut une période magique de grande créativité.

Le ministère de la défense, par l'ambition des systèmes et systèmes de systèmes nécessaires à la sécurité du pays, par la complexité de leur définition et de leur mise en œuvre, m'a fourni la matière de la partie III. Le contact direct avec les programmes et les architectes de la DGA, ainsi qu'avec les industriels, et plus particulièrement THALES, sont, pour ce qui me concerne, un témoignage de la prudence (au sens de la vertu cardinale, avec ses deux visages, la sagesse de l'expérience et la force de la jeunesse) avec laquelle il faut attaquer ces problèmes extrêmes.

Que tous ceux avec qui j'ai pu interagir et travailler durant toutes ces années soient ici remerciés, ce livre est un peu le leur. Ma secrétaire du CMSL, Jacqueline Pujol, en me soulageant du travail logistique, m'a permis de disposer de plus de temps pour le travail de fond.

Je remercie également les éditions Dunod qui m'ont fait confiance avec patience, car ce livre aurait dû paraître il y a déjà quelques années…

PARTIE 1

Qu'est-ce que l'architecture du logiciel ?

1

L'architecture dans les sciences de l'ingénieur

1.1 L'ARCHITECTURE AU SENS LITTÉRAL

Le terme architecture, dans le sens originel qu'il a en construction civile désigne tout à la fois :

- Une compréhension profonde des besoins présents et futurs (c'est un facteur d'incertitude) de ceux pour lequel le bâtiment est destiné et des contraintes de l'environnement, y compris les contraintes sociales, dans lequel le bâtiment doit être inséré ;
- Une connaissance approfondie des matériaux à assembler ainsi que de leurs limites (usinage, vieillissement, rupture, etc.) ;
- Une connaissance approfondie des procédés de construction (i.e. les méthodes du génie civil, la gestion de projet, l'acquisition) permettant d'organiser l'activité des différents corps de métiers qui concourent à la réalisation et au maintien en conditions du bâtiment (surveillance, maintenance, adaptation, etc.).

L'architecte est celui qui rassemble et sait mettre en œuvre ces différentes connaissances au service exclusif du but poursuivit par le commanditaire ou son mandant : le maître d'ouvrage. L'architecte est un chef d'orchestre qui doit savoir jouer de plusieurs instruments (pas de tous, car cela est humainement impossible !) et qui doit surtout connaître la musique.

Les architectes du moyen âge[1] comme Villard de Honnecourt, Robert de Luzarches ou Pierre de Montreuil, ont excellé à traduire le sentiment religieux de leurs contemporains dans ces cathédrales de pierres que l'on ne se lasse jamais de contem-

pler. Comme l'a dit un historien du moyen-âge : « La cathédrale est un problème technique victorieusement résolu, mais, dans sa conception, elle reste une inspiration grandiose de la Foi ». Sur les épitaphes de leurs pierres tombales, ou dans la crypte des cathédrales qu'ils ont construites, on trouve des qualificatifs comme « docteur es-pierres » ou « très élevé dans l'art des pierres » qui traduisent l'admiration et la reconnaissance de leurs contemporains dans la maîtrise du matériau de base des cathédrales qui ne sont que des pierres savamment taillées et assemblées.

Vauban quelques siècles plus tard érigera la construction des fortifications en une véritable « science »[2] qui traduisait parfaitement le besoin de défense des cités frontalières du nord et de l'est de la France en intégrant complètement dans le système défensif l'art de la guerre de cette époque. On peut encore admirer tout un ensemble de maquettes dont il fit démarrer la réalisation, au musée des Plans et Reliefs, aux Invalides, à Paris. De toute évidence, Vauban maîtrisait complètement l'art du maquettage, et il savait s'en servir pour expliquer à ses commanditaires la nature des travaux qu'il proposait d'entreprendre. La fortification enterrée était la seule façon de répondre au feu des canons désormais capables de briser n'importe quelle muraille verticale conçue initialement pour résister aux flèches et aux assauts de l'infanterie.

Compréhension du besoin, connaissances des matériaux, capacité d'organisation, motivation des corps de métiers, conviction et persuasion vis-à-vis des autorités religieuses et/ou militaires commanditaires des travaux, forment la trame culturelle et intellectuelle de ces architectes du passé qui étaient également, et avant tout, des ingénieurs[3].

Toutes ces constructions reposent sur une innovation technique au service d'une idée. Les cathédrales n'existent que par la croisée d'ogives, les arcs-boutants et l'art de la taille des pierres comme les clés de voûtes dont la précision requiert une grande connaissance de la géométrie[4], ainsi qu'une très grande capacité d'organisation de chantiers qui pouvaient rassembler plusieurs milliers de personnes dans une grande variété de métiers. La taille des pierres se faisant dans des carrières parfois fort éloignées de la cathédrale[5], donc sur plans ou sur modèles en bois, ce qui nécessitait déjà un excellent niveau de standardisation. L'innovation libère l'imagination et rend la construction possible sans en être pour autant une condition suffisante. La construc-

1. Cf. E.Viollet-Leduc, *Encyclopédie médiévale*, Interlivres – J.Gimpel, *Les bâtisseurs de cathédrales*, Le Seuil, et R.Bechmann, *Villard de Honnecourt – La pensée technique au 13ème siècle et sa communication*, Picard éditeur, 1993.
2. Cf. le remarquable ouvrage de J. Langins, aux MIT Press, *Conserving the enlightenment – French military engineering from Vauban to the Revolution*, 2004.
3. Cf. P.Rossi, *Aux origines de la science moderne*, Ed. du Seuil, collection Points Sciences (chapitre III, Les ingénieurs) ; Le Corbusier, *Vers une architecture*, Flammarion ; *L'art de l'ingénieur - constructeur, entrepreneur, inventeur*, Ed. Centre Georges Pompidou – Le Moniteur.
4. C'était un des arts libéraux enseigné dans nos universités du moyen-age ; il est certain que les bâtisseurs de cathédrales connaissaient Euclide, mais la taille des pierres requiert des savoir-faire précis qui se transmettaient par compagnonnage.
5. Les pierres de la cathédrale de Cantorbéry, en Angleterre, viennent de la région de Caen réputée pour la qualité de son calcaire.

tion doit également être « belle », selon les règles esthétiques de l'époque. Il faut donc rajouter du subjectif et du qualitatif pour rendre la construction socialement acceptable. Les soldats de Louis XIV se sentaient bien protégés quand ils étaient en garnison dans un fort conçu par Vauban ; ils avaient confiance dans leur commandement.

Une construction est « belle » si elle est adaptée à sa finalité, ce qui n'est jamais acquis d'avance. La recopie sans l'intelligence de l'environnement qui lui a donné naissance mène rapidement à la dégénérescence. Ce qui était beau devient conventionnel. Les lignes de forces, si visibles et naturelles dans les premières cathédrales disparaissent sous les fioritures du gothique flamboyant. L'intention primitive cède le pas à la prouesse technique, comme les voûtes du cœur de la cathédrale de Beauvais qui s'effondrent en 1284. La guerre de mouvement napoléonienne rend obsolète l'idée même de place forte, symbole de la guerre de position, qu'il est très facile de contourner avec une cavalerie et surtout une artillerie mobile, afin d'en désorganiser les arrières ; une évidence oubliée par les constructeurs de la ligne Maginot. On sait que Napoléon disposait d'un système de communications hérité de la révolution (le télégraphe de l'ingénieur Chappe) et d'une artillerie mobile et puissante (le canon Gribeauval) qu'il a magistralement exploitée dans des mouvements fulgurants, relativement à l'époque, jusqu'à ce que ses adversaires fassent comme lui.

Rien n'est donc jamais définitivement acquis et l'architecte doit évidemment faire preuve d'une très grande ouverture d'esprit et d'une vraie capacité d'anticipation : « il pense à cent ans » disait Le Corbusier.

Plus près de nous, Le Corbusier a su tout à la fois imaginer des solutions architecturales et urbanistiques qu'autorisaient les possibilités de nouveaux matériaux comme le béton armé ou l'usage systématique du fer, déjà utilisé dans les cathédrales. Le Corbusier définissait la ville comme un outil de travail qui doit être adapté à sa fonction[6] administrative, militaire, protectrice, marchande, conviviale, etc. L'ordre géométrique était pour lui d'une importance primordiale tout autant qu'un facteur d'efficacités des actions humaines ; il disait que « l'architecte, par l'ordonnance des formes, réalise un ordre qui est une création de son esprit »[7]. Il distinguait soigneusement l'architecture « un fait d'art » de la construction « c'est pour faire tenir ». Pour faciliter la reconstruction des villes détruites durant la seconde guerre mondiale, il avait même créé une unité de mesure : le *modulor*[8] qui permettait, disait-il, de construire en série des maisons variées à partir de blocs modulables mais respectant des proportions agréables aux humains. En quelques sortes, des règles de modularité et de réutilisation avant la lettre ! Une ville comme Le Havre, complètement détruite en 1943-44, fut rebâtie par A.Perret, disciple de Le Corbusier, sur ces principes ; elle est aujourd'hui classée au patrimoine mondial de l'UNESCO.

6. C'est le concept d'urbanisme, au sens littéral ; cf. Le Corbusier, *Urbanisme*, Flammarion.
7. Cf. Le Corbusier : *Vers une architecture*, Flammarion.
8. Cf. *The modulor*, Faber paperback, 1951.

1.2 LIMITES DES MÉTAPHORES ARCHITECTURALES

Tout ce qui vient d'être dit de l'architecture et du génie civil, se transpose quasiment mot pour mot dans celui des grandes « constructions immatérielles » que sont les logiciels systèmes ou les systèmes informatisés.

On peut cependant s'étonner de l'emploi de mot comme architecture ou urbanisme dans un domaine aussi abstrait que celui du logiciel. Il y a évidemment de grandes différences, et la métaphore architecturale et urbanistique cesse rapidement d'être pertinente, pour ne pas dire trompeuse. On n'agit pas de la même façon sur un logiciel que sur des pierres ou du béton, fut-il qualifié de « matière grise » comme disait Francis Bouygues ! Le « matériau » informatique qui intègre du sens et de la sémantique, une forme de « vie » avec des comportements bien particuliers selon les contextes, n'a rien à voir avec les matériaux inertes du monde physique. « Analogie ne veut pas dire similitude, il faut se méfier des fausses simplifications ».

Toute architecture est basée sur l'intuition que nous avons de la forme et de l'espace[9]. Les architectes plus que les autres ont évidemment besoin de cette intuition. Les formes géométriques pré-existent, au sens platonicien ; la construction révèle et matérialise la forme. On pourrait dire que l'architecture c'est l'art des formes que les procédés de construction et nos règles sociales autorisent.

Dans une construction artificielle comme l'est n'importe quel programme informatique, la forme géométrique n'existe pas. La forme logique d'un programme doit être créée par l'architecte du logiciel, ou par le programmeur. La seule analogie pertinente à laquelle on puisse se référer nous est donnée par le développement des mathématiques modernes (i.e. à la fin du XIX^{e} siècle) où émerge la notion de structure mathématique. La prolifération des faits mathématiques, des théorèmes et des théories particulières rendent progressivement indispensable la nécessité d'une remise en ordre, ne serait-ce que pour des raisons pédagogiques et de mémorisation des faits mathématiques, associées à un puissant besoin de généralisation comme on peut le voir dans la théorie des équations algébriques et les variétés de géométrie. Les mathématiciens considèrent qu'Henri Poincaré et David Hilbert, son cadet de dix ans, ont été les derniers géants des mathématiques à pouvoir embrasser toutes les mathématiques de leur époque. La matière mathématique est simplement devenue trop riche par rapport aux capacités des cerveaux les plus doués.

D. Hilbert, et son école de Göttingen, ont été les architectes de cette remise en ordre dont on peut dire que Bourbaki a été l'héritier putatif[10]. La construction d'abstraction efficace est un puissant moyen d'emmagasiner plus de faits mathématiques dans une vie de mathématiciens et donc d'en étendre encore le champ afin de faire reculer les limites

9. Sur la notion de forme, l'ouvrage de D'Arcy Thomson, *On growth and forms*, Cambridge University Press, est incontournable. Voir également, R.Thom, *Stabilité structurelle et morphogenèse*, Interéditions, 1977.

10. Sur cette re-fondation, voir l'ouvrage de C. Reid, *Hilbert*, Springer, et le dossier *Bourbaki* dans les cahier de *Pour la science*. Egalement, les deux ouvrages de J. Dieudonné, *Pour l'honneur de l'esprit humain* et *Le choix bourbachique*.

de ce qu'il pourra maîtriser. Sans cette vue abstraite, c'est l'enlisement immédiat dans le sable des particularismes. La même remarque vaut en physique[11].

Quiconque a fait un peu de mathématiques (disons au niveau des classes préparatoires ou de la licence scientifique) peut se faire une idée très précise de la puissance de l'abstraction dans le nombre de pas de raisonnement nécessaires à la démonstration de tel ou tel théorème pénible (Bolzano-Weierstrass, Sturm, la théorie classique des coniques, etc.). Un calcul absolument fastidieux en coordonnées cartésiennes devient trivial en coordonnées affines ou en coordonnées polaires. Le choix du système de représentation d'un problème géométrique, adapté à ce problème, est un élément simplificateur dans l'expression du problème qui va rendre les démonstrations sinon limpides du moins plus faciles à mémoriser, ce qui permettra d'autres rapprochements, et de nouvelles abstractions. Sans le symbolisme du calcul tensoriel, la théorie de la relativité générale serait quasiment inexprimable, et c'est un fait connu qu'Einstein mit plusieurs années à en maîtriser le maniement[12]. La même remarque vaut pour l'emploi des espaces de Hilbert par J. von Neumann en mécanique quantique[13]. Von Neumann, très impliqué dans la création des premiers ordinateurs, créateur de l'architecture dite de von Neumann, lui même mathématicien de première force, utilisait dans ses écrits[14] sur les « *computing instruments* » des expressions comme « *logical design* » pour bien distinguer la cohérence logique de la machine de sa réalisation matérielle qui en était la traduction. La forme architecturale d'un logiciel est sa forme logique qui ne se perçoit pas sans une certaine éducation à la rigueur mathématique et à la méthode scientifique[15]. Les notions d'automates, de système à états-transitions, de traduction, de machines abstraites sont des formes logiques fondamentales pour organiser et structurer les logiciels.

A contrario, un informaticien issu des sciences « molles », par définition peu familier des abstractions et de la rigueur logique, aura des difficultés à saisir l'essence abstraite de l'architecture qu'il ne distinguera pas de la réalisation matérielle représentée par le programme. N'ayant pas à sa disposition le référentiel culturel et les instruments de pensée indispensables à la perception des abstractions, il ne pourra pas s'en faire une idée précise, d'où des réactions de rejets face à ce qu'il considérera comme de la théorie sans intérêt.

Cependant, comme pour le gothique flamboyant, on peut faire de l'abstraction, en quelque sorte pour le plaisir, avec comme seul objectif de faire plus compact mais sans rien créer de nouveau. Créer un nouveau langage n'est pas la même chose que créer de nouveaux concepts ; une sténographie n'est pas un langage, c'est juste un moyen d'écrire plus vite. Ce fut beaucoup reproché à Bourbaki[16], ou à des théories comme la théorie des catégories[17].

11. Voir par exemple l'ouvrage de O. Darrigol, *Les équations de Maxwell*, Belin, 2005.
12. Cette histoire est fort bien racontée dans : J-P. Auffray, *Einstein et Poincaré*, Ed. Le pommier.
13. Cf. son livre fameux : *Fondement mathématique de la mécanique quantique* (1ère édition en allemand, en 1927, en français en 1946).
14. Cf. œuvres complètes, volume VI.
15. On peut recommander la lecture de l'ouvrage de K. Popper, *La logique de la découverte scientifique*, Payot, 1970 et 1984, 1ère édition en 1934 (en allemand).

De tout cet effort, il ressort qu'il y a deux catégories d'abstractions : a) les abstractions utiles, car non seulement elles rendent la description plus simple (i.e. plus courte ! en prenant le point de vue de la théorie algorithmique de l'information) mais en plus elles ouvrent des portes ; et b) d'autres, qui ne sont que des abstractions stylistiques, que R.Thom qualifiaient d'« algébrose ». On verra, avec l'exemple des techniques de compilation ébauchées en partie 2, la puissance de l'abstraction au service d'une des technologies sur laquelle repose toute la programmation. Une classe logique n'est véritablement intéressante que si elle contient plus d'un élément, sinon on a deux noms pour la même chose : un nom propre et un nom de classe. Les logiciens du moyen age, contemporains des bâtisseurs de cathédrales, dans leur effort de simplification de l'exposé de la théologie, disaient qu'il ne fallait pas créer de catégories inutiles. Guillaume d'Ockham[18], moine franciscain du XIVe siècle, nous a donné une des premières expressions du principe de simplicité ou de parcimonie, la bonne vieille règle dite du rasoir d'Ockham, sous différentes formulations comme : *Entia non sunt multiplicanda praeter necessitatem* (Il ne faut pas multiplier les choses au-delà du nécessaire) ou *Frustra fit per plura quod fieri potest per paucioria* (On fait vainement avec beaucoup de choses ce qu'on pourrait faire avec peu de chose). Thomas d'Aquin qui fut un grand professeur de l'université de Paris un siècle plus tôt, vers 1250[19], disait dans sa *Somme contre les gentils*, au livre I,I et II, XXIV,3 : « […] L'office du sage est de mettre de l'ordre. […] Ceux qui ont en charge d'ordonner à une fin doivent emprunter à cette fin la règle de leur gouvernement et de l'ordre qu'il crée : chaque être est en effet parfaitement à sa place quand il est convenablement ordonné à sa fin, la fin étant le bien de toute chose. Aussi, dans le domaine des arts, constatons-nous qu'un art détenteur d'une fin, joue à l'égard d'un autre art le rôle de régulateur et pour ainsi dire de principe. […] Il en va de même du pilotage par rapport à la construction des navires, de l'art de la guerre par rapport à la cavalerie et aux fournitures militaires. Ces arts qui commandent à d'autres sont appelés architectoniques, ou arts principaux. Ceux qui s'y adonnent, et que l'on appelle architectes, revendiquent pour eux le nom de sages. […] Ceux qui dans le domaine technique coordonnent les différentes parties d'un édifice sont appelés des sages par rapport à l'œuvre réalisée ». Les mots de Saint Thomas ont une saveur étrangement moderne : « piloter », « gouverner », « réguler », « finalité » viennent de la même racine grecque (*cyber*) que N.Wiener utilisa pour créer le mot cybernétique. L'architecte est fondamentalement quelqu'un qui met de l'ordre, qui crée de la

16. Cf. R. Thom, *Les mathématiques modernes, une erreur pédagogique et philosophique*, dans *Apologie du logos*, Hachette, 1990 ; également B. Beauzamy, *Méthodes probabilistes pour l'étude des phénomènes réels*, SCM 2004, ISBN 2-9521458-06.

17. N'étant pas mathématicien, ceci n'est pas un jugement mais juste une remarque, d'autant plus qu'avec A.Grothendieck et sa théorie des schémas (ou topos, ou motifs) on pourrait certainement discuter. Voir son ouvrage autobiographique, *Récolte et semailles* (jamais publié, disponible sur Internet).

18. Le personnage de Frère Guillaume, dans le roman de U. Ecco, *Au nom de la rose*, lui ressemble comme un frère.

19. A cette époque, les chantiers de nos grandes cathédrales étaient ouverts : 1163 Paris, 1190 Bourges, 1194 Chartres, 1211 Reims, 1220 Amiens, et la Sainte Chapelle à Paris, joyau de l'art gothique, en 1243, que Thomas d'Aquin, familier du roi St-Louis, a vu construire.

simplicité, pour la satisfaction du plus grand nombre, qui régule pour organiser le chantier de mise en œuvre.

Plus près de nous, l'épistémologie génétique de J. Piaget[20] a mis en évidence l'importance de la construction de ces abstractions dans les phénomènes d'apprentissage et du développement de la maturité, plus particulièrement chez l'enfant. L'assimilation cognitive va toujours du concret vers l'abstrait ; cet ordonnancement a une grande importance pour l'assimilation correcte et la maîtrise progressive des mécanismes cognitifs. L'inverse est anti-pédagogique. On retrouvera cet ordonnancement dans la démarche de conception des systèmes informatisés où l'on analyse en détail le besoin à l'aide de cas d'emplois et de modèles du système à informatiser, fut-il humain, pour, après coup, en abstraire la partie qui sera programmée et automatisée. De même, en théorie des fonctions on passe d'une description en extension à l'aide de tables à une description en intention avec des formules de calcul.

1.3 ARCHITECTURE DE L'INFORMATION

Dans les sciences de l'information, l'identification des processus à informatiser et l'organisation de ces processus, les acteurs humains et les équipements qui les animent, sont la matière première de l'information. À partir de cette réalité informationnelle, le travail d'agencement des instructions et des données pour construire les programmes, la prise en compte des événements qui déclencheront tels ou tels automatismes programmés, le choix de telles ou telles technologies, l'ordonnancement correct des opérations qui transforment progressivement les données initiales en résultats exploitables, etc., que l'architecte/programmeur effectue dans son activité quotidienne, les « preuves » et tests qu'il doit fournir de la justesse de ses décisions et de la validité de ses constructions, etc., a de nombreuses similitudes avec le travail du mathématicien et/ou du physicien face à la réalité du monde physique. La grande différence est que les abstractions fabriquées par le programmeur vont être exécutables et testables sur un ordinateur : l'erreur est immédiatement visible, et ses conséquences peuvent être graves selon les missions dévolues au système informatisé, mais cela permet la validation dans un monde virtuel simulé, avant la mise en service opérationnelle dans la réalité !

Comme le mathématicien ou le physicien, l'architecte / programmeur cherche à construire des entités logiquement cohérentes qui traduisent fidèlement l'intuition du sens profond qu'il a des choses qu'il veut modéliser et abstraire. Comme lui, il cherche les abstractions utiles qui vont diminuer la taille du programme qu'il fabrique, et cela non pas grâce à des astuces artificielles douteuses comme une sténographie, mais par une compréhension profonde de la structure logique et du sens des entités qu'il manipule. Dans sa forme achevée, cette compréhension se manifeste sous la forme de langages plus ou moins formalisés. Ce travail méthodologique

20. Cf. *Introduction a l'épistémologie génétique*, 3 Vol. et *L'équilibration des structures cognitives*, aux PUF.

s'apparente à la construction d'une théorie, un « système du monde » aurait dit Newton, ou plus près de nous P. Duhem[21]. Ainsi émergent, les sous-programmes, les interfaces, les structures de données, les objets et méthodes, les applications dont il sera abondamment question dans cet ouvrage. Les langages informatiques (modélisation et programmation, y compris programmation des tests) sont une aide puissante dans la matérialisation des abstractions ainsi créées qui elles ne dépendent que des capacités de compréhension de l'architecte/programmeur. Mais si ce dernier ne perçoit pas les abstractions, le langage informatique ne lui sera d'aucun secours.

La source de ces abstractions informatiques est dans la réalité informationnelle qui, une fois la programmation terminée, constituera le juge de paix de la validité de la construction. La « vérité » de la construction est dans le monde réel, pas dans le programme. Cette réalité informationnelle est plus ou moins complexe, selon la nature des systèmes envisagés. Dans un système technologique comprenant une grande variété d'équipements eux-mêmes complexes, comme un système de contrôle aérien, la complexité est dans le fonctionnement des équipements, et dans l'interface homme-machine. Le premier schéma ci-dessous (figure 1.1) montre une boucle de régulation pour gérer le trafic en zone aéroportuaire. Un second schéma montre un système de contrôle beaucoup plus élaboré avec des dizaines de Radar et de nombreux centres de contrôle du trafic aérien. NB : Les satellites de positionnement (GPS) ne sont pas représentés.

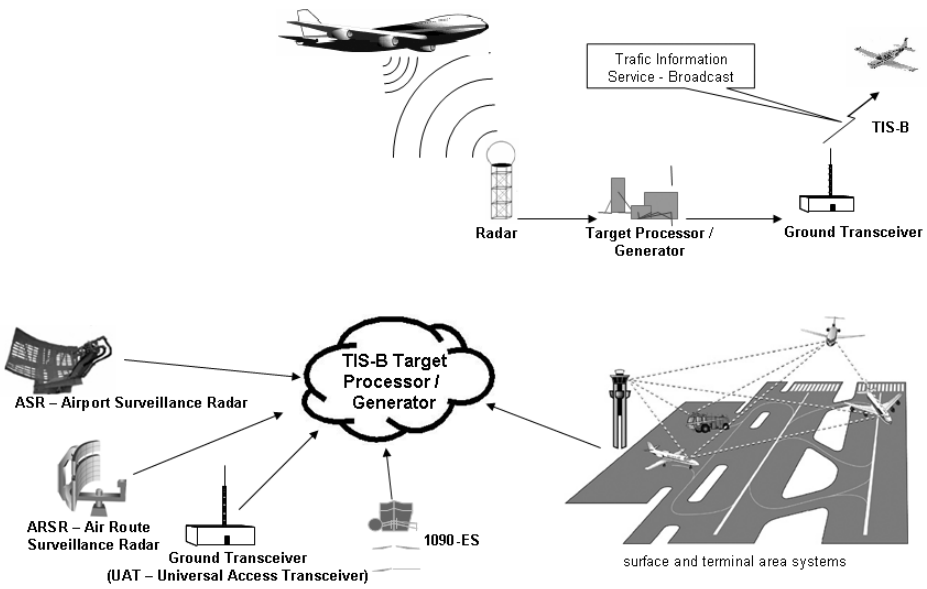


Figure 1.1 - Un système de contrôle aérien

21. Cf. son ouvrage, *La théorie physique, son objet – sa structure*, Vrin 1989 (1ère édition 1906) ; c'est un texte profond, bien plus intéressant que celui de T. Kuhn, *La structure des révolutions scientifiques*, souvent cités par les informaticiens.

Tous ces systèmes ont leur origine dans les systèmes de défense anti-aérienne, issus de la guerre-froide, comme le STRIDA en France. Une abondante littérature[22] existe sur le sujet.

Dans un système d'information d'entreprise, les équipements sont généralement plus simples (postes de travail, serveurs, équipements réseaux, imprimantes, robots d'archivage, etc.) ; la complexité vient cette fois des utilisateurs qui peuvent être nombreux et dont les comportements sont sujets à l'erreur humaine, voire à des actes de malveillances. Avec les systèmes ouverts et l'interopérabilité, des règles de sécurité inexistantes ou mal respectées par les utilisateurs, la complexité est de nature sémantique ; les comportements humains font partie du système ; ils sont généralement plus complexes que ceux d'une machine sans intelligence ! On peut schématiser la relation du système informatisé à la réalité comme suit (figure 1.2) :

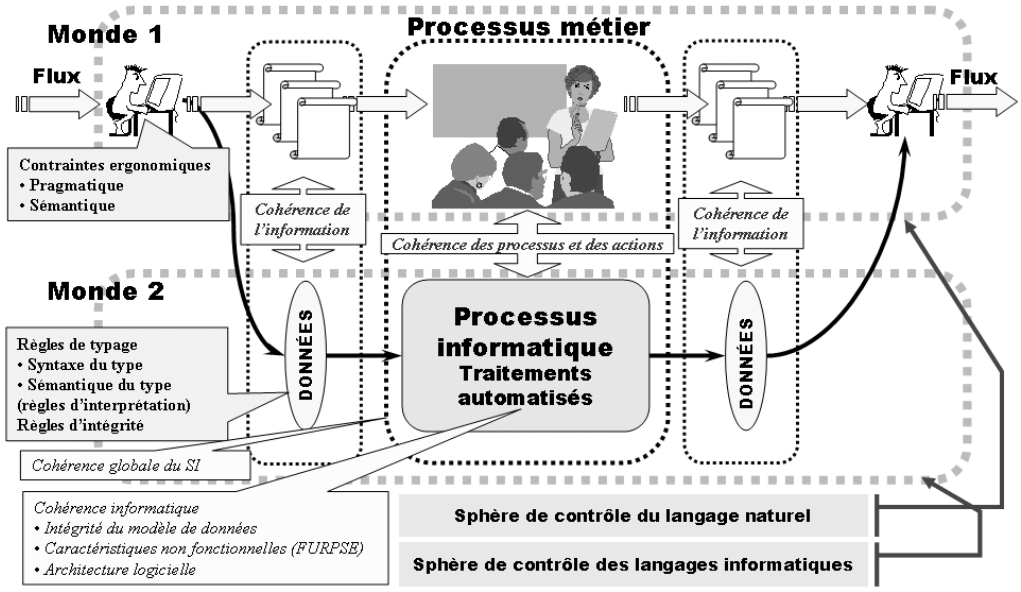


Figure 1.2 - Articulation du monde réel et de l'informatique

L'interface la plus fondamentale est à l'articulation des mondes M1 et M2. M1 est le monde du langage naturel et des connaissances accumulées au cours des ages ; il a la flexibilité et l'adaptabilité du cerveau humain. M2 est un monde figé, rigide qui ne se modifie que par des interventions humaines et dont l'auto-adaptation est limitée à ce qui a été prévu par l'architecte ; la sûreté de fonctionnement exige que les défaillances et pannes soient reproductibles. À un certain niveau de granularité, M2

22. Voir par exemple l'article de T.S. Perry, *In search of the future of air traffic control*, IEEE Spectrum, Vol. 34, N°8, August 97 qui relate les difficultés rencontrées dans la refonte du système de contrôle aérien américain, ainsi que le site de la FAA ; cette refonte fut un échec qui coûta 2,6 milliards de dollars au contribuable américain.

est un monde déterministe. Les « vérités » dans M1 et M2 sont régies avec des logiques quasi antinomiques, vérité modale dans M1 avec référence constante au sens commun et à la sémantique qui au bout du compte juge ; un mélange de déterminisme et de modalité dans M2 où les ambiguïtés (quand le système est capable de les détecter) nécessitent un arbitrage par l'opérateur. Dans la mesure où les systèmes informatisés se réinjectent dans la réalité, on peut parler de co-évolution, comme dans les écosystèmes biologiques.

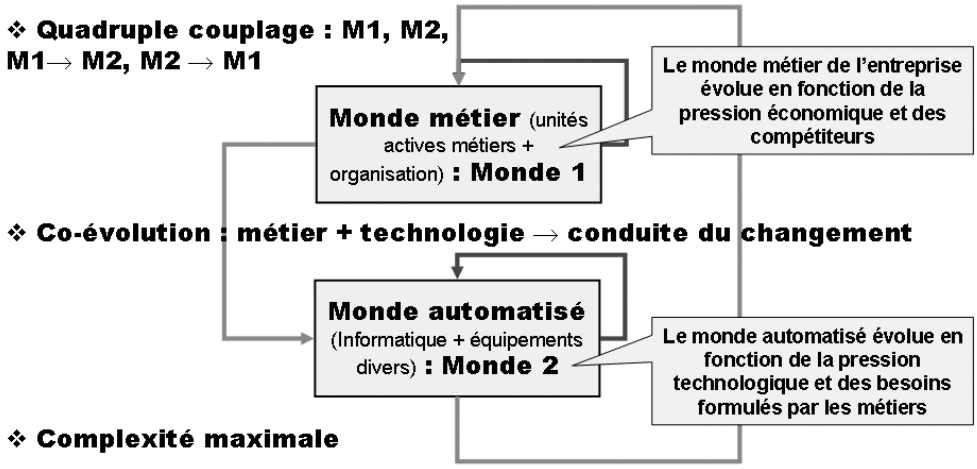


Figure 1.3 - Couplage du monde réel et de l'informatique

L'interface qui assure la cohérence monde réel ↔ modèles informatiques est l'interface homme-machine IHM, que l'on peut schématiser comme suit (figure 1.4).

L'architecture logicielle concerne la programmation du système IHM-GUI (GUI : *Graphical User Interface*) et celle de l'automatisation de tout ou partie des processus et des tâches métiers qui constituent « l'intelligence » du système vue de l'informatique.

Par rapport aux technologies de l'information, l'un des ouvrages marquants écrits ces trente dernières années est certainement *The sciences of the artificial*, d'Herbert Simon dont la première édition date de 1969, révisée en 1996. Le chapitre 8 de cette 3ème édition, *The architecture of complexity : Hierarchic systems* est particulièrement intéressant quant à l'attitude à adopter face à la complexité du monde de l'information. Pour organiser l'information, les civilisations humaines ont inventé deux types de systèmes : a) les systèmes linguistiques qui permettent de décrire et de mémoriser la réalité humaine afin de communiquer et de faire partager ses expériences et b) les systèmes de classification qui permettent d'organiser les connaissances selon une typologie partagée par les usagers du système, ce qui implique une négociation et un équilibrage entre les acteurs de cultures différentes, avec des points de vues différents. Il n'est donc pas étonnant de retrouver ces deux types de systèmes au cœur des

technologies de l'information. Les notions de classification et de langage jouent ici le rôle de structure « mère » pour l'informatique.

La présentation détaillée de différents modèles d'architecture, selon la nature des systèmes à réaliser est l'objet de la 3e partie.

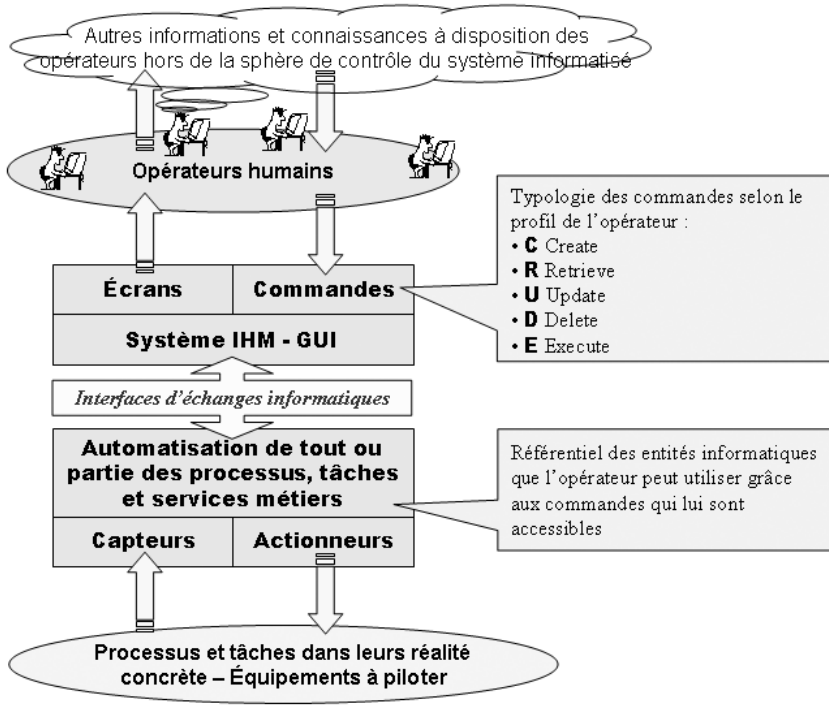


Figure 1.4 - La frontière IHM entre le réel et l'informatique

1.3.1 Le contenu informationnel d'une application

Il est intéressant de « sentir », aux normes de l'édition, ce que peut représenter le texte d'un grand système informatique. Un système d'exploitation totalise plusieurs millions d'instruction en langage source. C'est-à-dire, pour un million de lignes source, un texte brut équivalent à 100 volumes de 200 pages chacun (une instruction par ligne ; environ 50 lignes par page). En comptant les annexes documentaires indispensables à la compréhension des acteurs concernés par le texte, on peut doubler ou tripler la taille. Sans un principe organisateur, un tel texte est totalement hors de portée de compréhension des capacités du cerveau humain le mieux éduqué. Tous ceux qui ont eu à se confronter à ce type de système savent par expérience qu'une documentation qui ne fait que paraphraser le texte du programme est nuisible, car redondante et jamais à jour ! Ce qui est vraiment utile aux acteurs ce sont les raisons[23] qui ont conduit à faire les choses de telle ou telle façon.

Exemples de taille de quelques grands systèmes

Dans la brève histoire de l'informatique, les premiers très grands systèmes ont été les systèmes d'exploitation et les systèmes pour la défense. Comme exemple de systèmes d'exploitation on peut citer le volume de Windows NT : 3 800 000 lignes sources (LS). On notera que ce chiffre puisé aux meilleures sources[24] est assez différents des chiffres ridicules qui circulent sur Windows NT (selon certains auteurs, entre 10 et parfois jusqu'à 30 millions de lignes). Pour que les chiffres aient du sens, il faut une technique de comptage précise, sinon on risque de compter plusieurs fois les mêmes lignes, ou de compter des applications qui n'ont rien à voir avec le système d'exploitation. Le chiffre de 3,8 millions de LS est tout à fait conforme à ce que l'auteur a lui-même pu constater de visu lorsqu'il était l'un des directeurs du projet GCOS7, chez Bull, dont la taille était de l'ordre de 4,5 millions de LS[25], taille également comparable à d'autres systèmes d'IBM (MVS) ou de DEC (VMS), ou de LINUX (2,3 millions de LS).

Comme exemple de système de défense, nous pouvons utiliser les chiffres communiqués lors d'une soutenance de mémoire d'ingénieur CNAM[26] concernant le système SENIT (Système d'Exploitation Naval des Informations Tactiques) du porte-avions Charles de Gaulle. C'est une famille de systèmes temps réels, très complexes, dont l'origine remonte aux années 60, fondés sur le système NTDS[27] de l'US Navy (*Naval Tactical Data System*), premier du genre. La dernière version du SENIT, écrite en Ada et C++, avec ses applications, totalise environ cinq millions de LS. Un tel système réalise toute la surveillance (RADAR, SONAR, systèmes d'armes, etc.) nécessaire à la protection d'un porte-avions et visualise l'information sur consoles dans les centres de commandement du navire.

Comme exemple de système d'information[28], nous utiliserons les données communiquées par la MSA[29] concernant le produit AGORA utilisé par les Caisses de Mutualité Sociale Agricole, pour leurs cinq millions d'adhérents (100 milliards d'€ de prestations versées). Le système est organisé en 7 applications principales et 28 applications secondaires (environ 10 000 programmes COBOL) totalisant 26 millions de LS. La base de données principale du système comporte 800 types

23. Un très bon exemple de ce type de documentation est le rapport : *Rationale for the design of the GREEN programming language*, de l'équipe Bull qui gagna l'appel d'offre du DOD pour le langage Ada. N.Wirth disait que « ce qui est important dans un langage n'est pas ce qu'on y met, mais ce qu'on n'y met pas ! ».
24. M. Cusumano, R. Selby, *Microsoft secrets*, Free Press, 1995.
25. Ces chiffres sont vérifiables aux archives Bull, et ont été communiqués lors d'un colloque consacré au 30ème anniversaire de la première release de GCOS7, en 2004, au CNAM.
26. *Etude des limites du langage Ada pour la réalisation d'une application temps réel*, Mars 2001 ; aux archives du CNAM de Toulouse.
27. Sur l'histoire de ces systèmes, il faut lire de D. Boslaugh, *When computers went to sea*, The IEEE Computer society, 1995. C'est une mine d'information sur la naissance de l'ingénierie système.
28. Dans une communication privée, Y. Caseau nous a indiqué que le SI de Bouygues Telecom totalisait environ 2 millions de points de fonctions.
29. Voir le site du club des maîtres d'ouvrage www.clubmoa.asso.fr.

d'articles. Comme beaucoup de systèmes de gestion, le produit AGORA a un fort taux d'évolution lié aux contraintes réglementaires. L'accès en ligne à l'information via Internet a nécessité le développement de nouveaux modules écrits en JAVA, soit 1,5 millions de LS nouvelles.

À supposer que l'on soit capable de trouver un langage et des abstractions qui condensent de tels textes d'un facteur 10, on arrive, pour un système de 5 millions de lignes source, à 50 volumes, ce qu'une vie humaine ne suffit pas à produire. Pour arriver à quelque chose d'humainement assimilable, il faudrait parvenir à un facteur 100 pour tomber à 5 volumes. Ce texte ultime, véritable armature logique du système, constitue le référentiel architectural du système. Le contenu informationnel de ce texte régule le comportement de tous les acteurs qui ont affaire avec le système.

On peut cependant imaginer qu'un petit groupe d'individus particulièrement doués soit capable de le produire, et que d'autres groupes, plus nombreux, munis de bons principes de construction (i.e. la méthode), soient capables de le traduire en une variété de textes finalement exécutables sur un ordinateur. C'est un fait avéré qu'à la source de tous les grands projets qui ont réussi, il y a toujours une très petite équipe noyau, moins de 10 personnes, qui constitue le germe initial et le centre organisateur du système[30]. Dans la théorie du « chaos » et des systèmes dynamiques, c'est ce que l'on appellerait un attracteur.

C'est ce texte condensé, accompagné de son mode d'emploi et des justifications qui fondent les choix effectués, qu'on peut appeler « architecture du système » ; il est ce que l'on peut le mieux comparer aux plans qui sont produits par l'architecte en génie civil, plans à partir desquels le chantier va pouvoir s'organiser. Ce texte est un « génotype » comme celui que constitue notre ADN par rapport aux êtres que nous sommes (qui sont des phénotypes). Le texte architectural définit une classe de systèmes dont une instance réalise un système particulier.

1.3.2 Description de l'architecture par un langage – Machine abstraite

L'activité architecturale consiste à découvrir le langage le mieux adapté au problème à résoudre, puis à s'assurer que l'on saura « compiler » et/ou traduire ce texte jusqu'à obtention du texte exécutable sur une plate-forme d'exécution pouvant comporter plusieurs machines interconnectées, ce texte étant fabriqué via un environnement de programmation lui-même plus ou moins complexe. Cette traduction se fera :

- soit par traduction automatique et/ou traduction humaine, totale ou partielle, d'une forme syntaxique,
- soit par compilation et/ou traduction automatique, sans intervention humaine, d'un langage de programmation (syntaxe + sémantique parfaitement définies),
- soit par interprétation directe du texte à l'aide d'une machine abstraite.

30. C'est le cas pour l'OS/360 et le système AS/400 chez IBM, du système SAGE pour la défense antiaérienne du DOD, etc. les exemples abondent.

Dans le monde des SI, les générateurs d'application ont été un puissant moyen de programmation des très grands systèmes de gestion comme les MRP/ERP/PGI, impensables en COBOL de base, vue la taille. On peut donc s'attendre à ce que la création de langages *ET* la création de machines dites « abstraites » qui interprètent ces langages et en déterminent la sémantique, soient au cœur de l'activité architecturale. Ces deux notions fondamentales entretiennent des relations de dualité dont l'importance a été soulignée dès les débuts de l'informatique[31].

Dans la notion de machine abstraite, il y a cinq concepts fondamentaux, nécessaires et suffisants, qui constituent les organes logiques de la machine :

1. La mémoire de la machine qui contient les données et les opérations sur les données (i.e. un programme au sens basique du terme). Les données doivent être conformes à une nomenclature des types autorisés, connus de la machine ; d'un point de vue linguistique, les types de données sont des « signifiants », alors que les valeurs des données sont des « signifiés », porteur de sémantique. Les données peuvent être agrégées et organisées en tableaux, en ensembles, en arbres, en graphes, etc., selon le niveau de structuration souhaité. L'ensemble données et opérations associées à ces données est appelé TAD (Type Abstrait de Données) pour montrer qu'ils forment une entité sémantique bien définie. L'ensemble des données de type arithmétique et les opérations arithmétiques {+, -, ×, /, etc.} associées constitue un TAD primitif de la machine.
2. Les opérations (instructions et fonctions), qui permettent d'une part de référencer les données précédemment définies dans la mémoire (c'est un adressage, permettant de manipuler le « signifié » via le type de la donnée) et d'autre part de les transformer. Instructions et fonctions peuvent être exprimées au moyen des opérations de base de la machine agencées en programmes. On peut les classer en cinq catégories principales, résumées par le sigle CRUDE, pour : **C**reate (création d'entités), **R**etrieve (extraction-recherche d'entités), **U**pdate (mise à jour d'entités), **D**elete (suppression d'entités), **E**xecute (mise en exécution d'une entité procédurale, via un appel de procédure synchrone ou asynchrone).
3. La structure de contrôle de la machine — c'est un automate — qui permet de contrôler l'enchaînement des opérations par interrogations des mémoires de contrôle (compteur ordinal, mot d'état, ports de la machine, etc.) conformément aux règles établies par le programmeur et/ou d'occurrence d'événements en provenance de l'environnement avec lequel le programme interagit, et qui sont susceptibles d'en modifier l'ordonnancement. Le contrôle est effectué au moyen d'instructions et/ou de fonctions qui permettent de modifier le flot

31. Cf. la référence, dans ce domaine, est : Hopcroft, Ullman, *Formal languages and their relation to automata*, Addison Wesley, 1969 ; voir également, P. Denning, J. Dennis, J. Qualitz, *Machines, languages and computation*, Prentice Hall, 1978. Un exemple de réalisation concrète est donné dans le livre remarquable de F. Soltis, *Inside the AS/400*, Duke Communications, 1997, qui fut le *chief architect* de cette machine révolutionnaire d'IBM.

d'exécution et/ou d'analyser le statut de l'opération effectuée. NB : à la fin de chaque opération, l'automate d'enchaînement consulte les mémoires de contrôle pour trouver la prochaine opération à exécuter.

4. La structure de surveillance, qui est également une structure de contrôle, mais qu'il convient de distinguer soigneusement de la précédente car son rôle est de garantir l'intégrité des organes et du fonctionnement de la machine. C'est une structure purement interne qui améliore la disponibilité de la machine. Au sens de la théorie de l'information, c'est une redondance qui compense les aléas de toute nature (erreurs, incidents, détérioration, etc.) qui peuvent affecter le bon fonctionnement des organes de la machine. Dans un monde parfait, cette structure serait vide. Dans une machine réelle l'organe de surveillance est le processeur de service qui est lui-même une vraie machine.
5. À ces structures, il faut rajouter les organes nécessaires pour l'interaction avec l'environnement, les entrées-sorties et les différents ports de la machine, i.e. les canaux. À ce niveau, on parlera de canaux logiques. Dans une machine réelle, les canaux sont des machines spécialisées qui gèrent les protocoles d'interaction avec le « monde extérieur » ; leur nombre est une composante importante de la performance et du prix de la machine.

Structures de données et opérations constituent la partie TRANSFORMATIONNELLE du programme ; les automates de contrôle constituent la partie RÉACTIVE[32] du programme (NB : on pourrait utiliser la terminologie des « aspects » de la programmation dite par aspect).

Cette machine logique est strictement conforme à l'architecture dite de von Neumann. Le référentiel qu'elle détermine est applicable tout à la fois dans la réalité informationnelle, et dans la partie informatisée de cette réalité. On peut la schématiser comme suit (figure 1.5).

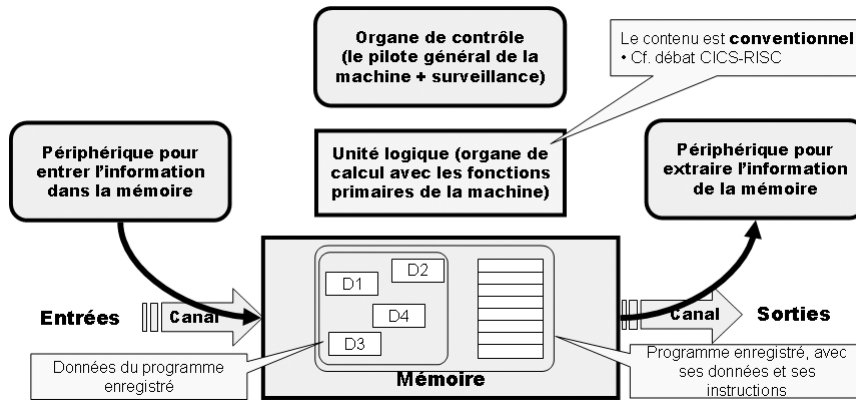


Figure 1.5 - Architecture logique d'une machine de von Neumann

32. Cf. Z. Manna, A. Pnueli, *The temporal logic of reactive and concurrent systems*, Springer-Verlag.

L'innovation fondamentale de cette architecture est la dualité code/données, interchangeables, dans une mémoire commune. C'est à partir du maniement de ces notions et de ces structures premières que l'on va construire l'architecture. C'est ce que nous allons tenter d'éclaircir dans cet ouvrage.

1.3.3 Point de vue systémique et contraintes

Pour déterminer l'architecture la mieux adaptée au contexte, il faut analyser en détail le réseau de contraintes auquel doit satisfaire le système. Le schéma 1.6 met en évidence deux niveaux de contraintes de nature très différentes : a) celles qui sont spécifiques à l'organisation qui utilise le système (le client) et sur lesquelles l'organisation peut agir, et b) celles qui résultent de l'environnement dans lequel le système et l'organisation sont immergés. Les premières sont les caractéristiques qualité FURPSE (*Functionality*, *Usability*, *Reliability*, *Performance*, *System Maintainability*, *Evolutivity*) au sens de la norme ISO 9126 ; les secondes sont les facteurs PESTEL, selon la terminologie de l'INCOSE[33].

Sur la base de travaux comme ceux de Mc.Call, Richards et Walters, *Factors in software quality,* chez General Electric, en 1977, on a commencé à dresser des nomenclatures de propriétés permettant de mieux cerner les caractéristiques qualité attendues du produit logiciel. Il en est résulté une norme ISO/CEI 9126, *Caractéristiques qualité des produits logiciel*, disponible en 1993, révisé en 2002. Il existe une abondante littérature sur le sujet[34].

L'essentiel de la norme, avec les caractéristiques principales et les sous-caractéristiques, peut être résumé par le schéma 1.6.

Chacune des caractéristiques, résumée par l'acronyme FURPSE, doit être suivie tout au long du cycle de développement du produit logiciel, ce qui garantit, au final, que le produit réalisé est conforme à l'attente de l'utilisateur.

Description des facteurs PESTEL (selon la terminologie de l'INCOSE) :

- **Politique** (Acceptabilité du système du point de vue de la vision des pouvoirs politiques et/ou des politiques publiques) ; si un acteur majeur (homme politique influent, syndicats, groupe de pression, etc.) est contre, le système s'enlisera.
- **Economique** (Pression du marché – Contraintes économique résultant de la compétition entre les acteurs – Indépendance nationale) ; la constellation GALIEO est un enjeu double pour l'Europe, à la fois économique et stratégique, vis-à-vis du GPS américain, sous contrôle du US DOD)

33. *International council on systems engineering* ; cf. le document *Systems engineering technical vision*.
34. Cf. L. Chung, et Alii, *Non-functionnal requirements in software engineering*, Kluwer, 2000.

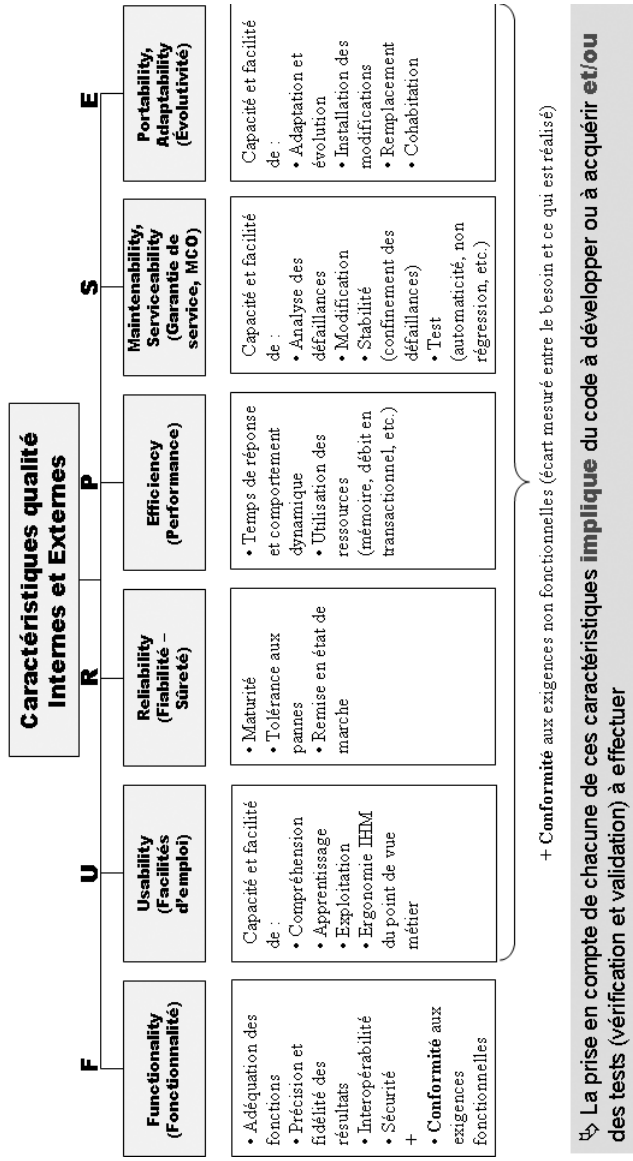


Figure 1.6 - Caractéristiques qualité des produits logiciel

- **Social** (cf. le *Human use of human being*, livre célèbre de N. Wiener, inventeur de la cybernétique – *Psychological and sociological aspects*) ; par exemple les OGN, rejeté par une partie de la population, ou le dossier médicalisé qui viole la vie privée des personnes, etc., avec au bout du compte un « principe de précaution » incontrôlable, à la limite de l'anti-science. C'est de la conduite du changement au niveau de la société, pas vraiment évidente.
- **Technologique** (*The characteristic of advanced technology systems are often unpredictable*) ; l'ordinateur en est le meilleur exemple, Internet en est un

autre. La technologie change le monde, mais jusqu'où faut-il aller sans faire peur (cf. le principe de précaution).

- **Ecologique** (*Environmentally friendly*), par exemple éviter une fracture numérique entre ceux qui savent ou qui peuvent, et les autres.
- **Légal** (*Code of ethic and public perception of risk and liability*) ; on en a des témoignages tous les jours avec le logiciel « libre », les téléchargements pirates, la copie de programmes, le viol de la vie privé via le téléphone portable, le dossier médical, les cartes de crédits, etc.

Du point de vue systémique de l'information on peut représenter le système global et les interactions entre les sous-systèmes de la façon suivante (figure 1.7) :

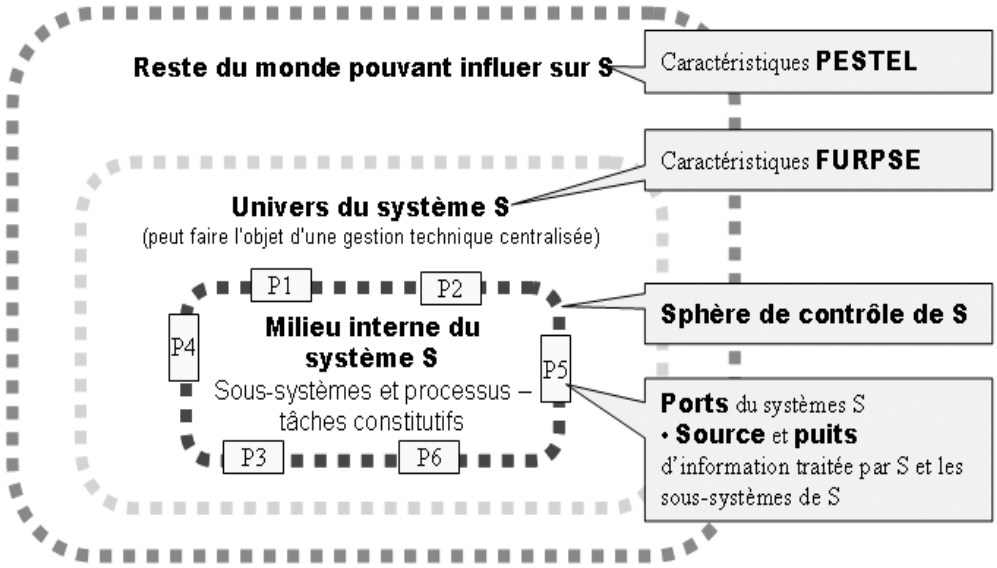


Figure 1.7 - le système d'information et son environnement

Les contraintes FURPSE et PESTEL, et les contraintes projet CQFD, font office de conditions aux limites. Il faut les répertorier et les valider avec le plus grand soin, car elles vont déterminer la solution architecturale. Elles devront être respectées sur tous les niveaux d'abstraction du système.

Niveau système

Le système automatisé est dans un univers (au sens logique du terme) système dont il peut contrôler certains aspects, via les caractéristiques qualité FURPSE. Cet univers est lui-même partie prenante d'un écosystème qu'il ne contrôle pas mais avec lequel il interagit (facteurs-critères PESTEL). À ces facteurs sont associées les notions de risques et de criticité pour ce qui concerne l'impact des défaillances et des pannes du système automatisé sur l'écosystème. On distingue :

a- la sûreté de fonctionnement du système (i.e. *safety critical*) et la tolérance aux pannes (i.e. *fault-tolerant systems*), qui concourent à la confiance des acteurs dans le système automatisé ;

b- la criticité de la mission (i.e. *mission critical*, cas des sondes spatiales, des constellations de satellites, etc.), avec abandon irréversible de la mission en cas de pannes ;

c- le risque de pertes économiques (i.e. *business critical*, cas des systèmes bancaires, du trading, des systèmes gouvernementaux, etc.) qui peuvent mettre enjeu la survie de l'entreprise, voire du pays ;

d- la vie humaine (i.e. *life-critical*, assistance médicalisée, handicapés, etc.) bien connu dans le contrôle aérien, le nucléaire et les transports.

Tout ce qui fait partie du système S est dans la sphère de contrôle de S. Ce qui entre et ce qui sort passe obligatoirement par les ports logiques du système. Le système, selon sa taille, est lui même décomposé en sous systèmes qui interagissent entre eux, comme on le verra ci-après.

Interactions entre les sous-systèmes et processus constitutifs de S – Percolation des sous-systèmes et interopérabilité – Couplages

En systémique, on définit un système[35] comme un ensemble d'éléments, ouverts sur l'environnement, qui interagissent en vue d'une finalité. Un élément peut lui-même être un système, ce qui en fait une notion récursive. Sur un plan général on peut représenter les interactions au sein d'un système de systèmes par les schémas suivants (figure 1.8).

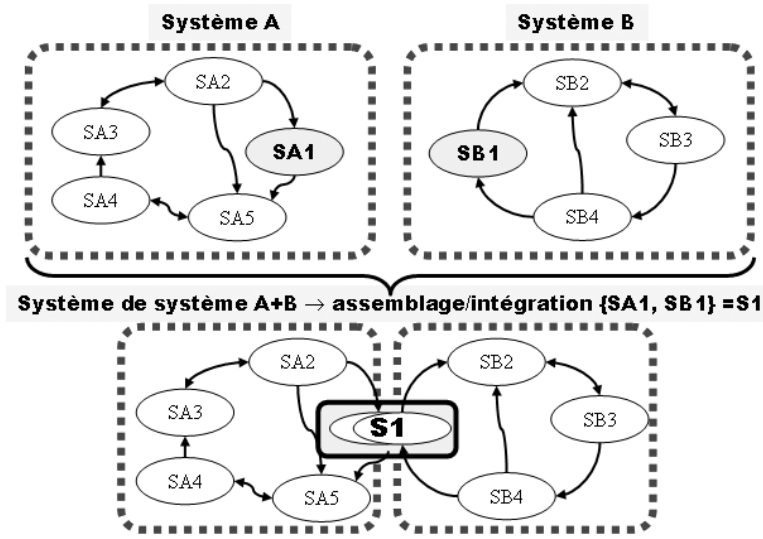


Figure 1.8 - Interopérabilité et percolation des sous-systèmes

35. Cf. J. Farrester, dans la bibliographie.

À un certain moment, l'architecte constate que les systèmes SA1 et SB1, appartenant à deux ensembles différents, font la même chose, ou des choses très voisines. Il devient économiquement rentable de les fusionner, ou tout au moins de les faire interagir, ce qui au final amènera une interaction plus globale entre SA et SB. En physique, ce phénomène s'appelle une percolation. Dans les systèmes d'information, c'est ce qu'on appelle interopérabilité : un ensemble de sous-systèmes qui partagent des informations d'intérêt pour chacun d'entre-eux se fédère pour devenir un système de systèmes (i.e. une fédération de systèmes qui acceptent des règles communes, conformes au principe de subsidiarité).

La communication entre les sous-systèmes se fait via des éléments (entités architecturales) communs au deux, mais représentés selon les règles de chacun d'eux. Pour que cela soit possible il faut que les caractéristiques FURPSE et PESTEL de chacun des sous systèmes soient compatibles (données, fonctions, événements émis/reçus, comportements et environnements).

Notion de sous-système pivot

Dans le schéma 1.8, l'élément S1 résulte de la fusion SA1/SB1 ce qui revient à reprogrammer chacun de ces éléments, et de ce fait à réaliser un couplage étroit entre les systèmes A et B. Une analyse plus poussée des interactions SA↔SB, qui ne sont pas nécessairement identiques, peut faire apparaître une abstraction commune à SA et SB, ce qui donnera une représentation dite « pivot » que l'on peut représenter comme suit :

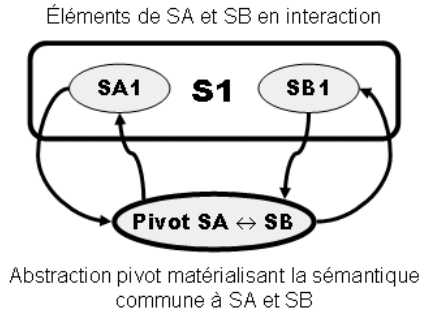


Figure 1.9 - Sous-système pivot permettant le couplage de deux systèmes

Dans cette architecture, le pivot agit comme un filtre qui laisse passer dans un sens ou dans l'autre les informations que les systèmes SA et SB ont décidé de s'échanger (aux erreurs près). À côté de sa fonction logique, il a une fonction fondamentale vis-à-vis de ce qui entre et de ce qui sort de chacun des systèmes, conformément au contrat de service négocié entre SA et SB. Le pivot a une fonction immunitaire et participe de ce fait à la surveillance générale du système. Toute ambiguïté, toute violation non détectée du contrat fait courir un risque quant à la sûreté de fonctionnement de chacun des systèmes et a fortiori de la fédération SA+SB.

L'architecture fonctionnelle de ce type d'interface très importante sera précisée aux chapitres 10 (Quelques modèles d'architectures) et 15 (*Interfaces)*.

Notion de ports logiques – Les sources et les puits d'information

Les interactions du système avec son environnement se matérialisent dans le système via des mémoires particulières (du point de vue sémantique) appelées « ports ». Un port est attaché à un équipement particulier, ou à une classe d'équipements génériques présentant des caractéristiques communes, qui agissent, du point de vue du système, comme des organes sensori-moteurs (i.e. capteurs et actionneurs). C'est par les ports que l'information entre et sort, et que le système agit sur son environnement.

Si l'on assimile un élément du système à une machine au sens de l'architecture de von Neumann, on est en droit de représenter un élément quelconque du système comme suit (figure 1.10) :

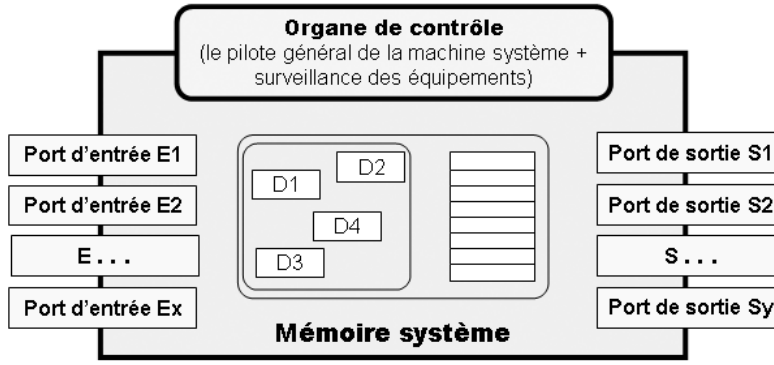


Figure 1.10 - Le système et ses ports logiques

Les ports sont des mémoires communes entre les différents équipements et organes périphériques qui agissent dans le système pour en réguler le comportement conformément à la finalité qui lui a été assignée. L'accès à ces mémoires particulières se fait de façon synchronisée avec des instructions particulières comme les sémaphores qui seront décrites au chapitre 7, via les organes de contrôles de la machine ou du système. Chaque port a un pilote spécifique (non représenté sur le schéma) qui opère sous le contrôle du pilote général.

2

Les matériaux de l'architecture logicielle

Tentatives de définitions et terminologie

2.1 DE QUOI SONT FAITS LES PROGRAMMES INFORMATIQUES ?

Dans les années 70, N.Wirth, l'inventeur du langage PASCAL qui fut pendant longtemps un modèle de clarté et de simplicité, a écrit un livre[1] resté célèbre, *Algorithms + Data structures = Programs*. C'est une première piste pour identifier ce que sont les constituants d'un programme.

Plus près de nous, Z. Mana et A. Pnueli, dans leur ouvrage *The temporal logic of reactive and concurrent systems*, classent les programmes en deux catégories :

1. Les programmes TRANSFORMATIONNELS dont la finalité est de produire un résultat conforme à la spécification exigée par l'utilisateur du système. Cette catégorie de programme est la plus connue, car elle correspond à des programmes qui transforment un état initial en un état final qui constitue le résultat du programme, à l'aide d'une fonction transformatrice, i.e. un algorithme. Un algorithme a un début et une fin. C'est un programme au sens de Wirth, ci-dessus.
2. Les programmes RÉACTIFS dont la finalité est de maintenir une interaction avec l'environnement dans lequel s'exécute le programme. Tous les moniteurs mis en œuvre par le système d'exploitation et le système d'exploitation lui-même sont des exemples de programmes réactifs. La finalité d'un programme

1. Chez Prentice-Hall, 1976.

réactif est de garantir le comportement attendu du programme, par exemple : a) de répartir équitablement les ressources entre des programmes transformationnels qui consomment ces ressources de façon aléatoire (mémoire, entrées-sorties, bases de données, fichiers, sessions de communications, etc.) ; b) de lancer une tâche prioritaire à la survenue de tel événement. Une fois lancé, un programme réactif peut ne jamais s'arrêter, sauf sur intervention d'un opérateur externe.

Dans les programmes réels, avec les langages de programmation de l'industrie, les catégories, TRANSFORMATION et RÉACTION, sont plus ou moins enchevêtrées. Comme langages de programmation, il faut non seulement considérer les langages généraux comme C, C++, Java, C# pour ne mentionner que les plus récents, mais également les langages dit de « *scripting* » ou de paramétrage qui généralisent la vieille notion de JCL, et qui produisent des textes très abondants, comme les langages PERL, PHP ou PYTHON ; certaines applications Web sont écrites exclusivement dans ces langages.

Sur cette base, nous considérerons qu'un programme est formé de trois constituants :

1. Le constituant DONNÉE, i.e. la structure de données manipulée et les mécanismes d'adressage qui lui correspondent.
2. Le constituant OPÉRATION, i.e. l'algorithme ou la procédure qui spécifie la fonction transformatrice effectuée par l'algorithme.
3. Le constituant RÉACTION, i.e. la spécification du comportement résultant des modes d'interactions du programme et de son environnement. Dans sa formulation la plus simple, par exemple en COBOL, c'est le début et la fin du programme COBOL. Dans sa formulation la plus complète, comme en Ada 95, c'est la gestion des tâches et des exceptions incorporée au programme.

Nous complétons l'équation de N.Wirth comme suit :

```
programme =  structures de données + algorithmes + comportements
```

Pour effectuer les transformations sur les données, le programme a besoin de ressources, ce qui nous permet d'écrire :

```
transformation = programme + ressources
```

Par analogie avec son lointain prédécesseur bâtisseur de cathédrale, l'architecte logiciel doit parfaitement maîtriser la « physique » des constituants des programmes pour garantir la qualité de l'assemblage final qui constituera le système répondant aux besoins des utilisateurs.

Tout programme agrège ces trois constituants dans des proportions variables pour constituer, selon les terminologies en usage, les {pièces – composants – modules – intégrats} de rang 0, qui eux-mêmes serviront à construire des {pièces – composants – modules – intégrats} agrégés de rang supérieur, selon les principes de la décomposition hiérarchique en classes disjointes, i.e. une taxonomie. La terminologie est fluc-

tuante car tous ces termes ont été également utilisés pour dénoter des concepts de structuration de la programmation, d'où parfois une certaine confusion entre les constructions architecturales et les structures syntaxiques des langages de programmation. La terminologie architecturale la plus simple est la notion de « module » au sens que lui a donné D. Parnas[2] ou de « bloc d'intégration/building-block » (c'est la même chose) que nous avons traduit par le néologisme « intégrat » pour indiquer que l'entité logicielle a un sens *du point de vue du processus d'intégration*. L'architecte définit les intégrats, i.e. modules ou « building-blocks » qui ont un sens : a) pour ceux qui les construisent, correspondant à des tâches bien définies dans le projet – c'est une condition première d'existence – puis : b) pour ceux qui les assemblent et qui auront à valider l'assemblage complet ou partiel pendant le processus d'intégration ; c) pour ceux qui les maintiennent , et enfin d) pour ceux qui les exploitent et qui auront à diagnostiquer les pannes en exploitation. D'où l'importance de trouver le bon compromis architectural entre tous ces acteurs et les « bons » intégrats. Un intégrat modélise un sous-ensemble de la réalité que l'architecte souhaite informatiser ; il peut en valider le comportement par rapport à la réalité.

La notion de composant logiciel, au sens que lui a donné C. Szyperski[3] – unité de déploiement indépendante, utilisable par des tiers via ses interfaces, dont l'état interne n'est pas observable – est identique. Cependant, le terme composant fait penser aux composants matériels ou puces électroniques, ce qui est une métaphore trompeuse, car le logiciel ne se comporte absolument pas comme le matériel.

La matérialisation de ces notions architecturales dans les langages de programmation se fait avec les constructions syntaxiques autorisées par les langages. L'entité architecturale existe indépendamment du langage qui sert à la représenter. Cela est d'autant plus vrai que la plupart des systèmes sont programmés avec différents langages de programmation ; on en verra un exemple avec les interfaces au chapitre 15. Il faut être très vigilant sur la terminologie car ce n'est pas le nom de la construction syntaxique qui donne la propriété architecturale, par exemple OBJET ou PACKAGE dans les langages objets, c'est usage qui en est fait. L. Wittgenstein, logicien célèbre de Cambridge, disait à ses élèves : « *Don't ask for a meaning, ask for a use* » ; c'est exactement ce dont il s'agit ici.

La définition de la hiérarchie des entités architecturales d'un système est la décision de l'architecte. C'est une organisation conventionnelle, résultant de compromis entre les acteurs métiers (performance et ROI attendus) et les acteurs ingénierie (développement, intégration, maintenance et exploitation), qui formera la nomenclature, ou ontologie, du système (i.e. ce qui porte « l'être » et l'intelligibilité du système). En suivant les normes comme IEEE 1220 ou ISO 15288, on peut définir des hiérarchies comme suit (figure 2.1) :

2. D. Parnas, *On the criteria to be used in decomposing systems into modules*, CACM, December 1972, Vol 15 N°12. Pour une définition plus récente, voir le chapitre 3 (27 pages), *Modularité*, du livre de B. Meyer, *Conception et programmation orientées objet*, Eyrolles, 1997. NB : le terme module était utilisé par les architectes bâtisseurs des cathédrales pour désigner la règle ou étalon définissant l'unité de mesure à utiliser pour la construction.

3. Cf. *Component software, Beyond object-oriented programming*, Addison-Wesley, 2002.

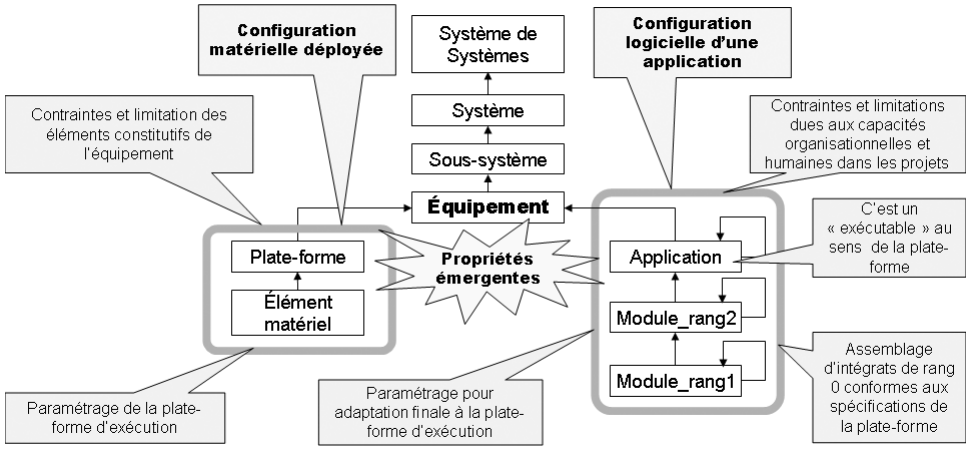


Figure 2.1 - Décomposition hiérarchique d'un système

Ce découpage initial est un acte de naissance architectural fort, car il doit anticiper la logique d'intégration et l'évolution qui viendront beaucoup plus tard dans le cycle de vie du système. Il est conventionnel et il n'y a pas de méthode type recette de cuisine pour découvrir la répartition et le placement le mieux adapté à la situation du projet. C'est la sémantique attribuée aux entités hiérarchiques qui va définir l'ontologie du système. La plate-forme et les éléments matériels constituent la ressource sur laquelle l'application va pouvoir s'exécuter, i.e. effectuer les transformations conformes à sa programmation. NB : l'ensemble plate-forme peut être remplacé par un simulateur.

Au niveau le plus fin de la décomposition hiérarchique proposée par l'architecte, nous avons les intégrats de rang 0 qui constituent les unités élémentaires de l'application. Ces unités sont les « instructions » ou « briques de base » de l'application. Elles peuvent être :

a- Les instructions de base du langage de programmation (en anglais « statement ») ;

b- Des fonctions (souvent appelées primitives) prédéfinies intégrées au langage de programmation comme les fonctions mathématiques ou des fonctions du système d'exploitation ;

c- Des fonctions en bibliothèques définies par l'architecte et construites conformément aux règles de l'application.

Cet ensemble constitue une « machine étendue » (ou machine virtuelle MV, ou machine abstraite à états MAE/ASM[4] qui met au premier plan la notion d'état, indispensable pour la surveillance et la reprise en cas d'incident) avec laquelle le programmeur va véritablement construire et programmer, au sens large, l'applica-

4. Cf. E. Börger, R. Stärk, *Abstract state machines*, Spinger, 2003.

tion. L'architecture logique de cette machine doit être conforme aux principes généraux de l'architecture de von Neumann.

Une bonne façon pour l'architecte de stabiliser les modalités du travail de programmation est de définir la « machine abstraite » que les programmeurs devront maîtriser pour programmer les modules/intégrats de différents rangs constitutifs de l'application. Cette interface MAE joue un rôle fondamental pour donner aux applications ainsi programmées les propriétés d'adaptabilité et d'évolutivité. Nous verrons dans la partie 3, *Architecture fonctionnelle logique*, les propriétés indispensables de cette machine, en particulier la notion de transaction.

L'architecte doit également maîtriser la chaîne de préparation des programmes (au sens du langage de programmation) qui, au final, générera les applications sous une forme exécutable. Le schéma de principe de cette chaîne d'assemblage est le suivant (figure 2.2) :

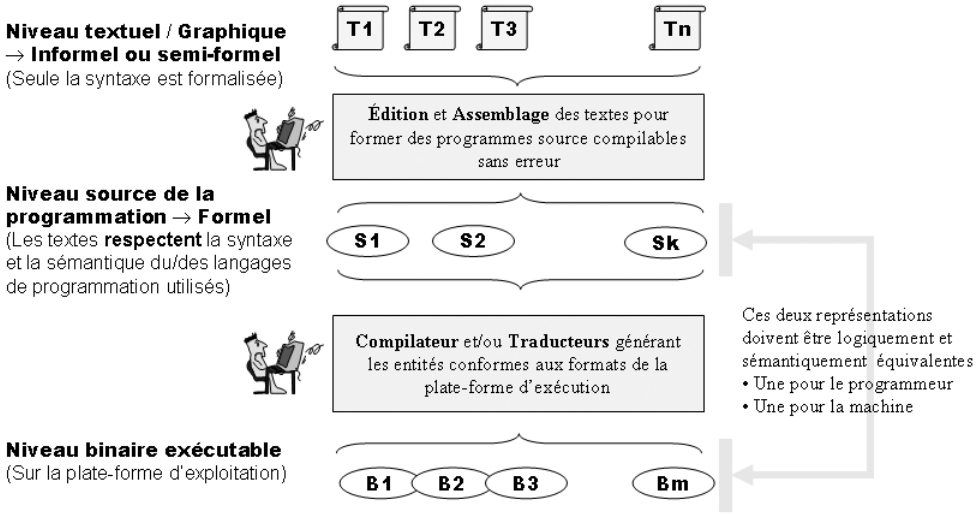


Figure 2.2 - Principe d'une chaîne d'assemblage de programmes

Ce schéma met en évidence la double nature du travail du concepteur-programmeur. Dans un premier temps, le référentiel est de nature textuel, au sens large. Il peut inclure des textes exprimés dans la syntaxe d'un langage de programmation, eux-même partie d'un langage de conception plus ou moins formalisé, généralement au niveau de la syntaxe ; il peut également inclure des schémas plus ou moins précis pouvant être fabriqués avec des éditeurs graphiques plus ou moins sophistiqués. Dans le cas des compilateurs que l'on examinera au chapitre 6, on verra que les grammaires des langages de programmation sont des textes parfaitement formalisés. L'ensemble de ces textes n'est pas exécutable sur une plate-forme.

Dans un second temps, le programmeur transforme et traduit les textes précédents dans les termes du ou des langages de programmation choisis, en respectant les

règles de programmation fixées par l'architecte du système, i.e. celle de la MAE. Ce nouvel ensemble textuel est cette fois exécutable sur la plate-forme, après compilation sans erreur des programmes sources et édition de liens.

Les deux textes sont élaborés de façon itérative jusqu'à obtention de textes complets corrigés, sans erreur et exécutable sur la plate-forme. Le niveau source et le niveau binaire exécutable doivent être sémantiquement équivalents, ce qui veut dire que les défauts dans l'un existent dans l'autre, et réciproquement (principe de déterminisme).

Mécanique du procédé de construction des intégrats – Chaîne d'assemblage

Pour bien comprendre la dynamique de l'intégration, il est essentiel d'analyser les mécanismes mis en œuvre par le procédé de construction des codes sources et de fabrication des binaires tel qu'on les trouve sur les plates-formes de développement et dans les environnements intégrés de programmation, également appelés *software factory*. C'est cet ensemble de mécanismes qui produit les intégrats de rang 0 ; ces intégrats peuvent résulter de l'assemblage, au sens informatique du terme, de plusieurs unités de compilation pour former un tout sémantiquement cohérent. La taille d'un intégrat de rang 0 est purement conventionnelle ; elle peut varier de quelques KLS (Kilo, ou millier, Lignes Source qui est l'unité d'œuvre en programmation) à quelques dizaines de KLS, selon la nature et la complexité des tâches de programmation à réaliser.

Les grandes lignes du procédé de construction mis en oeuvre en intégration peuvent être schématisées comme suit (figure 2.3). C'est la software factory.

Légendes de la figure 2.3

A1 et **A2** sont des adaptations par des textes source que l'on peut incorporer au programme soit avant la compilation à l'aide d'un éditeur de texte, un pré-processeur comme dans C/C++ ou un générateur de programme, soit pendant la compilation avec les fonctions du langage prévues à cet effet (`copy`, `include`, etc.).

A3 est un mécanisme de l'édition de liens statiques qui permet d'incorporer au programme des objets binaires au format des unités de compilation.

A4 est une fonction du moniteur de travaux de la plate-forme qui permet d'incorporer au programme, lors de son lancement, un texte (ce sont des données) qui pourra être lu par le programme à l'aide d'APIs système ad hoc disponibles sur la plate-forme.

A5 est tout simplement un fichier de paramétrage que l'application pourra consulter chaque fois que nécessaire (par exemple des « constantes » pas assez immuables pour pouvoir être déclarées ; 3.14159… est une vraie constante, tant qu'on est en géométrie euclidienne ; la TVA à 19,6% est une « constante » lentement variable).

Les possibilités A1 à A5 existent depuis plus de 30 ans dans les langages de programmation et les systèmes d'exploitation ; A6 et A7 sont beaucoup plus récentes

dans la pratique, bien que connues également depuis plus de 30 ans, car elles nécessitent beaucoup de ressource de performance mémoire + temps CPU.

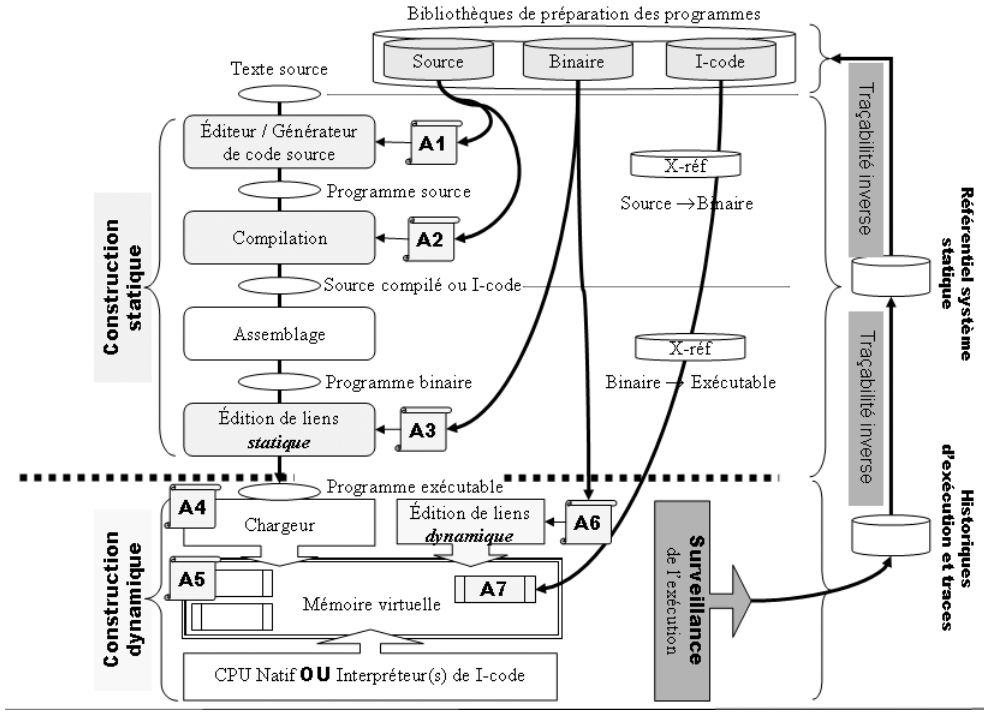


Figure 2.3 - Procédé de construction et d'intégration d'un système

A6 est la possibilité d'incorporer dynamiquement, au moment de l'exécution du programme, du code et/ou des données, lors de leur première référence. C'est le mécanisme d'édition de lien dynamique, apparu pour la première fois avec le système MULTICS du MIT, au début des années 70. C'est un mécanisme d'une extrême puissance, très consommateur de ressources plate-forme, qui permet de différer au plus tard la résolution des références à des informations qui a défaut auraient du être liées via l'éditeur de liens statiques (le terme *lazy evaluation* est parfois utilisé). C'est en fait une façon de mettre à jour dynamiquement l'application, sans avoir besoin de tout recompiler, et surtout de tout ré-intégrer (mais il faut tout de même valider). Seul le protocole d'appel du programme doit être standardisé, son corps viendra au dernier moment.

Les systèmes d'exploitation disposent généralement de la propriété d'introduire des « patches[5] » sur un site particulier. Cette propriété très intéressante, car réversible et locale, permet de corriger les anomalies constatées sur une exploitation particulière sans déstabiliser tous les sites. Le moment venu, on pourra intégrer tous les

5. Au sens littéral, c'est un terme de couture qui signifie « pièce », au sens de rapiéçage d'un vêtement.

patches à l'occasion de la fabrication d'une nouvelle version, ou d'une révision. Les patches sont la matérialisation de nos imperfections et de nos erreurs ; si l'on était dans un monde parfait, il n'y en aurait pas.

A7 est la possibilité de masquer la machine réelle hardware par une machine étendue purement logicielle, comme la JVM (*Java Virtual Machine*), ce qui permet de disposer d'autant de types de machines que de besoin, donc une très grande souplesse, au prix : a) d'une baisse de performance très importante (facteur 100, voire plus !), ce qui avec les microprocesseurs actuels ne pose plus de problème dans certains contextes comme les IHM, et b) d'une augmentation de complexité très importante, car le processus d'IVVT déjà problématique et difficile en mode statique, est cette fois complètement dynamique ; en conséquence, la stratégie d'intégration est intégralement à revoir.

Les machines abstraites à états (MAE/ASM) comme celles mentionnées ci-dessus, peuvent être réalisées sous la forme de machines virtuelles comme la JVM.

La possibilité de compléter dynamiquement une application avec de nouvelles fonctions mieux adaptées au contexte d'emploi, soit par édition dynamique, soit par ajout d'une machine virtuelle, est une capacité fonctionnelle particulièrement bien adaptée à la problématique de l'intégration et de l'évolution des systèmes. Mais encore faut-il soigneusement doser les parties dynamiques de l'application pour ne pas rendre l'ingénierie de l'intégration carrément ingérable, voir même infaisable, tant au niveau de l'équipe d'intégration, que de la logistique que cette nouvelle forme d'intégration implique. C'est ce qui figure sur la partie droite du schéma avec les mécanismes de surveillance et les traçabilités inverses. Sans ces mécanismes, sur lesquels repose toute l'intégrité du dispositif d'adaptation dynamique, il est exclu de faire fonctionner correctement une telle architecture et de donner au maître d'ouvrage une quelconque garantie de qualité.

Le caractère récursif des entités architecturales logicielles est un aspect fondamental de la « matière » logicielle qui a donné naissance, à la notion de langages à structure de blocs, dès les années 50 avec des langages comme ALGOL, et tous ses successeurs. La notion de récursivité, et de fonctions récursives qui viennent directement de la logique mathématique, bien antérieure à l'invention des ordinateurs, a été entièrement intégrée dans les langages informatiques.

Le logicien R.Carnap[6] énonçait, dans son ouvrage *Logical syntax of languages*, ce qu'il appelait Principe de tolérance en syntaxe : « *It is not our business to set up prohibitions, but to arrive at conventions* ». L'architecte doit être imprégné de cet état d'esprit, car en matière d'interfaces et d'intégrats, tout est affaire de compromis et de conventions à négocier entre les acteurs du jeu architectural, dont le nombre doit être minimisé. L'architecte doit être un pédagogue et un communicateur, sans rien céder à la rigueur logique des constructions qu'il échafaude. Il doit s'assurer que les conventions sont respectées par tous. Les contrats, au sens de B.Meyer, sont des con-

6. C'est l'un des membres fondateurs du cercle de Vienne, dont sont issus de très nombreux travaux d'épistémologie, dans les années 1930.

ventions négociées explicites que les programmeurs devront respecter. Les transactions, dont nous parlerons au chapitre 9, respectent une forme particulière de contrat caractérisé par les propriétés ACID.

2.2 NATURE SÉMANTIQUE DES CONSTRUCTIONS INFORMATIQUES

Dans le monde simple des années 70-80 architectes et programmeurs étaient des réalisateurs d'applications informatiques dont les règles de construction relevaient plus du bon sens que de définitions abstraites.

Une application était formée d'un ou plusieurs programmes COBOL (pour prendre un exemple de langage de cette époque) compilés séparément, et d'un ou plusieurs fichiers, appartenant en propre à l'application ou partagés en plusieurs applications. Les programmes d'une application pouvaient s'exécuter, soit de façon séquentielle, soit de façon concurrente (programmation transactionnelle) selon les évènements gérés par l'application, et transmis à l'application sous forme de fichiers de messages organisés en files d'attente où pouvaient jouer différentes priorités.

Une fois compilés, les programmes binaires étaient édités sous une forme exécutable (souvent appelé *Load Module*) via un programme du système d'exploitation appelé éditeur de liens (*Linker*) et/ou relieur (*Binder*). Initialement tout était monolangage, mais dans les années 80, il était déjà courant de rencontrer des applications multi-langages : les langages généraux comme COBOL, FORTRAN, PL1, PASCAL, C, etc., et des langages spécialisés pour les IHM (les FORMS des terminaux alphanumériques), la description des données dans les bases de données (DML et DDL), etc.

Les programmes compilés séparément devaient respecter les critères de modularité du type de ceux proposées par D.Parnas en 1972 et être conformes aux règles de la programmation structurée, afin de former des modules syntaxiquement cohérents. La notion d'interface était déjà un acquis important de cette période, par exemple une information de type donnée ou de type procédural, vu d'un côté de l'interface en COBOL et de l'autre en C.

À titre d'exemple, dans l'architecture en couche des protocoles de communication il existe un langage : ASN.1 (voir bibliographie), qui permet de décrire les données échangées, au bit près, sachant que d'un côté du protocole, on va voir la donnée dans le ou les langages utilisés de ce côté du protocole, lesquels peuvent être différents de l'autre côté. On a donc une articulation comme suit (figure 2.4).

La sémantique des données est un problème en soi (voir la figure 2.21).

La même articulation est possible avec une information de type procédural, sauf que l'on n'a jamais réussi à mettre tout le monde d'accord sur un langage procédural universel complet. D.Knuth dans son *Art of Computer programming* utilisait un assembleur (le langage MIX dont la spécification fait 60 pages) que les architectes-

programmeurs pouvaient reformuler dans le ou les langages de leur choix, à leur charge de garantir la cohérence des traductions. Algol 60 avait cet objectif d'universalité ; IBM avait essayé d'imposer PL1 au début des années 70 en remplacement de COBOL et FORTRAN, avec le succès que l'on connaît ; ce fut un énorme échec commercial, ce qui n'empêcha pas de refaire la même erreur 15 ans plus tard avec Ada. Avec le langage JAVA on peut en théorie faire la même chose avec du code JAVA que l'on distribue sur le réseau, pour autant que les machines déployées disposent de la JVM (*Java Virtual Machine*). Le byte-code de la JVM joue ce rôle pour la programmation en JAVA, mais il faut un interpréteur de la JVM sur la machine qui exécute le byte-code.

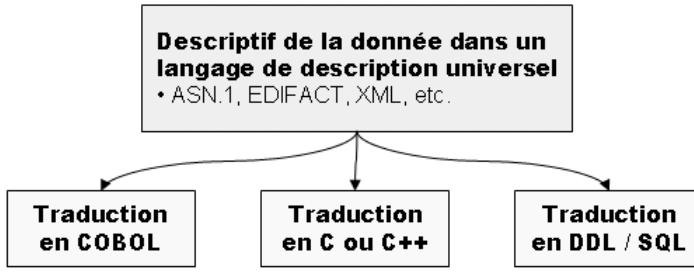


Figure 2.4 - Description des données

Dans la norme SQL, il existe un sous-ensemble dont la sémantique est garantie par le standard[7], ce qui permet d'échanger des requêtes sous forme caractère dans la mesure où elle se limite aux modalités du standard.

Cette approche de la construction a été utilisée plus ou moins judicieusement dans les générateurs d'applications qui à partir d'un texte initial vont construire un texte source compilable sur la plate-forme d'exploitation, ou dans un langage portable sur une grande variété de plates-formes comme COBOL, ou JAVA. C'est la notion de langage pivot, dont le rôle est de définir la sémantique de l'entité ainsi décrite, indépendamment des langages utilisés sur les différentes plates-formes.

La vraie rupture se produit avec le développement de l'informatique distribuée où l'application centralisée se répartit sur différentes machines, de puissance variable, reliées entre elles par des réseaux de débit très différents, eux-mêmes partagés entre différentes applications.

7. ISO 9075, Database language SQL.

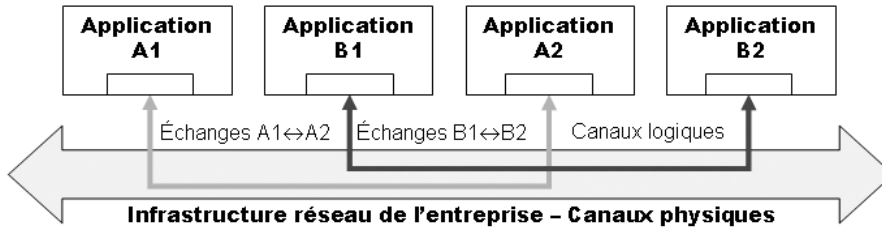


Figure 2.5 - Mutualisation de l'infrastructure réseau

Dans ce schéma les couples A1, A2 et B1, B2 échangent des messages, et/ou des fichiers, et/ou des requêtes, au moyen de canaux logiques, via l'infrastructure réseau de l'entreprise. Le réseau réalise de ce fait un couplage physique entre les canaux logiques de ces différentes applications, ce qui fait que les échanges A1, A2 peuvent perturber les échanges B1, B2 et réciproquement. La latence du réseau devient aléatoire, dans la mesure où aucune des applications ne peut contrôler le niveau de charge qu'elle induit chez les autres applications. C'est un aspect fondamental des systèmes centrés réseau, i.e. NCS *Network Centric Systems*.

2.2.1 Organisation hiérarchique – Niveaux d'abstraction

Il est indispensable de préciser rigoureusement la terminologie et le sens véhiculé par les différents niveaux de la décomposition hiérarchique et de clarifier la notion de référentiel. Une application répartie est formée de plusieurs applications (composants applicatifs) non réparties qui interagissent, et dont la cohérence est obtenue par l'application de règles communes partagées par toutes les applications non réparties. La sémantique ainsi définie correspond exactement à ce qui est communément appelé système (au sens systémique du terme).

Il est nécessaire de préciser le sens de deux termes :

a) **Application**, qui est réservée à ce qui s'exécute sur une machine et qui peut être en interaction avec d'autres applications, sur la même machine ou sur des machines différentes. De façon plus précise, nous utiliserons le terme **composant applicatif**.

b) **Système**, qui dénote un ensemble d'applications apparentées, partageant une sémantique et des règles communes garantissant la cohérence des actions vis-à-vis des utilisateurs du système (cohérence métier).

Un composant applicatif est lui-même organisé en « grains » plus fins, qui peuvent être eux-mêmes agrégés en grains plus gros : c'est la configuration de l'application.

Le niveau système peut lui-même être agrégé pour former un système de rang supérieur, et ainsi de suite, de façon récursive.

Dans une norme comme ISO 9126 qui définit une nomenclature de caractéristiques qualité du produit logiciel (i.e. une application), c'est sur le niveau application, ou le niveau système, que ces caractéristiques s'appliquent. Selon la position de l'entité dans la hiérarchie (cf. figure 2.1), on aura, du côté application des contraintes à satisfaire, et du côté intégration système des propriétés émergentes, non spécifiées, et éventuellement non conformes à ce qui est souhaité par l'utilisateur (ce que nous appelons contraintes PESTEL et CQFD/FURPSE système), selon le schéma 2.6 :

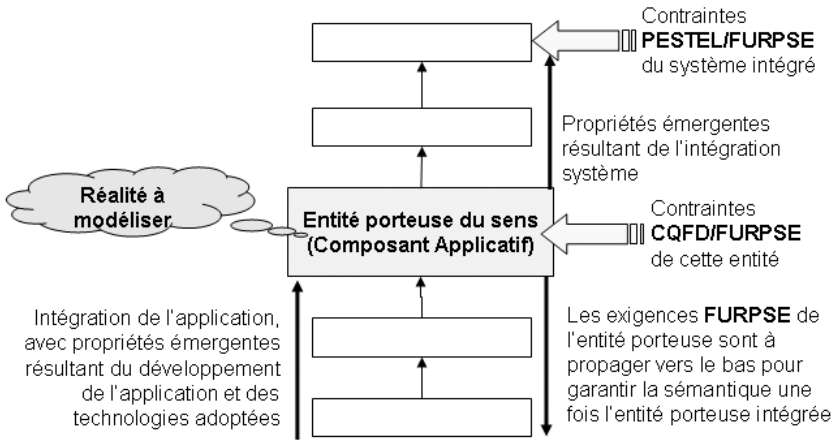


Figure 2.6 - Propagation des contraintes et propriétés émergentes

Ceci étant, les règles applicables au niveau des composants applicatifs résultent de ce qui a été spécifié explicitement au niveau du système intégré. Si la spécification système est incomplète, il y aura émergence de propriétés indésirables lors du processus d'intégration qui n'est pas qu'un simple assemblage de composants.

Pour bien comprendre la logique de construction, il est essentiel de procéder par étape.

1re étape : Fabrication d'une application générique adaptable sur différentes machines.

Dans le processus de développement de l'application générique, il faut soigneusement distinguer :

a) La partie programmatique de l'application qui se fait avec le ou les langages autorisés pour cette programmation, conformément aux règles de programmation édictées par l'architecte.

b) Le référentiel de construction qui contient des éléments et des règles prédéfinis que les équipes de développement devront utiliser pour respecter la cohérence globale de l'application. Ce référentiel est lui-même structuré en deux parties : ce qui est spécifique à l'application et ce qui est spécifique aux machines sur lesquelles

sera déployée l'application. Nous avons donc le référentiel application et le référentiel système, conformément au schéma 2.7 :

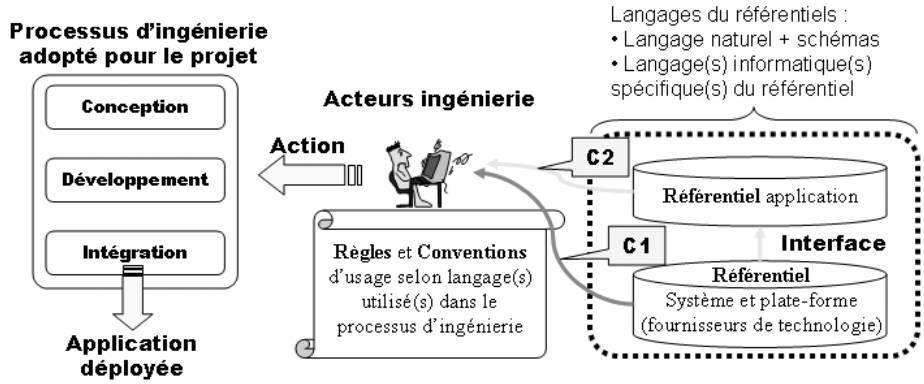


Figure 2.7 - Le bon usage des référentiels

Concernant le référentiel système, deux cas sont à considérer :

1. Les équipes de développement peuvent utiliser librement le référentiel système : chemin C1.
2. Les équipes de développement ne peuvent utiliser le référentiel système qu'à travers le référentiel application selon le chemin C2.

Dans le 1er cas, le travail architectural est simplifié pour autant que les équipes de développement connaissent bien le référentiel système et ses limitations, ce que dans le modèle d'estimation COCOMO[8] on appelle facteurs de coût ACAP et PCAP, *Architect* et *Programmer capability*. La contrepartie est une dépendance de l'application vis-à-vis du référentiel système ; s'il est modifié, c'est l'application dans son ensemble qui est à reprendre pour s'adapter.

Dans le 2ème cas, un niveau d'abstraction (i.e. des interfaces) doit être élaboré, de façon à ce que le développement ne soit dépendant que du référentiel application. Cette stratégie est préférable pour les applications dont le cycle de vie est long, car elle a un impact important sur le TCO)

2ème étape : Construction d'une application exécutable sur des machines M1, M2, …, Mn non nécessairement identiques (en fait, c'est le cas général). Les machines (ou serveurs) sont des équipements particuliers (au sens de la figure 2.1). Le schéma de principe du déploiement est le suivant (figure 2.8) :

8. Pour tout ce qui concerne la productivité des programmeurs et les modèles d'estimation, y compris COCOMO, nous renvoyons sur nos ouvrages publiés chez Hermès, *Productivité des programmeurs* et *Coûts et durée des projets informatiques*.

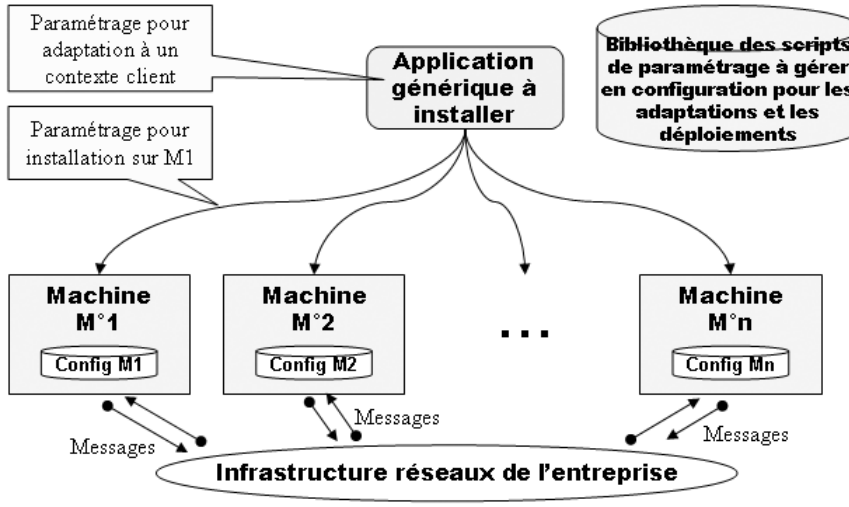


Figure 2.8 - Déploiement d'une application et échanges sur le réseau

Le paramétrage de l'application générique AG se fait avec le ou les langages de scripts (JCL, etc.) spécifiques à une machine particulière.

A l'issue de ces deux étapes, quelle que soit la stratégie adoptée, le résultat du développement est une construction que l'on peut représenter : a) de façon hiérarchique, conformément aux règles du ou des langages de programmation utilisés, pour ce qui concerne les aspects statiques et b) de façon dynamique, à l'aide de diagrammes d'activités, d'états – transitions, etc., tels qu'on les trouve dans des langages de modélisation comme UML. Il est essentiel de comprendre que ce n'est pas UML qui décide de la structure de la hiérarchie et des enchaînements, mais l'architecte.

Chacune des entités ainsi répertoriées est ensuite programmée, ou acquise sous forme de COTS, conformément aux règles fixées par le référentiel.

On peut illustrer cette problématique à l'aide des schémas suivants :

a) Une représentation hiérarchique : dans le langage de la gestion de projet, c'est l'arbre produit de l'application (figure 2.9), à l'aide des diagrammes statiques (diagrammes de classes).

Dans cet exemple, l'application a une profondeur logique de 3, correspondant aux niveaux N1, N2, N3, le niveau 0 étant constitué des livraisons des programmeurs ou des fournisseurs de COTS qui entrent dans le processus d'intégration ; ce sont les intégrats considérés comme atomiques qui se présentent au processus d'intégration avec le niveau de service et la qualité requise par ce processus. Chacun des blocs identifiés dans l'arbre produit correspond à une tâche de développement à réaliser (organigramme des tâches – WBS, selon la méthodologie adoptée pour le processus d'ingénierie). On remarquera que la profondeur logique dépend du « pas » d'agrégation des intégrats de rang 0. Si l'on agrège par paquet de n intégrats les N

intégrats de rang 0, ce nombre croît comme un logarithme de base n, soit $\log_a$(*Nombre d'intégrats de rang* 0) que l'on peut approcher avec des fonctions de forme N^{α} avec $\alpha < 1$.

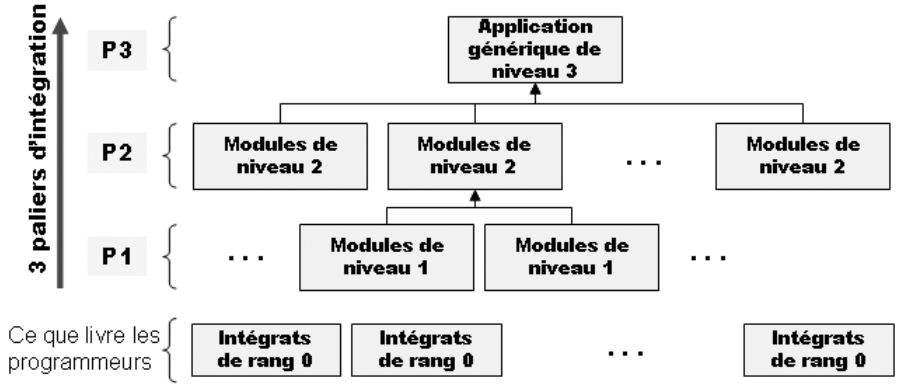


Figure 2.9 - Arbre produit d'une application et intégration

b) Une représentation dynamique à l'aide des diagrammes dynamiques (activités et/ou états-transitions), soit (figure 2.10) :

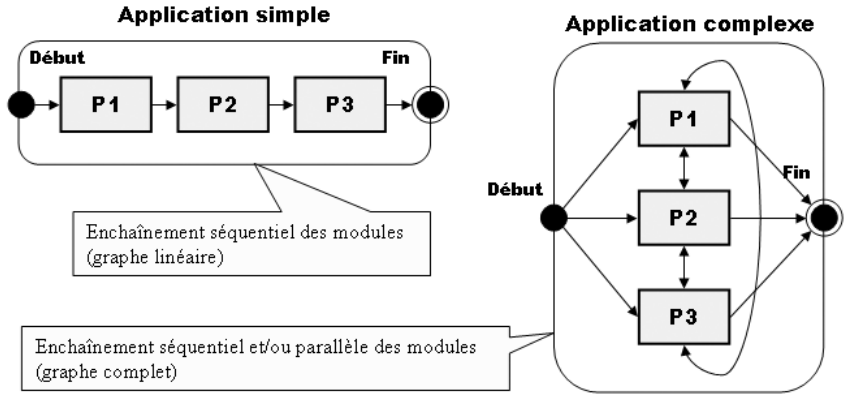


Figure 2.10 - Diagrammes dynamiques d'une application

Le diagramme dynamique matérialise le graphe des appels ; sa complexité est fonction du niveau de granularité considéré. Dans l'application complexe, tous les modules sont susceptibles de s'appeler ; le graphe est complet et comporte $\frac{n(n-1)}{2}$ connexions. Ce graphe fondamental peut être représenté sous-forme matricielle facile à manipuler à l'aide de programmes (cf. tableau 2.1).

En programmation objet, on aura des complexités très différentes selon que l'on considère le niveau paquetage, le niveau objet/classe, le niveau méthode. Le niveau de complexité corrèle l'effort à fournir par l'équipe d'intégration.

C'est ce facteur que l'on retrouve en exposant dans la formule d'effort du modèle d'estimation COCOMO : $Eff = k(KLS)^{1+\alpha}$.

On peut donc dire, à ce stade, que :

a) La décomposition hiérarchique (arbre produit) organise le processus de développement mis en œuvre par le responsable projet de l'équipe développement.

b) Les diagrammes dynamiques organisent la nature des travaux à effectuer dans le processus d'intégration (choix des chemins à privilégier, programmes « bouchons » et « drivers » à réaliser, cas d'emplois à privilégier, tests à réaliser, etc.).

À ce stade, nous avons complètement défini la nature du grain application qui est tout à la fois une unité de réalisation d'un point de vue projet (équipe nominale projet) et une unité d'intégration qui peut être exécutée dans les différents contextes où elle sera déployée.

Reste maintenant à examiner la nature du nouveau système qui intègre une ou plusieurs applications, avec les équipements nécessaires pour la mise en exécution effective de l'application.

Commençons par l'exemple d'une application client serveur répartie, ce qui, compte tenu de la terminologie introduite, se traduira par un système formé de trois applications (i.e. trois composants applicatifs), a minima :

1. L'application IHM / serveur de présentation dont l'équipement peut être un PC, une imprimante locale, un scanner, un lecteur de badge d'authentification, etc.
2. L'application métier proprement dite, ou serveur métier, qui contient les transactions nécessaires au traitement de l'information, dont l'équipement peut être constitué d'un ou plusieurs serveurs, de plus ou moins grande puissance, avec éventuellement des capacités de récupération de l'un sur l'autre.
3. L'application de gestion des données, ou serveur de données, qui gère les fichiers et bases de données utilisées par le système, lesquels peuvent être partagés entre différents systèmes, dont l'équipement est un serveur de données qui peut être constitué de plusieurs machines qui seront vues du serveur métier comme une ressource centralisée pour garantir l'intégrité de l'information codée dans les données.

Une première représentation d'un tel système est donnée par le schéma 2.11, sur lequel sont indiqués des nombres d'occurrence qui entreront dans la configuration déployée.

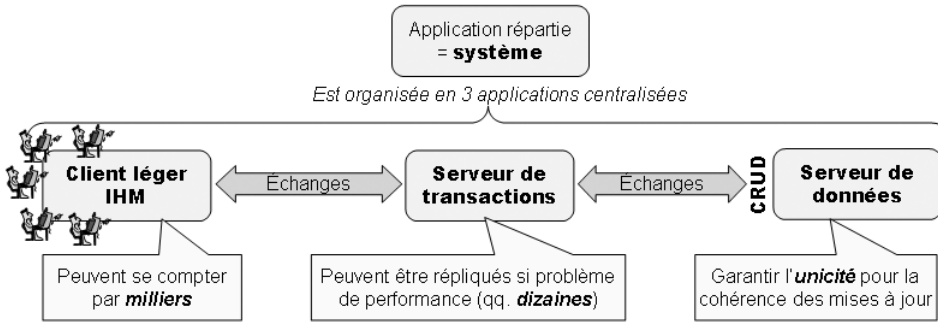


Figure 2.11 - Une architecture client-serveur élémentaire

Dans un second schéma, nous faisons apparaître un niveau de détail dans chacun des serveurs qui pourra donner lieu à des intégrations particulières pour chacun d'eux, en parallèle avec des intégrations générales concernant les trois serveurs, quand bien même l'intégration de chacun d'eux ne soit que partielle.

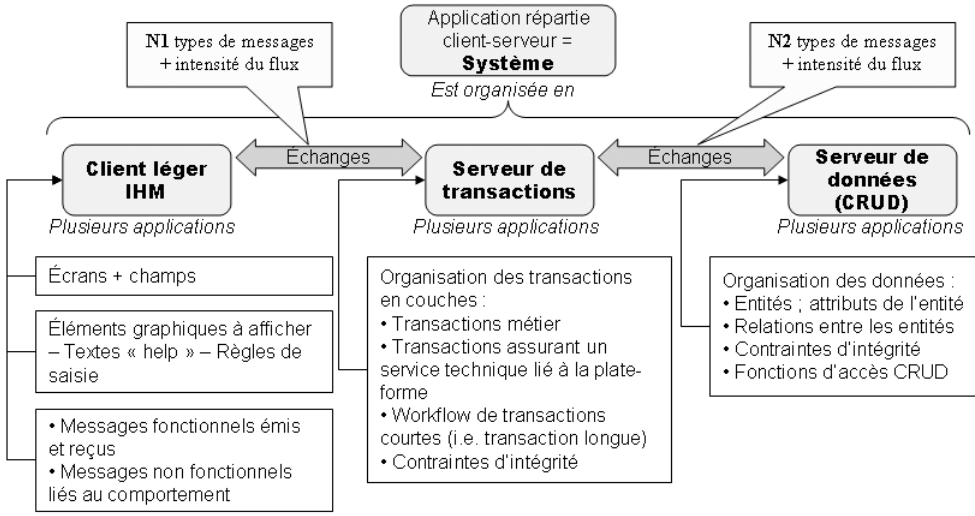


Figure 2.12 - Détails de l'arbre produit d'une architecture client-serveur

Dans une telle architecture, les applications communiquent entre elles via des messages. La cohérence logique est assurée par le respect des règles du référentiel application + système, ainsi que des règles et conventions de programmation. Les équipes de développement doivent parfaitement connaître ces différentes catégories de règles.

La figure 2.12 donne quelques détails sur les entités logicielles constitutives de ces trois applications. La syntaxe et la sémantique des messages, ainsi que les caractéristiques non fonctionnelles qui leur sont associées doivent être définies de façon

rigoureuse, et ne laisser la place à aucune interprétation qui pourrait générer des incohérences logiques (ambiguïté, non déterminisme, contradiction, etc.).

Pour les systèmes de très grande taille et/ou de très grande complexité il est indispensable de décomposer le niveau système global, appelé système de systèmes SDS, en un ensemble de systèmes, eux-mêmes hiérarchisés, conformément aux bonnes pratiques recommandées par les normes d'ingénierie système. Cette hiérarchie définit la profondeur de l'intégration système. On a alors des décompositions comme suit :

a) Cas d'une entreprise ou d'un organisme de taille moyenne.

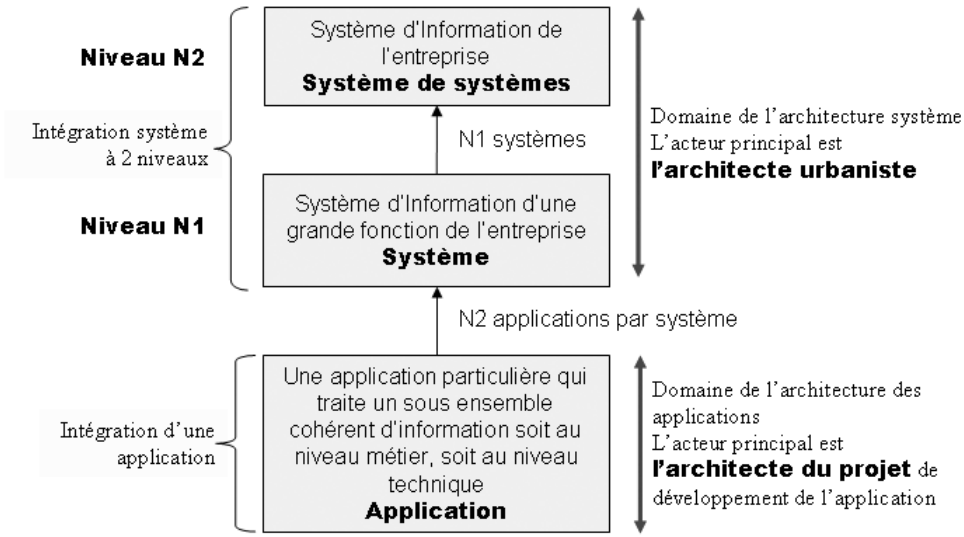


Figure 2.13 - Référentiel du SI de l'entreprise – Cas N°1

À chacun des niveaux est attaché un référentiel particulier qui fixe les règles à respecter pour garantir la cohérence du niveau considéré.

b) Cas d'une entreprise ou d'un organisme de très grande taille.

Un ou plusieurs niveaux de sous-systèmes s'intercalent entre le niveau système et le niveau application, soit, dans le schéma 2.14, une profondeur d'intégration égale à 3.

Idéalement, un niveau de référentiel est constitué de connaissance implicite des acteurs de l'entreprise, auquel s'ajoutent des connaissances explicites qui peuvent être soit de type informel textuel, soit de type méta-modèle avec, a minima, une syntaxe explicite, ou une sémantique formalisée.

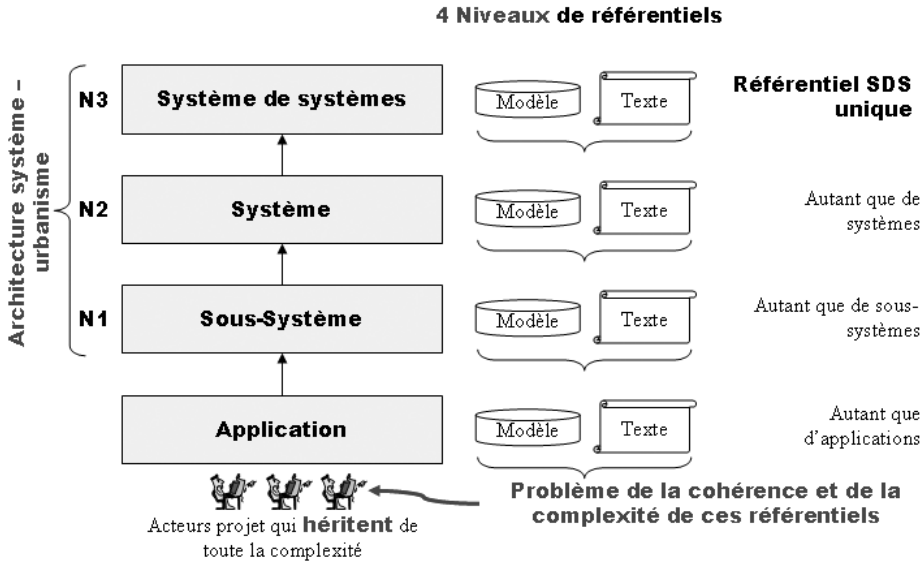


Figure 2.14 - Référentiel du SI de l'entreprise – Cas N°2

On remarque que les arbres d'héritage en ingénierie de projet et en intégration de système sont inverses :

- En intégration système, le processus d'intégration va du bas vers le haut (héritage *bottom up*), ce qui peut faire émerger des propriétés indésirables qu'il faudra vérifier et valider ;
- En développement, l'équipe de développement doit utiliser le référentiel de l'application, auquel va venir se rajouter le référentiel du sous-système dans lequel l'application s'intègre, et ceci jusqu'au niveau SDS (héritage *top down*), ce qui peut faire apparaître des contradictions ou des ambiguïtés entre les niveaux qu'il faudra lever.

2.2.2 Relations, interactions et couplages des entités architecturales

Du point de vue de la programmation de l'application l'équipe de développement doit connaître les niveaux d'appartenance des autres applications avec lesquelles elle interagit, de façon à identifier les règles qui vont régir les interactions.

Du point de vue du référentiel, il faut considérer :

a- La matrice de couplage des applications A1, A2, …, An constitutives du système considéré (matrice N2) ;

b- Les relations d'appartenance (arbres hiérarchiques) qui définissent un ordre partiel entre les éléments de la hiérarchie ;

c- Le chemin de données logique que parcourt un message qui s'échange entre deux applications.

En théorie, tout ceci relève de problèmes classiques rencontrés en théorie des graphes (connexité, chemins, etc.), et plus généralement en recherche opérationnelle[9].

Représentation des couplages par une matrice N2

On considère *n* applications A1, A2, …, An et l'on place le nom des applications sur la diagonale de la matrice comme suit :

Tableau 2.1 - Matrice N2 des applications intégrées

					… …				
	A1	✔							
		A2							
	✔		A3						
				A4					
… …									
									An

Pour chaque ligne, les cases marquées ✔ indique un couplage ordonné de l'application ligne vers l'application colonne. Par exemple A1 → A2, A3 → A1, etc. Si toutes les cases sont marquées, le graphe de couplage est complet. Une ligne donne tous les Ai appelés ; une colonne donne tous les Aj appelants.

Cette matrice fondamentale en ingénierie système[10] permet de savoir qui appelle qui, et dans quel sens ; la case correspondante peut servir à décrire la nature de l'interface : messages échangés, modalités d'appels, etc.

Relations d'appartenance des applications

La représentation la plus intuitive est l'arborescence (i.e. un diagramme de classes, au sens de la conception objet), avec des conventions : {SDS} pour la racine, {S1, S2, S3} pour le niveau de profondeur 1, {SS11, SS12, …} pour le sous-niveau de profondeur 2 rattaché à S1, etc. Soit par exemple (figure 2.15) :

9. Cf. R.Faure, *Précis de recherche opérationnelle*, Dunod, 1993.
10. Cf. NASA *Systems engineering handbook*.

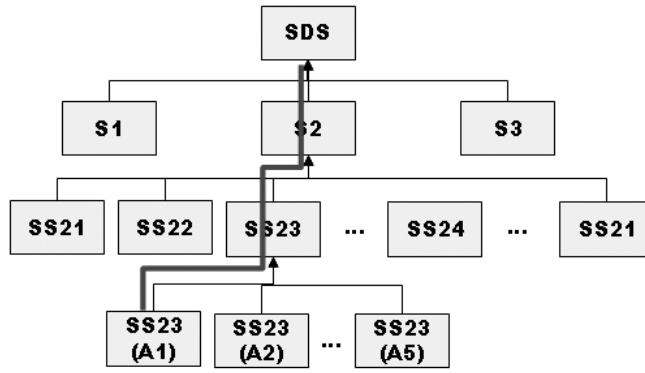


Figure 2.15 - Arbre de décomposition hiérarchique du SDS

Cette arborescence va permettre d'identifier les frontières de systèmes et de sous-systèmes à franchir pour déterminer les référentiels applicables à l'échange. Deux applications Ax et Ay ont en commun un point de rencontre, soit au niveau SS, soit au niveau S, soit au niveau SDS.

NB : on peut également représenter cette arborescence à l'aide d'une matrice N2 en disposant sur la diagonale les éléments de l'arborescence, soit : S, {liste des systèmes}, {liste des sous-systèmes}, {liste des applications}. Seule la 1/2 matrice supérieure est utile. On pourrait également utiliser une notation grammaticale classique, comme suit :
SDS (S1 (SS11(A1, A2, …, Ax1), SS12(A…), …, SS1x(…)), … S3 (…))

Chemin de données – Nomenclature des messages échangés – Référentiel des messages

Pour chacune des applications il faut connaître les types de messages émis et reçus, ainsi que les applications destinatrices et réceptrices. Pour une application quelconque Ax parmi 10, on dispose d'une table de références croisées en faisant ressortir des messages émis et reçus (voir tableau 2.2).

Si le message mx est émis par Ax vers l'application réceptrice Ai, la case correspondante dans la colonne Ai comporte un e ; si le message est reçu de l'application émettrice Aj, la case correspondante comporte un r ; e/r si le message est à la fois émis et reçu par A.

Au niveau système, on peut construire une matrice d'échange global, en intégrant les messages par niveau, à l'aide des relations d'appartenance, soit :

- Messages du niveau SDS commun à tous : c'est le langage commun de la fédération de systèmes constitutifs du SDS.
- Messages spécifiques des systèmes S1, S2 et S3.
- Messages spécifiques des sous-systèmes SS11, …, SS21, …, SS31, …
- Messages privés aux applications.

On pourrait également utiliser la matrice de couplage N2, en utilisant les cases comme un index vers une liste de messages émis ou reçus.

La matrice globale des échanges est la fusion des matrices spécifiques à chacune des applications.

Tableau 2.2 - Echanges de messages d'une application

	Ax	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10
m1	E	e									
m2	-										
m3	R			r							
m4	-										
	-										
m17	E/R		e		r		e/r				
	E										
m75											

2.2.3 Synthèse du paragraphe

En synthèse de ce paragraphe, différents aspects de la sémantique des entités constitutives d'un système informatisé sont identifiés :

Aspect nomenclature du système

Les entités sont répertoriées et organisées en hiérarchie. Le niveau le plus fin de la hiérarchie est une entité de type « application » qui transforme l'état d'une mémoire (fichiers et/ou bases de données), elle-même organisée en hiérarchie. Toutes les entités sont nommées ; le nom est soit un nom de classe (classe d'entités), soit un nom d'individu dans la classe (par exemple, le collaborateur Dupont, de la classe employé). Ceci identifie deux domaines sémantiques :

1. le domaine données, ou plus exactement des classes de données regroupant des individus partageant des attributs caractéristiques de la classe (cf. le

problème des classifications des données). La donnée est caractérisée par son type.

2. Le domaine programmes, qui est une nomenclature d'opérations applicables (i.e. procédures) sur les données.

L'ensemble {données, opérations} peut être organisé en types abstraits de données.

Les entités ainsi répertoriées peuvent appartenir à différents niveaux d'abstraction du système global de l'entreprise ; soit : Système de Système, Système, Sous-système, Application.

Aspect transformationnel ou opérations effectuées par le système

C'est le descriptif de l'opération qui transforme un ensemble de types en entrée de la transformation, en un ensemble de types, résultat final de la transformation, en sortie. Par commodité, on peut dire que la transformation reçoit des messages en entrée et restitue des messages en sortie. Au sens propre du terme, l'opération, quel que soit le niveau d'abstraction, caractérise ce que fait le système. Les transformations réalisées correspondent à des changements d'état de la réalité : un calcul a été effectué, un compte bancaire a été débité, un abonné du téléphone a été connecté, un avion a décollé, etc. moyennant une fourniture de ressources pour effectuer la transformation.

Aspect service ou commandes à disposition de l'opérateur

On a vu en §2.1. que certaines opérations (intégrat de rang 0) jouent le rôle d'instructions d'une « machine étendue ». Pour une application ou d'un module appelant, une telle opération apparaît comme un service dont l'appelant peut commander l'exécution. La sémantique de l'entité appelante est caractérisée par la nature des services appelés et par les modalités de l'appel (séquentiel synchrone, ou parallèle asynchrone), et par les messages de suivi de la bonne exécution du service qu'elle est censée recevoir.

Aspect réactif ou enchaînement des opérations et services effectués

C'est le descriptif du contrôle des opérations effectuées par le système, en fonction du résultat des opérations et des événements pouvant affecter le déroulement des opérations. Ce descriptif est un automate de contrôle caractéristique du workflow mis en œuvre conformément aux actions commandées par les opérateurs du système. Cet automate peut être localisé dans un intégrat précis du système (cas des chaînes batch) ou distribué dans différents intégrats (cas des systèmes transactionnels et des architectures orientées services).

Dans un monde parfait, ceci serait suffisant pour caractériser la sémantique. Dans le monde réel où il y a des aléas, des erreurs, des pannes, des ressources épuisées ou indisponibles, il faudra rajouter des règles d'intégrité et des mécanismes de surveillance faisant office de démon de Maxwell. Tout ceci sera repris et complété dans la partie 3.

2.3 INDÉPENDANCES DES DONNÉES ET DES PROGRAMMES

L'indépendance des données et des programmes est un vieux problème, mais il est toujours actuel. C'est l'une des préoccupations centrales de l'architecte car les données sont le constituant le plus stable de l'équation de Wirth. Les données, c'est-à-dire les états que les données représentent, sont le socle de toutes les architectures.

Le problème apparaît avec les périphériques bandes et disques magnétiques qui permettent de stocker les données, indépendamment des programmes qui les manipulent, et ceci dès les années 60. Ces périphériques permettent d'organiser un espace logique permettant de stocker les données de façon durable. On pourra :

a- Récupérer les données d'une exécution à l'autre,

b- Partager les données entre plusieurs applications, lorsque la multiprogrammation deviendra possible, vers la fin des années 60 et début des années 70.

La nature du périphérique permet d'inventer la notion de fichier séquentiel, avec les bandes magnétiques, puis, avec les disques, la notion de base de données, de fichiers indexés, etc. en accès direct sur les données. Cet espace logique a ses propres règles d'adressage, indépendamment de celles qui prévalent dans la machine où les programmes sont stockés et exécutés.

Le schéma logique d'une telle organisation des traitements est le suivant (figure 2.16).

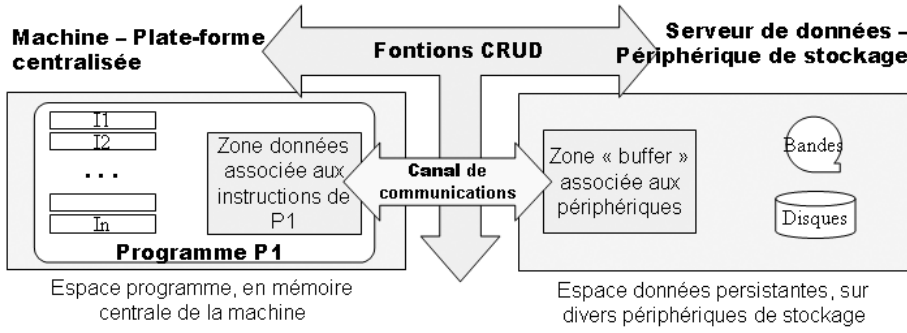


Figure 2.16 - Indépendance des données et des programmes

Le programme P1 gère les données stockées avec un jeu de fonctions CRUD qui réalisent la transduction entre l'espace données et l'espace stockage. Le format des données stockées est différent du format des données programmes qui est celui des données autorisées par la machine (types de données connus de la machine : entiers courts/longs, caractères, nombres décimaux « packés » / « non packés », flottants, booléens, etc.).

Cette transduction rend l'espace programme indépendant de l'espace données. Les règles de transformations se font au moyen des fonctions CRUD que l'on peut configurer conformément à la structure logique et physique du périphérique. Le programme P1 peut évoluer indépendamment du stockage.

Par exemple, un stockage séquentiel peut s'implémenter de différentes façons en fonction du nombre d'éléments stockés, de leur taille variable ou fixe, de l'ordre que l'on choisit de privilégier.

De fait, ce sont les caractéristiques non fonctionnelles qui permettront de choisir le meilleur compromis. Toutefois, au début de l'histoire de l'informatique, les ressources de stockage sont tellement coûteuses que l'on fabrique, sans vraiment s'en rendre compte, des problèmes qui se révèleront des désastres économiques avec le passage de l'an 2000 ou le changement d'étalon monétaire, lors du passage à l'Euro.

Aujourd'hui, avec les techniques de compression de données, et les très grandes mémoires centrales, le problème peut être envisagé d'un point de vue strictement logique, du moins si l'architecte le considère sous cet angle.

Dans un schéma comme ci-dessus, trois formats logiques existent (figure 2.17) :

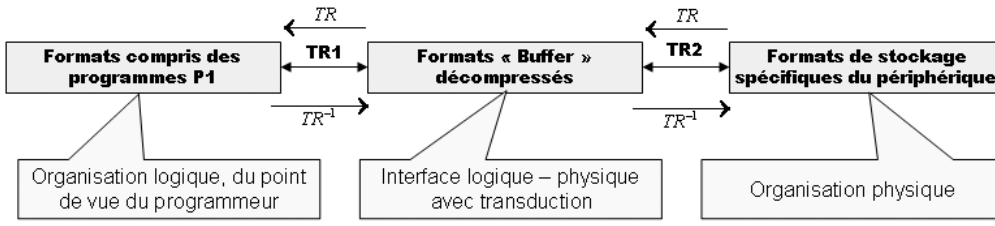


Figure 2.17 - Traduction des formats de données

Tous les formats peuvent être traduits les uns dans les autres, dans les deux sens. Les opérateurs *TR* et TR^{-1} sont des opérateurs inverses l'un de l'autre. Si la contrainte de performance est forte, ou si l'architecte se laisse aller à la facilité, il est tentant de tout confondre, ce qui fait que toute modification du stockage entraînera une modification du programme, et réciproquement. L'organisation des données programmes est alors totalement rigidifiée.

Avec l'arrivée des bases de données, vers la fin des années 60, le problème se complexifie, mais une solution logique d'une grande élégance lui est apportée : la notion de schémas à trois niveaux du rapport ANSI-SPARC que l'on retrouvera ultérieurement dans tous les SGBD. Toutefois, avec la puissance des machines des années 70, la solution restait coûteuse à implémenter en totalité, ce qui n'est plus le cas aujourd'hui.

Le schéma logique de la solution ANSI-SPARC est le suivant (figure 2.18) :

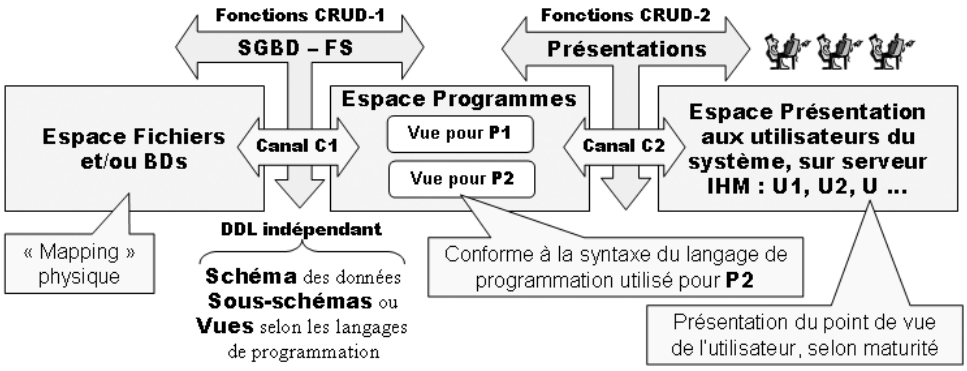


Figure 2.18 - Le modèle d'indépendance des données ANSI-SPARC

Le modèle ANSI-SPARC fait apparaître deux nouvelles interfaces : la notion de vue logique et la notion de présentation (également appelée vue externe).

Une base de données est une structure complexe qui n'a pas besoin d'être vue dans tous ses détails par les différents programmes ; ceux-ci ne verront que le sous-ensemble qui leur est nécessaire. C'est la notion de sous-schéma ou vue, selon les modèles de SGBD envisagés.

Les programmes P1, P2, …, ont besoin d'éditer leurs résultats et les données manipulées. Les formats correspondants doivent être compréhensibles par l'acteur (humain ou système) pour qui ces éditions sont effectuées.

C'est la notion de présentation, encore appelé schéma externe, car c'est ce que l'acteur verra effectivement. D'un point de vue ergonomique, la présentation a toute latitude pour visualiser les données dans le format le plus « simple » du point de vue de l'usager[11], sans affecter pour autant les représentations internes qui sont tributaires de la technologie informatique disponible.

Tous ces formats peuvent être adaptés à leur usage, indépendamment et de façon ad hoc. La souplesse que donne cette architecture permet une évolution de l'application au moindre coût, si toutefois l'architecture a été correctement déployée.

D'un point de vue logique, cette architecture fait apparaître trois niveaux d'abstraction que l'on peut schématiser comme suit (figure 2.19) :

11. A ce sujet, voir en particulier les règles de présentation préconisées par le Mil-Std 2525B, *Common warfighting symbology*, du DOD et de l'OTAN, pour les systèmes de gestion de crise.

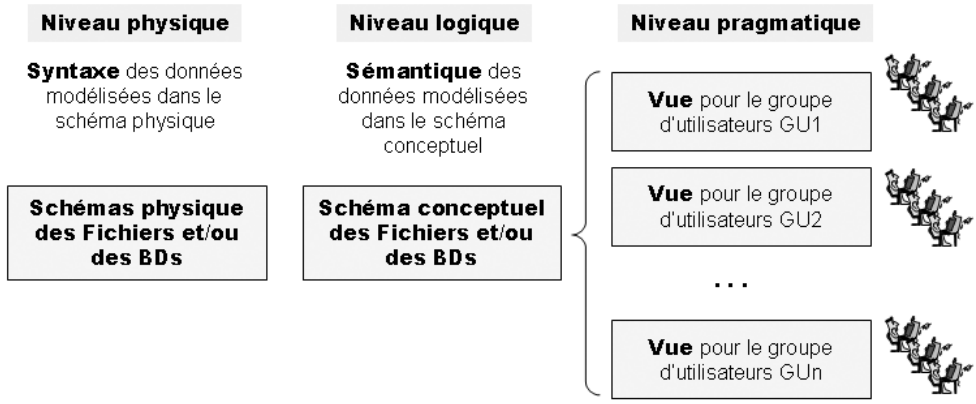


Figure 2.19 - Les trois niveaux d'abstraction de l'architecture ANSI-SPARC

Les considérations sur l'architecture à trois niveaux occupent une place modeste dans les ouvrages sur les SGBD, généralement le chapitre 1, bien qu'elles imprègnent la totalité de ces ouvrages. Cette place est plus importante dans les ouvrages consacrés exclusivement à la modélisation des données comme par exemple :

- D. Tsichritzis, F. Lochovsky, *Data Models*, Prentice Hall, 1982 (très intéressant mais épuisé, disponible en bibliothèque).
- T. Teorey, *Database modeling and design*, Morgan Kaufmann, 1999 (3ème édition, l'un des meilleurs ouvrages sur le thème de la modélisation des données).

NB : Le rapport ANSI-SPARC et ses annexes expliquant le pourquoi du modèle totalisent environ 200 pages.

Pour un architecte de SI, il est presque inutile de connaître le détail des mécanismes logiques du SGBD ou les subtilités des formes normales et l'algèbre relationnelle, s'il n'a pas parfaitement assimilé ces notions fondamentales, car les erreurs à ce niveau sont véritablement catastrophiques du point de vue de l'économie globale du SI (CQFD des projets + TCO). Ces erreurs sont malheureusement fréquentes et les plates-formes distribuées n'ont pas simplifié le problème. Le point crucial est la séparation et le confinement strict de deux aspects de l'information gérée par le SGBD :

- Les données qui représentent le domaine sémantique pris en considération dans la modélisation, par exemple les différentes catégories d'employés d'une entreprise.
- Les données de structure (i.e. les méta-données) qui définissent a) l'organisation du domaine sémantique considéré (visible du modélisateur métier) et b)

la façon dont tout cela va être représenté dans le monde physique (invisible du modélisateur métier, mais visible de l'architecte du SGBD).

Cette deuxième catégorie de données est un méta-langage par rapport à la première catégorie. Ce méta-langage sert tout à la fois a) à organiser les données à traiter : c'est le DDL du modèle de données, et b) à effectuer les différentes traductions impliquées par la structure des formats et interfaces adoptés par le SGBD.

Modifier les méta-données, ou modifier les données n'a pas le même impact économique. Il faut considérer :

a- les fonctions CRUD sur les données qui constituent le langage de manipulation des données que le programmeur de requêtes sur la BD va utiliser de façon courante.

b- les fonctions CRUD sur les méta-données qui sont réservées à l'administrateur de la BD et aux concepteurs-programmeurs du SGBD. Certains SGBD permettent en effet de modifier le méta-modèle dans des limites strictement contrôlées par le SGBD lui-même.

Dans des modèles comme le modèle des données en réseaux (modèle CODASYL), les deux langages sont différents. Dans le modèle relationnel, les deux langages sont identiques, ce qui peut occasionner une grande confusion dans la tête de ceux qui n'ont pas saisi la nuance, d'où une catastrophe économique prévisible quand on confond les niveaux. Si l'on reprend les niveaux introduits dans la section 2.2, on aura une structure de référentiel (figure 2.20).

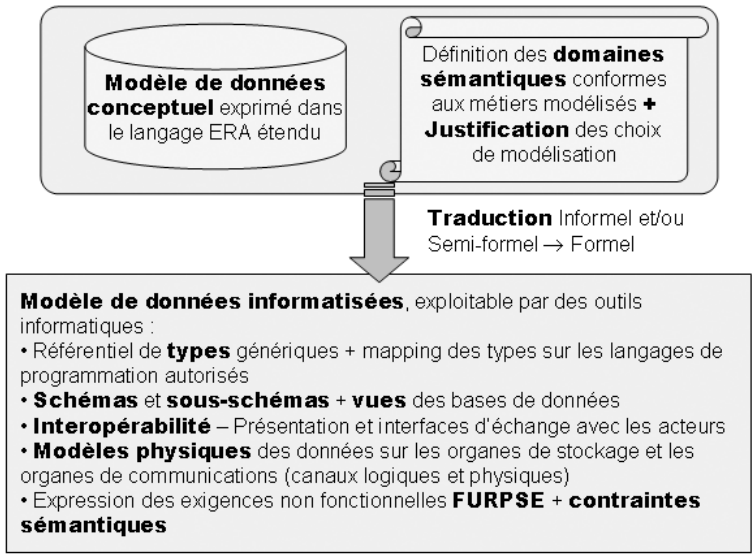


Figure 2.20 - Niveaux informel et formel en modélisation de données

D'autres aspects seront analysés dans la 3e partie de l'ouvrage, en particulier les aspects transactionnels.

Synthèse du paragraphe

Les données correspondant à l'information modélisée dans le système informatique peuvent être vues sous trois représentations différentes :

1. Une représentation conceptuelle centrée sur la sémantique de la donnée et des relations qui existent entre les données, et ceci quelles que soient les modalités de représentation de ces relations (en extension dans le modèle, ou en intention dans les programmes CRUD). Ce niveau est à l'articulation des mondes 1 et 2.
2. Une représentation spécifique aux langages de programmations utilisés avec filtrage de ce qui est vu par le programme CRUD correspondant.
3. Une représentation physique spécifique aux périphériques de stockage de l'information. En cas de défaillance, c'est la représentation physique qui est vue.

Ceci définit pour les données trois représentations distinctes selon les points de vue, mais interdépendantes et convertibles les unes dans les autres au moyen de traducteurs fondés sur la syntaxe et la sémantique ainsi représentées.

2.4 TENTATIVE DE DÉFINITION DE L'ARCHITECTURE

Le débat sur l'architecture logicielle est aussi ancien que celui concernant l'ingénierie du logiciel, dont il constitue un thème récurrent. Il suffit de relire les deux rapports du *NATO Science Committee*, *Software Engineering*, (en 1968 et 1969) pour s'en convaincre. On y trouve, entre autre, une brève communication de E. Dijkstra, *Complexity controlled by hierarchical ordering of function and variability*, très intéressante, et tout à fait révélatrice de problèmes qui sont toujours actuels. Des textes comme *Cooperating sequential processes*, (1965) dans lequel le concept de sémaphore est rigoureusement défini, ou encore *THE multi programming system* dans lequel est défini le concept d'architecture en couches, gardent toute leur fraîcheur ; voir la recension : *The origin of concurrent programming*, de P.B.Hansen, Springer 2002, qui rassemblent de nombreux textes fondamentaux.

Le livre : *The mythical man-month* de F. Brooks, 1re édition en 1975, toujours ré-édité depuis, peut à bon droit être considéré comme un ouvrage d'architecture de système logiciel (F. Brooks fut l'architecte du système d'exploitation MVS de la série 360 d'IBM). Tout architecte et/ou chef de projet soucieux de son art devrait l'avoir à portée de main ! Idem pour le livre d'H. Simon, *The sciences of the artificial*, 1re édition 1981, toujours ré-édité, mais d'une toute autre nature que celui de Brooks. La 3ème édition (1996) contient un chapitre 8, *The architecture of complexity : hierarchic systems*, qui fait écho à ce qu'écrivait Dijkstra 30 ans plus tôt, d'une très grande pro-

fondeur car il ébauche une analyse quantitative entre les trois aspects qui nous préoccupent ici (données, transformations, événements).

Depuis les années 90, le thème ***Software Architecture*** est à la mode tant au niveau des publications, nombreuses mais de qualité très inégale, que des manifestations.

Dans le livre de M. Shaw et D. Garlan, *Software Architecture*, Prentice Hall 1996, on trouve une définition « *The architecture of a software system defines that system in terms of computational components and interactions among those components. […] Interactions among components […] can be simple […] or complex and semantically rich, as client-server protocols, data-base-accessing protocols, asynchronous events multicast, and piped streams. […] More generally, architectural models clarify structural and semantic differences among components and interactions. Etc.* »

Définition intéressante, certes, mais oh combien réductrice ! Où est la liberté de choix de l'architecte, quelle est sa stratégie, que doit-il optimiser ? La définition ne le dit pas, ni d'ailleurs le livre. Notons cependant que le terme *computational components* renvoie à la notion de calcul (i.e. *business computing*), et donc de machine ; ce que nous avons appelé composante transformationnelle.

Le cadre d'architecture développé par le ministère de la défense américain[12] qui est un peu la référence en matière d'architecture de systèmes complexes, n'a évidemment pas échappé à cette vague. Pour la terminologie, il s'appuie sur le corpus de normes de l'IEEE *Software Engineering Standards Collection*, ce qui est bien, mais pas vraiment plus utile en pratique. On y trouve, des définitions comme : « les architectures fournissent des mécanismes permettant de comprendre et de manager la complexité » ou encore : « l'architecture décrit la structure des composants, leurs relations, les principes et les directives pilotant leur conception ».

Ce que toutes ces définitions révèlent est qu'il est très difficile, voire impossible et/ou vain, de parler de l'architecture dans l'abstrait. L'architecture se réfère toujours à quelque chose de concret. Brooks s'est avant tout préoccupé de l'architecture des systèmes d'exploitation, et il cultivait la métaphore des cathédrales : avec le temps, et les problèmes rencontrés par les premiers systèmes d'exploitation, l'expression est devenu plutôt péjorative. Son livre contient quelques recommandations très fortes comme :

« *By the architecture of a system, I mean the complete and detailed specification of the user interface* » ; aujourd'hui, on dirait les API, fléchant ainsi l'importance cruciale des interfaces. Où encore : « *Representation is the essence of programming.* […] *The data or tables* […] *is where the heart of a program lies* » ; aujourd'hui on dirait que le modèle de données est le cœur du système.

Aucun architecte de systèmes informatisés sérieux ne renierait ce genre de recommandations.

12. Il s'agit d'un ensemble de travaux intitulés DODAF, pour *DOD Architecture Framework*.

Le problème fondamental de l'architecte est d'organiser les communications et les interactions, non seulement entre les composants constitutifs du système, mais également entre les équipes en charge de la réalisation de ces composants. L'architecture du système et de son logiciel rétro-agit sur l'architecture du processus de développement, et vice et versa ; stabiliser cette relation est l'une des tâches les plus difficiles qui incombent à l'architecte.

Parmi les équipes, celle en charge de l'intégration des composants a un rôle bien particulier ; elle ne développe pas au sens classique du terme, mais elle assemble les composants dans un ordre qui n'est pas quelconque.

Il est évident que l'architecture du système doit inclure une stratégie d'assemblage, faute de quoi il est certain que la construction risque d'avoir des problèmes de fiabilité dans ses étapes intermédiaires.

La métaphore de la cathédrale reprend ici tout son sens : toutes les pierres (i.e. les composants logiciels) ayant été taillées correctement (c'est très difficile pour les clés de voûtes, les rosaces, les ogives), le problème est de comment les assembler sans que tout s'écroule en cours de montage (grâce aux échafaudages qui sont aussi des éléments d'architecture) ? L'épitaphe de Robert de Luzarche, architecte de la cathédrale d'Amiens, le qualifie comme « docteur es-pierres », tout un programme !

Dans une recension récente du SEI, *How do you define software architecture* ? on trouve les définitions suivantes :

Booch, Rumbaugh, Jacobson, dans : The UML modeling language user guide

« *An architecture is the set of significant decisions about organization of a software system, the selection of the structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaboration among those elements, the composition of these structural and behavioral elements into progressively larger subsystems, and the architectural style that guides this organization – these elements and their interfaces, their collaborations, and their composition.* »

NB : Cette définition met l'accent sur l'aspect organisation et les interfaces, ainsi que sur le modèle de croissance du système.

B. Boehm (le créateur du modèle d'estimation COCOMO)

« *A software system architecture comprises :*

- *A collection of software and system components, connections, and constraints.*
- *A collection of system stakeholders' need statements.*
- *A rationale which demonstrates that the components, connections, and constraints define a system that, if implemented, would satisfy the collection of system stakeholders' need statements.* »

NB : Ici, nous avons une relation très forte entre le donneur d'ordre MOA et la solution proposée par le MOE. Par « *connection* » il faut comprendre interface. Le « *rationale* » explicite la régulation et les choix conjoints MOA/MOE permettant de rechercher l'optimum CQFD + FURPSE + TCO + PESTEL. À son niveau, l'architecte met en œuvre le principe de moindre action, ou d'économie de pensée[a], que l'on peut considérer comme une variante du rasoir d'Ockham.

a. Formulé par E. Mach, *La mécanique*, chapitre IV, §IV, La science comme économie de pensée, réimpression chez J. Gabay ; H. Poincaré, *Science et méthode*, Livre I, chap. I, Le choix des faits, chap. II, L'avenir des mathématiques, Flammarion, 1913.

US Army technical architecture (ATA) et DODAF

« *A technical architecture is the minimal set of rules governing the arrangement, interaction, and interdependence of the parts or elements that together may be used to form an information system. Its purpose is to ensure that a conformant system satisfies a specified set of requirements. It is the build code for the Systems Architecture being constructed to satisfy Operational Architecture requirements.*

An Operational Architecture is a description, often graphical, which defines the force elements and the requirement to exchange information between those force elements. It defines the types of information, the frequency of its exchange, and what warfighting tasks are supported by these information exchanges. It specifies what the information systems are operationally required to do and where these operations are to be performed

À system architecture is a description, often graphical, of the systems solution used to satisfy the warfighter's Operational Architecture requirement. It defines the physical connection, location, and identification of nodes, radios, terminals, etc., associated with information exchange. It also specifies the system performance parameters. The Systems Architecture is constructed to satisfy the Operational Architecture requirements per the standards defined in the Technical Architecture. »

NB : Ici, l'accent est mis sur la recherche d'un extremum, i.e. compromis optimum, entre besoins opérationnels (y compris les performances) et solution proposée.

Règles d'architecture du point de vue projet

On peut compléter les définitions précédentes de l'architecture par une définition constructive plus opérationnelle, intégrant la problématique projet et donnant un critère de terminaison formulé en termes de réponses aux interrogations de l'architecte (cf. les 5 W : « *why, what, who, where, when* », auxquels on pourrait rajouter « *how* »)

Règle d'architecture dans une perspective projet

L'architecture d'un système est terminée quand, dans le projet de réalisation chaque acteur sait ce qu'il doit faire (aspects fonctionnels de l'architecture), comment il doit le faire (aspects non fonctionnels prenant en compte l'environnement du système, i.e. l'écosystème complet du projet selon PESTEL, donc les interfaces avec cet environnement) compte tenu des contraintes économiques de coût, de qualité et de délai, i.e. CQFD/TCO et FURPSE, conformément aux règles de gestion du portefeuille projet. **Tous les intégrats et leurs relations (donc leurs interfaces) sont identifiés**.

Dans cette définition, « chacun » peut être un individu, une équipe nominale (environ 7 personnes ± 2 est une bonne pratique[13]), un ensemble d'équipes partageant une même finalité technique et/ou métier ; ce que, dans les sciences de l'action, on appelle une unité active UA[14]. La vision hiérarchique du système, conformément aux bonnes pratiques de l'ingénierie système, est fondamentale : c'est la structure la plus simple qui permet de travailler efficacement. Toute autre structure doit être évaluée en terme de risque, en particulier à cause de la combinatoire induite qui peut vite devenir exponentielle (les interactions varient comme l'ensemble des parties, soit une complexité $O(k^n)$.

Ce qui a été dit pour l'équipe d'intégration est parfaitement pris en compte dans cette définition. Cette équipe teste et assemble les intégrats définis par l'architecte, avec comme critère la finalité d'emploi du système : pour travailler, elle a besoin des API, du plan d'assemblage et de scénarios d'emploi représentatifs de situations réelles, y compris en cas de défaillances qu'il faudra caractériser. Si elle n'en dispose pas, l'architecture n'est pas terminée, il y a donc un risque considérable à démarrer la réalisation.

Dans cette définition, on ne dit pas ce qu'est l'architecture du système dans l'abstrait, on se contente de dire, par rapport au projet de réalisation, quand le travail d'architecture pourra être déclaré fini. Dans cette approche, la réalité est le projet et son écosystème, et c'est dans le cadre du projet que se développe l'architecture, ce qui est une bonne façon d'éviter les dérapages conduisant à des développements inutiles : si l'on ne sait pas intégrer ou maintenir, ou comment former les usagers, mieux vaut s'interroger avant d'être face à ces échéances.

Sur cette base, on peut alors faire jouer des critères d'optimalité concernant a) la taille du système en points de fonctions ou en volume de code ou en nombre d'intégrats ; b) la taille des tests, son coût, sa qualité, le délai de la réalisation de la 1re version ; c) sa durée de vie (combien de versions est-il raisonnable de prévoir) ; d) le délai d'une intégration complète ou partielle, etc.

Compte tenue de la variété de ces différents critères, on sait qu'il n'y a pas de solution unique au problème posé. La situation est tout à fait analogue à celle du second théorème de Shannon qui montre qu'il existe toujours un moyen de compen-

13. Explication dans J. Printz, *Puissance et limites des systèmes informatisés*, Hermès.
14. Cf. J. Printz, *Ecosystème des projets informatiques*, Hermès, 2005.

ser les erreurs résultant du bruit du canal de communication (dans notre cas, le canal est le processus de développement lui-même) par un code correcteur ad hoc, mais que la construction de ce code (dans notre cas, le système qualité associé au processus de développement qui est clairement une redondance destinée à corriger les défauts du processus de développement) est à réaliser au cas par cas.

Plus qu'une chose, ou un concept, réductible à un nom, l'architecture est un processus de construction étroitement couplé à son environnement socio-économique. L'ordre de prise en compte des différentes problématiques, et les transformations opérées sur les différentes représentations qui vont alimenter le processus de programmation (« l'implémentation », dans le rapport du *NATO science committee*) est fondamentalement non linéaire et résulte des interactions entre les acteurs.

Règles du point de vue de la programmation

Selon le niveau des exigences, à compétence de programmeur égale, le code est plus ou moins coûteux à réaliser. En suivant l'approche du modèle d'estimation COCOMO, on considère trois catégories de programmes auxquelles vont correspondre trois équations d'effort (ce sont celles du modèle basique) :

Equation 1 : $Effort_{en\ hm} = 2{,}4 \times (Taille_{en\ kls})^{1{,}05}$,

Equation 2 : $Effort_{en\ hm} = 3{,}0 \times (Taille_{en\ kls})^{1{,}12}$,

Equation 3 : $Effort_{en\ hm} = 3{,}6 \times (Taille_{en\ kls})^{1{,}20}$,

soit pour 100 KLS respectivement : 302 hm, 521 hm, 904 hm. Toute chose égale par ailleurs, il y a un facteur ***trois*** selon la nature du code à réaliser.

Résumons la ligne de raisonnement qui justifie le bien fondé de cette catégorisation (pour une présentation en détail, nous renvoyons à nos ouvrages déjà cités) :

Règle d'architecture dans une perspective programmation

Tout programme comprend une ***composante réactive*** (i.e. sa structure de contrôle qui agit sur le flot ; régie par l'équation 3) et une ***composante transformationnelle*** (i.e. ce qui régit les changements d'état de la mémoire) qui peut être ***simple*** ou ***complexe***, régie par les équations 1 et/ou 2. Les transformations effectuées constituent des opérations dont l'architecte peut minimiser la variété et le nombre, et leur donner un caractère de généralité permettant la réutilisation dans différents contextes.

La composante transformationnelle peut elle-même être décomposée en deux sous-composantes :

Transformation simple (une transcription) régie par l'équation 1, comme par exemple une règle de gestion métier dont la programmation n'est qu'une traduction en langage informatique d'une règle exprimable en langage naturel propre au métier ; c'est l'idée fondamentale du langage COBOL, à l'origine. Une telle programmation peut être validée par une simple comparaison aux textes définissant les règles métiers.

Transformation complexe, régie par l'équation 2, à l'aide d'algorithmes, d'un ensemble d'états pouvant présenter une grande variabilité, et un volume quelconque, en un résultat fini (traitements statistiques, optimisation, traduction, paramétrage, etc.). Pour être intéressant, au-delà du simple cas particulier, l'algorithme doit présenter un caractère de généralité plus ou moins poussé, ce qui permet d'englober un grand nombre de cas en une seule formulation abstraite. L'abstraction est l'essence du travail de programmation. Valider un algorithme, c'est démontrer que tous les cas, et rien que les cas (i.e. la robustesse, ou résilience, de l'algorithme), auxquels il s'applique sont effectivement pris en compte, quelle que soit la méthode de démonstration adoptée, expérimentale à l'aide de tests, ou formelle avec un démonstrateur.

Pour la composante réactive de l'entité programme selon le niveau d'intégration (intégrat, module, application, système), ont peut appliquer la règle suivante:

Règle d'architecture de la composante réactive

La composante réactive est elle-même une structure que l'on peut organiser en hiérarchie, depuis le workflow global jusqu'aux enchaînements d'intégrats élémentaires. L'identification des hiérarchies nécessite une compréhension profonde de la mécanique des enchaînements en distinguant soigneusement ce qui est purement fonctionnel métier et ce qui est induit par la logique informatique. La hiérarchie du contrôle permet de visualiser la chronologie de la transformation, de faciliter le diagnostic en cas de défaillance (la trace, ou chronique, est le langage de l'automate correspondant à l'enchaînement) et l'optimisation de l'effort de VVT.

Règle de simplicité de l'architecture, du point de vue de l'intégration – Architecture testable

Une architecture testable (TOA, *Test Oriented Architecture*) minimise le volume de tests nécessaires à la validation du produit logiciel correspondant, et ceci dans le contexte d'exploitation réelle du logiciel et de son cycle de vie qui peut être très long. Le critère de simplicité de l'architecture est corrélé au volume de tests à produire et à l'effort nécessaire à les gérer. On peut proposer la règle suivante :

Règle de simplicité de l'architecture : Une architecture est plus simple qu'une autre si a) le volume de test à produire pour la valider, et que b) l'effort correspondant pour produire les tests est significativement moindre.

La testabilité résulte de la prise en compte des exigences SE de FURPSE (i.e. MCO du système). Voir le chapitre 12, Simplicité - complexité, pour une analyse plus fouillée.

2.5 TERMINOLOGIE INTRODUITE DANS CE CHAPITRE

Termes utilisés pour la nomenclature : nous recommandons l'usage des normes de la profession (en particulier IEEE *Software engineering*, *standards collection*).Voir la figure 2.1.

Module – Intégrat : un intégrat est un module qui satisfait aux critères d'intégration (ce qui présuppose une plate-forme particulière, éventuellement une plate-forme générique que l'on sait traduire sur une plate-forme physique).

Application – Application intégrée sur un équipement : c'est un ensemble de modules (architecture statique) assemblés dans un certain ordre. L'assemblage peut être statique ou dynamique (lors de l'exécution de l'application). Une application est dite intégrée lorsque tous ses modules ont satisfait à tous les critères d'intégration : une application intégrée est un assemblage d'intégrats. Une application non intégrée n'a aucune utilité pour l'usager du système. Quand on parlera d'application on présupposera toujours qu'il s'agit d'une application intégrée.

Terminologie utilisée pour caractériser le degré de formalisation des entités architecturales

Les constituants définissent la sémantique de l'entité. Les phases définissent le degré de formalisation par rapport au cycle de vie adopté pour l'ingénierie du système. Nous utilisons les termes phases/états, faute d'une meilleure dénomination, pour préciser les modalités d'existence d'une entité architecturale. Le fait, pour une entité, d'être exécutable ou non, est un véritable état, au sens physique du terme (NB : l'eau existe sous l'état gazeux, liquide et solide). Le fait d'être non exécutable, peut toutefois signifier une conformité par rapport aux règles du référentiel système qui joue le rôle d'un méta-langage. Nous allons considérer cinq modalités d'existence qui constituent des états dans lesquels peuvent se trouver les entités architecturales :

1. La première modalité est d'exister en tant que besoin.
2. La seconde est d'exister en tant qu'entité nommée du système, avec un nom explicite qui la situe dans l'organisation du système : c'est la conception générale, avec les interfaces associées (relations avec d'autres entités).
3. La troisième est d'exister en tant que transformation à effectuer ; c'est le passage de la description en extension à une description en intention. Les transformations peuvent être nommées et leurs enchaînements précisés. Les ressources nécessaires doivent être précisées.
4. La quatrième est d'exister en tant que programme écrit dans les langages du référentiel système, avec les tests unitaires associés aux programmes.
5. La cinquième et dernière modalité est d'exister en tant que service intégré disponible, c'est-à-dire qu'à côté du texte programmatique de l'entité, on dispose des tests d'intégration qui garantissent son contrat de service du point de vue métier.

NB : une transformation est caractérisée par ses interfaces. Le résultat d'une transformation est un certain état mémoire, complété par un vecteur d'état qui fournit une information qualitative sur la façon dont cet état a été obtenu (cf. chapitre 8).

Le tableau ci-dessous résume les différents états possibles et leurs transitions.

Tableau 2.3 – Etat des entités architecturales

	Etat (ou phase) de l'entité architecturale →				
Nature (FURPSE) **du constituant ↓**	EB/EC En tant que besoin [R0]	CG – Interfaces externes	CD – Interfaces internes	Programmée	Intégrée
Données (y compris les sources et les puits)	Cycle de vie acteurs [R1]	Vue externe des données (langage pivot) [R2]	Vue interne selon modèles de données utilisés	Vue interne selon le(s) langage(s) de programmation	Vue physique, selon nature des périphériques de stockage
Opérations	Nommées (nomenclature des opérations)	Définies en termes de fonctions et/ou services	Décrites en termes de transactions sur les données	Décrites selon les règles du/des langages de programmation	On dispose de la correspondance logique ↔ physique
Evénements à contrôler - Interfaces	Métier	Messages externes	Messages internes	Evénements logiciels programmés	Evénements plate(s)-forme(s)
Nature des représentations adoptées →	En extension. [R3]	En intention, via les différents langages utilisés tout au long du cycle de vie. [R4]			
Propriété →	Non exécutables + conformité au référentiel			Exécutables + Tests	
		Représentation logique		Représentation physique	

[R0] Il s'agit de l'expression de besoin et des exigences comportementales (EB/EC) qui traduisent le FURPSE en tant que besoin, plus les contraintes liées à l'environnement, appelées contraintes PESTEL, selon la terminologie IS et INCOSE.

[R1] Décrit le cycle de vie de la donnée (cycle CRUD) du point de vue des acteurs métier (voir figure 2.21).

La sémantique de la donnée s'obtient en analysant les besoins de l'ensemble des acteurs qui en ont l'usage ; c'est la réponse à la question à quoi et à qui sert la donnée (cf. les 5 W déjà cités dans la règle d'architecture projet).

Voir également la figure 2.4, description des données, qui fait partie de la création effective dans le SI.

[R2] Le diagramme de collaboration (diagramme de contexte) permet l'identification des ports logiques de l'entité architecturale aux différents niveaux d'abstraction (système, sous-système, application, intégrat).

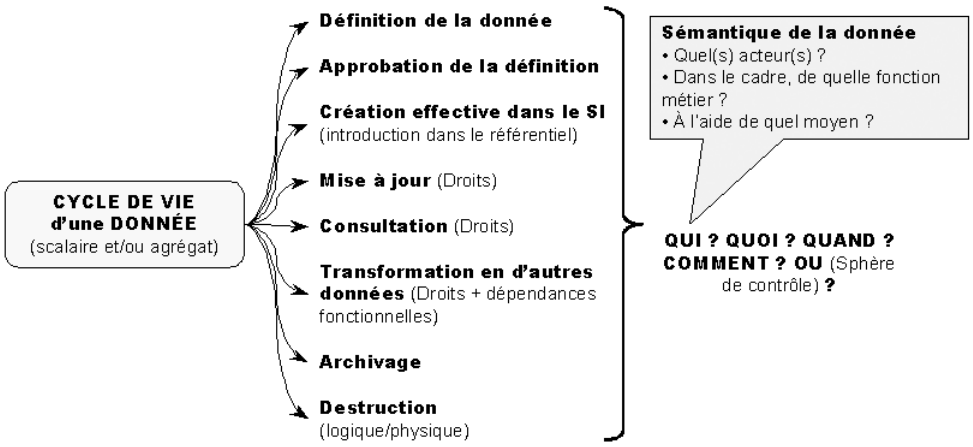


Figure 2.21 - Cycle de vie des données et définition de la sémantique

[R3] Les cas d'emplois sont une représentation en extension des processus métiers. Ils sont élaborés par les experts métiers, avec l'aide éventuelle d'un expert en modélisation dont le rôle est de donner une forme standard au cas d'emplois pour améliorer sa compréhension vis-à-vis des architecte/programmeur. En tout état de cause la sémantique intuitive est la prérogative de l'expert métier qui doit être consulté dès qu'il y a doute. C'est une technique dont l'objectif est de mettre de l'ordre et de la cohérence dans un ensemble de cas particuliers, à l'aide des processus métiers. Le cas d'emploi est par définition non exécutable, mais c'est un support à la discussion entre les acteurs métier et la MOA, en particulier pour définir les tests d'intégration et la recette du point de vue du métier.

NB : En théorie des ensembles, la représentation en extension d'un ensemble est l'énumération de ses éléments. Une table de logarithme est une représentation en extension de la fonction Log dont on connaît depuis longtemps différents algorithmes de calcul qui sont des représentations en intention.

[R4] La représentation en intention nécessite un effort de conceptualisation, à partir des cas d'emplois, afin d'en extraire une procédure générale englobant un ensemble de cas particuliers, sous la forme d'une procédure métier ou d'un algorithme au sens informatique.

La représentation en intention du besoin EB/EC se matérialise via différents types de langages :

Les langages naturels, éventuellement améliorés avec des représentations graphiques, et un glossaire précis des termes utilisés dans ce langage. Ils ne sont pas exécutables, car il n'y a pas de sémantique opérationnelle explicite à ce niveau, ce qui n'interdit pas de s'exprimer de façon rigoureuse.

Les langages formalisés au niveau syntaxique peuvent être manipulés via des éditeurs syntaxiques. XML est un bon exemple de ce type de langage.

Les langages formalisés au niveau sémantique peuvent être compilés et exécutés sur une plate-forme (éventuellement à l'aide d'une machine abstraite). C'est le cas de tous les langages de programmation (avec quelques précautions pour les langages objets dont la sémantique est mal définie, à cause des nombreux aspects dynamiques propres à ces langages), y compris les langages formels comme B, SDL, ESTEREL, LUSTRE, etc.

Dans la philosophie MDA, les modèles peuvent être traduits les uns dans les autres. Dans l'état CG/CD l'entité architecturale est indépendante, en théorie, de la plate-forme d'exécution (i.e. un PIM dans le jargon MDA) ; dans l'état Programmée/Intégrée, l'entité est exécutable sur une plate-forme (i.e. un PSM). Le passage automatique PIM→PSM implique la formalisation de la sémantique de la plate-forme ; le traducteur correspondant est un compilateur (voir le chapitre 6).

3

Propriétés indésirables des entités architecturales

Défauts et anomalies de fonctionnement, correction des logiciels

3.1 DÉFAUTS ET ANOMALIES DE FONCTIONNEMENT DES ENTITÉS ARCHITECTURALES

Dans les sciences de l'ingénieur classiques, les matériaux et les phénomènes que l'ingénieur utilise à son profit présentent tous des défauts, des anomalies temporaires ou permanentes qui ne doivent pas empêcher les machines de fonctionner ou d'accomplir leur service. C'est une contrainte inhérente à la nature du monde physique que l'on a su progressivement théoriser. La résistance des matériaux [RDM] est la science de l'ingénieur qui étudie les conditions de rupture de tous les matériaux que nous utilisons. C'est grâce à la résistance des matériaux que l'on a pu optimiser la quantité de matière à utiliser pour faire un pont qui tient.

En constructions civiles, en mécanique, en électrotechnique, les ingénieurs sont familiers depuis toujours des phénomènes d'usure qui progressivement dégradent les conditions de bon fonctionnement des machines ou des constructions, selon des lois que l'on sait modéliser, du moins pour les matériaux connus depuis un certain temps. Chacun se rappelle les problèmes rencontrés avec les premiers ouvrages en béton précontraint ou les premiers ponts suspendus exposés à des vents violents. Pour surveiller ces différents phénomènes les ingénieurs ont intégré dans leur production toute une mécanique de surveillance qui permet de suivre dans le temps le comportement des matériaux et prévoir l'arrivée des pannes.

Dans les sciences de l'ingénieur qui traitent de l'information, la théorie de l'information, créée par l'ingénieur C. Shannon, née avec les premières communications et le radar, est la première science entièrement fondée sur le traitement de

l'erreur. R. Hamming[1], l'inventeur des codes qui portent son nom, disait : « *Most bodies of knowledge give errors a secondary role, and recognize their existence only in the later stages of design. Both coding and information theory give a central role to errors and are therefore of special interest, since in real-life errors (noise) is everywhere.* » L'ingénieur O. Heaviside inventa une aberration mathématique, le calcul opérationnel (ou calcul symbolique), pour calculer les propagations des signaux électromagnétiques dans les conducteurs que L.Schwartz expliqua rigoureusement cinquante ans plus tard avec la théorie des distributions[2]. En fonction de l'amortissement du signal, cela permettait de calculer le nombre de répéteurs à placer sur la ligne de communication. Qu'en est-il des abstractions informatiques issues du travail des concepteurs, des architectes et des programmeurs ? Comment garantir la sûreté de fonctionnement d'une chaîne de traitements qui n'est jamais qu'une propagation d'information doublée d'un calcul entre un acteur émetteur et un acteur récepteur ?

Toutes les abstractions informatiques, y compris la logique câblée dans les microprocesseurs, sont créées de toutes pièces par l'homme, et de ce fait soumise à l'erreur humaine. Dans les abstractions mises en œuvre pour réaliser l'ordinateur et les programmes qui lui donneront son « intelligence » (système d'exploitation, progiciels, programmes utilisateurs), seul le langage diffère, et la malédiction de Babel[3] est déjà au cœur de notre construction : comment assurer la cohérence d'un ensemble hétéro-logique aussi hétéroclite ? Le totalitarisme de la langue informatique unique rend pensable, donc possible, les entreprises les plus arrogantes qui toujours échouent (ALGOL, PL1, Ada, PCTE, l'IA, etc.) ; la sagesse, c'est le principe de subsidiarité qui implique l'interopérabilité et la recherche de conventions sémantiques (qu'est ce que ça fait, à quoi ça sert !) partagées et acceptées par tous.

La qualité des intégrats définis par l'architecte et assemblés dans le processus d'intégration, est directement fonction de la qualité du travail du programmeur pris dans un sens extensif : un ingénieur électronicien qui programme un SOC (*System On Chip*) est un programmeur, un ingénieur qui réalise le paramétrage d'un PGI comme SAP pour le compte d'un client X est également un programmeur, un usager qui personnalise son poste de travail est un programmeur. Un ingénieur en aéronautique qui programme une loi de pilotage pour l'Airbus A380 est évidemment un programmeur. Chacun de ces programmeurs est sujet à l'erreur.

Il est donc fondamental de comprendre la relation complexe que le langage informatique utilisé entretient avec la machine sous-jacente et avec l'usager du langage, et de prendre en compte la présence inévitable de défauts dans les logiciels (s'il n'y en a pas, c'est tant mieux !).

---

1. Dans, *Coding and information theory*, Prentice Hall, 1980.
2. Voir son livre autobiographique : *Un mathématicien aux prises avec son temps*, chap. VI, L'invention des distributions, chez Odile Jacob, 1997, et l'introduction de *La théorie des distributions*, chez Hermann, 1957.
3. Dans Genèse, 11, 7 : « Mais le seigneur descendit pour voir la ville et la tour […] Allons, descendons pour mettre la confusion dans leur langage, en sorte qu'ils ne se comprennent plus l'un l'autre. »

Examinons les conditions de production d'un texte informatique par un « programmeur ».

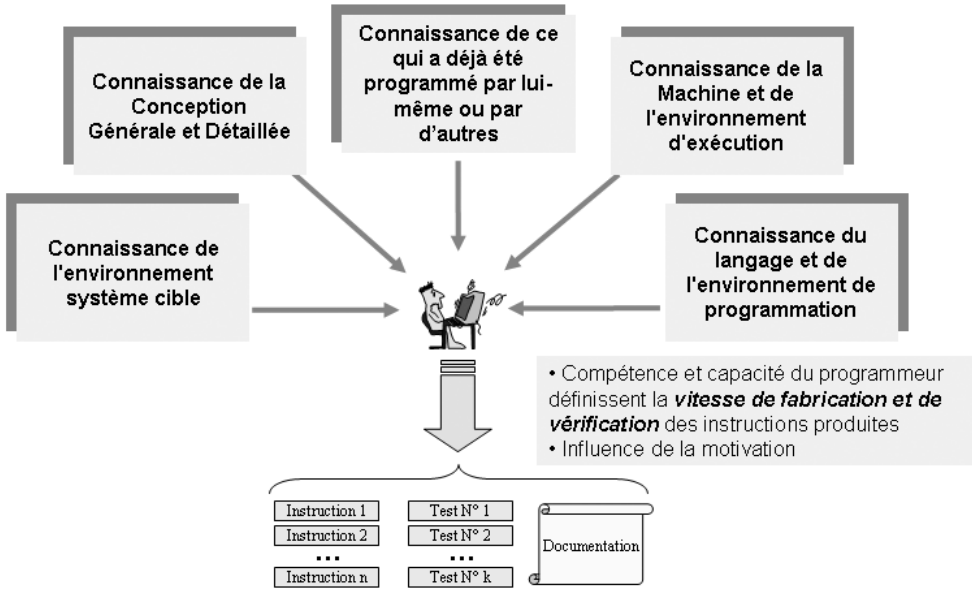


Figure 3.1 - Ecosystème du programmeur

Le cerveau du programmeur est une « machine » à faire des choix et à prendre des décisions, sous la forme :

```
si les conditions C1, C2,… Cn sont satisfaites, alors écrire la séquence
d'instructions I1, I2,…, Ik
```

On sait depuis longtemps que le taux d'erreur d'un opérateur humain[4] est une fonction croissante du nombre de décisions prises par unité de temps et des perturbations, de la fatigue, etc., endurées par le programmeur. Plus il y a de décisions, plus le nombre d'erreurs humaines (i.e. défauts dans le texte du programme) est élevé.

Depuis les premières statistiques produites par la NASA avec le programme navette spatiale, on a pris l'habitude de mesurer la qualité d'un programme du point de vue des défauts par le taux d'erreur résiduelle ramené au millier d'instructions effectivement livrées, soit :

- Partie embarquée : 700 KLS ; 0,1 Err./KLS.
- Partie sol : 1.400 KLS ; 0,5 Err./KLS.

4. Les ergonomes signalent des taux de 5 à 10 erreurs par heure d'activité effective ; cf. F. Vanderhaegen, Analyse et contrôle de l'erreur humaine, Hermès ; Salvendy, Handbook of human factors and ergonomics, Wiley interscience.

Aux normes de l'édition, un texte de 1 000 instructions correspond à une vingtaine de pages, ce qui représente un effort certain de mémorisation et de rigueur logique pour arriver à une seule erreur résiduelle.

Rappelons, pour donner une échelle de comparaison, que la preuve du théorème de Fermat apportée par A.Wiles en 1993 est un texte de 200 pages, organisé en 6 sections : « la preuve est un argument énorme, construite à partir de centaines de calculs enchevêtrés, et collés ensemble par des milliers de relations logiques » (cf. S. Singh, *Fermat's last theorem*, Fourth Estate, 1997). Une erreur détectée par les re-lecteurs nécessita 130 pages additionnelles, et la preuve ne fut définitivement acquise qu'en 1995.

Dans un compilateur, un générateur de code avec optimisation est un texte d'environ 50 000 instructions de langage de type C/C++, soit environ 1 000 pages. C'est le fruit du travail d'une équipe de 4-5 programmeurs connaissant parfaitement les techniques de compilation, sur une durée d'environ deux ans. C'est aussi un défi logique d'une autre nature que le théorème de Fermat, étant entendu qu'une défaillance du générateur de code se traduira par un programme généré faux, dont l'exécution révèlera, peut-être, la fausseté, beaucoup plus tard. Le compilateur ne doit pas ajouter ses erreurs à celles commises par le programmeur utilisateur du langage, sinon ce dernier perdra toute confiance dans le langage.

B.Beizer[5] a fait une analyse détaillée portant sur 3,6 millions de LS qui fait apparaître un taux moyen de 2,3 défauts/KLS.

En examinant le phénomène sur le cycle de vie (CDV), il y a consensus des experts sur les chiffres suivants (figure 3.2).

En fin de programmation, on observe des taux d'erreurs de l'ordre d'une erreur toutes les 10-15 lignes de codes. Pour installer dans de bonne condition, les processus IVVT doivent ramener ce taux aux alentours de 1-2 Err/KLS.

Plus près de nous, l'échec du vol Ariane 501 dû à une erreur triviale de programmation qui était restée cachée sur Ariane 4, montre qu'il nous faut être modeste vis-à-vis du phénomène des erreurs résiduelles. Même les codes les plus éprouvés, parfois dans un contexte mal défini, nous révèlent de mauvaises surprises.

Le défaut introduit lors de la création du texte pourra produire une défaillance beaucoup plus tard, ce qui nécessite des tests et des diagnostics ad hoc pour effectuer la correction (c'est une contrainte de maintenabilité, i.e. *System Maintainability*, le S de FURPSE). Idéalement, comme c'est la règle dans l'ingénierie matérielle (construction, machines, équipements électroniques, etc.), le produit intègre les tests et diagnostics en ligne (OLTD, *On Line Test and Diagnostic*) pour analyser le phénomène in-vivo, et non post-mortem. De tels tests diagnostiques sont indispensables si les circonstances de la défaillance sont difficilement reproductibles (cf. chapitre 5).

5. Dans son livre, *Software testing techniques*, Van Nostrand Reinhold, 1990.

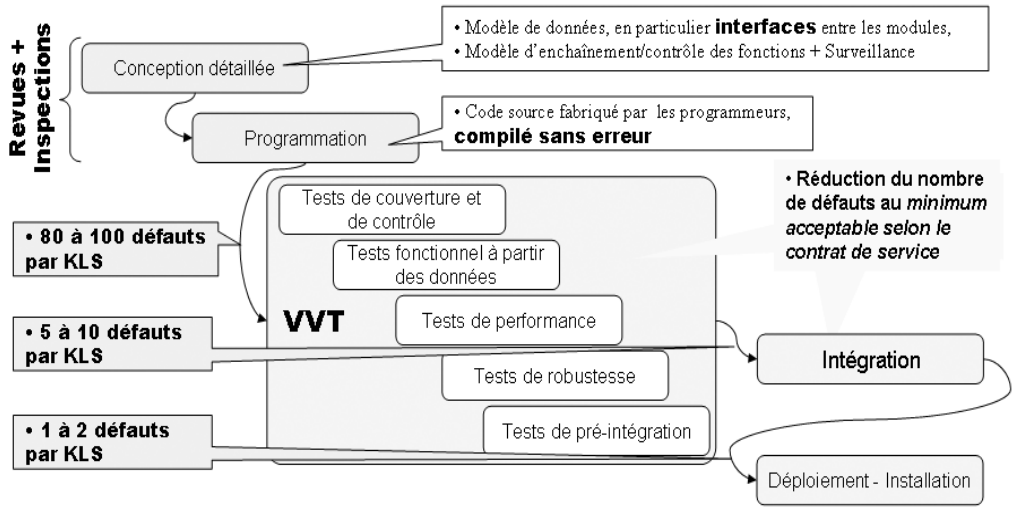


Figure 3.2 - Introduction et correction des défauts sur le CDV

Sur ces bases, la règle de l'architecte est la suivante :

Règle de qualité des intégrats

Tout intégrat ayant satisfait à tous les tests de conformité résultant du CDV adopté par l'organisation de développement révèlera, durant son exploitation, un taux d'erreurs résiduelles fonction de la durée et du nombre d'expositions de l'intégrat résultant du déploiement effectué. **Ce taux n'est jamais nul**.

La problématique de l'erreur humaine a été détaillée dans notre ouvrage *Puissance et limite des systèmes informatisés*. Contentons-nous ici de rappeler deux grandes catégories d'erreurs :

1. Catégorie N°1 : Celles résultant d'erreurs de jugement avérées du programmeur, quand bien même celui-ci dispose d'une information précise et complète du contexte décisionnel (c'est un jeu à information complète, comme le jeu d'échec).
2. Catégorie N°2 : Celles résultant de décisions erronées dues au manque d'information ou aux ambiguïtés du contexte décisionnel (c'est un jeu à information incomplète, comme le poker).

Dans le premier cas, le programmeur est seul en cause ; dans le second, c'est l'écosystème qui est en cause. Face à cette réalité qui est celle de tous les ingénieurs, quelle doit être l'attitude de l'architecte informaticien ? Il ne doit surtout pas croire qu'il a affaire à des entités mathématiques aux comportements parfaitement définis, conformes à l'idéal déterministe que Laplace a si bien défini : « Nous devons donc envisager l'état présent de l'univers comme l'effet de son état antérieur, et comme la cause de ce qui va suivre. Une intelligence qui pour un instant donné connaîtrait toutes les forces dont la nature est animée et la situation respective des êtres qui la

composent, si d'ailleurs elle était assez vaste pour soumettre ces données à l'analyse, embrasserait dans la même formule les mouvements des plus grands corps de l'univers et ceux du plus léger atome : rien ne serait incertain pour elle, et l'avenir, comme le passé, serait présent à ses yeux. »[6]. Une illusion scientiste, dans l'euphorie positiviste du XIX[e] siècle, détruite par H.Poincaré, le vrai découvreur du chaos, dans les années 1890, avec ses études de mécanique céleste.

L'architecte a affaire à des entités « physiques » qui peuvent contenir des défauts et dont le comportement peut ne pas être stable dans le temps. Les réseaux, les ressources partagées et les progiciels introduisent de l'aléatoire dans le comportement et de l'incertitude sur la durée des opérations et la validité des résultats. Ce qui veut dire que sur une suite d'intégrats qui s'enchaînent en séquence, I1→I2→I3, la succession ne peut pas être garantie à 100% mais seulement à 1–ε. Si on lit la séquence comme une suite d'implications logiques sur le modèle : état *E0* VRAI, si la transformation $E0 \xrightarrow{I1} E1$ via I1 VRAI, alors *E1* VRAI (NB : c'est la règle de détachement de la logique classique, i.e. SI *a*, et SI $a \supset b$, ALORS b, dite du modus ponens), et ainsi de suite, il est évident que ce n'est pas l'implication classique qui s'applique, car ce que permet l'implication logique, à savoir $F \supset V \rightarrow V$, n'a pas de sens dans ce contexte d'exécution). Retenons que la règle logique du tiers exclu qui nous est familière ne va pas s'appliquer à la logique des enchaînements[7].

Le schéma logique de l'implication des enchaînements est le suivant (figure 3.3) :

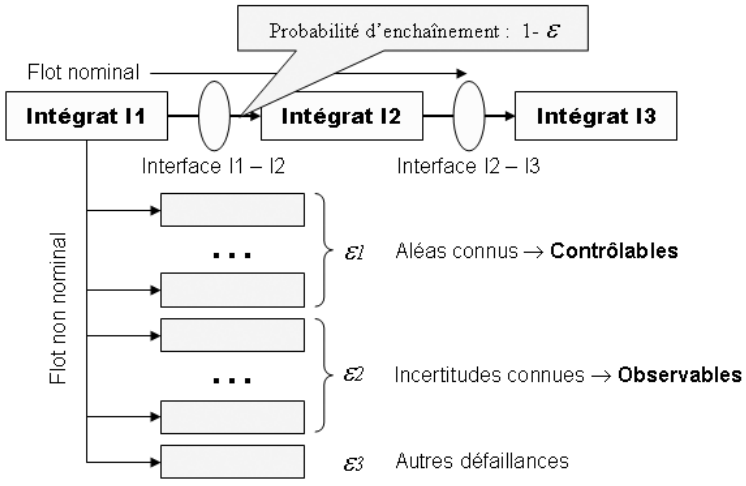


Figure 3.3 - Modalités de l'enchaînement des intégrats

6. Pierre-Simon Laplace, *Calcul des probabilités*, qui contient *l'essai philosophique sur les probabilités*, d'où est extrait la citation.

7. Voir von Neumann, *Theory of self-reproducing automata*, *1ère partie : Theory and organization of complicated automata*, University of Illinois Press, 1966.

Si les situations incertaines, dont la probabilité est estimée à , ne sont pas séparées du flot nominal distingué, le flot nominal I1→ I2 va être dégradé par une incertitude ε, et ainsi de suite, à chacune des étapes de l'enchaînement. L'incertitude d'une séquence I1, I2, I3 suit en première approximation une loi additive du type $I_{I2,I2,I3}=1-(\varepsilon_{I2}+\varepsilon_{I2}+\varepsilon_{I3}+résidu)$ qui peut tendre vers une limite <1, ou vers 0 selon les stratégies d'enchaînements adoptées (par exemple la tolérance aux pannes, i.e. fault-tolérant, comme dans les codes correcteurs d'erreurs qui fiabilisent les communications).

Pour l'architecte, cela se traduit par la règle suivante :

Règle de l'enchaînement des intégrats

- Vérifier aussi précisément que possible (i.e. information connue du programmeur) que le « contrat » qui valide la transition I1I2 est satisfait.
- Si le contrat est satisfait, mais que la transition I1→I2 conduit tout de même vers un comportement non nominal, on a alors deux cas : a) il y a eu erreur de la part du programmeur (catégorie d'erreur N° 1, ci-dessus) ou b) l'information permettant d'établir le contrat est incomplète, il faut la compléter et « durcir » le contrat de façon à le rendre plus conforme à la réalité « physique » (catégorie d'erreur N° 2 ci-dessus).

Notons qu'en ramenant les ε à 0, on se met dans le cas limite d'un monde parfait qui ne connaît pas l'erreur, un peu comme la théorie des gaz parfait en thermodynamique, ou comme la physique du point matériel en mécanique, ou comme la mécanique des milieux continus en résistance des matériaux qui ignore les défauts de structure.

Dans le premier cas, avec ε=0, l'ingénieur se comporte en mathématicien, et ne traite qu'une partie du problème ; dans le second, il assume pleinement le risque inhérent aux constructions humaines et prépare des contre-mesures appropriées qui font partie de sa réflexion architecturale.

Ecoutons H. Poincaré[8] : « Le but principal de l'enseignement mathématique est de développer certaines facultés de l'esprit et parmi elles l'intuition n'est pas la moins précieuse. C'est par elle que le monde mathématique reste en contact avec le monde réel et quand les mathématiques pures pourraient s'en passer, il faudrait toujours y avoir recours pour combler l'abîme qui sépare le symbole de la réalité. […] L'ingénieur doit recevoir une éducation mathématique complète, mais à quoi doit-elle lui servir ? à voir les divers aspects des choses et à les voir vite ; il n'a pas le temps de chercher la petite bête. Il faut que, dans les objets physiques complexes qui s'offrent à lui, il reconnaisse promptement le point où pourront avoir prise les outils mathématiques que nous lui avons mis en main. »

8. Dans *Science et méthode*, chapitre II, Les définitions mathématiques et l'enseignement, Flammarion.

Que n'aurait-il dit des objets du monde de l'information aux contours sémantiques souvent très mal définis ! La réponse est venue quelques cinquante ans plus tard avec J. von Neumann et sa théorie des automates fiables[9].

Le problème que l'architecte doit résoudre peut se poser comme suit :

- Existe t-il un dispositif, une façon de faire, qui garantit que l'incertitude des enchaînements I1, I2, … In, soit

  $$\sum_{1}^{n} I_k \leq \varepsilon$$

  avec ε fixé à l'avance ; et si oui, comment construire un tel dispositif ?
- Existe-t-il un dispositif qui garantit que le ε d'un intégrat est strictement 0, moyennant certaines hypothèses sur l'environnement de l'intégrat (preuve formelle de l'intégrat logiciel).

Les ingénieurs des télécommunications ont réussi à résoudre le problème, grâce à la théorie de l'information et à l'invention des codes correcteurs d'erreurs (NB : c'est le 2e théorème de C. Shannon qui en établit l'existence). Il n'y a pas de fatalité à l'erreur, il faut regarder le problème en face, avec lucidité et pragmatisme.

Notons que l'existence d'une preuve formelle, quand elle existe, donne une information précieuse en cas de défaillance de l'intégrat. En effet, si la construction de l'intégrat logiciel est prouvée correcte, la cause de l'erreur est à rechercher hors de l'intégrat, dans le contexte d'exécution, ce qui diminue la taille de l'espace de recherche (voir chapitre 13). Dans ce cas, le problème de l'architecture est ramené à un problème de construction dont chaque étape fait l'objet de validation et de vérification méthodique (cf. notre ouvrage, *Ecosystème des projets informatiques*, Hermès, 2005).

Dans la partie 3, nous expliquerons comment intégrer ces contraintes dès l'analyse fonctionnelle.

3.2 COMPORTEMENTS DÉGÉNÉRATIFS DES ENTITÉS ARCHITECTURALES EN COURS D'EXÉCUTION

Le phénomène équivalent à l'usure d'un matériau ou d'une machine est beaucoup plus étrange quand il s'agit d'intégrat logiciel. Le comportement du logiciel est affecté par sa durée de fonctionnement ce qui va le faire progressivement sortir des contraintes définissant son fonctionnement nominal (cf. définition de la sûreté de fonctionnement, chapitre 13) si tant est que celles-ci soient définies explicitement.

9. Cf. sa monographie : *Probabilistic logics and the synthesis of reliable organisms from unreliable components*, Œuvres complètes, Vol. N°5.

Le phénomène est apparu dans les années 70 avec la multiprogrammation et des mécanismes comme la mémoire virtuelle. Il s'est amplifié avec le traitement en ligne des transactions (OLTP) et les bases de données partagées entre des milliers d'utilisateurs (cf. les premiers transactionnels avec les systèmes de réservation type SABRE pour American airlines ou AMADEUS pour Air France) nécessitant des mémoires caches sophistiquées pour minimiser le nombre d'entrées-sorties disques et/ou réseaux. Le phénomène est aujourd'hui à son maximum avec les architectures distribuées massives, et est aggravé par de mauvaises pratiques de programmation.

Il est important de comprendre comment, à partir d'une situation en apparence déterministe, on évolue rapidement vers une situation non déterministe, pouvant même devenir chaotique. Le mécanisme de mémoire virtuelle est un excellent exemple de ce type de situation.

L'invention de la mémoire virtuelle est la réponse à un besoin de simplification de la programmation et d'une opportunité technologique offerte par les mémoires de masse (tambours au départ, puis exclusivement sur disques), à accès aléatoire rapide.

Le programmeur des années 60-70 gérait lui-même sa mémoire, avec des partitions, des overlays, etc. ce qui compliquait singulièrement le travail de programmation. L'exiguïté des mémoires centrales d'alors (128 K étaient considérés comme une grande mémoire !) et la nécessité de charger en mémoire plusieurs programmes (i.e. la multiprogrammation) pour optimiser le partage de ressources de la machine, rendait les interactions programmeurs↔machine encore un peu plus complexes et interdépendantes, un vrai « casse-tête ».

La mémoire virtuelle libère le programmeur, du moins en apparence, du souci de savoir si son programme va effectivement tenir en mémoire, et des dialogues avec le système d'exploitation pour allouer les partitions nécessaires à l'exécution. Toute la logique d'allocation est reportée vers le système d'exploitation qui régule automatiquement les demandes générées par les différents programmes en exécution dans la machine à l'instant t.

Sans rentrer dans le détail de l'algorithme de gestion de la mémoire virtuelle (il y a de très bons livres sur le sujet) rappelons-en les principes, afin d'éclairer notre propos sur les phénomènes d'usure.

La mémoire centrale MC et les disques servant à l'extension physique de la MC sont organisés en pages, de taille 1 à 2 K octets, de façon à former un continuum qui devient la taille de la mémoire visible du programmeur qui peut alors contenir plusieurs dizaines, voire centaines, de programmes. L'image mémoire est construite par le linker et le loader du système d'exploitation (SE).

Le schéma de principe du dispositif est le suivant (figure 3.4) :

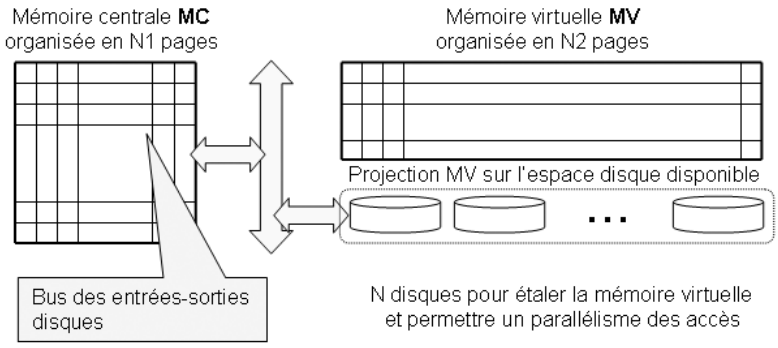


Figure 3.4 - Topologie de la mémoire virtuelle

L'image du programme dans la mémoire virtuelle respecte la topologie du programme tel qu'il a été créé par le programmeur, ce qui veut dire que dans une construction du type :

```
Instr1; Instr2; IF C1 THEN Instr31; IV32; . . . ELSE Instr41; Instr42; . . . END
IF; Instr5; . . . etc.
```

On aura une topologie du type :

```
Page_N : {… Instr1; Instr2; Instr_C1; }, Page_N+1: { Instr31 ; Instr32
; . . . }, Page_N+2: { Instr41; Instr42; . . . },  Page_N+3: { Instr5;
. . . etc. }
```

Il est évident que la topologie dynamique du programme en MC est différente de la topologie du programme en MV qui résulte de sa structure statique. Ce qui est contigu en MV ne l'est pas nécessairement en MC. Par exemple si la fréquence d'utilisation de la partie `ELSE` est beaucoup plus élevée que la partie `THEN`, la topologie dynamique sera Page_N Page_N+2 Page_N+3, avec un trou correspondant à la Page (N+1) qui ne sera pas chargée.

Or l'unité de transfert Disque → MC dépend de la topologie statique sur le disque, et de la contiguïté des pages. La topologie MC ne sera équivalente à la topologie statique que si en MC il y a suffisamment de pages contiguës pour charger les quatre pages N, N+1, N+2, N+3.

D'ou la contrainte fondamentale, pour les algorithmes de gestion de la mémoire virtuelle, et plus généralement de toute « *garbage collection* », de gérer les trous de façon à créer les plus grands trous possibles, lors de la libération d'espace, pour optimiser les transferts MV → MC[10]. Il faut réaliser qu'une MC d'aujourd'hui de 512 Méga-octets, comprend 256 000 pages de 2K chacune qui sont les unités d'échange entre le disque et la MC, ce qui peut faire beaucoup de trous.

10. Cf. l'algorithme de D. Knuth, *Buddy System*, universellement utilisé pour cette gestion, dans *Art of computer programming*, Addison Wesley, Vol. N°1, *Fundamental algorithms*.

Le comportement du programme en MC est fonction de la topologie du programme en MC, i.e. de la fragmentation de la MC (i.e. topologie des trous). La fragmentation de la MC est aléatoire. Elle dépend du nombre de programmes en compétition pour être chargés en MC, des règles de priorité adoptées par le système d'exploitation (par exemple la règle LRU) qui ne sont pas nécessairement les mêmes que celles fixées par l'administrateur système, et de celles résultant de la programmation elle-même. Ces différences, qui se traduisent par un nombre d'entrées-sorties aléatoires pour exécuter le programme, vont se traduire pour l'usager, et les autres programmes avec lesquels il interagit, par un temps de réponse aléatoire qui peut perturber le bon fonctionnement de l'ensemble du système. Si la mémoire est surchargée par une abondance de programmes en MC qui tous sont en compétition pour être présents en mémoire, on peut concevoir que le seul programme qui aura la priorité absolue est le gestionnaire de MV. Dans ce cas, le système ne travaille plus que pour lui-même, correspondant à une situation de saturation (i.e. phénomène dit de « *thrashing* »), qui effondre les performances globales de la plate-forme. C'est une logique à seuil, spectaculaire par ses effets, qui surprend toujours le programmeur néophyte lorsqu'elle se manifeste. Avec les architectures distribuées et les applications en clients-serveurs réparties, la situation empire.

Dans un univers statique, comme celui des applications *batch* des années 70 – 80, ou tout est prévisible (ou presque) car tout est déclaré statiquement, y compris si la ressource n'est pas utilisée dynamiquement pour telle ou telle raison locale à une exécution particulière, le comportement moyen du système reste prévisible, i.e. déterministe. Dans un univers où tout devient dynamique, le système ne peut prendre des décisions intelligentes que s'il est régulé par le programmeur, l'administrateur du SI, et par défaut l'administrateur de l'exploitation (cf. les approches ITIL et l'*Autonomic Computing* d'IBM) qui va privilégier le contrat de service (*Service Level Agreement*, SLA) ; le comportement global du système devient non déterministe, avec possibilité de régime chaotique, au détriment du contrat de service.

Toute entrée-sortie réseau, en plus des entrées-sorties disques, crée des zones de travail pour lesquelles il va falloir allouer des pages de MC qui seront verrouillées, et non pas relogeables. C'est cette fois la structure physique de la MC qui est perturbée, d'où l'impossibilité progressive pour le SE de récupérer la mémoire par le gestionnaire de MV. Cette situation correspond à une fuite de ressource mémoire qui va porter préjudice à tout l'écosystème qui dépend de la disponibilité de ces ressources.

La gestion des ressources, au sens large, est le phénomène fondamental que le programmeur doit impérativement contrôler. L'ouverture d'un fichier ou d'une session de communication verrouille des ressources en MC. Même si le programmeur adopte une stricte discipline de programmation, i.e. toute ressource allouée doit être explicitement restituée, il n'est pas à l'abri d'un arrêt de son programme provoqué par le système d'exploitation. Le nettoyage des ressources allouées est alors à la charge du système d'exploitation qui peut se trouver dans des situations où il n'a pas de critère de décision ; dans ce cas, il ne libère pas la ressource, et la performance de la plate-forme se dégrade progressivement.

Pour garantir durablement le bon fonctionnement global du système, le programmeur doit adopter une stricte discipline de programmation qui échappe quasiment à toute vérification effectuée par des outils de programmation, car ceux-ci ignorent la sémantique de la transformation et ne considèrent que la syntaxe du programme ; c'est la difficulté majeure des outils de tests. Le respect de ces conventions ne peut être vérifié que par des revues et des inspections de code faites par des pairs connaissant la logique métier. Si ce n'est pas fait, il est certain que le code comportera de nombreuses anomalies dont l'effet ne se manifestera que progressivement lors de l'exécution des programmes. Ceci est particulièrement préjudiciable lorsque le code est fabriqué par des sous-traitants, et a fortiori par des sous-traitants *off-shores*. De telles anomalies sont extrêmement difficiles à localiser a posteriori, d'où une nouvelle règle d'architecture.

Règle de construction

L'architecte avec l'aide de l'équipe en charge de l'assurance qualité, doit s'assurer que les règles et conventions de programmation ont été effectivement respectées, lors de la livraison de l'intégrat (ce doit être un élément du contrat). Toutes les ressources manipulées par l'intégrat doivent être explicitées et gérées en configuration. Toutes les ressources doivent être gérées sous le contrôle d'un pilote (i.e. driver) afin de gérer la chronologie des ordres CRUDE afférents à ces ressources, et les tests diagnostiques (OLTD) indispensables au dépannage en cas d'anomalie.

Ces règles font partie du référentiel d'architecture ; elles doivent être appliquées par tous les acteurs, y compris les mainteneurs et les exploitants qui doivent en connaître la raison d'être pour ne pas les invalider lors des installations ou lors des corrections.

La gestion des ε qui sont liés à la nature du phénomène de gestion des ressources et des incertitudes inhérentes à la réalité physique, nécessite la mise en place d'une fonction de surveillance qui dépend totalement de l'écosystème. Dans une logique projet, c'est du code très difficile à développer et à maintenir (Type E du modèle COCOMO), il est donc prudent d'en minimiser le volume.

Le contrôle des enchaînements se dédouble en enchaînements prévisibles, sous contrôle du pilote P1, conformément à la logique nominale des enchaînements, et en enchaînements imprévisibles, sous contrôle du pilote P2, conformément aux aléas de l'environnement (mécanisme de supervision) qui seront détaillés dans la partie 3, Architecture fonctionnelle. Dans un monde parfait, le mécanisme de surveillance serait vide.

Le schéma de principe de ce dédoublement est le suivant (figure 3.5) :

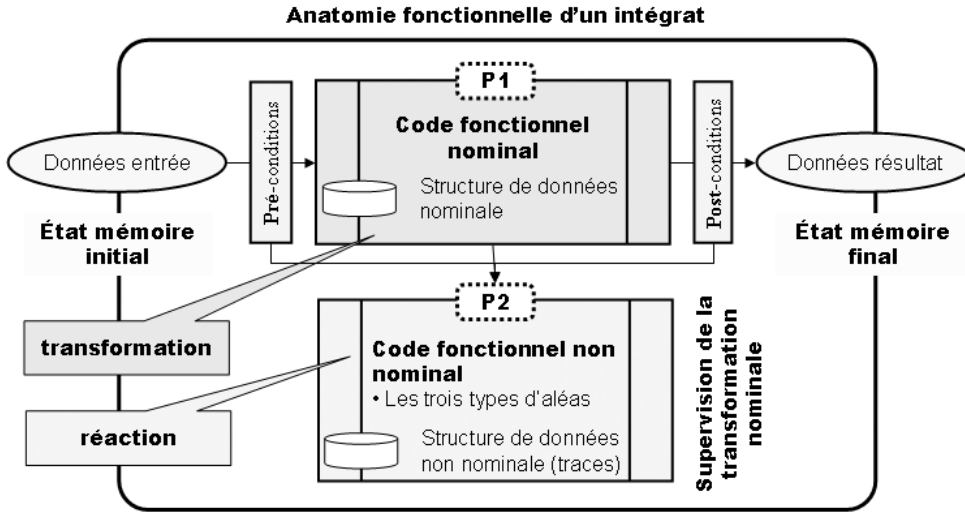


Figure 3.5 - Enchaînement nominal et enchaînement non nominal

Si l'interaction code fonctionnel nominal et non nominal n'est pas organisée dans l'architecture initiale, la sûreté de fonctionnement est durablement compromise. La programmation correspondante est enchevêtrée de façon inextricable. C'est le principe de l'architecture testable qui sera analysée en détail dans les parties 3 et 4.

En terme de simplicité/complexité, on peut concevoir que l'enchevêtrement des deux composantes augmente significativement la complexité de l'ensemble.

Le phénomène d'usure est dégénératif. La dégradation du contexte est comme l'entropie d'un système physique : elle ne fait que croître. Seul un arrêt de la plate-forme et une remise à l'état initial sont susceptibles de restaurer un état nominal propre. Pour cela, il est indispensable d'arrêter la plate-forme d'exécution temporairement. Si pour des raisons de continuité de service cela est impossible, il faut basculer l'exploitation sur une plate-forme de secours, afin de procéder aux réinitialisations, puis re-basculer vers la plate-forme « nettoyée ».

3.3 CONTRÔLES ASSOCIÉS AUX DÉFAILLANCES – SYSTÈME DE SURVEILLANCE – ADMINISTRATION

En terme de ressources R consommées par les programmes on peut dresser le bilan comme suit :

- Une première partie des ressources sert aux transformations effectuées dans le cas nominal où tout va pour le mieux dans le meilleur des mondes.

- Une seconde partie va être consommée par la gestion des aléas d'exécution qui sont contrôlables (compensation par le programme et/ou par le système d'exploitation). C'est ce qui correspond aux 1 de la figure 3.3.
- Une troisième partie correspond aux incertitudes qui ne sont que de simples observables. La dépense correspond à ce qu'il faut conserver comme traces et historiques permettant un diagnostic ultérieur.
- Une quatrième partie correspond aux ressources nécessaires à la gestion des bascules et des réinitialisations.

Cette dernière partie mettra en jeu des fonctions d'exploitation automatisées ou assistées par un opérateur qu'il faudra former de façon ad hoc. Le problème pour l'architecte n'est pas de savoir s'il doit se préoccuper de tout cela, c'est indispensable, mais de savoir quelle quantité de ressource R il lui faut garantir pour que le contrat de service aux usagers soit respecté. La définition du contenu de ce contrat en tant que besoin est un élément indispensable de l'expression de besoin et de l'ingénierie des exigences, afin de définir une architecture fonctionnelle qui en soit la traduction fidèle.

Ce bilan peut être présenté comme suit :

R = R0 (Nominal) + R1 [Aléas contrôlables] + R2 [Aléas observables] + R3 [Supervision et Surveillance]

Si R1, R2, R3 sont faibles ou nuls, la disponibilité ne pourra pas excéder telle ou telle valeur, constatée a posteriori.

Si la disponibilité doit être garantie à telle valeur, il faut allouer des ressources R correspondant au contrôle, à l'observation et à la supervision que cette garantie impose. Ceci sera analysé au chapitre 13, Disponibilité.

4

Représentations de l'architecture – Symboles architecturaux – Diagrammes d'architecture

4.1 INTRODUCTION – DIFFÉRENTES VUES DE L'ARCHITECTURE

Il n'est pas question dans cet ouvrage de décrire en détail les notations et langages disponibles aujourd'hui pour représenter les différentes vues de l'architecture. Il est cependant fondamental de s'interroger sur la nature et le rôle de ces différentes représentations du point de vue des acteurs qui en ont l'usage. Le réseau d'acteurs directement ou indirectement concernés par l'architecture est aujourd'hui complexe, i.e. nombreux et divers. Nous en donnons un aperçu au moyen de trois visions, selon des domaines projet concernés.

4.1.1 La vue acquisition

Le détail du processus d'acquisition est présenté dans différents standards comme par exemple IEEE 1220. Nous ne nous servirons ici que de la partie sous-traitance, en distinguant deux catégories de sous-traitants :

1. ceux qui livrent de la programmation,
2. ceux qui livrent des boîtes noires dont seules les interfaces sont connues de l'intégrateur MOE (par exemple : des équipements sur lesquels les logiciels seront intégrés, des progiciels comme SAP, etc.).

La situation est décrite par le schéma 4.1.

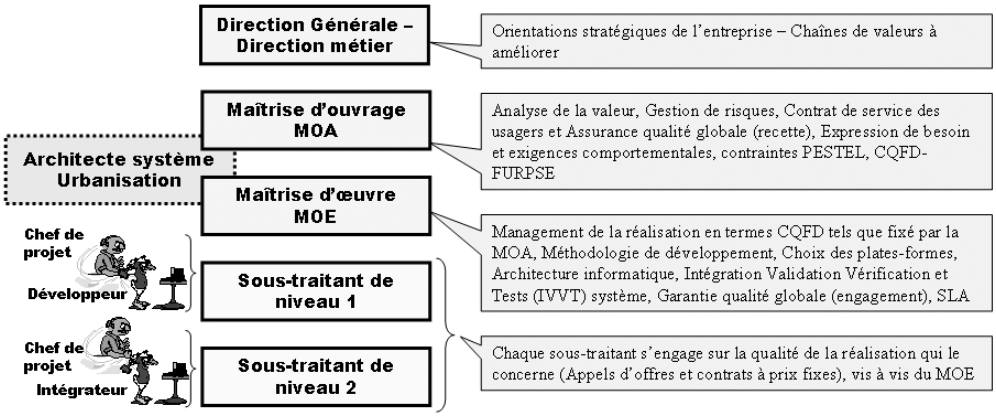


Figure 4.1 - Vue de l'architecture pour les sous-traitants dans le processus d'acquisition

Chacun des sous-traitants a ses propres chefs de projets et équipes de développement qui, pour agir efficacement, ont besoin de connaître la partie de l'architecture qui les concerne et la vue d'ensemble du système qui pour eux est une contrainte à expliciter.

Réciproquement, le MOE doit connaître ce que font les sous-traitants, avec une réserve pour les acquisitions boîtes noires, pour intégrer correctement l'ensemble et garantir le contrat de service. Dans le cas d'une acquisition boîte noire, le sous-traitant peut faire jouer des clauses de propriété industrielle et ne communiquer aucune information sur l'organisation interne de la boîte noire, sauf ce qui est strictement nécessaire à la garantie de service à laquelle il s'est engagé par contrat vis-à-vis du MOE. La connaissance réciproque est le juge de paix en cas de litige avec les sous-traitants. En cas de développement offshore, cela peut devenir compliqué.

4.1.2 La vue engineering

Cette vue concerne directement les acteurs engineering indépendamment des sous-traitances éventuelles. Là encore, le détail du processus d'ingénierie peut être trouvé dans les normes comme ISO 12207.

L'acteur principal, du point de vue de l'engineering est un couple architecte-chef de projet qui interagit avec l'ensemble des acteurs concernés, comme le montre le schéma 4.2.

Le couple architecte-chef de projet est lui-même une entité complexe comme on le verra ci-après. La matérialisation du travail de ce couple est la spécification du système et le référentiel, autour desquels tout le travail effectué au sein des projets va se synchroniser. La bonne compréhension par les acteurs concernés est un facteur qua-

lité fondamental et en conséquence une propriété impérative des représentations adoptées pour décrire l'architecture et le référentiel système.

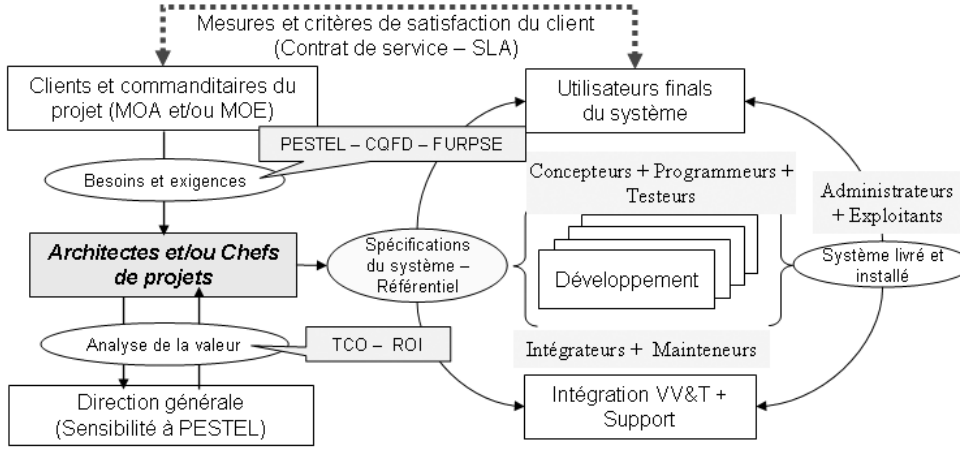


Figure 4.2 - Vue de l'architecture par les acteurs engineering

4.1.3 La vue projet

Comme indiqué dans l'une de nos définitions de l'architecture, la bonne compréhension de l'architecture par les acteurs projets est le véritable critère de terminaison du processus de conception du système. Le schéma 4.3 présente un aperçu simplifié du réseau d'acteurs concernés par cette compréhension.

Comme on le voit sur le schéma, cela fait beaucoup de monde, de culture très variée, mais il est impératif que chacun ait une perception cohérente de ce qu'il doit faire par rapport à la finalité recherchée, faute de quoi il est inéluctable que le chaos s'installe. Compte tenu du nombre d'acteurs concernés, il est impossible de laisser les interactions s'établir au gré des évènements rencontrés. La cellule [architecte système/urbanisation] est l'organe central de coordination des actions entreprises au sein des projets, pour la satisfaction des besoins des clients et usagers du système.

4.1.4 Exigences concernant la représentation de l'architecture

Face à cette diversité de point de vue, on peut considérer qu'il y a deux modalités de représentation de l'architecture.

Modalités N°1 : Représentations et notations pour faire comprendre l'architecture

Il y a de nombreux acteurs dont le seul besoin est de comprendre qualitativement de quoi il s'agit. Dans ce cas, la notation fait largement appel à l'intuition que ces acteurs ont du système. La sémantique est implicite ; seuls les symboles graphiques utilisés font l'objet de conventions. Les notations SADT/SART ou MERISE sont un bon exemple de ce type de besoin.

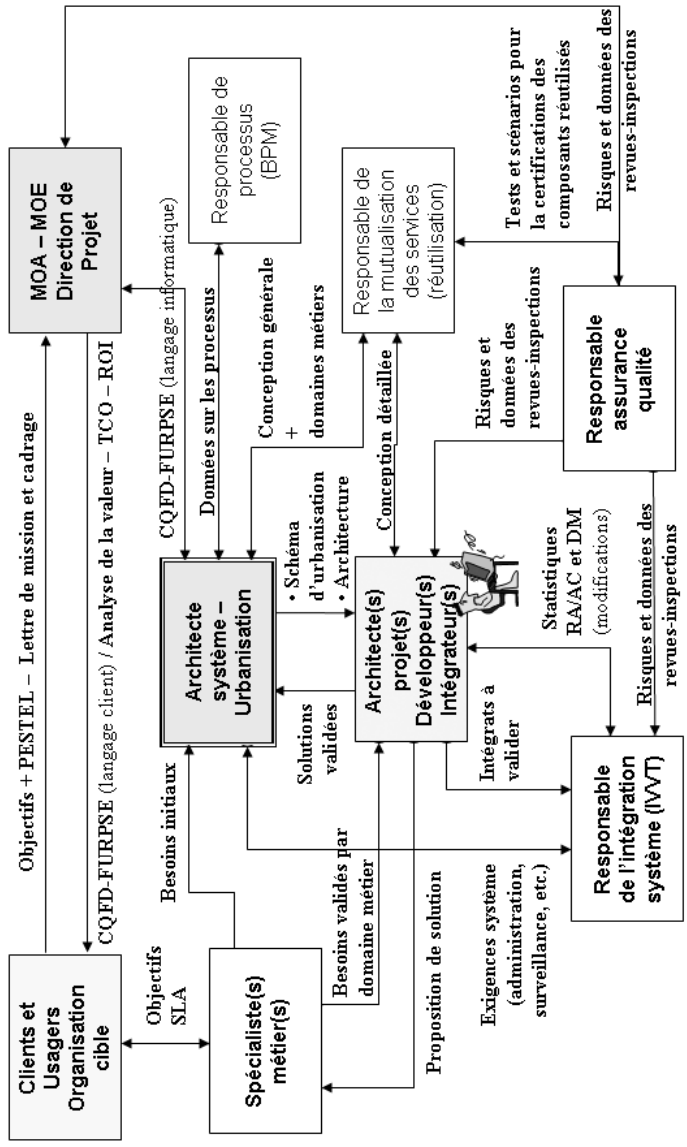


Figure 4.3 - Vue de l'architecture par les acteurs de l'écosystème projet

Modalités N°2 : Représentations et notations pour générer/construire tout ou partie de certaines entités architecturales.

Ce type de représentation est spécifique aux acteurs de l'ingénierie qui contribuent à la fabrication du système : ceux qui réalisent ou modifient le code source du système, ceux qui fabriquent les tests de toute nature utilisés pour valider le système, ceux qui fabriquent la documentation pour les exploitants et le support du système.

Dans ce cas, la notation doit être un vrai langage avec une syntaxe et une sémantique rigoureusement formalisées. La définition de la sémantique du langage fait appel à une machine abstraite indépendante de toute plate-forme réelle dont les structures (données, opérations, contrôle) pourront être traduites dans le ou les langages des plates-formes sur lesquelles le système pourra être installé.

Dans cette famille de notations nous trouvons des formalismes comme celui du modèle ERA Entités – Relations – Attributs qui permet de générer les schémas des bases de données, ou des formalismes issus de la théorie des automates qui fondent des langages comme SDL, utilisé pour spécifier les protocoles de communications.

Le cas du langage UML est particulier car il peut être utilisé simplement pour faire comprendre, par exemple au moyen des cas d'emploi, ou plus précisément pour générer du code source dans des approches du type MDA. Dans sa version 2, le langage UML reste incomplet du point de vue de la sémantique, ce qui pose un vrai problème pour les outils de transformation de modèles pour lesquels une sémantique précise doit être fournie. Avec une sémantique ad hoc, on pourrait parler d'UML exécutable.

Pour ce qui concerne la vue projet, la représentation de l'architecture doit permettre au chef de projet, soit MOA, soit MOE, d'effectuer les études d'estimation des coûts de la réalisation. Quel que soit le modèle d'estimation utilisé, COCOMO ou Point de Fonction, la connaissance de l'architecture est un paramètre indispensable du modèle d'estimation. Un défaut de connaissance de l'architecture se traduira, dans l'application du modèle, par un niveau d'incertitude élevé quant à la précision de l'estimation (NB : comme son nom l'indique, l'incertitude est un risque qu'il faut lever).

Pour ce qui nous concerne, nous pensons qu'il est vain de rechercher un langage unique qui puisse satisfaire des besoins d'une aussi large variété d'acteurs.

La MOA, la DG, les utilisateurs ont besoin d'une représentation intuitive proche du langage naturel. La seule chose qu'il faut exiger, c'est une rigueur terminologique indispensable pour qu'ils puissent se comprendre, matérialisée par un glossaire de termes. Les données qui sont la matière première des transformations opérées peuvent être décrites de façon précise avec le langage XML qui est de type grammatical, que tout le monde peut comprendre avec un effort minime. Les événements à considérer sont ceux du monde métier et de l'écosystème PESTEL, mais en y intégrant ceux associés à des situations de défaillances ou d'erreurs qui existent indépendamment de l'informatique. Le MOA doit disposer de l'information nécessaire pour « vendre » le projet auprès des commanditaires ; il est le maître de l'état EB/EC des entités architecturales et des cas d'emploi associés.

Les architectes/urbanistes ont besoin d'une grande rigueur pour tout ce qui traite de la sémantique et des interfaces qui vont constituer les contrats de réalisation. Tout ce qui est nécessaire aux analyses de CQFD/FURPSE doit être modélisé à ce niveau. Les projets à lancer doivent pouvoir être définis avec précision (cf. la règle de l'architecture pour les projets). La logique globale doit pouvoir être appréhendée à ce niveau ainsi que les capacités d'évolution du système sur le moyen-long terme

(exigence pour le TCO, vis-à-vis du MOA et de la DG). La machine abstraite qui matérialise l'architecture à ce niveau doit être suffisamment précise pour « exécuter » les cas d'emploi formulés par la MOA.

Les concepteurs/programmeurs/testeurs, en particulier en cas de sous-traitance du développement, ont besoin d'un formalisme permettant un pilotage au plus près de la programmation et des tests associés, y compris les tests d'intégration des applications et de pré-recette système. Toute la complexité informatique doit être appréhendée à ce niveau (architecture testable, performance, maintenabilité, évolutivité).

Le vrai besoin est plus dans l'articulation de ces trois formalismes que dans l'existence hypothétique d'un noyau commun.

4.2 LES PREMIÈRES NOTATIONS – LE MONDE DE LA PROGRAMMATION STRUCTURÉE

Le tout premier système de notations utilisé par les architectes programmeurs a été celui des organigrammes, y compris par von Neumann lui-même, dans les années 50. Ce type de notation met l'accent sur la logique d'enchaînement des opérations effectuées par la machine.

Très rapidement, dès les années 60-70, le mécanisme d'enchaînement se raffine avec des notations comme celles de MERISE (Modèle Conceptuel de Traitement) ou de SADT/SART[1] où l'on voit apparaître explicitement les notions de modules, tâches et processus qui concourent à la transformation des données représentant l'information traitée par le système.

L'ouvrage le plus représentatif de cette période est : T. Demarco, *Structured analysis and system specification*, Yourdon Press, 1re édition 1978, avec des diagrammes de flots de données (DFD – *Data Flow Diagram*) qui sont construits à l'aide de quatre structures de base : les processus, les fichiers (i.e. la mémoire), les sources et puits de données, les enchaînements.

Le mérite de ce type de représentations est qu'elles sont très intuitives et permettent un dialogue informel entre les acteurs usagers, MOA, MOE, architecte-urbaniste dans la phase de définition du système : tout le monde comprend. Ce très grand avantage est également son inconvénient car en allant vers la réalisation, il faudra plus de rigueur, avant d'attaquer la programmation (ou l'acquisition d'un progiciel) proprement dite.

1. Cf. P. Ward, S. Mellor, *Structured development for real-time systems*, Yourdon Press, 3 Vol., 1985.

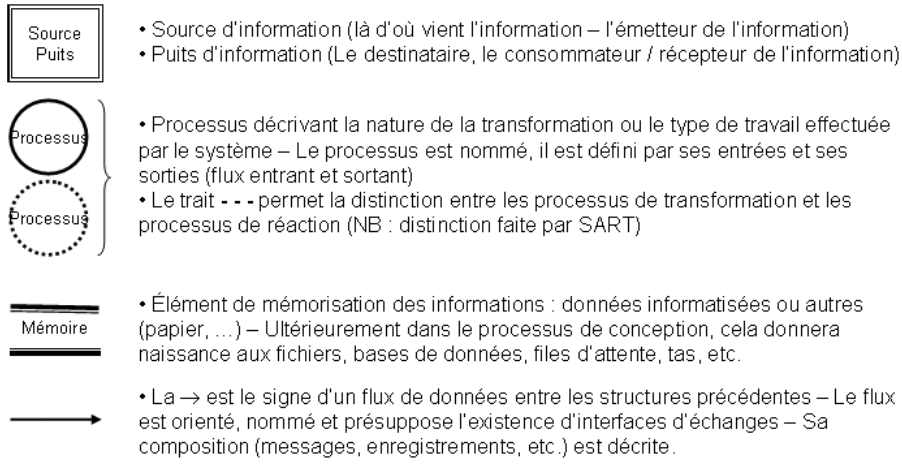


Figure 4.4 - Structures de base des DFD

Voici un exemple simple de DFD décrivant un automate bancaire rudimentaire (figure 4.5).

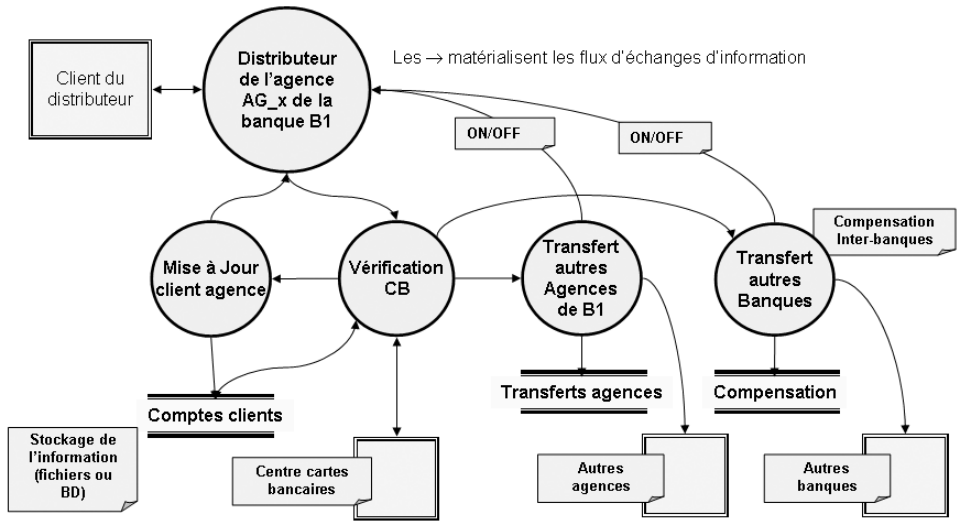


Figure 4.5 - Exemple de DFD – Un automate bancaire

Dans un DFD, si l'on ne retient que les sources et puits d'information, en faisant abstraction des processus constitutifs de l'application on obtient un diagramme de contexte, également appelé diagramme de collaboration. Sources et puits peuvent servir à représenter les acteurs interagissant avec le système, c'est-à-dire les ports du système.

Le point culminant de cet effort sera le Standard IEEE *Functional modeling language – Syntax and semantics for IDEFO*, IEEE Std 1320.1 et 1320.2, dernière version 1998.

Dans tous ces formalismes, la structure centrale est la fonction, définie comme ci-dessous (figure 4.6), que l'on peut appliquer à elle-même de façon récursive.

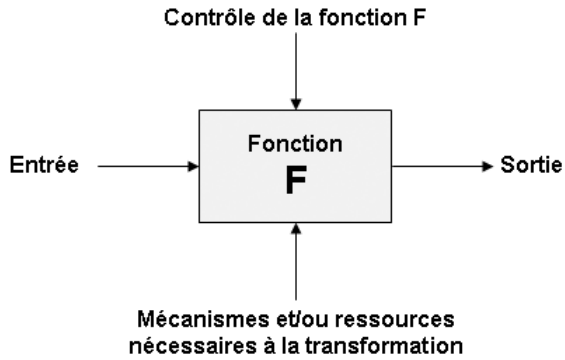


Figure 4.6 - Fonction selon IDEF

La fonction, au sens IDEF, est définie comme une transformation des entrées en sorties (résultat de la fonction) au moyen d'un mécanisme ad hoc et soumise à certains contrôles (les ressources pour effectuer la transformation spécifiée).

Le mécanisme de décomposition récursive de la fonction initiale en sous-fonctions plus simples est naturel pour quiconque a fait des études scientifiques et des mathématiques de niveau élémentaire.

Le principe de décomposition récursive en fonction de plus en plus simple s'arrête lorsque la fonction finale correspond à une instruction ou à un service de la machine étendue sur laquelle va s'exécuter le programme correspondant. Notons que la machine correspondante peut être de bas niveau, comme dans la programmation de micro-contrôleurs, ou de très haut niveau comme dans la manipulation de données dans une base de données ou comme dans une architecture orientée services.

Beaucoup de reproches ont été adressés à ces notations fonctionnelles, avec parfois une certaine mauvaise foi. Elles mettent l'accent sur la dynamique des enchaînements quand tout se passe bien dans le système. C'est un cas limite de fonctionnement idéal, hors de toute défaillance. Il est clair que si l'on veut signifier autre chose, la notation n'est plus adaptée. Leur immense mérite est qu'elles sont comprises par à peu près tout le monde qui a du bon sens et un minimum de culture scientifique (cf. application du principe KISS). Pour décrire les structures de données associées à ces DFD, la notation du modèle ERA est supposée connue. Là encore, c'est une notation intuitive (jusqu'à un certain niveau) que tout le monde comprend, fondée sur une notion d'ensemble naïve (au sens des ouvrages de E.

Kamke, *Théorie des ensembles*, ou de P. Halmos, *Introduction à la théorie des ensembles*, dont le titre anglais est *Naive set theor*y ; ces deux ouvrages sont disponibles en réimpression chez J. Gabay) et de la logique classique.

Des notations comme celles utilisées pour les modèles conceptuels de données (MCD) de MERISE ou les schémas de C. Bachman développés pour les bases de données du modèle CODASYL, sont directement inspirées du modèle ERA.

Pour représenter la composante réactive, la notation de référence est issue de la théorie des automates et des systèmes à états-transitions, connue dès les années 50, soit sous forme d'automates à états finis, soit sous forme de grammaires équivalentes comme on en verra un exemple au chapitre 6, avec les compilateurs. Avec ces notations, nous ne sommes plus dans le domaine de l'intuition ; pour les utiliser correctement de solides connaissances en mathématiques discrètes sont indispensable. Néanmoins, la notion d'automates est assez intuitive et facile à comprendre, ce qui n'est pas le cas des notations fonctionnelles, comme le λ-calcul, ou algébriques.

Pour re-situer ces notations dans leur contexte historique et ne pas faire de paralogisme, il faut se rappeler que jusque dans les années 80, l'interface homme-ordinateur n'était que textuelle. Tout ce qui nécessitait de vrais dessins devait être fait sur d'autres supports ; les tables traçantes étaient tellement chères et encombrantes que leur diffusion restait limitée aux professionnels du dessin. L'arrivée des écrans bitmap haute définition avec le MacIntosh dans les années 80 et des imprimantes laser allaient bouleverser le paysage et rendre les interfaces graphiques accessibles à tous. De purement textuels, les langages, en particulier ceux utilisés en conception, vont pouvoir devenir graphiques, ce qui est le support de travail habituel des ingénieurs de conception dans les bureaux d'études, et ce depuis déjà quelques siècles (cf. le dicton « un bon schéma vaut mieux qu'un long discours »).

4.3 LES NOTATIONS RÉCENTES – LE MONDE OBJET

Les notations de première génération sont relativement qualitatives, à l'exception du modèle ERA, dont la précision peut être poussée aussi loin que l'on veut . Ces notations servent d'intermédiation entre les acteurs du schéma de la figure 4.5. Elles deviennent très insuffisantes si l'on considère les besoins de l'ingénierie système et logiciel, en particulier avec les problèmes suscités par la maîtrise des architectures distribuées et des clients-serveurs répartis.

- Un premier besoin vient du monde des protocoles de communications, avec un langage phare : SDL.
- Un second besoin vient du besoin de maîtriser l'ingénierie des logiciels de grande taille concomitant avec le langage Ada[2] et surtout les premiers langages objet.

2. Un très bon langage dans sa version 95, mais un énorme échec stratégique, technique et commercial du DOD US qui a également coûté cher aux européens.

Le concept fondateur pour exercer cette maîtrise est celui de Type Abstrait de Données (TAD, ou ADT, *Abstract Data Type* en anglais), avec un langage phare : UML, qui intègre bien d'autres choses que ce qui relève des TAD, stricto sensu.

4.3.1 La notation SDL

Le langage SDL a été défini par le CCITT, devenu depuis l'ITU (*International Telecommunication Union*), pour standardiser et matérialiser le travail de conception des architectes de protocoles. La première version date de 1976, et n'a pas cessé d'évoluer régulièrement depuis. Actuellement, il est question de fusionner ou d'intégrer le langage dans UML, ce qui n'est pas forcément une bonne idée.

Il n'est pas question de présenter ici le langage SDL, il y a d'excellents ouvrages sur le sujet, mais d'en comprendre les principes et les besoins auxquels le langage répondait. Pour cela il faut se rappeler les principes de fonctionnement d'une ligne de télécommunication ; là encore les ouvrages spécialisés peuvent être consultés, pour plus de détail.

Dans son principe, la problématique des architectes de protocoles est relativement simple. Les schémas qui suivent donnent un aperçu des problèmes fondamentaux que l'architecte de protocoles doit résoudre, et pour cela modéliser, avant de développer. Le schéma 4.7 montre quelques-uns des organes de la machine qui vont être sollicités dans une opération de communication.

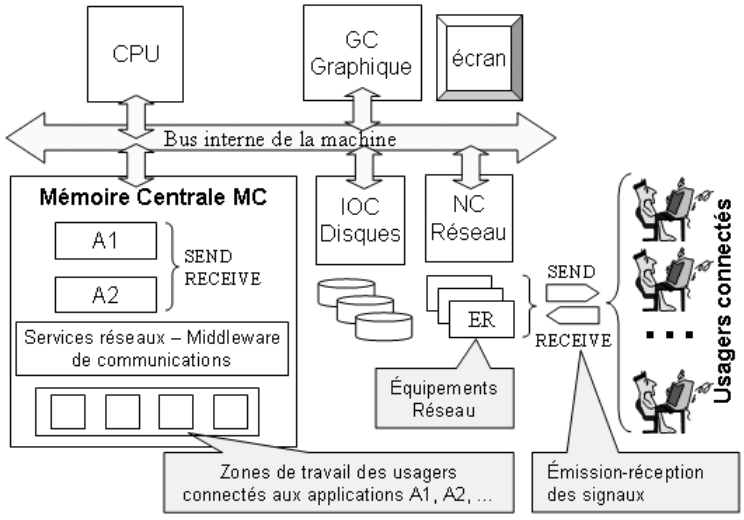


Figure 4.7 - Emission et réception des données dans une application

L'application A1 peut être sollicitée par un ou plusieurs usagers pour lesquels il va falloir créer dynamiquement dans la mémoire de la machine serveur des zones de travail qui serviront à enregistrer les données d'entrée et de sortie des transformations

effectuées par l'applications pendant la durée de l'interaction. L'application communique avec son environnement via des ordres du type SEND/RECEIVE qui sont en fait des ordres d'entrées et de sorties vers les postes de travail des usagers. Elle reçoit et émet des signaux pour être notifiée de l'arrivée d'un message dans sa zone de travail ou au contraire notifier à un service de communication que telle ou telle donnée est prête à être expédiée vers son destinataire.

Pour effectuer ces différentes opérations, le système d'exploitation, les services réseau et les *middleware* de communications vont créer des processus de traitement et des zones de travail spécifiques aux processus de communications, pour acheminer ces différentes données vers le contrôleur E/S qui gère les équipements réseau.

Le schéma 4.8 montre une partie de la logique à mettre en œuvre.

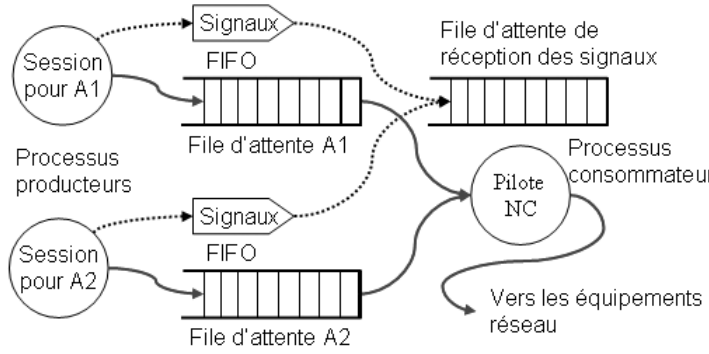


Figure 4.8 - Processus et files d'attentes nécessaires à la communication

Les ordres SEND déposent des messages dans les files d'attentes (généralement *First-In First-Out*, FIFO). Le processus consommateur de la file d'attente, pour acheminement des messages vers le contrôleur, est notifié par un signal du dépôt d'un nouveau message. Les files d'attente se vident et se remplissent en fonction des opérations effectuées par l'usager et de la capacité du réseau (bande passante exprimée en bit/seconde, perturbations, etc.) à acheminer les messages plus ou moins rapidement. Tous les mécanismes sont asynchrones et requièrent une gestion mémoire rigoureuse car tout est dynamique et largement aléatoire même si quelques régularités statistiques sont observées.

Si l'on regarde du côté des équipements réseau, il faudra gérer la transmission d'information et les évènements associés.

Les équipements d'émission et de réception assurent divers transcodages, selon la nature des codes utilisés. L'accès à la ligne est commandé via des connecteurs normalisés qui assurent la liaison avec l'équipement gérant la ligne (typiquement, un modem). L'horloge permet de gérer la durée des transmissions en fonction des caractéristiques physiques de la ligne. Des anomalies dans la durée peuvent être un signe

de défaillance de la ligne qu'il faut alors déclarer en panne ; auquel cas, il faut que l'application cesse d'envoyer des messages qui ne feront que saturer les files d'attentes intermédiaires, occasionnant ainsi d'autres pannes du côté ordinateur.

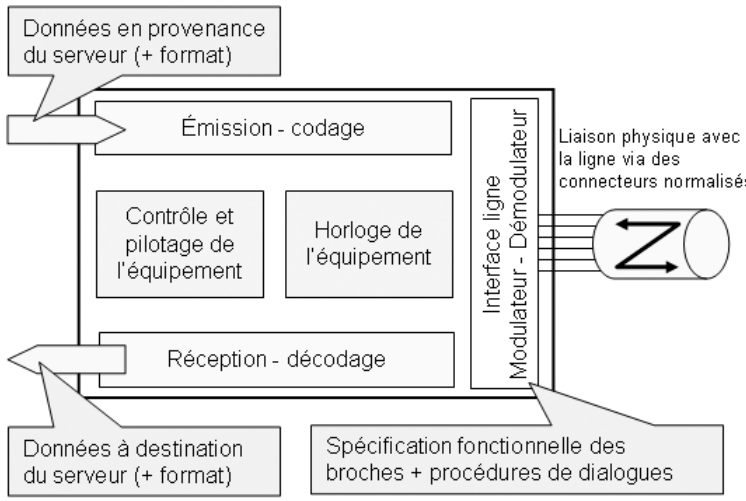


Figure 4.9 - Interfaces génériques pour la transmission de données

Le contrôle des temporisations est fondamental pour optimiser le débit de la ligne. Les temps de latence dépendent de phénomènes physiques liés à la structure du circuit électrique. Si la durée est trop longue, la ligne perd en efficacité ; si elle est trop courte, il y aura des défauts au niveau du signal qui se traduiront par des erreurs de transmission (corrigés à l'aide de codes correcteurs d'erreurs) ce qui est une autre façon de perdre en efficacité.

Pour visualiser les dialogues entre équipement et/ou service de communication, les architectes de protocoles ont inventé des diagrammes particuliers : les MSC (*Message Sequence Chart*) que l'on trouve dans tous les ouvrages traitant des réseaux et des communications.

La commande et le contrôle des équipements et/ou des services se fait à l'aide de primitives de services (Interactions atomiques munies de paramètres entre l'usager du service et le fournisseur du service, selon la définition du CCITT/UIT) qui doivent s'enchaîner dans un certain ordre dépendant des réponses fournies. Le parallélisme est possible, il y a nécessité de prévoir des points de synchronisation et de contrôle de cohérence de la transmission.

En résumé, l'architecte de protocoles est confronté à un certain nombre de problèmes difficiles comme :

- La création d'entités informatiques (processus, zones de mémoire, réservations de ressource, etc.) totalement dépendantes des requêtes des usagers, donc largement imprévisibles.

- Le contrôle pas à pas d'opérations effectuées à l'aide des primitives pilotant le fonctionnement des équipements, d'où les diagrammes d'états.
- Le recueil de signaux dont certains ultra-prioritaires.
- Le parallélisme et la synchronisation de certaines opérations.
- Le transcodage de format de données en fonction des codes utilisés tout au long de la chaîne de transmission, d'où une norme permettant la spécification rigoureuse de ces transcodages (cf. Recommandation X208, du CCITT/IUT, *ASN.1 – Abstract Syntax Notation 1*).
- La surveillance des équipements permettant les tests et diagnostiques dynamiques, le paramétrage, la reconfiguration, les performances et le contrat de service, etc.

Le système de notation utile à l'architecte de protocole met l'accent sur ce qui est difficile, et ne s'occupe pas de ce qui est trivial, et nécessairement connu de quiconque, et est confronté au problème de la spécification et de la mise en œuvre de protocole.

Pour ne mentionner qu'un seul problème dont on a déjà parlé, celui de la gestion dynamique des ressources, il doit être clair que la préoccupation fondamentale de l'architecte est la libération des ressources partagées entre les usagers, et ceci quelle que soit la situation système (nominale, interrompue, suspendue, etc.) de l'usager. La moindre faille dans la gestion de la libération des ressources se traduira par des fuites de ressources qui rendront le serveur progressivement inexploitable. Les pannes correspondantes sont les plus difficiles à diagnostiquer car non reproductibles.

Si la notation ne met pas l'accent sur les vraies difficultés, elle ne sert à rien, sinon à produire de la simple paraphrase de programmes, ce qui a comme effet pervers d'augmenter la complexité et donc le coût d'ingénierie.

4.3.2 La notation UML

Le désir d'unification et d'harmonisation des différentes notations utilisées depuis une trentaine d'années est un effort louable à mettre au crédit de l'OMG et des experts qui ont essayé de mettre de l'ordre dans le fatras des symbolismes utilisés. La tentative n'est pas sans risque, d'autant plus que l'effet comité, inhérent aux organismes de standardisation, va malheureusement jouer dans le sens du « toujours plus ».

Le manuel de référence de la version 2.0 totalise 700 pages (*The Unified Modeling Language Reference Manual*, Addison Wesley, 2005). Elle intègre, au niveau graphique, un certain nombre de notions du langage SDL, mais pas le langage lui-même ; elle permet également de faire figurer des contraintes associées à certains symboles, de façon textuelle, en les plaçant entre accolades {texte de la contrainte}, sans privilégier un langage particulier.

Dans sa version 2.0, le langage est d'une très grande richesse, ce qui amène quelques questions de fond, qui ne sont d'ailleurs pas spécifiques à UML.

Question N°1 : UML pour qui ?

Si l'on reprend les acteurs principaux de l'ingénierie (cf. figure 4.2), la bonne question est : de quoi ont-ils besoin ? et la seconde : pour quoi faire ? En d'autres termes : peut-on communiquer avec UML comme on le ferait avec des notations DFD, IDEF ou ERA (ce sont juste des exemples, sans aucune arrière-pensée) ; quelle est la précision requise dans la communication pour que l'action des acteurs qui communiquent soit efficace ?

Sans rentrer dans des considérations sémiologiques hors de propos, communiquer requiert un langage commun, une culture et des valeurs communes qui vont donner le même sens aux actions entreprises. Si l'on n'a pas de langage commun, l'un des acteurs doit parler la langue de l'autre avec suffisamment de précision pour effectuer une traduction fidèle.

Pour un DSI qui se pose la question de savoir s'il a fait le bon investissement en TCO et que le ROI sera là, ou si les usagers seront satisfaits et que la conduite du changement ne sera pas trop douloureuse, ou s'il pourra mettre le système en TMA sans trop de difficultés, la réponse est évidente : il ne trouvera quasiment rien dans UML dans le cas de la TMA. Si le développement est familier d'UML et de ce fait travaille en UML, sans que l'équipe de TMA le soit, c'est un problème de plus pour la direction MOA/MOE du projet !

Pour un usager face à de nouvelles IHM, on peut concevoir que la cinématique des affichages soit expliquée avec des diagrammes d'états qui sont bien adaptés à ce problème ; encore faut-il que l'explication ne fasse appel à aucune considération d'implémentation.

Pour les sous-traitants intervenant dans l'ingénierie, c'est utile s'ils sont au même indice de connaissance que le MOE ; si ce n'est pas le cas, c'est un risque au minimum temporaire s'il y a des formations organisées pour les sous-traitants.

Le cas de la relation MOA / MOE est intéressant à examiner car ce sont deux acteurs importants qui doivent travailler en bonne coopération mais dont les domaines de préoccupation et la vue dans la durée du projet sont différents.

Comme le montre la figure 4.1, l'intersection entre le monde MOA et le monde MOE est l'architecture système et l'urbanisation. C'est le lieu où se fait la traduction (transduction serait préférable, on y reviendra dans la partie 3, Architecture fonctionnelle logique) entre l'architecture du SI au niveau de l'entreprise et des métiers qui en font la valeur, et entre l'architecture au sens informatique du terme (architecture logicielle + architecture plates-formes + architecture réseaux – i.e. le maillage et la collaboration des acteurs – au sens NCS, *Network Centric Systems*).

Pour le versant MOE, tout est bon à prendre dans UML, ou presque (à la réserve près qui est la taille du manuel de référence).

Pour le versant MOA, rien n'est moins sûr, si l'on considère les missions premières de la MOA :

- Ingénierie de l'expression des besoins et des exigences comportementales du point de vue des différentes catégories d'acteurs qu'il représente : usagers des métiers, utilisateurs exploitants et maintenance, support.
- Qualification et recette du point de vue de ces mêmes acteurs.
- Suivi des aspects CQFD/FURPSE/PESTEL des projets coordonnés qui fondent le TCO et le ROI espérés du système.

En matière d'EB/EC le MOA peut tirer partie de la notion de cas d'emploi (*Use Case*) telle que définie dans UML.

Deux situations peuvent se présenter :

Situation N°1 : les différents usagers sont eux-mêmes familiers, à un certain niveau, du langage UML, et savent manipuler les diagrammes constitutifs des cas d'emploi. Dans ce cas le travail de la MOA consiste à s'assurer de la cohérence des différents cas d'emplois et à en déterminer la priorité, avec des échelles qualitatives du type **U I P** (**U**tile, **I**mportant-**I**ndispensable, **P**rimordial).

Situation N°2 (la plus fréquente) : les usagers ignorent tout de UML. Seules les interviews en langage naturel, avec des diagrammes informels type DFD, et le bon sens de tout un chacun sont partagés. Dans ce cas, tout le travail de définition des cas d'emplois est à faire par le MOA qui doit s'assurer auprès des usagers réels que ce qu'il formalise traduit fidèlement l'intuition des usagers réels. Si ce n'est pas le cas, c'est un risque important pour le projet. Ce niveau étant acquis, on retombe dans la situation N°1.

En matière de suivi CQFD/FURPSE/PESTEL, il s'agit de contrôler la traduction des cas d'emploi issus de l'étape EB/EC en termes de besoins fonctionnels du point de vue de l'ingénierie. Cette correspondance est fondamentale pour le MOA car c'est la seule façon de garantir que la réalisation restera en cohérence avec les besoins et les exigences exprimés par les usagers. Ce suivi nécessite l'harmonisation des systèmes qualité MOA et MOE, y compris celui des sous-traitants éventuels.

En matière de qualification et recette, il est évident que des cas d'emplois partagés vont faciliter l'élaboration des tests correspondants. En conséquence le langage des cas d'emploi améliore le dialogue MOA/MOE.

La question : pour qui ? a comme éléments de réponse la définition d'un noyau de langage commun, avec des enveloppes successives en fonction du rôle de l'acteur dans le processus d'ingénierie. Notons qu'une première tentative a été faite avec un ouvrage comme : *UML distilled*, M. Fowler, Addison Wesley, 2004, dont il serait intéressant d'avoir des retours d'expérience des MOA qui l'auraient pratiquée.

Ce noyau commun, si tant est qu'il soit identifiable, devrait être partagé par tous les acteurs informaticiens quel que soit leur niveau d'implication dans le cycle de vie logiciel et système, un peu comme les mathématiques élémentaires font partie de notre culture commune (arithmétique, équations simples du 1er et 2ème degré, fonctions linéaires, calcul différentiel pour calculer des longueurs et des surfaces de formes géométriques simples, théorie naïve des ensembles et rudiment de logique). Une

bonne expérimentation serait de définir avec les MOA se dont ils pensent avoir besoin pour assurer leur rôle d'intermédiation entre les usagers réels et les informaticiens de la MOE.

Question N°2 : UML pour le quoi et le comment faire ?

La littérature UML, et ce qu'en disent les « évangélistes », peut laisser penser que UML en l'état peut ou pourrait satisfaire le besoin de tous les types de réalisation.

Une analyse de simple bon sens permet de voir qu'il n'en est rien. Si l'on prend comme axe d'analyse la typologie introduite au début du chapitre à partir des constituants que sont les données, les transformations / fonctions / objets-méthodes, les comportements / contrôles et les caractéristiques qualité FURPSE – PESTEL (cf. figure 1.7), on peut déterminer la couverture du langage dans un référentiel dont la structure est la suivante (tableau 4.1) :

Tableau 4.1 - Couverture UML versus besoins des applications

Elément de modélisation	F	U	R	P	S	E	P	E	S	T	E	L
Constituant données												
Constituant opérations												
Constituant contrôle												
Constituant ports (sources, puits)												
Ressources utilisées												

Dans un système à données prépondérantes comme un système d'information au sens classique du terme, l'architecte a deux problèmes fondamentaux sur sa table de travail :

1. l'architecture des données, sur la base des attributs répertoriés qu'il faut regrouper en entités grâce à des critères sémantiques.
2. l'architecture des traitements qu'il faut organiser en transactions de façon à garantir l'intégrité des données pour tous les usagers quoiqu'ils fassent, y compris des erreurs de manipulations (propriétés ACID).

Si le système d'information inter-opère avec d'autres SI de l'entreprise, voire avec les SI des entreprises partenaires (clients, fournisseurs, administrations, etc.) l'architecture des données doit identifier les données susceptibles d'être échangées de façon à élaborer le modèle conceptuel de données pivot permettant de réaliser les connecteurs des différents SI sur l'infrastructure d'intégration de l'entreprise. Une telle architecture nécessite une modélisation de la sémantique des données.

Si le SI évolue vers une architecture orientée services (SOA), il faut rajouter les protocoles d'appels des différents services et la nomenclature de messages et d'événements que cette architecture implique.

Si données et services sont répartis sur une architecture de serveurs distribuée, il est indispensable d'évaluer la performance dès que le nombre de clients dépasse quelques dizaines. Le problème devient primordial quand le nombre de client se chiffre en milliers.

Si l'architecture des données manipule des données réglementaires qui évoluent sur des rythmes législatifs courts (moins de un an), il faut complètement séparer les données et les traitements correspondants pour gérer leur évolution de façon optimale, et ceci sans avoir à reconfigurer l'ensemble des données et des traitements. Ceci peut amener à des découpages données-traitements qui n'ont pas de justification du point de vue de l'usager, mais uniquement du point de vue économique du projet.

De tels SI ont souvent des durées de vie très longues, vingt ou trente ans ne sont pas rares, ce qui pose le problème de la pérennité des modèles et des outils de gestion de ces modèles, d'où la nouvelle question : que faut-il pérenniser ? et la réponse qui est rien moins que simple : l'atelier de gestion, le modèle conceptuel de l'atelier, le méta-modèle.

Une chose est sûre, plus le modèle est complexe, moins la sémantique du langage est précise, et plus le risque de non pérennité est élevé, en application du principe KISS.

À l'opposé des SI, on va trouver les systèmes à contrôle prépondérant qui mettent en œuvre des automatismes. À l'époque des notations SART et Statechart, les systèmes temps réel étaient quasiment les seuls de cette catégorie. Aujourd'hui, avec l'amélioration de la puissance des machines, avec les réseaux à haut débit, le besoin d'interactivité s'est généralisé. La seule différence est la nature des échéances temps réel :

- Pour des équipements, cela reste de l'ordre de la milliseconde, voire moins dans des circonstances très exceptionnelles (missiles à grande dynamique, processus instables comme dans le nucléaire, etc.).
- Pour des acteurs humains, un temps de réponse à la seconde donne un bon confort ergonomique.

La difficulté vient de la nature des services offerts à l'usager et du nombre d'entrées-sorties que le service implique. Or la performance des sous-systèmes d'entrées-sorties n'a pas suivi celle des unités centrales ; la « loi » de Moore ne s'applique pas. Par exemple un service qui nécessite 1 000 entrées-sorties disque à 10 ms chaque, fabrique (en l'absence de parallélisme) un temps de réponse de l'ordre de 10 secondes, donc très mauvais.

Dans le cas des systèmes temps réel classiques les problèmes sont bien connus. Le langage SDL en est un très bon exemple et l'on peut dire que tant que UML n'aura

pas intégré l'intégralité de la notation et surtout sa sémantique, ainsi que d'autres langages qui lui sont étroitement associés comme ASN.1 et TTCN[3] (c'est un langage de test), il ne pourra pas se substituer valablement à SDL.

Si UML réussi à intégrer SDL, ASN.1 et TTCN la question est : pourquoi encombrer le langage avec des notations qui ne sont d'aucun usage pour ceux qui n'ont pas le problème du temps réel ?

Dans le cas de système interactif avec des humains dans la boucle, comme par exemple le travail collaboratif, le besoin d'interactivité entre les acteurs peut être décrit avec des langages de workflow dont BPML est un bon exemple. Il s'agit cette fois de langages émergents, qu'il peut être tentant d'intégrer à UML, avec d'ailleurs un risque supplémentaire par rapport à SDL car la situation est inverse. SDL, et quelques autres langages de même nature, n'ont fait que matérialiser les connaissances des architectes des systèmes temps réel accumulées depuis 20-30 ans. Avec BPML, c'est l'inverse, et une normalisation serait tout à fait prématurée. En particulier le problème de l'enchevêtrement des aspects transactionnels et des IHM est loin d'être résolu, et il n'est même pas évident, compte tenu de la nature sémantique et pragmatique de cet enchevêtrement, qu'il y ait une solution générale indépendante de la nature du problème traité. Des solutions de type boîtes à outils restent envisageables, mais leur déploiement sous la forme de langages spécialisés ad hoc – on parle dans ce cas de DSL, *Domain Specific Language* – nécessite la manipulation de méta-modèles et la connaissance des techniques de méta-compilation (voir partie 2) ce qui rend problématique une large diffusion de ce type de méthode.

Un exemple de notation confuse : l'agrégation en UML

Dans leur effort de se distinguer à tout prix des notations antérieures, les inventeurs d'UML ont créé une grande confusion autour du concept d'attributs agrégés, pourtant intuitivement évident et bien connu des architectes de bases de données. Comme le dit l'un des bons auteurs[4] : « *One of the most frequent sources of confusion in the UML is aggregation and composition* ». Si l'on a la curiosité de se reporter au manuel de référence UML[5] voici les définitions :

- **Agregation** : "*a form of association that specifies a whole-part relationship between an aggregate (a whole) and a constituent part. It is also applicable to attributes and parameters*."
- **Composition** : "*a strong form of aggregation association with strong ownership of parts by the composite and coincident lifetime of parts with the composite. Etc*."

Difficile de faire pire en matière d'obscurité. Aucun de ces ouvrages ne donne les références incontestables de l'origine de ces notions, à savoir : P. Chen, *The entity-relationship model*, ACM Transactions on database systems, Vol 1, N° 1, March 1976,

3. Cf. C. Willcock, *An introduction to TTCN-3*, Wiley and Sons, 2005 ; Normes de l'IUT.
4. M. Fowler, *UML Distilled*, Addison Wesley, 2003.
5. J. Rumbaugh, I. Jacobson, G. Booch, *UML reference manual*, Addison Wesley, 2005.

et T. Teorey, *A logical design methodology for relationa databases using the extended Entity-Relationship model*, ACM Computing Surveys, Vol. 18, N°2, June 1986.

Le modèle ERA a été massivement utilisé dès la fin des années 60 dans des méthodes comme MERISE, Warnier-Orr, etc., dans toutes les bases de données de l'époque, et bien sûr dans le modèle relationnel (E.F. Codd, *The relational model for database management*, Addison Wesley, 1990 ; plus de nombreux articles antérieurs, à partir de 1969). C'est la notation de référence du modèle ANSI-SPARC des années 70.

Dans la modélisation des données il y a deux notions fondamentales :

- La notion d'entité, avec ses attributs, par exemple :

  `EMPLOYE (EMP-ID, DATE-EMBAUCHE, SALAIRE, NUM-SS, DOMICILE, BANQUE-ID, NOM, PRENOM, AFFECTATION)`

- La notion de relation entre les entités, par exemple :

  `RELATION EMP-ET (EMPLOYE, ETABLISSEMENT)` qui établit une relation entre un employé et son établissement d'appartenance.

Dans la notion de navigation introduite par C. Bachman (un des piliers du CODASYL et du DBTG à l'ANSI et à l'ISO, récipiendaire d'un ACM Turing Award, que l'auteur a bien connu chez Honeywell), cela se représente de façon simple et intuitive, comme suit (figure 4.10) :

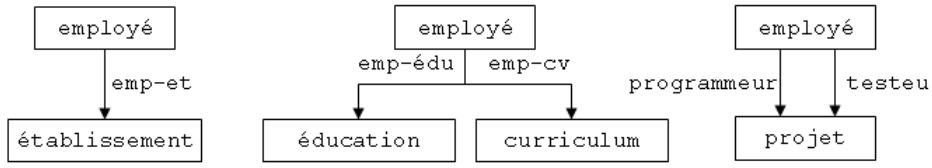


Figure 4.10 - Navigation dans les relations

Tout type de structure de données peut être représenté dans le formalisme des diagrammes de C. Bachman[6]. Dans le 3ème cas, on peut retrouver les projets dans lesquels l'employé X a le rôle de testeur, et d'autres où il serait programmeur. Le concept de navigation a été remis à la mode avec les bases de données objet dans les années 90, mais entre-temps, avec le tsunami « tout objet », le monde académique avait oublié les bases de données en réseau[7].

6. Pour une bonne synthèse, voir T. Olle, *The CODASYL approach to data base management*, Wiley ; également, D. Tsichritzis, F. Lochovsky, *Data models*, Prentice Hall, 1982.
7. Voir par exemple F. Bancilhon, C. Delobel, P. Kanellakis, *Building an object-oriented database system, the story of O2*, Morgan Kaufmann, 1992 (NB : la société O2, issu de l'INRIA, a disparu dans les années 2000).

La notion d'agrégat existe indépendamment de l'informatique. Il est intéressant, du point de vue de la direction du personnel de disposer, pour l'employé X, de la liste de toutes ses augmentations de salaires, depuis sa date d'embauche. Du strict point de vue logique, cela ne pose aucun problème de notation ; il suffit d'introduire une notion de liste dans la description de l'employé, soit, par exemple le groupe : `(SALAIRE,DATE,MONTANT)` itéré autant de fois que nécessaire pour chaque augmentation de salaire. Idem si l'employé a changé de domicile ou de banque.

Nous manipulons quotidiennement des documents administratifs organisés de cette façon. Une DTD XML (ou schéma XML) permet de représenter cela, sans la moindre ambiguïté.

Du point de vue de la représentation dans la mémoire de l'ordinateur (mémoire centrale ou mémoire disque), les choses sont différentes, du fait des contingences de l'ordinateur et de la performance des accès mémoire (mémoire centrale, disques). Pour optimiser les accès, il est préférable d'organiser la mémoire avec des entités de taille fixe, que l'on a l'habitude de représenter comme suit (figure 4.11) :

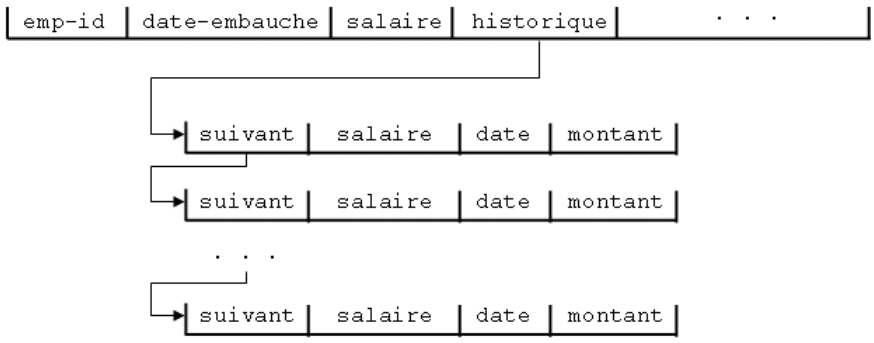


Figure 4.11 - Représentation des données en mémoire

Les flèches → représentent des pointeurs, c'est-à-dire des adresses de rangement dans l'espace d'adressage de l'ordinateur. Là encore, il est complètement inutile de vouloir cacher à l'architecte et aux programmeurs ce type de contraintes qui sont fondamentales pour la performance et l'intégrité des données manipulées, et d'une façon plus générale les caractéristiques non fonctionnelles qui exprime le comment faire et le pourquoi.

Avec Internet et les URL, la notion d'adresse, i.e. de pointeur, est partout. Il est parfaitement contre productif de vouloir la masquer. La liste chaînée ci-dessus correspond à la notion d'agrégation en UML ; la notation proposée est la suivante (ce que dans le langage des bases de données on appelait un « repeating-group » correspondant à la notion de tableau *on* liste) :

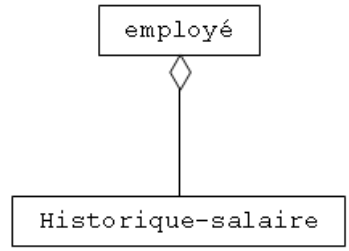


Figure 4.12 - Une agrégation en UML

À l'inverse, si l'on a décidé de renseigner systématiquement les deux dernières affectations de l'employé, dans le descriptif on pourra remplacer AFFECTATION par AFFECTATION-1 et AFFECTATION-2. La notation UML est la suivante (agrégation forte, *on composition*) :

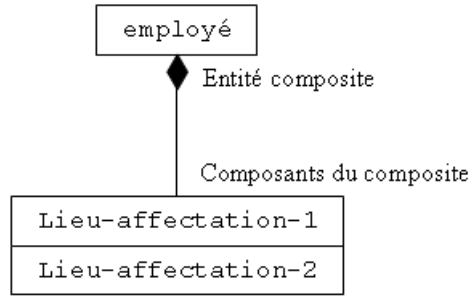


Figure 4.13 - Une composition en UML

Tout cela a des liens très forts avec la programmation. En Ada, par exemple, l'association R[composite, composant] correspond à la notion de sous-type et de constructeur de type.

La distinction agrégation/composition que fait UML n'est compréhensible que si l'on prend le point de vue du programmeur. Invoquer la notion de « *garbage collection* » comme cela est fait dans le manuel de référence pour justifier la distinction, est la preuve absolue qu'il ne s'agit pas là d'une notion logique ; et d'ailleurs le modèle ERA ne le fait pas. La gestion d'une liste chaînée, de longueur variable, nécessite à l'évidence une gestion mémoire soignée, surtout si la liste est partagée entre plusieurs programmes (i.e. transactions) qui peuvent la manipuler, ce qui rajoute cette fois une notion de gestion de ressources associée aux opérations effectuées sur l'ensemble {entité + entités agrégées}.

En prenant le point de vue du modèle relationnel, la notion de table relationnelle exclut la notion de groupe répétable (i.e. *repeating group*) pour des raisons qui

ne se justifient que par les performances d'accès. Si les entités logiques correspondent à des entités physiques de taille fixe sur disques, cela permet de gérer de façon optimale les accès aux disques.

En effet, si les entités ne sont pas de taille fixe, tout ajout d'éléments dans la liste oblige à une restructuration de l'espace disque qui, si elle n'est pas faite régulièrement, conduit à une fragmentation de l'espace qui désoptimise le fonctionnement des mémoires caches et réduit fortement la performance.

NB : une lecture disque amène en mémoire la page qui contient la donnée et généralement les pages adjacentes. Si les données associées sont dans ces pages, une seule lecture amène toute l'information, sinon il en faut d'autres. Du point de vue des performances, la topologie en mémoire est significative d'un point de vue fonctionnel et ne peut être passée sous silence s'il y a des exigences métiers de temps de réponse.

Moralité, la distinction agrégation/composition n'a rien de conceptuelle ; elle n'a de sens que du point de vue de l'implémentation dans le modèle physique. Ce sont les caractéristiques non fonctionnelles qui le moment venu permettront d'effectuer les choix d'implémentation les mieux appropriés. Du point de vue métier, la seule distinction qui aurait un sens serait de savoir si l'information visée se présente de façon déjà agrégée dans un seul document, ou de façon fragmentée dans plusieurs documents qu'il faudra manipuler pour en extraire le morceau intéressant (cf. le modèle ETL du chapitre 10) ; le travail à effectuer pour accéder à l'information, dans l'un ou l'autre cas, est différent.

La notation XML utilisée comme schéma de données est bien meilleure car elle permet de distinguer l'aspect purement logique des aspects liés à la sémantique des représentations via la notion d'attributs incorporés au langage XML.

Dans le modèle ERA étendu de T. Teorey[8], on distingue les entités dites « faibles » dont l'existence est liée à une entité dites « forte ». La suppression d'une entité « forte » comme `EMPLOYE(Dupont)` doit entraîner automatiquement la suppression des entités « faibles » rattachées à `Dupont` qui en dépendent. Si ce n'est pas le cas, l'espace alloué aux augmentations de salaire de `Dupont` est irrémédiablement perdu. Ce type de dépendance peut occasionner des fuites de ressources lors des opérations CRUD effectuées sur les entités « fortes ».

Le fait d'avoir physiquement séparé l'entité `employé` de l'entité `salaire` permet de gérer de façon différentielle la dépendance existentielle des augmentations par rapport à l'employé et le partage éventuel de cette information entre différentes transactions qui peuvent opérer isolément. On peut imaginer qu'une statistique sur les augmentations n'utilise que la partie historique, indépendamment de qui a été effectivement augmenté. Le problème sous-jacent est celui de la mise à jour des données entre différents programmes qui partagent les mêmes données : c'est le problème de

8. Cf. *ci-dessus*.

la programmation transactionnelle. Le vrai concept est celui de transaction ACID dont on parlera au chapitre 9.

Tout cela est parfaitement clair dans des notations comme les diagrammes de C. Bachman, longtemps utilisés pour illustrer la navigation dans les bases de données, dès la fin des années 60. Des ouvrages un peu anciens (disponibles à la bibliothèque de l'INRIA), mais toujours très intéressants comme : D. Tsichritzis, F. Lochovsky, *Data models*, Prentice Hall et T. Olle, *The CODASYL approach to data base management*, Wiley, en sont le témoignage.

Il est vraiment regrettable que la communauté UML ait comme tiré un trait sur ces acquis du passé que nous sommes en train de réinventer, au détriment des usagers qui n'y comprennent plus rien. Par ailleurs, il y a un vrai problème, de nature presque philosophique, entre l'approche objet et les bases de données dont B. Meyer fait état dans son livre encyclopédique : *Conception et programmation orientée objet*, Chapitre 31, Persistance d'objets et bases de données, Eyrolles, 1997. À ce jour, la technologie des bases de données objets a fait long feu ; les acteurs industriels ont quasiment tous disparu sans espoir de retour. L'architecture des données est un sujet en soi qu'il faut prendre au sérieux et non à la légère comme l'ont fait de trop nombreux « experts » de la communauté objet : les problèmes classiques de la mise à jour, des vues partielles du schéma, du partage et de l'accès concurrent, des performances, d'intégrité et de sécurité demeurent ; ils sont d'ailleurs indépendants de l'informatique, pour la plupart. Les données mémorisées, structurées ou non (textuelles, graphiques, etc.), représentent un état du monde à l'instant t, état qui évoluera ; c'est cette relation [monde réel M1, modèle informatisé M2 du monde réel] que le modèle de données, et les transformations qui lui sont associées (transactions effectuées par les acteurs), doit gérer. C'est un problème d'une autre ampleur qu'un « simple » problème de style de programmation.

4.4 LA LIBERTÉ DE L'ARCHITECTE – LA PRAGMATIQUE DES REPRÉSENTATIONS

La liberté de manœuvre de l'architecte est conditionnée par le réseau d'acteurs avec lesquels il interagit et qui vont chercher à l'influencer à leur profit. Un responsable métier sera exigeant sur le délai, alors qu'un responsable qualité sera exigeant sur le volume de test à produire pour effectuer la recette, ce qui allongera le délai. Au centre de ces contraintes parfois contradictoires, l'architecte doit impérativement garder sa liberté de jugement et rester lucide, sans complaisance, en évitant de croire aux miracles annoncés de telle ou telle technologie.

Par-dessus tout, il devra d'une part convaincre et communiquer efficacement avec son environnement, et d'autre part démontrer un charisme et un leadership pour motiver et animer l'équipe de développement. C'est un équilibre particulièrement instable, surtout dans les projets de grande taille. Le dilemme de l'architecte réside principalement dans la nature du problème qu'il doit résoudre et dans la com-

munication obligatoire avec les parties prenantes MOA, DG, client qui attendent une solution.

Du côté problème, il est confronté à la complexité, aux incertitudes, à des choix sans espoir de retour pour la prise en compte des caractéristiques dites non fonctionnelles, mais qui du point de vue de l'architecture sont tout à fait déterminantes. Ne pas s'en occuper à temps peut conduire à l'échec du projet.

Du côté solution, il y aura une focalisation sur les fonctionnalités offertes, le **F** de FURPSE, sur le contrat de service (SLA, *Service Level Agreement*) du point de vue de l'usager, sur les paramètres économiques CQFD du projet dont il faudra toujours maximiser QF et minimiser CD.

Un bon architecte sait qu'il doit impérativement s'occuper des performances et de la disponibilité très tôt dans le cycle de vie du système, et veiller à la complexité engendrée par des exigences non justifiées. En environnement distribué ce sont toujours des problèmes difficiles dont la résolution nécessitera beaucoup d'effort, mais à propos desquels il lui sera difficile de communiquer avec les parties prenantes. L'architecte devra traduire les exigences exprimées en langage compréhensible pour les parties prenantes, le pourquoi et le comment, par exemple en termes de SLA ou de CQFD projet.

Inversement s'il cède à la facilité d'une communication non fondée sur des faits mais sur des opinions et des ouï-dire, il se met dans la situation dangereuse d'être contredit par les faits, avec comme conséquence une perte de confiance des parties prenantes. Par exemple, annoncer des performances dont l'estimation ne repose pas sur un modèle de performance, lui-même basé sur une analyse quantitative des flux d'information, est un pari perdu d'avance qu'il ne faut jamais prendre. Idem pour la disponibilité.

À la question inévitable : quelle sera la performance du système une fois déployé ? mieux vaut répondre qu'on ne sait pas plutôt que de jouer à la devinette, mais proposer simultanément un plan pour effectuer des mesures dès que possible. À un risque avéré, l'architecte doit toujours répondre par une contre-mesure adaptée à la situation qui sera gérée comme telle dans le plan projet. Cela requiert une certaine force de caractère, d'autant plus que tout le monde a entendu parler de la « loi » de Moore. La loi de Moore reste encore vraie pour la performance des microprocesseurs, mais elle est souvent fausse au niveau système, car elle ne prend pas en compte les entrées-sorties disques et réseau dont l'amélioration de performance n'est pas exponentielle ; or, de cela, personne ne parle ! Dans une architecture client-serveur en environnement distribué, la contrainte principale est le plus souvent les entrées-sorties, très rarement le temps CPU. Cette non connaissance, dénommée illettrisme technique par les experts du Standish Group[9], qui s'apparente parfois à de la désinformation (les constructeurs de plates-formes et les éditeurs logiciels n'aiment pas parler de ce qui fâche), rend la communication entre experts et non-experts problé-

9. Cf. *Chaos chronicles*, version 3, 2003.

matique, surtout lorsque la défiance règne. Les uns parlent « problème », les autres « solution » ; avec le jargon des informaticiens l'incompréhension est totale.

Une autre difficulté à laquelle est confrontée l'architecte tient à la nature des notations et « langages » utilisés pour décrire le résultat du processus d'architecture qu'on appelle également architecture. C'est une polysémie dangereuse et sans doute vaudrait-il mieux ne parler que de conception pour ce qui relève du processus et d'architecture pour ce qui relève du résultat du processus. Sur ce dernier point, le langage des normes IEEE en Software Engineering est sans équivoque, et doit être préféré à tout autre. Le processus unifié[10] qui est présenté comme le compagnon méthodologique d'UML est également un bon document, compatible avec les normes IEEE, bien que les auteurs ne les citent pas alors qu'il est certain qu'ils les connaissent. Le processus unifié est qualifié d'« architecture centric » pour bien focaliser l'objectif à atteindre.

À propos de la distinction entre notation et langage, il convient d'être précis. Une notation n'est pas un langage, mais un langage a toujours besoin d'une notation. La tentation de confondre les deux et de faire passer la notation pour un langage est quasi inévitable si l'on n'est pas rigoureux. On ne peut parler de langage que si l'on a défini une sémantique opérationnelle qui permet d'interpréter la notation. Pour rester simple et drastique, ne peut être qualifié de langage que la notation exécutable ou interprétable sur une machine bien définie. Une notation qui n'est pas exécutable est généralement ambiguë car chacun peut l'interpréter à sa façon, selon son humeur. SDL est un langage ; B est un langage ; UML n'est pas un langage, bien que certains diagrammes (activités, états-transitions) en soient assez proches.

Les architectes des années 70-80 faisaient la distinction entre le « pseudo code » utilisé pour exprimer l'architecture – c'était une paraphrase de langage de programmation, utilisé comme une sténographie – et le code du programme qui en résultait ; c'était plus honnête.

Pour l'architecte, cela peut conduire à de graves ambiguïtés s'il perd sa lucidité sur ce point précis. L'architecture ne se réduit pas à l'emploi d'un langage, surtout si ce dernier n'est qu'une notation. Cela peut donner l'apparence de la rigueur, alors que la rigueur est absente.

L'une des vraies difficultés de l'architecture est d'identifier ce qu'il faut réellement modéliser de façon rigoureuse. Dans un système il y a toujours une partie simple et une partie compliquée, dans une proportion qui suit souvent une distribution de Pareto 80-20. Si l'on ne modélise que les 80% faciles et qu'on laisse le reste en jachère, qu'a-t-on fait réellement : enfoncé des portes ouvertes ou résolu le problème ? Sur quoi faut-il communiquer : sur les 80% ou sur les 20% ? Chacun appréciera !

En matière de communication, on peut communiquer sur le fait que l'on modélise le cas fonctionnel nominal quand tout se passe bien (c'est le monde parfait M1).

10. Cf. I. Jacobson, G. Booch, J. Rumbaugh, *The unified software development process*, Addison Wesley, 1999.

Pour le reste, performance, disponibilité, sûreté de fonctionnement, on verra à l'intégration (c'est évidemment trop tard), quand à l'évolutivité et l'adaptabilité on verra dans les versions futures ! La tentation de ne pas s'occuper des 20% qui fâchent est très forte.

Pour ces 20%, l'architecte va d'ailleurs se heurter à des difficultés d'ordre notationnel, car il ne va pas trouver dans le « langage » ce qui est essentiel à ses yeux pour effectuer une modélisation utile. On peut certainement interpréter la tentative d'intégration de SDL à UML de cette façon car il est évident qu'un architecte confronté à des problèmes de protocoles et connaissant SDL sera frustré par UML qu'il jugera incomplet sur ce point particulier. C'est d'ailleurs un vrai danger pour UML car est-il raisonnable de vouloir absorber tout ce qui a été fait ailleurs ? C'est refaire l'erreur qui fut fatale au PL1 dans les années 70, et plus récemment à Ada dans les années 90. L'interopérabilité entre langages, au moyen de règles de correspondances et/ou de traductions, est certainement une meilleure approche car elle respecte la liberté de l'architecte et l'ouverture des systèmes, mais ceci est une autre histoire.

L'architecte est un créatif qui au besoin ne doit pas hésiter à compléter la notation si cela est justifié, par ses propres notations. Auquel cas il est préférable d'avoir une notation plutôt qu'un langage qui est toujours un monde clos. L'ambiguïté devient un facteur d'évolutivité, ce qui peut paraître paradoxal, mais c'est une propriété bien connue des langues naturelles. À l'opposé, il est très difficile de faire évoluer le langage mathématique. Cependant, de nombreux physiciens savent très bien bricoler le langage mathématique. Retenons que l'ambiguïté à parfois des avantages, et que l'architecte peut les tolérer à condition que cela soit fait en toute lucidité.

On peut conclure ce paragraphe par une recommandation du logicien R. Carnap[11] appelé principe de tolérance en syntaxe (i.e. la structure de la notation) : « *It is not our business to set up prohibitions, but to arrive at conventions* ». La convention, par définition, fait l'objet d'un consensus. Elle est acceptée par tous. C'est une forme de contrat de communication entre les acteurs usagers de la notation. Dans cette perspective, la notation XML, qui n'est qu'une BNF légèrement améliorée (cf. partie 2), est une excellente convention, promise à un grand usage.

4.5 ORGANISATION DU RÉFÉRENTIEL D'ARCHITECTURE – LE RÉFÉRENTIEL COMME MÉTA-LANGAGE

La quantité d'information à stocker dans le référentiel est à l'échelle de la taille du système d'information. Pour un système d'information comme AGORA (cf. section 1.3, Architecture de l'information) cette taille sera très importante. Si l'on compte une ligne de documentation, au sens large, à faire figurer dans le référentiel pour 20 à 30 lignes de code source, nous aurons environ 20 000 pages de texte soit 50 volumes de

11. Dans, *Logical syntax of language*, Routledge and Kegan Paul.

400 pages. On comprend tout de suite que si cette « masse » n'est pas elle-même bien organisée, elle peut devenir un réel problème pour ceux qui en ont l'usage, en premier lieu les équipes de maintenance, d'exploitation et d'administration.

Le référentiel, dans la mesure où c'est lui qui donne la clé d'accès au contenu informationnel du système d'information, doit lui-même faire l'objet de conventions de présentation extrêmement strictes entre auteurs et usagers de l'information qu'il gère. Le référentiel joue le rôle du thésaurus dans une encyclopédie. Il est un élément fondamental d'une communication efficace.

En reprenant le schéma de la figure 2.7, Le bon usage des référentiels, on peut décrire la logique de fabrication du référentiel comme suit (figure 4.14) :

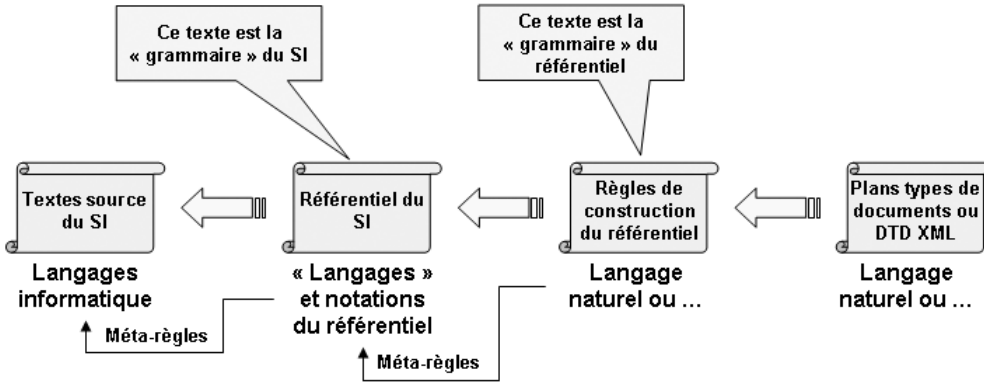


Figure 4.14 - Logique de construction du référentiel

Chaque niveau de prescription est un méta-langage pour le niveau décrit ou, en langage compréhensible par tous, une grammaire. Pour utiliser le jargon du MOF de l'OMG, on pourrait dire que les règles de construction du référentiel sont un méta-méta-modèle du système informatisé. Cette terminologie pédante n'a aucun intérêt pour celui qui a compris la nuance fondamentale entre le langage et le méta-langage (i.e. la grammaire) qui décrit ce langage ; elle est nuisible pour celui qui n'a pas saisi la nuance et qui va se perdre dans les méta-niveaux.

L'une des qualités indispensable à l'architecte est de bien percevoir ces différents niveaux de description car c'est un facteur d'ordre et de structuration important de l'information. Si lui-même n'est pas à l'aise dans cette hiérarchie d'abstraction, cela créera des difficultés de communication et des incompréhensions avec les équipes de développement. Comme nous l'avons souligné dans le chapitre 1, le rôle de l'architecte qui est peut-être le plus important est de mettre de l'ordre dans le système, mais d'abord et avant tout dans la tête des acteurs.

Pour l'organisation proprement dite du référentiel on pourrait s'en tenir aux recommandations des normes IEEE (en particulier IEEE 1220 et IEEE/EIA/ISO 12207) ou équivalentes qui proposent toute une variété de plans types validés par la

profession pour les livrables issus des différents processus d'ingénierie. Il n'y a quasiment rien à rajouter car tout est dit : il suffit de faire son marché dans l'IEEE software engineering standards collection qui regroupe une vingtaine de standards tous très intéressants.

Pour aller plus loin, et utiliser pleinement l'outil informatique, on pourrait envisager une XML-isation des documents, ce qui permettrait de faire des recherches en utilisant les langages de requêtes basés sur XML que l'on voit fleurir un peu partout, y compris en OPEN SOURCE. Notons que les DTD (XML schéma) sont un méta-langage par rapport aux plans types des documents.

Pour les très grands systèmes et les systèmes de systèmes, tous caractérisés par une longue durée de vie, c'est une question que l'architecte doit poser aux parties prenantes, en leur faisant comprendre les enjeux économiques. Un désordre dans le référentiel provoquera mécaniquement un désordre dans les équipes de développement qui rejaillira tôt ou tard en désordre dans les projets.

L'organisation du référentiel est un coût qualité au sens de P. Crosby (cf. son livre célèbre, *Quality is free*) qui fera faire des économies tout au long de la vie du système. Le coût de non qualité occasionné par l'absence, ou l'inorganisation, du référentiel est un coût récurrent qui s'amplifie avec l'age car tout les implicites sont rapidement oubliés et disparaissent avec la mobilité des détenteurs de l'information. Ce coût compense largement le coût d'obtention d'un référentiel de qualité, d'où le titre du livre de Crosby.

Terminons ce chapitre en mentionnant l'approche « framework » dont J. Zachman[12] a été l'un des initiateurs, suivi et imité par beaucoup d'autres[13]. La présentation en framework a le mérite de la simplicité. Le référentiel est présenté sous la forme d'une grille synthétique qui visualise le questionnement de l'architecte et l'ensemble des travaux à effectuer.

Les lignes de la grille donnent les points de vues et les acteurs concernés par le point de vue (le contexte, le modèle d'entreprise, le modèle du système informatisé, la technologie utilisée, les représentations détaillées). Les colonnes organisent le questionnement (*What, How, Where, Who, When, Why*) sur les aspects jugés importants (Données, fonctions, Réseaux, Acteurs, Temps, Motivation), ce qui donne une grille comme suit (tableau 4.2) :

---

12. Cf. IBM Systems journal, Vol 26, N°3, 1987 : *A framework for information systems architecture* ; et IBM Systems journal, Vol 31, N°3, 1992, *Extending and formalizing the framework for information systems architecture*. Son site : www.zifa.com.

13. En particulier sous l'égide du US-DOD avec TAFIM (*Technical architecture framework for information management*), TOGAF (*The open group architecture framework*), DODAF (*DOD Architecture framework*), etc. Quelques ouvrages intéressants sur le sujet des « *Enterprise IT architecture* » : M. Cook, *Building enterprise information architectures*, Prentice Hall, 1996 (le point de vue de Hewlett-Packhard) ; C. Perks, T. Beveridge, *Guide to Enterprise IT architecture*, Springer, 2003 ; P. Bernus, et alii, *Handbook on enterprise architecture*, Springer, 2003.

Tableau 4.2 - Présentation du référentiel en grille

	What Données	**How** Fonctions	**Where** Réseaux	**Who** Acteurs	**When** Temps	**Why** Motivation
Contexte	Essentiellement des nomenclatures					
Modèle d'entreprise	Modèles sémantiques	Modèles de processus métier		Workflow	Ordonnancement entreprise	Plans opérationnels
Modèle du SI	Modèles de données logiques	Modèle de traitements	Architecture de l'infrastructure	IHM	Cycles produits services de l'entreprise	Procédures de l'entreprise
Technologies utilisées	Selon offre des éditeurs + Outils spécifiques développés pour le projet					
Représentations détaillées	Selon SGBD	Selon langages	Infrastructure réseau et middleware	Sécurité	Ordonnancement fin, Interruptions	Spécification fine des procédures

C'est de la bonne pédagogie de communication car elle montre aux parties prenantes tout ce qu'il faut faire pour réaliser des systèmes utiles dans un langage qu'elles peuvent comprendre. Du point de vue de l'ingénierie il faudrait nuancer, car des aspects fondamentaux sont absents de la grille, à ce niveau de synthèse, comme par exemple les interfaces. Seules les IHM sont explicitées car elles ont un impact direct sur la performance des acteurs métiers. Pour les autres interfaces, on pourrait dire que la colonne réseau implique nécessairement des interfaces, mais c'est insuffisant car il y a d'autres types d'interfaces comme par exemple les modèles d'échanges pour l'interopérabilité (cf. chapitre 10).

Dans le standard IEEE 1016, *Recommended practice for software design description*, on trouve un petit tableau tout à fait révélateur de la problématique d'ingénierie, (figure 4.15).

Dans ce tableau, l'accent est clairement mis sur les dépendances et les interfaces dont il est dit, dans le corps du standard, qu'elles sont les contrats qui matérialisent les conventions passées entre les architectes, les programmeurs, les testeurs et les clients (pour la recette). Nous reviendrons plus en détail sur ce point au chapitre 15, Interfaces. Sur ce tableau, on voit la lucidité et la largeur de vue dont doit faire preuve l'architecte, et les deux facettes de la communication, soit avec les parties prenantes, soit avec les équipes d'ingénierie, sous-traitants inclus. La colonne scope inclut la description du vecteur d'état du service fait ou non fait de l'entité architecturale (cf. chapitre 8 pour le détail).

Il est essentiel de faire prendre conscience aux parties prenantes tout ce qu'il y a à faire de leur point de vue pour produire des systèmes de bonne qualité et dont il faudra payer le juste prix. Le grand mérite de J. Zachman est d'avoir proposé une pré-

sentation percutante pour les parties prenantes, mais pour l'ingénierie, mieux vaut s'en remettre à l'IEEE. Ce n'est pas incompatible.

Table 1—Recommended design views

Design view	Scope	Entity attributes	Example representations
Decomposition description	Partition of the system into design entities	Identification, type, purpose, function, subordinates	Hierarchical decomposition diagram, natural language
Dependency description	Description of the relationships among entities and system resources	Identification, type, purpose, dependencies, resources	Structure charts, data flow diagrams, transaction diagrams
Interface description	List of everything a designer, programmer, or tester needs to know to use the design entitites that make up the system	Identification, function, interfaces	Interface files, parameter tables
Detail description	Description of the internal design details of an entity	Identification, processing, data	Flowcharts, N-S charts, PDL

Figure 4.15 - Vue de l'architecture selon IEEE 1016, Software design description

Pour bâtir des projets sains, que l'on pourra intégrer sans trop de difficultés, la définition préalable des interfaces et des dépendances est impérative, faute de quoi le parallélisme entre les différents projets sera compromis. Les responsables projets négocieront, ou renégocieront, en permanence ce qui aurait dû être fait d'entrée de jeu avec tous les risques d'ambiguïté, d'incohérence et d'oubli que cela comporte. Ils perdront beaucoup de temps. Dans une architecture orientée services qui est un thème à la mode, et pour une fois à juste titre, les interfaces sont la donnée première. Il faut leur donner une visibilité très grande. Le plus surprenant est que la notion d'interface existe non seulement dans l'ingénierie, mais également dans le métier. Quand deux acteurs organisationnels et/ou humains négocient, ils le font conformément à certaines règles, soit culturelles, soit juridiques. Dès qu'il y a interaction, il y a interface. On pourrait d'ailleurs aménager la grille de J. Zachman en transformant la colonne réseau en une colonne interface, dont les réseaux seraient un cas particulier.

5

Place de l'architecture dans les projets informatiques

Aspects économiques de l'architecture et ROI système

Ce chapitre n'est qu'un rappel. Les notions de cycle de vie et de cycle de développement imprègnent l'ingénierie des systèmes et du logiciel depuis plus de trente ans. C'est un acquis définitif du savoir faire en ingénierie des TIC qu'il faut connaître. Voir notre ouvrage, *Ecosystèmes des projets informatiques*, chez Lavoisier-Hermès, et bien sûr les normes de la profession.

5.1 CYCLE DE VIE D'UN SYSTÈME – CYCLE DE DÉVELOPPEMENT

Pour traduire un besoin utilisateur en une application informatique utilisable, il faut dérouler tout un ensemble de processus dans un certain ordre et avec certaines règles, ce qui n'interdit nullement de paralléliser les développements et de multiplexer les ressources affectées aux différents processus.

Tout ceci est aujourd'hui parfaitement consigné et documenté dans des standards internationaux comme les normes ISO 12207, *Software life cycle*, et ISO 15288 / IEEE 1220, *System Engineering – System life cycle processus*, qu'il est impératif de connaître.

On peut les résumer par les deux schémas 5.1 et 5.2.

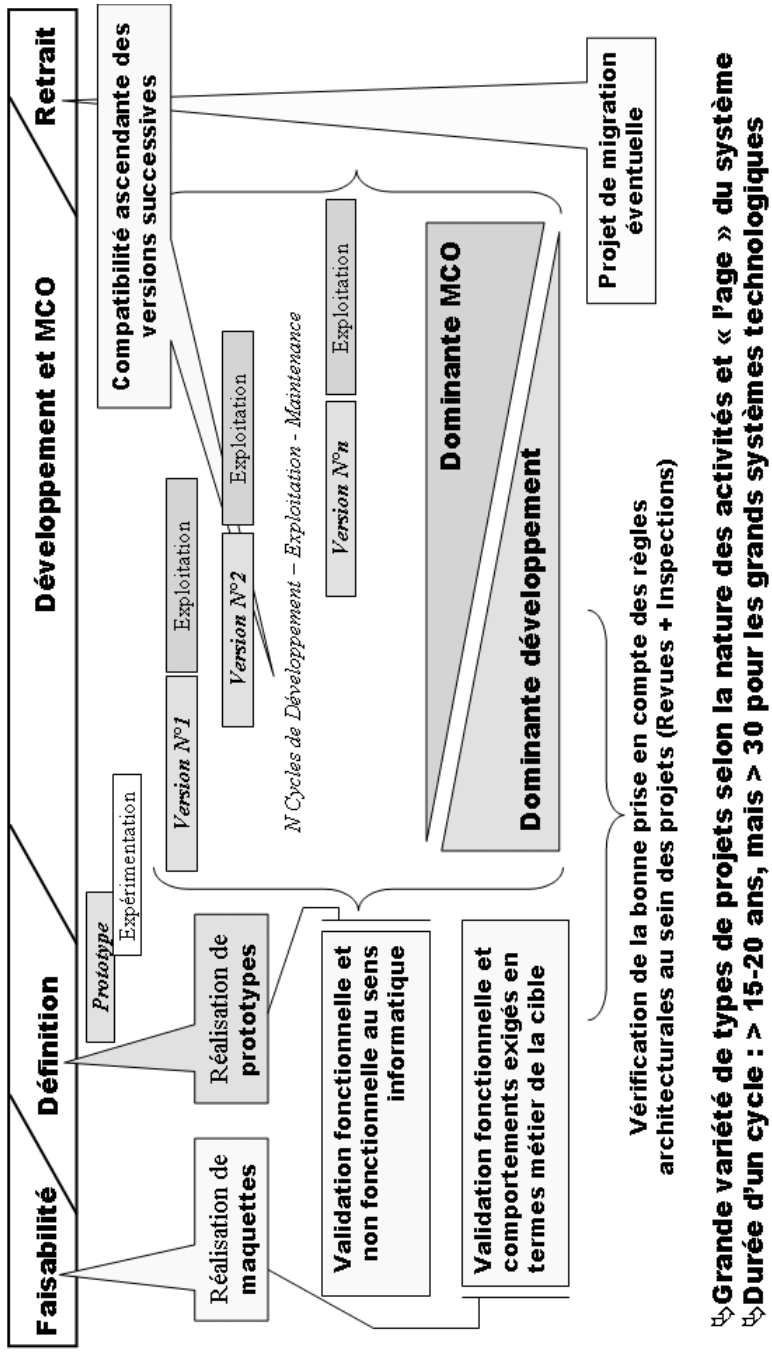


Figure 5.1 - Cycle système complet

Dans ce cycle système, chacune des versions donne naissance à un cycle de développement conforme au schéma 5.2.

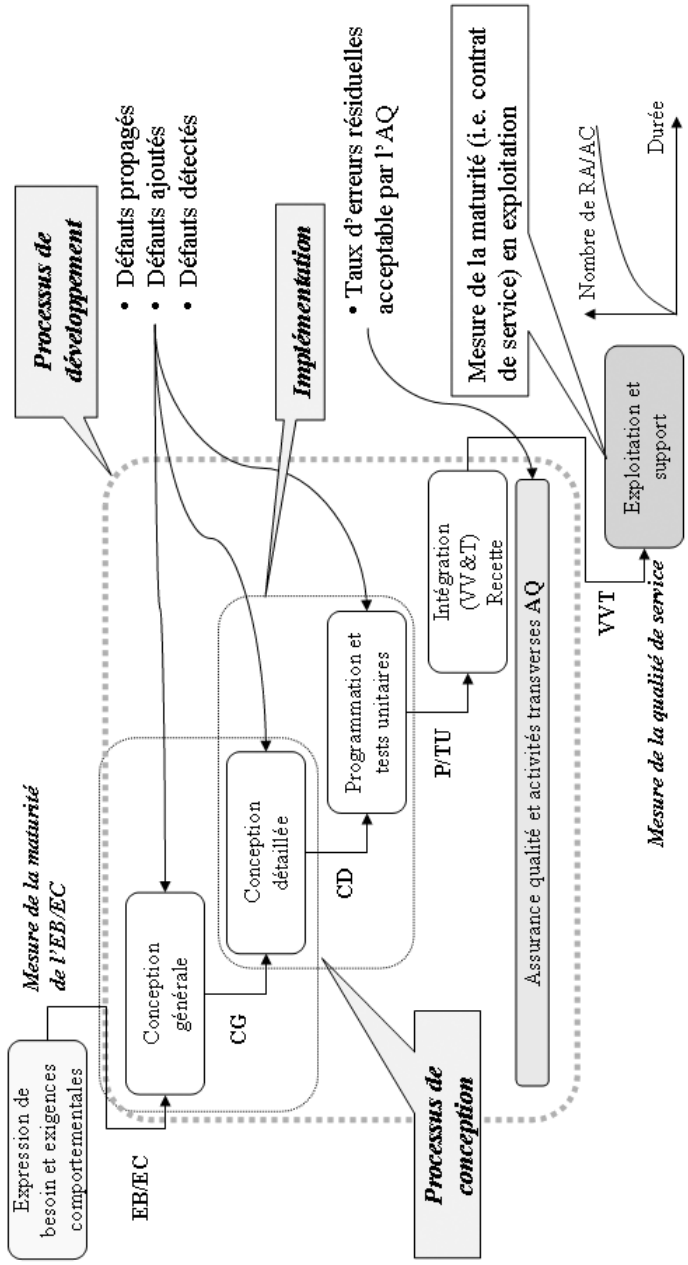


Figure 5.2 - Cycle de développement d'une version

Par rapport à ces deux schémas, on peut dire que le rôle de l'architecte est prépondérant dans les phases de faisabilité et de définition du cycle système, et dans les processus de conception et d'intégration des premières versions du cycle de Développement – MCO (Maintien en Conditions Opérationnelles). L'architecte pourra se

retirer quand le développement du système aura trouvé son rythme de croisière et quand les équipes MCO auront effectué les premières évolutions et adaptations.

5.2 RÔLE ET PLACE DE L'ARCHITECTE DANS LA RELATION MOA/MOE

La relation dynamique MOA (les parties prenantes) ↔ MOE (l'ingénierie) peut être représentée par le schéma 5.3. L'architecte y a un rôle prépondérant. Le MOA est dans une logique moyen/long terme (TCO) en fonction de la stratégie de la DG. Le MOE est dans une logique projet court terme.

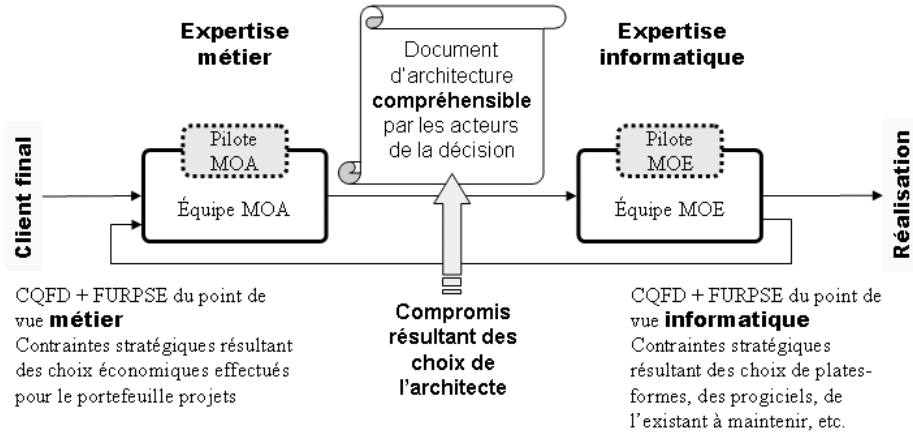


Figure 5.3 - Relation dynamique MOA/MOE

Le MOE reçoit une spécification de besoin (cahier des charges fonctionnelles et techniques) réalisée sous l'autorité de la MOA. Cette spécification de besoin doit définir à minima :

- La nature des flux échangés entre les acteurs métiers.
- Le descriptif des fonctions à informatiser en tout ou partie, et la définition, en tant que besoin, des interfaces à réaliser avec les fonctions non informatisées. Cette description peut être faite à l'aide de *use-cases* qui sont des représentations en extension des fonctions à réaliser.
- Les contraintes d'intégrité du point de vue de l'usager, en particulier la disponibilité.
- L'ensemble des caractéristiques non fonctionnelles de façon à définir d'un point de vue usager les contraintes à prendre en compte impérativement pour le contrat de service (i.e. chacune des parties a des droits ET des devoirs).

En retour, le MOE propose une solution (ou plusieurs) et le devis correspondant en terme CQFD ; mais cette fois, **C** est le coût de développement estimé, **Q** est l'ensemble des caractéristiques qualité non fonctionnelles que la solution garantit, **F** est le périmètre fonctionnel réalisé (ou intégré, s'il s'agit d'un progiciel) dans une unité d'œuvre convenue entre la MOA et la MOE par exemple les points de fonctions ; **D** est le délai de réalisation de la solution. La description en extension a été remplacée par une description en intention qui généralise les cas particuliers ; c'est un modèle fonctionnel sur lequel l'équipe de développement en charge de l'ingénierie va pouvoir travailler.

Sur cette base réciproque, la négociation contractuelle proprement dite peut s'engager jusqu'à l'obtention d'un compromis CQFD acceptable (et donc d'un risque) pour les deux parties.

L'intégration des contraintes PESTEL et FURPSE, sur la base d'une solution architecturale (ou d'une famille de solutions compatibles) que l'entreprise peut se payer en terme CQFD et TCO peut se représenter comme suit (figure 5.4).

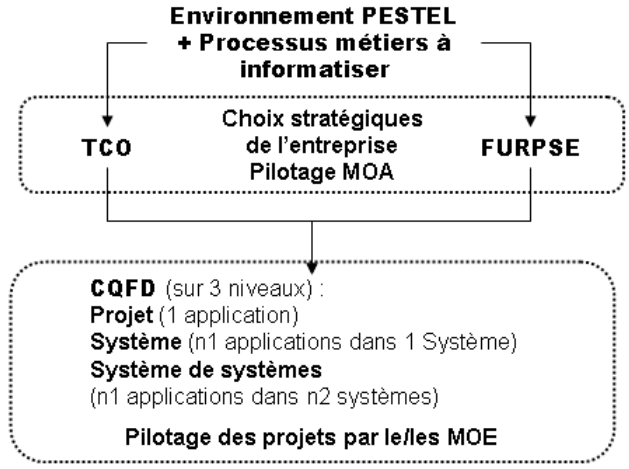


Figure 5.4 - Articulation des contraintes PESTEL/FURPSE et CQFD/TCO

L'architecte est présent à tous les niveaux car c'est lui qui doit veiller à ce que les décisions prises pour un projet particulier (CQFD), ne fabriquent pas un problème pour les projets futurs ce qui dégraderait le TCO (*Total Cost of Ownership*) et ruinerait le retour sur investissement.

5.3 INFLUENCE DE L'ARCHITECTE SUR LE RETOUR SUR INVESTISSEMENT ROI

Pour les entreprises dont le système d'information est un moyen au service des directions métiers, et de ce fait un actif très important de l'entreprise, le seul problème qui

devrait préoccuper le décideur est celui de la création de valeur pour l'entreprise et pour ses clients. Comment faire en sorte pour que l'informatique optimise et améliore la performance des chaînes de valeur des métiers ? Comment utiliser la technologie informatique pour créer de nouveaux services qui procurent un avantage compétitif sur les concurrents de l'entreprise ? Et bien sûr, comment mesurer ou estimer qualitativement le ROI, le retour sur investissement, et la performance des processus ainsi améliorés ?

Le profil classique d'un investissement informatique a l'allure suivante, quand tout se passe bien (figure 5.5).

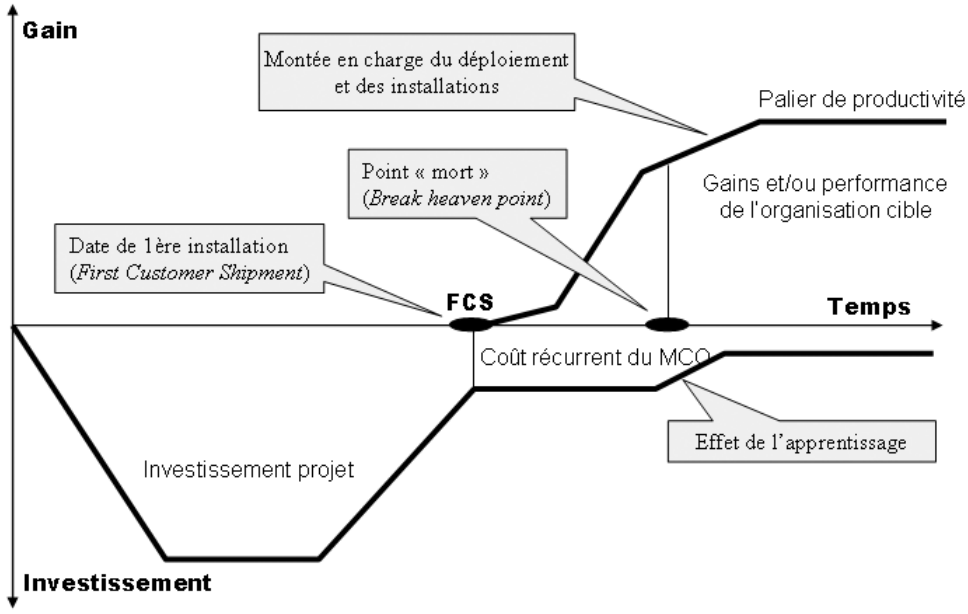


Figure 5.5 - Courbe d'investissement et ROI

L'investissement réalisé, c'est-à-dire la dépense faite, se mesure relativement facilement car il s'agit de coûts de développement, d'acquisition de plates-formes et de postes de travail, d'équipements réseau, etc., et du coût récurrent associé. C'est un pur problème de comptabilité analytique.

Le calcul du gain réalisé ou de l'amélioration de la performance de la chaîne de valeur est beaucoup plus délicat, voir paradoxal pour reprendre le terme de Robert Solow[1] : « *We see computers everywhere but not in the productivity statistics* »[2]. Cepen-

1. C'est un prix Nobel d'économie, Docteur Honoris Causa du CNAM.
2. Une très bonne référence sur l'économie de l'information est la série d'ouvrages de Paul Strassmann, en particulier : *The squandered computer*, 1997, et *Information productivity*, 1999, tous deux chez Information Economics Press.

dant, si les processus constitutifs de la chaîne de valeur ont été raisonnablement modélisés (NB : c'est le rôle des modèles d'entreprise), si des indicateurs de performance ont été établis pour chacun d'eux, alors on peut effectuer une estimation du gain engendré, à défaut d'une véritable mesure quantitative. Une informatisation correcte permet :

a- D'améliorer la productivité – on fait plus de choses pour un coût moindre – ;

b- D'augmenter la réactivité de l'organisation – on travaille plus vite, on interagit mieux avec les clients de la chaîne de valeur – ;

c- De proposer de nouveaux services qui simplifient la vie du client, ou qui améliorent les chaînes de valeur du client, ou qui permettent de conquérir de nouveaux clients tout en fidélisant les anciens.

Sur une période d'exploitation donnée, on peut calculer le ROI selon la formule classique $ROI = \frac{Gain–Investissement}{Investissement}$, l'investissement étant la somme de deux coûts :

1. l'investissement projet,

2. le coût récurrent englobé sous l'appellation MCO qui regroupe l'exploitation, le support et la maintenance.

Dans un monde parfait, sur une longue durée, le ROI tend vers une limite qui est le rapport :

$$ROI \xrightarrow{\text{limite}} \frac{Gain–Investissement}{Investissement}$$

La structure de coût du MCO et son évolution jouent un rôle déterminant dans l'économie générale du SI, car c'est lui qui fixe l'horizon que l'on ne pourra pas dépasser (une asymptote).

Un retard projet provoque un décalage du point mort, ce qui entraîne mécaniquement une perte de productivité. Une livraison de mauvaise qualité, à supposer qu'elle n'occasionne pas de retard, provoque une augmentation du coût récurrent de MCO, donc une baisse du ROI. Les décisions de l'architecte sont un paramètre fondamental de la durée des projets et des coûts récurrents MCO, via la qualité du référentiel mis en place. Tous ces paramètres sont fondamentaux pour que l'entreprise conserve ses avantages compétitifs.

Pour une analyse plus fine des pathologies projets, associées à la courbe d'investissement, voir notre ouvrage *Ecosystèmes des projets informatiques*, chez Lavoisier-Hermès, déjà cité.

PARTIE 2

Analyse de deux chefs-d'œuvre d'architecture

La meilleure façon de comprendre ce qu'est l'architecture, à défaut de pouvoir la pratiquer, est d'étudier ce qui s'est fait de mieux dans le domaine, et si possible avec ceux qui y ont participé, en évitant les deuxièmes ou troisièmes mains car alors le message initial se perd. L'architecture est un processus dont l'architecte est le pilote ou, si l'on préfère, le chef d'orchestre dont les équipes projets sont les instruments.

Comme on va le voir, l'architecture est toujours un compromis entre les besoins formulés, l'état courant et prévisible de la technologie, les savoir-faire méthodologiques. L'architecture est une solution parmi beaucoup d'autres possibles, ce qui requiert de la part de l'architecte, à la fois imagination et créativité, mais aussi un vrai talent pédagogique pour expliquer le pourquoi et le comment des choix, doublé d'une force de conviction et d'une capacité de décision en univers incertain. Cela fait beaucoup dans une même tête, et c'est pour cela que les bons architectes sont rares.

Pour illustrer cette démarche, nous allons analyser deux exemples d'architecture que l'on peut à bon droit qualifier de chefs-d'œuvre, au sens compagnonnage du

terme : l'architecture des compilateurs et l'architecture des processus dans les machines.

L'architecture des compilateurs, et plus généralement des traducteurs, connaît un regain d'intérêt avec l'arrivée dans le grand public informatique de langages comme XML ou WSDL (*Web Service Definition Language*) qui nécessitent de bien comprendre la relation syntaxe-sémantique. A moins grande échelle, la maîtrise d'une démarche comme le MDA (*Model Driven Architecture*) préconisée par l'OMG requiert un bon niveau d'expertise en théorie des langages et des compilateurs.

Quant aux processus, depuis l'arrivée massive des architectures distribuées, ils sont partout, tant au niveau métier avec le *business process engineering* (cf. un langage comme le BPML, *Business Process Modeling Language*) qu'au niveau des composants applicatifs et des systèmes informatisés. La notion de processus est un concept nouveau[1], une pure abstraction, que les architectes des systèmes d'exploitation ont utilisé à leur profit dès les années 60 et sans lesquels ces grandes constructions que sont les systèmes d'exploitation n'auraient jamais vu le jour.

Parmi beaucoup d'autres choix possibles, celui-ci n'est pas fortuit, pour deux raisons. La première est d'ordre conjoncturel, lié à l'état de la technologie informatique et à la nature des problèmes auxquels elle doit apporter des solutions. On en a vu un aperçu au chapitre 1. Par bien des aspects, les problèmes qui se posent aujourd'hui aux architectes des systèmes informatisés, se sont déjà posés, certes en d'autres termes, aux architectes des systèmes d'exploitation et des progiciels systèmes comme les SGBD. La mise en perspective est éclairante, et l'on se sent moins seul.

La seconde raison est de nature plus personnelle, car l'auteur a eu la chance durant vingt années passées chez Bull, au départ Bull-General Electric puis Honeywell-Bull, d'être acteur pour ce qui concerne les compilateurs et les bases de données, et d'être collègue, à quelques bureaux d'écart, des acteurs en charge des processus, notions qui bien évidemment n'étaient pas sans incidence sur les langages de programmation et les compilateurs (cf. le langage Ada).

Le témoignage donné ici n'a aucune prétention historique[2]. Son seul but est d'illustrer la permanence de certains problèmes, de montrer comment telle abstraction a émergé au milieu des particularismes, de décrire l'état d'esprit des architectes qui ont mis au point ces mécanismes fondamentaux et permis, in fine, le décollage des TIC.

Mon activité d'enseignement au CNAM et de consultant auprès de grands groupes sont autant de témoignage du caractère d'urgence que la situation actuelle de l'industrie informatique, et des difficultés qu'elle rencontre, impose à notre réflexion concernant le métier d'architecte et de la transmission des savoir-faire. Les problè-

1. Même en philosophie, cf. N. Withehead, *Process and reality*, Free Press, 1978 (1ère édition en 1929) ; il est le co-auteur, avec B. Russel, des *Principia Mathematica*, la bible de la logique mathématique.

2. Voir le site de la fédération des équipe Bull, www.feb-patrimoine.com qui gère les archives Bull.

mes que rencontre l'industrie de l'information sont nombreux, comme on peut le voir dans le rapport du Standish Group, C*haos chronicles*, mais il ne faudrait pas faire l'erreur de penser que les architectes sont moins bons qu'avant. Ils ne sont surtout pas assez nombreux compte tenu du développement fulgurant de la technologie de l'information, et le métier est encore très mal compris des décideurs, pour les mêmes raisons. Dans le BTP, il ne viendrait à l'idée de personne de se lancer dans une construction, sans l'aide d'un cabinet d'architecte. On n'en est pas encore là pour ces grandes constructions abstraites que sont les systèmes informatisés.

6

Principes d'architecture des compilateurs

6.1 LE PROBLÈME DE LA TRADUCTION DES LANGAGES INFORMATIQUES

L'architecture des compilateurs a été le premier grand défi de l'ingénierie du logiciel, dans les années 60-70. Sans compilateur, pas de langage de haut niveau et donc pas de programmation de masse. Les problèmes posés par la fabrication des compilateurs ont l'avantage d'être simple dans leur formulation et complexe dans leur résolution à cause de la nature combinatoire des problèmes posés par la compilation.

Les langages de programmation ont une double fonction :

- Programmer les applications en utilisant de façon optimale les ressources de la plate-forme d'exécution.
- Communiquer entre programmeurs (relecture, maintenance, évolution, etc., des programmes) et avec les acteurs du processus d'ingénierie.

Les critères FURPSE/PESTEL sont applicables.

Il s'agit de traduire un texte écrit dans un langage proche de celui de l'usager — c'était l'un des objectifs avérés de FORTRAN et COBOL — en séquence d'instructions exécutables sur une machine M. Si l'on veut simplifier le travail de l'usager, il faut masquer le plus possible les idiosyncrasies de la machine tout en conservant la possibilité de générer un code optimum sans pour autant créer des difficultés de traduction qui ne feraient que complexifier la réalisation du compilateur ; d'où les divers modes de compilation indiqués sur la figure.

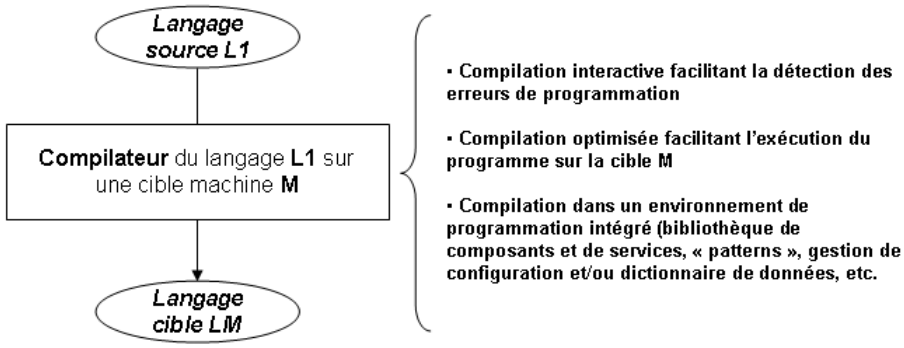


Figure 6.1 - Différentes utilisations d'un compilateur

Les deux premiers langages pour des raisons historiques faciles à comprendre ont été :

- FORTRAN (*FORmula TRANslator*) qui au départ n'était qu'une simple transposition de formules mathématiques utilisées par les physiciens et mathématiciens de Los Alamos.
- COBOL (*COmmon Business Oriented Language*) qui était une adaptation des procédures comptables utilisées en gestion aux nouvelles machines destinées à remplacer les machines à cartes perforées.

Ces deux langages ont évidemment joué un rôle de premier plan dans la naissance et l'évolution des techniques de compilation.

La formulation complète du problème de compilation peut se résumer comme suit, avec trois interfaces fondamentales : l'éditeur de lien, le moniteur de contrôle associé au langage (appelé RTP, *Run Time Package*, très différent d'un langage à l'autre) et le système d'exploitation (figure 6.2).

Le langage **L1** est un premier compromis car il doit servir de support à la communication homme-machine (c'était le problème initial) mais également à la communication homme-homme afin de faciliter la relecture, les modifications, les évolutions des programmes écrits en L1. Le langage **L1** ne doit donc pas rompre les habitudes langagières courantes des langages naturels tout en offrant des notations rigoureuses et non ambiguës (comme celles des mathématiques) permettant de générer du code machine LM sémantiquement équivalent à L1.

Le premier problème qu'il a fallu résoudre est celui de la description du langage L1 de façon à ce qu'un locuteur A1 puisse produire un texte non ambigu compréhensible par un locuteur A2.

Par analogie avec le langage humain les architectes de langages, dans les années 60, ont mis au point des techniques de description à l'aide de grammaires formelles issues des travaux des linguistes, comme N. Chomsky[1], et des logiciens comme S.C.

Kleene[2]. Ces travaux représentent une performance unique par quelques-unes des meilleurs esprits du siècle dernier dont les *Principia Mathematica*[3] (3 Vol, 1910-1913) sont la face visible. Cet effort, sans précédent, a marqué pour des décennies la problématique dont les langages de programmation sont un résultat lointain. Ces techniques sont aujourd'hui un standard de présentation de tout langage informatique. ALGOL 60 fut le premier langage dont la syntaxe fut décrite de manière formelle avec la notation BNF (*Backus Naur Form* ou B*ackus Normal Form*), standardisée par l'ISO, Norme 14977, *Extended BNF.*

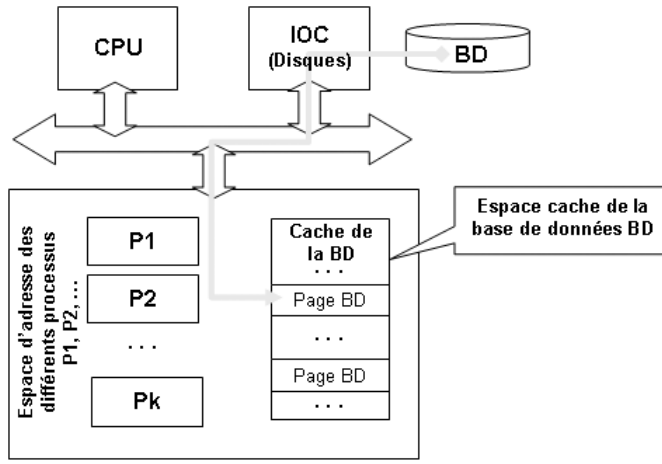


Figure 6.2 - Transformation du programme source en exécutable

Pour l'architecte de compilateur la question qui se pose est de savoir s'il peut utiliser directement dans le compilateur la notation grammaticale utilisée pour décrire L1, ou bien s'il doit utiliser une notation directement compréhensible par la machine (par exemple le langage machine lui-même) ce qui revient à dupliquer l'information syntaxique, et de ce fait ouvrir la porte aux erreurs d'interprétation et aux contresens dans le changement de notation.

Le problème sous-jacent est difficile car la notation grammaticale G(L1) a comme seul objectif, pour l'usager du langage L1, de permettre de générer ou de reconnaître les phrases du langage L1 alors que la notation G(L1→M) doit *en plus* permettre de traduire le langage L1 en langage cible LM ; c'est une grammaire transformationnelle.

1. Cf. son article, *On certain formal properties of grammars*, *Information and control*, Vol 2, N°2, June 1959 et ses nombreux ouvrages comme *Introduction to the formal analysis of natural languages* ; beaucoup sont traduits en français.
2. Cf. son ouvrage, *Logique mathématique*, réimpression chez J. Gabay.
3. Ecrit par B. Russel et N. Withehead. C'est à partir de ce texte que K. Gödel démontra son célèbre théorème.

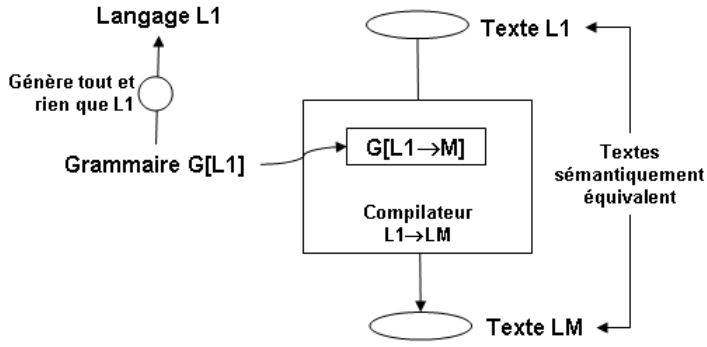


Figure 6.3 - Prise en compte de la grammaire du langage

Le dilemme pour l'architecte est de choisir entre a) tout faire en même temps, avec le risque de déboucher sur un programme de très grande taille difficile à gérer, ou bien b) de procéder par étapes successives avec le risque d'avoir à éclater le composant G(L1→M) en sous-composants ayant entre eux de très fortes relations de dépendance ; dans ce dernier cas, quel est le nombre optimal d'étapes intermédiaires et comment garantir la cohérence de l'ensemble ?

La notion d'étape de traduction n'est vraiment apparue de façon claire qu'avec le langage PL1 qu'IBM essayait d'imposer en lieu et place de l'ensemble hétéroclite {FORTRAN + COBOL} dans les années 70. Ce fut un énorme échec commercial mais un formidable accélérateur dans la compréhension de l'architecture des compilateurs.

6.1.1 Analyse lexicale

Vu la taille du langage PL1 et les difficultés de réalisation d'un compilateur PL1 efficace, il était urgent de réfléchir en profondeur, d'autant plus que PL1 apparaissait comme un bon langage d'écriture système, d'ailleurs choisi par les architectes de MULTICS au MIT comme un langage de développement, en accord avec leur sponsor General Electric, puis Honeywell, donc de façon sereine et neutre vis-à-vis d'IBM qui était leur concurrent, ainsi que par la NASA avec le langage HAL dans lequel sont programmés les logiciels de la navette. On peut dire que le langage C est un descendant putatif de tout cet effort, mais réduit à l'essentiel, comparé à PL1.

Il fut désormais acquis qu'il fallait systématiquement travailler sur trois niveaux, ce qui se matérialisait par trois étapes de compilation, avec trois interfaces nouvelles matérialisées par les langages intermédiaires LI_1, _2 et _3, comme suit (figure 6.4).

Si l'on considère une expression algébrique usuelle comme :

racine_1 = (–coef _2+SQRT(coef _2xx2–4xcoef _3))/(2xcoef _1)

correspondant à $x' = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

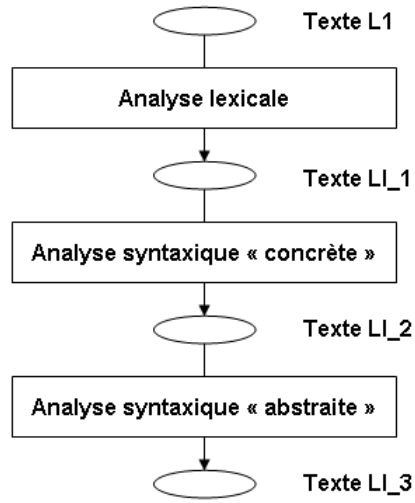


Figure 6.4 - Les trois niveaux d'analyse du texte source

On peut classer les termes de l'expression en :

a- Noms propres et constantes, comme : SQRT, 2, 4.

b- Noms de variables : *racine_1*, *coef_1*, … pouvant contenir de 1 à n symboles alphabétiques ou alphanumériques.

c- Symboles mathématiques avec leurs règles de priorité usuelle.

d- Caractères et symboles spéciaux comme : >=, <=, etc.

De plus, sur les ordinateurs usuels, il n'y a pas d'instruction permettant de faire en un coup, une opération à 3 arguments ou plus comme **4×*coef*_1×*coef*_3**. Les instructions sont, soit unaire, de type {OP, R1} par exemple ajouter 1 à R1, soit binaire, de type {OP, R1, R2} (par exemple multiplier R1×R2 et ranger le résultat dans R2). Il faut donc créer, *pour le besoin du seul ordinateur*, des entités abstraites intermédiaires inconnues de l'utilisateur, de façon à ce que, du point de vue de la machine, le texte soit complet.

La logique de regroupement des traitements sur trois niveaux apparaît donc clairement.

1. Le niveau lexical traite de tout ce qui concerne la structure des noms (ou lexème, en linguistique) ; il identifie les noms propres et les constantes ; il réalise un codage réversible du texte de façon à faciliter les traitements ultérieurs (NB : les structures internes des machines sont optimisées pour manipuler des mots de 32 ou 64 bits, et non pas les chaînes de caractères de longueur variable qui désoptimisent l'unité centrale, car le non alignement des données dû à l'absence de codage peut faire chuter des performances d'un facteur 1.5 à

2 !). La complexité des traitements dépend du nombre de caractères manipulés.

2. L'analyse syntaxique concrète vérifie la syntaxe des expressions, le bon parenthésage, etc. ; elle reconnaît les opérateurs et restructure l'expression de façon à éliminer ce qui ne sert à rien, du point de vue de la machine, en particulier toutes les ( ) vont disparaître. La complexité des traitements dépend du nombre d'entités lexicales et de leur imbrication.
3. L'analyse syntaxique abstraite rajoute tout ce qui était implicite dans le texte initial pour présenter à la machine un texte complet où tout est explicite ; par exemple les tableaux où les types abstraits qui ne sont pas dans les types de base de la plupart des machines sont « digérés » à cette étape. Ce code intermédiaire peut d'ailleurs être interprété comme cela est fait dans certains systèmes (cas du byte-code de la Java-Machine JVM).

Concernant la syntaxe concrète, une première remarque s'impose. Une expression comme celle donnée en exemple ne pose aucun problème d'appréhension à un œil humain qui peut parfaitement lire dans le sens Gauche→Droite ou Droite→Gauche, et même de haut en bas (ou inversement) pour les matrices. Nos années de lycée, depuis des décennies (mais pas depuis des siècles !) nous ont familiarisé avec ce genre d'exercice. L'ordinateur, compte tenu de la structure de l'adressage de la mémoire centrale, ne peut lire que de façon linéaire. Ce qui est naturel à l'œil humain, ne l'est plus pour la machine, il faut donc transformer la notation infixe en autre chose, plus simple pour la machine.

6.1.2 Syntaxe concrète – syntaxe abstraite

Il se trouve que dans les années 30, les logiciens polonais (Lukasiewicz, etc.) ont inventé une notation qui évite les ( ) et le système de points utilisé dans les *Principia*, notation appelée depuis cette époque, **notation polonaise** (pour des raisons typographiques) afin d'éviter les empilements de ( ) que l'on retrouve dans un langage comme LISP et qui rendent ce langage illisible à l'œil humain.

Dans cette notation, une expression comme a = b+c×d va s'écrire :

a- En notation post-fixe : a b c d ×+=, ainsi nommée car les opérateurs sont placé *après* leurs opérandes.

b- En notation pré-fixe : = a+b×cd (les opérateurs sont *avant* leurs opérandes).

Infixe	Post-fixe	Pré-fixe
a = b+c+d	abc+d+=	= a+b+cd
a = b×c +d×e	abc×de×+ =	= a+×bc×de

Cela revient à parcourir le graphe unique de l'expression de différentes façons (figure 6.5) :

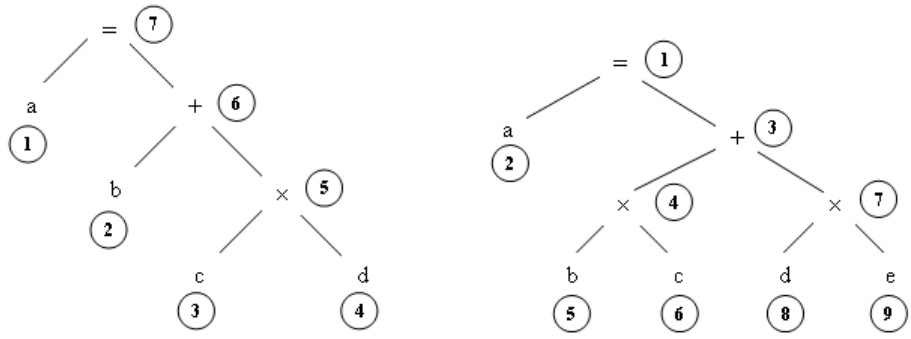


Figure 6.5 - Représentation arborescente des expressions

Le graphe en question, selon qu'il est scruté par un œil humain ou par un ordinateur sera matérialisé sous une forme dont l'usage est le plus commode (et la plus économique en ressource de calcul).

L'architecte du compilateur choisira la notation qui est la mieux adaptée au contexte de traitement, à savoir :

- Pour la structure de contrôle du programme → notation préfixe.
- Pour les instructions impératives (modification des états mémoire) et expression arithmétique et/ou logique → notation post-fixe.

Cette structure à trois niveaux, caractéristique des compilateurs, est tout à fait générale ; on la retrouvera en de multiples occasions. On peut la résumer par le schéma 6.6.

Le second problème résolu par les compilateurs est celui de la sémantique du langage de programmation. Lorsque l'on écrit une expression telle que `a = b + c d` la syntaxe s'applique à des nombres. Dans un ordinateur, il y a plusieurs types de nombres qui sont en fait des nombres modulo *n*, et non pas les nombres réels que l'on a étudié en mathématiques, soit au minimum :

- les entiers signés ou non signés, courts ou longs (soit 4 formats) qui sont des nombres modulo la précision, en représentation binaire,
- les décimaux « packed » ou « unpacked », signés ou non, de longueur variable soit 2x2x*n* formats, *n* étant le nombre de chiffres autorisés,
- les flottants court, moyen, long (soit 3 formats).

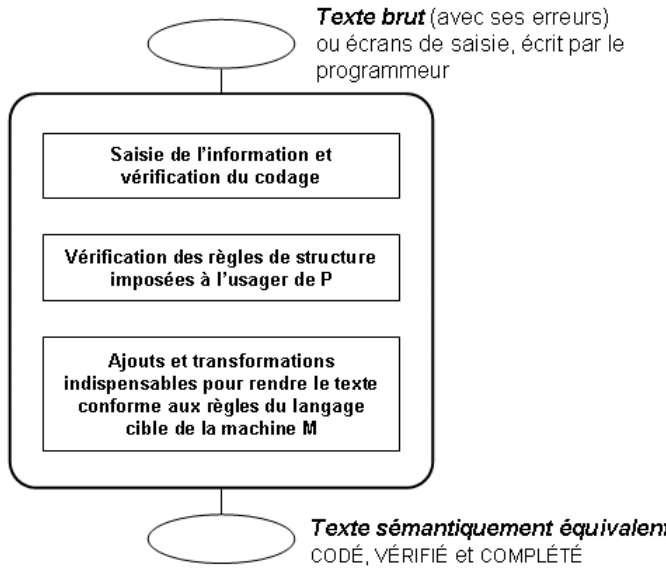


Figure 6.6 - Les trois niveaux de traitements

La même expression arithmétique, selon le type et le format des nombres, aura donc plusieurs représentations en langage assembleur, ce qui est évidemment une gêne considérable pour la maintenabilité des programmes, d'où l'idée fondamentale de séparer clairement :

- la déclaration du type de la variable,
- l'emploi de la variable dans les expressions et les instructions du langage.

La modification d'un type permet la mise à jour du calcul, *sans* modifier l'expression algébrique correspondante ; on dit que l'expression hérite de la sémantique portée par les types.

6.1.3 Les types

Les types abstraits introduits dans les langages Pascal puis Ada, appelés pour cette raison langages fortement typés, permettront d'effectuer des contrôles de nature sémantique beaucoup plus rigoureux car ils permettent aux programmeurs de spécifier leurs propres types, conformément à la sémantique de leurs programmes, pour autant que le concepteur du programme ait effectué une analyse fouillée.

L'expression devient un modèle de calcul qui sera adapté aux différents types de nombres disponibles sur la machine. Cette adaptation est faite par le module d'analyse syntaxique abstraite car elle est susceptible de nécessiter de nombreuses opérations de conversions qui sont implicites dans le texte initial.

La combinatoire des formats, si l'on prend la description ci-dessus, est de : 4x [2x2xn] x3 = 48 xn

Si l'on ne veut pas être débordé par la combinatoire, il est impératif de distinguer le format du nombre, tel que l'usager l'a souhaité, des formats internes adoptés par le compilateur pour tous les résultats intermédiaires.

Par exemple si l'on a les descriptions de variables :

```
a décimal (2)
b décimal (2)
c décimal (2)
```

et que l'on trouve des expressions du type `a = b x c`, il faudra que le format du calcul intermédiaire soit `décimal (4)` pour signaler les débordements de capacité nécessités par les calculs intermédiaires, et ce d'autant plus que le format des nombres provient de conventions adoptées pour le stockage sur disque.

Le rôle de l'analyse syntaxique abstraite est donc :

a- De fournir des modèles de calcul généraux, indépendants de la logique de la machine sous-jacente, qui vont désolidariser le mode de représentation des expressions arithmétiques et/ou logiques usuelles, des règles de calcul propres à une machine particulière. Il est par exemple commode de raisonner avec des opérateurs typés symétriques à trois adresses : `OP-type, R1, R2, R3`. À ce stade de la compilation, il n'y a plus que des opérateurs binaires et des opérateurs ternaires.

b- De rendre explicite tout ce qui était implicite dans l'expression initiale, comme l'adressage et/ou les conversions de formats, la précision requise pour les variables intermédiaires et les réservations mémoire correspondantes.

Ce mécanisme est donc un moyen fondamental d'adaptation des représentations dans le langage de l'utilisateur à n'importe quel type de machine. À ce stade de l'analyse, le texte du programme initial est sémantiquement correct du point de vue de la sémantique du langage de programmation. Tous les calculs et vérifications à effectuer sont explicités. Il n'y a plus qu'à exécuter le programme, soit par interprétation directe du texte, soit par génération de code sur une machine particulière. Dans ce dernier cas, il faut effectuer une dernière transformation qui ne concerne plus que les opérateurs qu'il faut traduire dans ceux de la machine cible.

Le schéma logique d'une telle transformation est le suivant (figure 6.7) :

On voit sur ce schéma que les règles sémantiques mêlent de façon imbriquée :

- les combinaisons de types autorisées par le langage,
- les opérations disponibles sur la plate-forme machine cible,

mais que l'on a un intérêt économique évident à séparer les deux logiques si l'on veut faire évoluer le langage indépendamment de la machine, ou la machine indépendamment du langage.

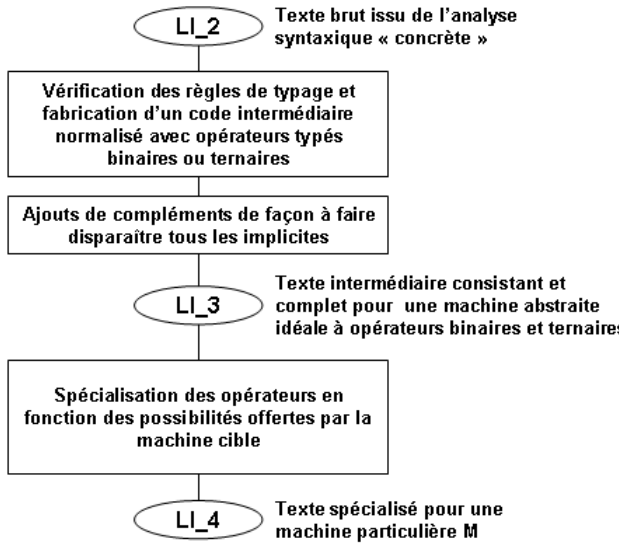


Figure 6.7 - L'analyse syntaxique abstraite

6.1.4 La génération de code et l'optimisation

Le troisième problème est celui de l'obtention du code généré à partir du texte intermédiaire qui n'est au fond qu'une paraphrase du texte initial complètement explicité. La structure du compilateur est alors la suivante (figure 6.8) :

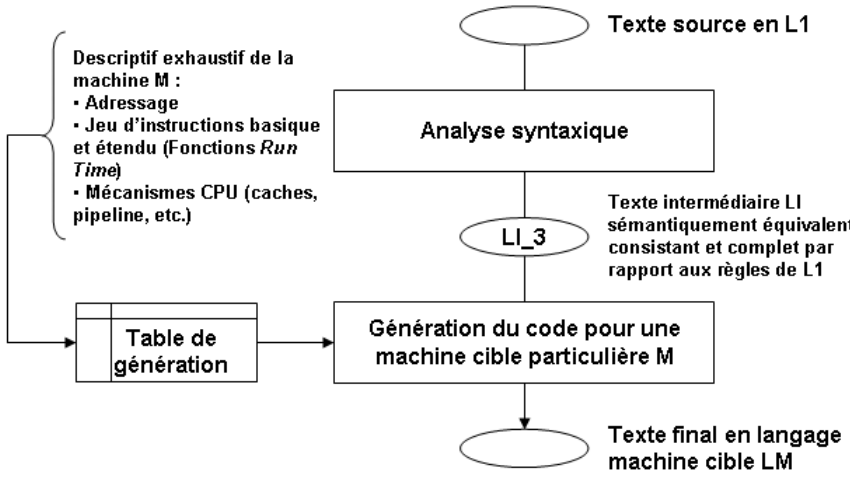


Figure 6.8 - Compilateur complet

Le générateur de code, quant à lui, établit une correspondance entre les instructions du langage intermédiaire LI et les instructions de base de la machine cible M. Chacune des instructions du langage intermédiaire fonctionne comme une macro-instruction, ce qui a d'ailleurs été largement exploité dans les macro-assembleurs des années 60 et 70. Cependant, un tel dispositif ne permet pas de garantir un code machine de bonne qualité du point de vue des performances, surtout avec l'arrivée des langages de haut niveau comme PL1, Pascal, Ada.

Pour être optimale, chacune des séquences doit pouvoir être adaptée au contexte de génération comme par exemple la réutilisation d'un résultat intermédiaire dans un registre.

La combinatoire d'un tel programme est, au minimum, de l'ordre $\mathbf{O}(n_1,n_2,\ldots,n_k)$, mais compte tenu des optimisations locales possibles d'un motif de génération d'une instruction du LI à la suivante, il faut compléter par l'existence d'un contexte qui résulte du traitement des opérations précédentes (sur le schéma, c'est la profondeur d'optimisation), ce qui peut engendrer une complexité exponentielle, soit (figure 6.9) :

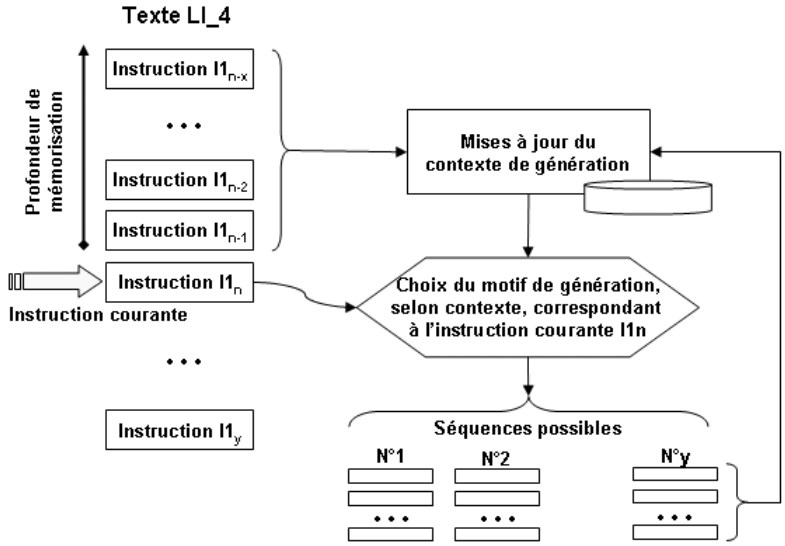


Figure 6.9 - Choix des séquences d'instructions à générer

Dans une telle structure, il faut distinguer clairement entre, a) d'une part la logique de choix de la bonne séquence d'instructions, et b) d'autre part, la prise en compte du contexte de génération afin de soigneusement contrôler le « produit » des deux logiques qui peut être exponentiel. Plus ce contexte est riche et précis, mieux le choix des instructions peut être adapté à la situation, mais plus le générateur de code est complexe et coûteux en termes CQFD. Toutes les alternatives possibles doivent être sémantiquement équivalentes, et l'on peut facilement imaginer que la validation de ce type de programmes relève plus de la démonstration de propriétés que

de tests au sens habituel du terme. Avec l'optimisation globale, on atteint ici les limites de l'ingénierie des compilateurs que la vertu cardinale de prudence incite à ne pas franchir.

Toutefois, une telle structure ne préjuge pas du bien fondé des instructions du langage intermédiaire. Dans tous les langages de programmation nous rencontrons des expressions (cf. les notations indicées du calcul matriciel ou du calcul tensoriel) du type `a(i,j)=b(i,j)+c(i,j)` avec `{a,b,c}` déclarées comme tableaux. Ce traitement des indices `(i,j)` va produire des expressions de calcul d'adresse du type `h1xi+h2xj` qui vont se répéter un certain nombre de fois dans le texte ; idem avec les expressions d'adressage par pointeur qui sont en fait des fonctions d'accès aux variables d'état du programme.

Si l'on veut éviter de générer *n* fois le même texte, ou des textes presque équivalents, il faut que le compilateur soit capable de reconnaître des motifs identiques de façon à les factoriser, ce qui revient à optimiser les ressources disponibles en terme d'instructions réellement utiles, sans dénaturer nos habitudes linguistiques. En fait, l'optimisation du compilateur reconstruit, grâce à des algorithmes ad hoc issus de la théorie des graphes, des sous-programmes, c'est-à-dire qu'il détecte automatiquement des abstractions et en fait la synthèse sur la base de critères économiques comme la taille du programme généré et/ou la vitesse d'exécution. La structure finale du compilateur est alors conforme au schéma 6.10 :

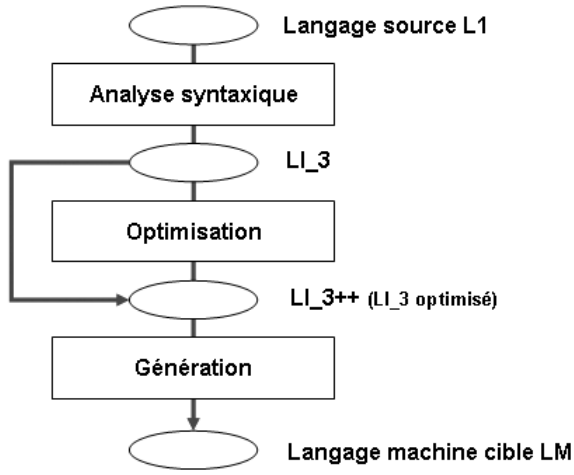


Figure 6.10 - Structure générale d'un compilateur optimisé

Si pour des raisons quelconques, on décide de débrancher l'optimisation (par exemple en mise au point du programme) on voit qu'il est impératif que LI_3 et LI_3++ soient identiques. LI_3++ est simplement un texte plus compact que LI_3.

Cette notion de langage intermédiaire est l'une des plus fondamentales qui soit, et pas seulement dans le compilateur. Elle est liée à la notion de distance entre repré-

sentations. On la retrouve tout aussi bien dans les IHM, dans les éditeurs syntaxiques, dans les systèmes de gestion des bases de données ou dans les couches de protocoles de télécommunication, mais c'est dans les compilateurs qu'elle a, à la fois, sa plus parfaite et sa plus simple (mais non simpliste) formulation, soit environ 15 ans de R&D intense.

Entre un texte en langage de haut niveau, et son équivalent en langage machine cible LM on peut facilement concevoir qu'il soit commode de passer par une succession d'étapes :

$$LS \xrightarrow{T1} LI1 \xrightarrow{T2} LI2 \xrightarrow{T3} LI... \xrightarrow{Tx} LM$$

chacune des étapes étant matérialisée par une interface qui est un langage intermédiaire sur lesquels agissent des fonctions transformatrices T1, T2, …, jusqu'à obtention du texte final.

Une telle structure amène deux questions :

1. Quel est le nombre minimum d'étapes à effectuer pour garantir une traduction optimale du point de vue de l'efficacité du texte généré ?
2. Quel est le nombre raisonnable d'étapes pour garantir une ingénierie optimale, c'est-à-dire un compromis économique en terme CQFD : coût de réalisation, qualité du texte généré, fonctionnalités offertes par le langage et délai de réalisation ?

Par exemple, une structure telle que : $Lx \xrightarrow{T(x\to y)} Ly$ remplacée par $Lx \xrightarrow{T(x\to u)} Lu \xrightarrow{T(u\to y)} Ly$ aura un coût et un délai de réalisation plus faible que la transformation T(x→y) si les transformations T(x→u) et T(u→y) sont indépendantes et peuvent être développées en parallèle.

Il s'agit ici d'un vrai choix architectural, car c'est la seule présence de *Lu* qui garantit le parallélisme de réalisation deT(x→u) et T(u→y).

6.1.5 La notion de langage intermédiaire pivot

On peut dire que l'architecture des compilateurs se réduit a) au choix d'un jeu de langages intermédiaires qu'il faudra spécifier de façon rigoureuse, et b) au placement de fonctions transformatrices les mieux adaptées aux contraintes propres de l'ingénierie des compilateurs, c'est-à-dire les traitements liés au langage de programmation et ceux liés à la machine cible pour laquelle on souhaite compiler. Ceci conduit à une structure idéale du type (figure 6.11) :

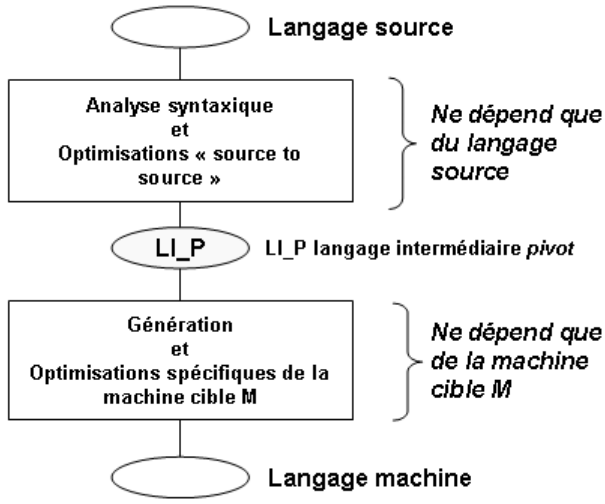


Figure 6.11 - Langage intermédiaire pivot

Une telle structure permet de construire des compilateurs multi-langages et multi-cibles, qui voient le jour dans les années 80 (cf. l'architecture des compilateurs du projet GNU, de la *Free Software Foundation*), comme suit (figure 6.12) :

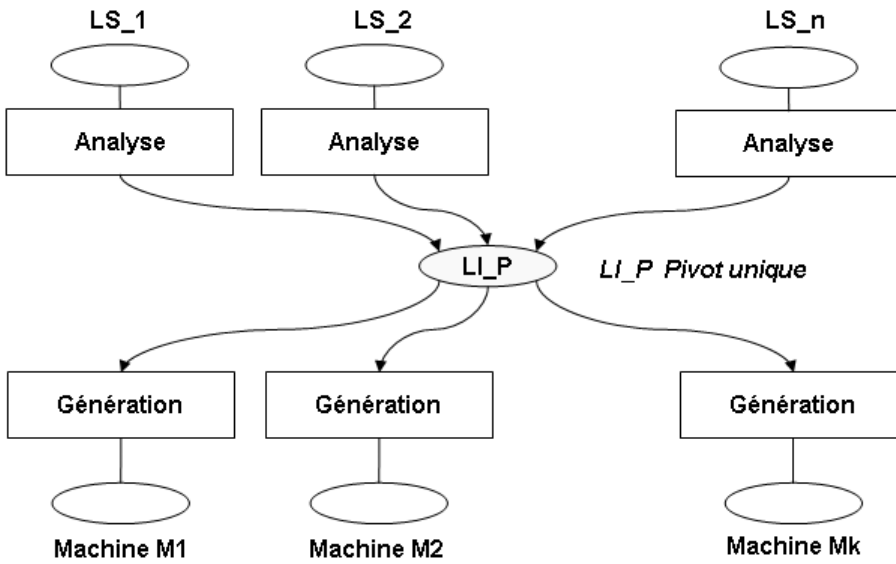


Figure 6.12 - Compilateurs multi-langages et multi-cibles

Dans une telle architecture, tout le savoir-faire et la sémantique du langage qui en est le résultat, se concentre dans la structure du langage intermédiaire pivot LI_P.

Pour un architecte de compilateur focalisé sur UN langage source et UNE machine cible, il est très tentant de faire référence, dès la phase d'analyse syntaxique à la structure de la machine M, et d'utiliser dans le générateur de code des spécificités propres au langage L, même s'il existe un niveau intermédiaire équivalent à LI.

Une telle approche rend impossible un dispositif LI_P comme ci-dessus, sinon au prix d'une complication très grande des phases d'analyse et de génération. Dans ce cas, la vraie structure en terme de dépendances fonctionnelles est la suivante (figure 6.13) :

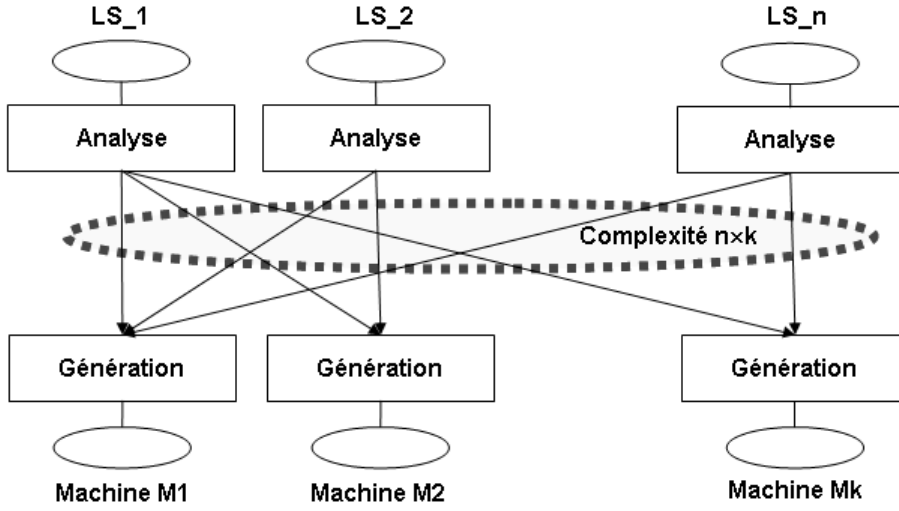


Figure 6.13 : Dépendances fonctionnelles

Dans un tel schéma, la structure intermédiaire LI n'est pas autonome (i.e. complet, au sens de la logique) et ne constitue pas un véritable langage. Une notation est un langage dans la mesure où elle se suffit à elle-même. Dès lors que pour comprendre le sens il faut faire appel à des informations qui ne font pas partie de *LI*, alors la structure *LI* ne peut pas être qualifiée de langage.

De fait, tout compilateur d'un langage L doit pouvoir être écrit en L, ce qui est une exigence pour pouvoir certifier un compilateur Ada. Une interface entre deux modules M1 et M2 n'est véritablement utile que si elle a une structure de langage, c'est-à-dire qu'elle contient tout et rien que l'information nécessaire au traitement de M2, c'est du moins la grande leçon à tirer de la technologie des compilateurs.

On peut justifier la nécessité de ce type d'interface pour les raisons de fiabilité (voir chapitre 12).

Soit par exemple une structure de programmation comme suit (figure 6.14) :

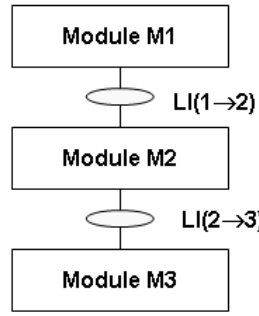


Figure 6.14 - Encapsulation de modules à l'aide d'interfaces

Si une défaillance apparaît dans M3 et que M3 ne dépend que de LI(2→3) et de lui-même, alors le défaut est à chercher uniquement dans LI(2→3) ou dans M3. Si ce n'est pas le cas, le défaut est dans M1 ou M2, ou LI(1→2), ou dans une combinaison de ce qui peut se passer dans chacune de ces structures (NB : l'ensemble des parties est à considérer).

On voit donc que la présence de langages à des endroits ad hoc de l'architecture est une façon d'étanchéiser les modules M1, M2 et M3 (et de réduire la quantité d'information globale de la structure, c'est-à-dire la complexité). Ces interfaces constituent les langages « cachés » de l'architecture. De plus, la présence de vrais langages tel que LI(2→3) permettra de simuler la présence de M1 et de M2 pour autant que l'on sache construire des textes conformément à la syntaxe et à la sémantique de LI(2→3). Le simulateur se réduit alors à un simple éditeur de langage, ce qui est une technologie parfaitement maîtrisée.

Il y a, à minima, trois grandes leçons à tirer de l'architecture des compilateurs :

- **Règle N°1** : Transformation par étapes successives matérialisées par de vrais langages autonomes qui matérialise le travail d'abstraction effectué par l'architecte. Ces langages intermédiaires sont des interfaces ultra-stables.
- **Règle N°2** : Respect absolu des domaines sémantiques (celui du langage, celui de la machine dans le cas des compilateurs).
- **Règle N°3** : Contrôles des enchaînements effectués à l'aide du couple dual grammaire /automates de façon à bien séparer la structure d'entraînements de la structure de traduction.

6.2 CAS DES MÉTA INFORMATIONS

Tous les langages de programmation contiennent des instructions comme :

```
MEASURE (structure), DISPLACEMENT (… … …)
```

Ces instructions sont indispensables pour paramétrer correctement certains programmes. En effet, si le programmeur de langage de haut niveau LHN doit manipuler la taille d'un objet alloué par le compilateur, il est fondamental que le compilateur lui communique ce qui a été effectivement alloué. D'où les ordres en question. NB : la pire erreur serait de les calculer « à la main » comme le font certains programmeurs qui n'ont rien compris.

Du point de vue de l'architecture du compilateur, cela veut dire que certaines constantes sont le résultat de tout ou partie du processus de compilation (i.e une méta constante). Ce qui conduit à des situations où le résultat du processus de compilation (i.e le programme en langage cible LM) dépend du processus de compilation lui-même. Cette forme d'auto-référence est, si l'on n'y prend pas garde, des plus gênantes pour plusieurs raisons :

- Il n'est pas évident que le processus converge, ce que l'on pourrait exprimer par une formule comme :

Compilation ($\boldsymbol{P_{L1}}$, , Constantes_de_ $\boldsymbol{P_{L1}}$) $\rightarrow$ $\boldsymbol{P_{LM}}$ + Constantes_de_ $\boldsymbol{P_{L1}}$

Ce qui exprime le fait que les constantes sont un résultat de compilation qu'il faut réinjecter dans le programme à compiler. Soit, en termes plus simples, des objets X tels que F(X)=X. La solution d'une telle équation, quand elle existe, s'appelle un *point fixe*. C'est d'ailleurs le problème général de l'optimisation qui ne doit pas désoptimiser d'autres régions du programme. On peut imaginer que le critère de convergence est rien moins que simple, car les optimisations sont parfois contradictoires. Ce qu'il faut obtenir, c'est un équilibre statistique du type de ceux que l'on calcule en théorie des jeux (équilibre de Nash).

L'ensemble des entités qu'un programmeur normalement constitué qualifiera de constantes peut ne pas être évident à décrire, et ce d'autant plus que les machines sous-jacentes ont généralement des instructions qui manipulent certaines constantes (appelées parfois constantes immédiates) qui, de ce fait, n'ont pas besoin d'être allouées dans la partie donnée de la mémoire.

Rien n'est plus gênant pour la lisibilité et la compréhension d'avoir à considérer des catégories d'entités dont la classification dépend de critères qui ne sont pas dans la logique du programmeur.

La catégorie « constante » se subdivise, pour le compilateur, en deux sous-ensembles :

1. Constantes immédiates (dépendent du jeu d'instruction)
2. Constantes allouées, soit :
   – $\rightarrow$ constantes explicites (déclarées par le programmeur),
   – $\rightarrow$ constantes implicites (calculées par le compilateur).

On voit que la notion apparemment simple de constante dépend en fait de trois choses :

1. de ce que le programmeur a écrit dans son programme,

2. de la structure du jeu d'instruction de la machine cible,
3. du processus de compilation lui-même pour les constantes « calculées ».

Selon l'architecture du compilateur adopté, il faudra un, deux, ou trois types de traitements pour résoudre un « simple » problème de compilation et manipulation des constantes ; donc, dans le pire des cas, 3 traitements pour faire la même chose, et donc 3 fois plus de possibilités d'erreurs ! Sans compter les interactions.

Dans la logique du programmeur, certaines « constantes » fonctionnent comme des abréviations ; ce qui revient à dire, par exemple : chaque fois que vous rencontrez « `pi` », remplacez-le par 3.14116, ou « `TVA` » par 0.206.

Ces deux constantes ont cependant un statut sémantique bien différent. Il est peu vraisemblable que π puisse valoir autre chose que 3.14… (sauf en géométrie non euclidienne !), c'est une vraie constante (i.e. un nom propre, en logique) alors que la TVA est susceptible de changer. La question de savoir comment il faut adapter le programme est très intéressante, car deux possibilités sont offertes :

1. modifier le texte du programme, ce qui amène à le recompiler (et donc de valider à nouveau),
2. modifier le contenu d'une variable qui contient la valeur de la constante.

Dans le second cas, on peut être tenté de programmer comme suit :

```
TVA = 0.206
… …
Montant_TTC = Montant_HT¥(1+TVA)
```

Un compilateur un peu malin s'apercevra sans difficulté que l'expression `1 + TVA` est un substitut pour `1+0.206` et que la bonne constante pour l'instruction N°2 est en fait 1.206, ce qui ramène au cas 1, probablement contre la volonté du programmeur dans ce cas particulier. Pour éviter cela, le programmeur devra débrancher certaines options d'optimisation concernant la propagation des constantes.

La notion de constante dépend cette fois de l'usage qui en est fait dans telle ou telle partie du programme, ce qui rajoute une dépendance vis-à-vis de la structuration du programme.

Pour qu'une notion soit facile d'emploi, il ne faut pas que son usage dépende de critères qui, du point de vue du programmeur, sont arbitraires comme la machine sous-jacente ou le fonctionnement du compilateur. Une des notions les plus puissantes introduites dans les langages de programmation est celle de « type ». Pour rester sur les constantes examinons comment cette notion de type clarifie le contexte d'emploi de certaines constantes et rajoute de l'ordre (au sens de Thomas d'Aquin).

Des langages comme Pascal, Ada, JAVA offrent des constructions comme :

```
Type couleur (bleu, blanc, rouge)
```

ce qui permet de déclarer des variables comme

`a : couleur //` (ce qui veut dire que ça ne peut valoir que `bleu, blanc ou rouge`).

et d'écrire des instructions comme :

```
IF a = bleu THEN ..... ELSE ...
```

Dans les langages qui n'ont pas cette construction il faut faire des conventions et dire que `bleu` sera représenté par `1`, `blanc` par `2` et `rouge` par `3`. Au lieu d'écrire :

```
IF a = bleu ... ; on écrira IF a = 1 ...
```

ce qui ne permet plus de distinguer le « `1` » qui veut dire « bleu » des autres « `1` » ! la quantité d'information attachée à « `1` » est plus forte que celle attachée à « bleu » car tous les « `1` » ne sont pas « bleu ». L'entropie textuelle du programme sans type est donc plus élevée que celle du programme avec type, même si ce dernier est plus long car il faut déclarer les types. Un langage typé est donc plus précis, sémantiquement parlant, qu'un langage non typé équivalent. La quantité d'information algorithmique est plus importante, le texte est plus long. Le coût correspondant est cependant quasi marginal : c'est le prix à payer d'un paramétrage qui sera bien utile en maintenance.

Si le sens attribué à la constante est purement lexical, alors la constante peut être traitée dès la phase d'analyse lexicale. Si le sens de la constante dépend du contexte d'emploi dans le programme, alors la constante ne peut être traitée que dans la phase de syntaxe abstraite, là où l'on dispose de toute l'information sur la structure du programme. Si, de plus, la machine dispose d'instructions immédiates, certains traitements ne pourront être faits qu'à la génération.

Cet exemple simple montre que les choix architecturaux initiaux prédéterminent la façon dont le langage pourra évoluer (sauf à refaire le compilateur !).

Si l'on a décidé de traiter les constantes dès la phase lexicale comme le bon sens pourrait le laisser croire, alors il sera très difficile d'introduire ultérieurement des notions comme les types énumératifs. De même, la structure des langages intermédiaires va permettre un enrichissement progressif du texte, et ce, en faisant abstraction de la façon dont les textes sont traduits. Cela revient à dire que tout ce qui peut advenir au texte *ne dépend que de lui seul.*

On peut dire que l'Etape *i* voit la suite des opérations dans et rien que dans la syntaxe et la sémantique du texte $\boldsymbol{LI_{i+1}}$. Ce texte intermédiaire est donc l'organe *exclusif* de communication entre les différentes étapes.

À partir de l'exemple du traitement des constantes, on voit se dessiner l'un des aspects fondamentaux de l'architecture, à savoir la suite des fonctions de traduction à opérer, lesquelles dépendent à leur tour des *LI* adoptés. On voit la relation qui peut alors exister entre le volume de ces fonctions, le lieu où l'on peut les déclencher (c'est un problème de placement) et la structure du langage intermédiaire. L'art de l'architecte est de découvrir le meilleur compromis entre ces trois structures, dans le respect des contraintes CQFD du projet. On peut dire que l'architecture est à la recherche d'un équilibre entre différents critères antagonistes. Une mauvaise appréciation des critères peut rendre la solution envisagée caduque.

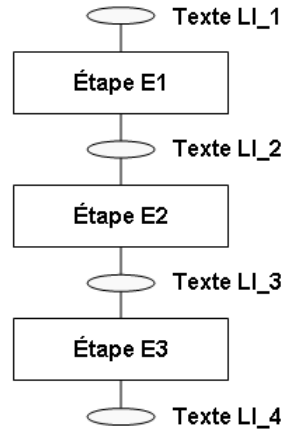


Figure 6.15 - Transformation par étapes successives

L'étude des langages et des compilateurs est le meilleur moyen de comprendre la relation parfois subtile entre syntaxe, sémantique, métalangage et les règles de transformations qui permettent de passer du langage source au langage machine par étapes successives. Les langages intermédiaires sont des interfaces fondamentales du compilateur.

7

Architecture des processus et de leurs interactions dans une machine

Le concept de processus et la communication inter-processus sont aux sciences de l'information ce que la physique des particules est à la science de la matière.

- Les processus sont « des grains » de matière informationnelle, de différentes tailles, comme le sont les fermions dans la matière physique (i.e. les protons, les neutrons, les électrons, etc.).
- Les messages qui s'échangent entre les processus, lors d'une communication entre processus, et qui rendent les processus solidaires au sein d'une même application, sont analogues aux bosons qui sont les particules qui assurent la cohésion de la matière aux moyens des différentes forces que nous connaissons (i.e. la gravitation, l'électromagnétisme, les forces nucléaires, etc.).

La notion de processus de programmation émerge très tôt dans le développement de l'informatique, dès le début des années 60 avec la multiprogrammation, et quelques années plus tard, le *time-sharing* et le transactionnel. Dès le départ, on savait que les organes périphériques de la machine fonctionnaient beaucoup plus lentement que l'unité centrale, d'où l'idée de réutiliser les temps morts, correspondant aux attentes d'entrées-sorties, pour avancer l'exécution d'un autre programme présent dans la mémoire de la machine. C'était une façon de rentabiliser le matériel, très cher à l'époque. Pour exploiter cette possibilité, il faut que la machine soit dotée d'organes de traitements asynchrones[1] pour toutes les opérations concernant la périphérie, une ou plusieurs unités centrales (CPU, *Central Processor Unit*) selon la puissance souhaitée et un pilote qui orchestre l'ensemble.

Du point de vue de la machine, nous avons plusieurs organes d'exécution matériels, et une collection de programmes en mémoire auxquels vont être alloués les organes d'exécution, en fonction des évènements rencontrés en cours de traitement. Cette allocation est faite par un organe matériel-logiciel (au départ, des micro-programmes) qui pilote l'ensemble : le dispatcher (cf. la notion de micro-noyau /micro-kernel dans certains systèmes).

NB : Dans une machine mono-programmation, cette fonction était assurée par l'opérateur de la machine. Si l'on souhaite gérer plusieurs programmes, il faut faire rentrer l'opérateur dans la machine : c'est le rôle du dispatcher.

Le schéma de principe est le suivant (figure 7.1) :

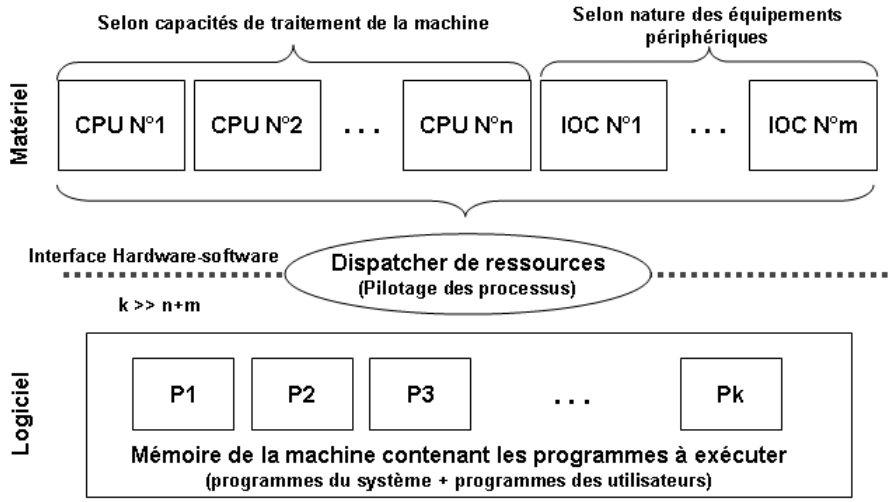


Figure 7.1 - Dispatchings des ressources matérielles

7.1 LE CONCEPT DE PROCESSUS

Le concept de processus donne un sens très précis à la notion de programme qui est et reste un concept flou, au contour mal défini.

Un processus est une unité de transformation élémentaire, séquentielle par définition, dont la configuration est connue. Le processus interagit avec son

1. Historiquement, la première machine asynchrone fut le Γ60 de Bull, projet qui mis la compagnie Bull en faillite, et occasionna son rachat par General Electric en 1964 ; voir J. Bouboulon, *L'aventure Gamma* 60, Lavoisier, 2005.

environnement ; il peut émettre et recevoir des évènements, sous contrôle d'un moniteur de processus qui pilote l'ensemble des processus actifs. Un processus contient a) une partie impérative, qui sont les instructions reconnues par l'organe de traitement qui exécute les instructions dans l'ordre défini par le programmeur (NB : la terminologie a été très fluctuante : on a parlé de « routine », de « procédure », etc., aujourd'hui de « méthode »), et b) une partie donnée qui définit l'état mémoire du processus tout au long des transformations effectuées pendant l'exécution du processus.

NB : on remarquera l'analogie avec la notion d'unité active introduite dans la partie 1.

La taille d'un processus, en nombre d'instructions machines et de données élémentaires est quelconque et ne dépend que des choix et décisions du/des programmeurs. Un composant applicatif, du point de vue de l'utilisateur peut regrouper un ou plusieurs processus, qui opèrent de concert, en se synchronisant conformément à leur programmation. Le processus est également une unité de construction, tant du point de vue du processus d'intégration (i.e. du point de vue métier) que des équipes de développement dont l'unité de livraison « naturelle » est le processus.

Au départ, les machines ont un seul CPU et des IOC pour les mémoires externes (bandes magnétiques, disques), la console système, le lecteur de cartes perforées, l'imprimante. C'est du côté des programmes que les choses se gâtent.

La facilité offerte par la multiprogrammation est contrebalancée par un accroissement de complexité. Ce qui était une histoire séquentielle simple avec les premières machines, pilotées par un automate déterministe, devient un ensemble d'histoires enchevêtrées globalement non déterministes. Un programme P avec des défauts peut contaminer d'autres programmes, et rendre l'ensemble globalement incohérent et non testable. Or comme tous les programmes ont des défauts… le chaos semble inéluctable. Le prix à payer de ce qui est objectivement une vraie facilité nouvelle, dont l'utilisateur est le bénéficiaire, va être la mise en place d'un certain nombre de concepts nouveaux, comme les processus et les sémaphores, de façon à garantir un déterminisme au moins local, permettant de construire de façon méthodique des programmes de grande taille dont on pourra garantir la qualité.

Pour simplifier la vie du programmeur et lui épargner une programmation complexe, les constructeurs de machines vont fournir avec le matériel un ensemble de programmes système pré-définis constituant le système d'exploitation. La rupture se fait vers la fin des années 60, avec le premier grand système d'exploitation réalisé par IBM[2], avec beaucoup de difficultés. Cette histoire est relatée dans le livre célèbre de F. Brooks, architecte de ce système, « *The mythical man-month* », qui est un grand classique du génie logiciel, toujours réédité et admiré.

2. Sur cette histoire, voir : C. Bashe et al., *IBM's early computers*, MIT Press, 1986, et E. Pugh et al., IBM's 360 and early 370 Systems, MIT Press 1991.

Les processus vont jouer un rôle fondamental dans toutes ces opérations en fournissant le matériau essentiel à la construction de ces premiers grands programmes, avec une double assurance qualité au niveau :

- du processus élémentaire (i.e. nos intégrats de rang 0) résultant de la programmation,
- des agrégats de processus, résultat du procédé d'intégration mis en œuvre conformément à l'architecture du logiciel.

Le processus est un espace de confinement, protégé par le hardware de toute contamination extérieure à lui-même et dont on ne peut entrer ou sortir que par des mécanismes parfaitement contrôlés qui sont les « ports du processus ». Pour reprendre la terminologie de la première partie, le processus est une sphère de contrôle élémentaire.

Organisés en couches ou en clients-serveurs, on peut former des structures hiérarchiques (cf. les articles de E. Dijkstra[3] et H. Simon[4]) dont l'augmentation de complexité en fonction de la taille reste humainement gérable, en gros en O(n×log (n)). Si ces principes ne sont pas respectés, la complexité devient exponentielle, donc ingérable (voir chapitre 12).

Nous allons nous intéresser à deux aspects de ces processus qui ont été au cœur des architectures de systèmes d'exploitation comme MULTICS, GCOS7, VAX/VMS, etc. :

- l'espace d'adresse du processus,
- les sémaphores et la communication entre les processus.

Nous expliquerons en quoi ces deux concepts architecturaux ont été des éléments simplificateurs, i.e. des concepts simples mais profonds, qui ont permis la construction de systèmes dont la taille a atteint environ cinq millions de lignes source pour GCOS7 qui restent à ce jour le plus grand programme réalisé en France.

Dans un serveur moderne de grande puissance, il peut y avoir, à l'instant t, plusieurs milliers de processus présents à des titres divers dans la machine, dont certain en cours de validation donc comportant des défauts, sans que cela nuise à la continuité du service de l'ensemble. Sans les processus, et sans la méthode pour en faire un usage correct, une telle performance serait inatteignable.

7.1.1 Topologie d'un processus

L'espace d'adresse définit la partie statique du processus et ses éléments constitutifs. Il contient toutes les données et instructions qui permettent d'effectuer les traitements qui lui sont dévolus.

3. Cf. son texte *Complexity controlled by hierarchical ordering of function and variability*, de la conférence OTAN Software engineering, Garmisch, 1968.
4. Cf. son livre *Science of the artificial*, MIT Press. Réédition 1996.

Un processus est fait de deux parties :

1. Le bloc de contrôle du processus, PCB dans le jargon (*Process Control Block*).
2. Un ensemble de « pages » mémoire qui contiennent les données et les instructions du processus, appelé espace d'adressage.

Un processus existe dans le système s'il a un PCB. L'ensemble des processus qui existent à l'instant t est défini par l'ensemble des PCB. Le nombre de processus est limité par le hardware. Un PCB contient toute l'information nécessaire à sa mise en exécution par la machine, sur un CPU ou sur un IOC. Du point de vue de la machine, c'est une machine virtuelle qui devient réelle lorsqu'on lui associe un processeur (CPU) ou un contrôleur d'entrée-sortie (IOC). Sur une machine comme le DPS7 la taille d'un PCB variait de 164 à 296 octets selon la configuration. La notion de PCB, au sens de l'interface hardware-software, était doublée d'une notion équivalente du point de vue du système d'exploitation contenant beaucoup plus d'informations nécessaires à la surveillance de la machine et de son logiciel (cf. notion de PGCS, *Process Group Control Segment*). Sans entrer dans des détails qui n'auraient aucun intérêt pour cet ouvrage (cf. les ouvrages classiques de Henessy-Patterson[5], qui donnent beaucoup plus détails) donnons un aperçu sur les attributs logiques associés à l'entité PCB.

La structure logique d'un PCB est la suivante (figure 7.2) :

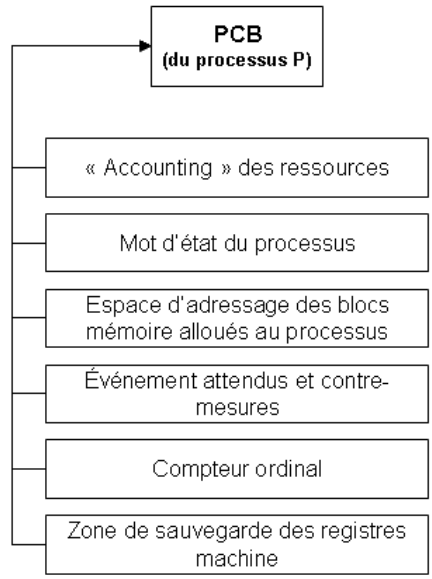


Figure 7.2 - Structure logique d'un PCB

5. Cf. les ouvrages : *Computer architecture, a quantitative approach* et *Computer organization and design, the hardware/software interface*, tous deux chez Morgan Kaufmann.

Accounting

La partie accounting contient un certain nombre d'horloges (i.e. les *timer*) qui permettent de connaître le comportement temporel du processus :

- H1 : durée de vie dans l'état ready (prêt).
- H2 : durée de vie dans l'état running (en cours).
- H3 : durée de vie dans l'état waiting (en attente).
- H4 : durée de vie résiduelle, le time-out.

La somme H1+H2+H4 définit ce qu'on appelle le temps écoulé (i.e. *elapsed time*) du point de vue d'un opérateur externe.

H2 est le temps CPU consommé par le processus ou le temps d'occupation d'un CPU.

H4 est une durée maximale assignée au processus. Le temps est décrémenté jusqu'au passage à 0 (zéro) qui déclenche un événement. Cette horloge permet de répartir les ressources CPU équitablement, en fonction des règles et des priorités d'attribution.

Au niveau du système d'exploitation, cette partie est considérablement renforcée pour effectuer l'accounting général (*capacity planning*, *system management*, etc.).

Etat du processus

C'est un ensemble d'information qui détermine l'état courant du processus, comme :

- La priorité du processus (sur une machine comme le DPS7, il y avait 16 niveaux de priorité possibles).
- L'état du processus du point de vue de l'exécution, soit quatre états principaux (et des sous-états) : READY (le processus est prêt à l'exécution), RUNNING (le processus est en cours d'exécution, un processeur, CPU ou IOC, lui a été alloué), WAITING (le processus est en attente d'un événement généré par un autre processus), SUSPENDED (le processus est à l'arrêt).
- Le processus nécessite une capacité de calcul scientifique, ou un jeu d'instructions étendu, ou nécessite telle particularité hardware, etc.

Espace d'adressage

Ensemble d'information définissant, la structure et l'organisation de l'espace d'adresse des entités en mémoire. C'est en fait un modèle conceptuel de données. Nous y reviendrons plus en détail ci-après.

Evènements attendus et contre-mesures (exception, interruptions)

C'est un ensemble d'attributs très importants du point de vue de la sûreté de fonctionnement du processus.

Lorsque le processus est en cours d'exécution (i.e. état RUNNING) un certain nombre d'anomalies d'exécution peuvent survenir ; elles dépendent du type d'instructions et des données référencées, et plus généralement du type de traitement et de l'état des ressources. Toutes les anomalies possibles sont répertoriées et organisées en classes de façon à associer une réponse, i.e. le driver d'exception qui convient, à telle ou telle classe d'évènements, et ceci, localement à chaque processus.

La cellule évènements pointe vers une table (i.e. un référentiel) qui contient toutes ces informations. Il est possible de la modifier en cours d'exécution pour adapter la réponse. Elle peut être différente d'un processus à l'autre. L'information correspondante correspond à ce que nous avons appelé vecteur d'état dans la partie 1.En architecture distribuée, cette table est fondamentale car elle donne la nomenclature des événements connus, observables et contrôlables, c'est-à-dire de ce qui peut survenir dans la machine du point de vue du comportement. Un événement non répertorié est une incertitude majeure qu'il faut diagnostiquer le plus rapidement possible, ce qui implique un style de programmation constructif que l'architecte devra faire respecter.

Compteur ordinal

C'est un registre de la machine contenant l'adresse de l'instruction en cours d'exécution. En fin d'exécution, ce compteur indique la prochaine instruction à exécuter. Une instruction comme `GOTO label` est une façon de modifier la valeur du compteur ordinal.

Zone de sauvetage des registres

Sur une machine comme le DPS7 (comme sur MULTICS et beaucoup d'autres machines) il y avait trois catégories de registres : registres de base (pour l'adressage), registres généraux (calculs arithmétiques, indexage des tableaux) et registres scientifiques (calculs en nombres flottants).

Lorsque le processus passe à l'état RUNNING, les informations contenues dans le PCB permettent d'initialiser les registres hardware correspondant à l'état du processus. L'inverse est réalisé lorsque le processus repasse dans l'état WAITING ou SUSPENDED.

La configuration initiale du processus est déterminée par l'état READY, qui correspond au démarrage des opérations.

7.1.2 Organisation de l'espace d'adressage d'un processus

C'est un des problèmes les plus difficiles auxquels ont été confrontés les architectes des machines et des systèmes d'exploitation. Autour de cette organisation gravite des problèmes comme : la structure du jeu d'instructions, l'adressage des instructions et des données, la structure des accès à la mémoire et la largeur des bus, la taille des pages (en relation avec la gestion de la mémoire virtuelle et la structure de l'espace disques), etc.

Tout l'équilibre de l'interface hardware-software dépend de cette organisation, ainsi que les performances de la machine et ses capacités d'évolution. C'est un enjeu économique de première importance pour quiconque prétend réaliser des machines.

Le problème logique sous-jacent est général, on le retrouve tout aussi bien dans la définition d'une machine virtuelle comme la JVM du langage JAVA ou dans la définition du modèle de données d'une application transactionnelle où les transactions ACID jouent le rôle des instructions machine (cf. chapitre 9).

Plusieurs choix sont possibles pour l'organisation de l'espace d'adresse, selon qu'on le considère du point de vue hardware ou du point software.

Organisation en pages

Turing disait : *« Keep the hardware as simple as possible »* ; il n'avait pas tort, surtout avec les machines de son époque, mais cela revenait à mettre toute la complexité du côté software, ce qui n'était pas trop gênant compte tenu de la taille, en nombre d'instructions, des premiers programmes.

Du point de vue du hardware, la structure la plus simple est le tableau, où les cellules de rangement de l'espace d'adressage sont des pages de taille fixe. Adresser une cellule revient à donner l'index de cette cellule dans le tableau des pages.

Dans cette organisation l'espace d'adressage du PCB est un tableau de pages. L'organisation de l'espace d'adresse, très simple, est la suivante (figure 7.3) :

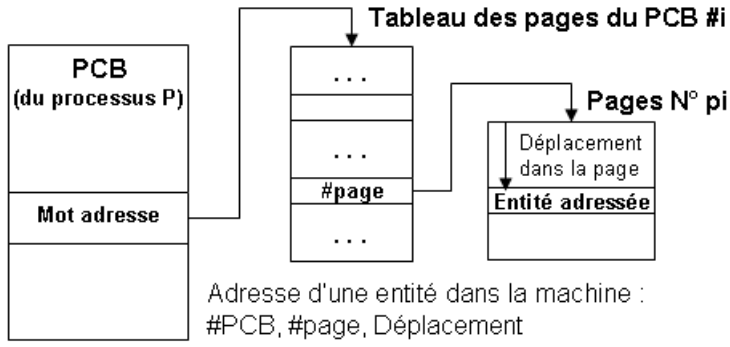


Figure 7.3 - Espace d'adressage en pages

Dans un tel espace, une adresse d'entité (donnée ou instruction) est repérée par deux nombres (# page, D/déplacement dans la page). On ne peut pas faire plus simple, ce qui est parfait pour le hardware et suffisant pour le calcul scientifique qui manipule essentiellement des tableaux de nombres.

Les choses se compliquent avec l'arrivée des premières bases de données, vers la fin des années 60 ; avec l'arrivée du *time-sharing*, et plus généralement du traitement

conversationnel qui prend son essor avec le transactionnel (IMS chez IBM et IDS2/TDS chez Bull, General Electric et Honeywell, rivaux d'IBM dans le BUNCH[6]).

Avec les bases de données arrive le problème des caches, sans lesquels les temps d'accès disques seraient rédhibitoires en terme de performance, selon le schéma 7.4 :

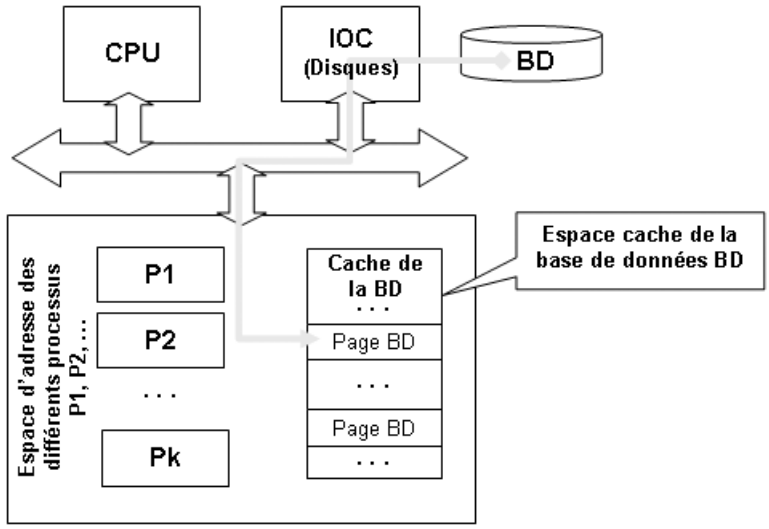


Figure 7.4 - Cache de base de données dans un espace pagé

Le remplissage du cache est aléatoire ; il dépend des accès qui eux-mêmes dépendent de l'extérieur. À qui appartient le cache ? A aucun des processus ou à tous ? Si on veut que les accès effectués par P1 bénéficient également à P2, il faut que P2 « voit », via son espace d'adresse, ce qui a été fait par P1, et réciproquement.

C'est une énorme complexité qui incombe, dans cette organisation de l'espace d'adressage, au software. On notera au passage que l'organisation des pages de la base de données doit être compatible avec l'organisation des pages en mémoire centrale et en mémoire virtuelle.

Même problème, en un peu plus complexe, avec les premiers systèmes en temps partagé (*time-sharing*) qui voient le jour vers la même époque. Chaque utilisateur a besoin a) d'un espace de travail qui lui est propre, c'est-à-dire des données et des programmes qui lui appartiennent, et b) des outils comme les compilateurs qu'il partage avec tous. Plusieurs processus sont nécessaires à chacun des utilisateurs. Là encore, toute la complexité est reportée du côté du software. Les premiers systèmes en *time-sharing* étaient plutôt du style « ça tombe en marche » ; et c'est l'époque où l'on commence à parler de crise du logiciel.

6. Burroughs, Univac, Nixdorf, Contro Data, Honeywell : beaucoup ont disparu ou ont abandonné les ordinateurs.

Les systèmes transactionnels, dont tous les bons stratèges ont compris la nécessité économique, pour faire véritablement décoller l'industrie informatique, cumulent toutes ces complexités, en les aggravant, car le nombre d'utilisateurs connectés augmente considérablement : plusieurs milliers, voire des dizaines de milliers, alors qu'en *time-sharing* on était dans la centaine.

Le nombre de processus physiquement possibles, n'est même plus suffisant pour donner un nom unique à chacune des entités présentes dans la machine ; il faut multiplexer les noms disponibles avec tous les risques d'ambiguïté que cela comporte. La sécurité, la sûreté de fonctionnement, la disponibilité deviennent des enjeux majeurs.

Conséquence : la simplicité initiale fait la part trop belle au hardware, il faut complètement revoir l'équilibre de l'interface hardware-software, et singulièrement la structure de l'espace d'adresse.

Organisation en segments et pages

C'est à cette époque, que, sous l'égide de la DARPA, le projet MULTICS voit le jour au MIT, avec General Electric qui vient de racheter Bull, comme partenaire industriel. Avoir un lien direct avec le MIT, c'est dialoguer avec les inventeurs de toute la technologie informatique. Ce n'est pas le lieu de raconter ici le rôle du MIT dans ce développement, mais il fut énorme[7].

C'est donc une chance inespérée pour les ingénieurs de Bull de travailler dans un tel contexte, dans la proximité la plus étroite avec les créateurs de ce système unique en son genre, un vrai chef-d'œuvre, au sens du compagnonnage.

Du point de vue de l'espace d'adresse, une nouvelle organisation a été testée et validée avec MULTICS. L'interface hardware-software du DPS7 va en bénéficier, ce qui en fait une des machines les plus modernes dans sa catégorie. Ce que proposent les architectes du MIT est une vraie révolution.

L'entité fondamentale n'est plus la page, qui est une structure physique, mais le segment qui est défini comme un container logique auquel on va pouvoir attacher des propriétés particulières, via un descripteur de segment.

La structure de cet espace d'adressage original est la suivante (figure 7.5) :

7. Cf. le livre de K. Redmond, T. Smith, *From whirlwind to MITRE – The R&D story of the SAGE air defense computer*, MIT Press, 2000 explique l'implication du MIT dans le début des grands systèmes informatiques.

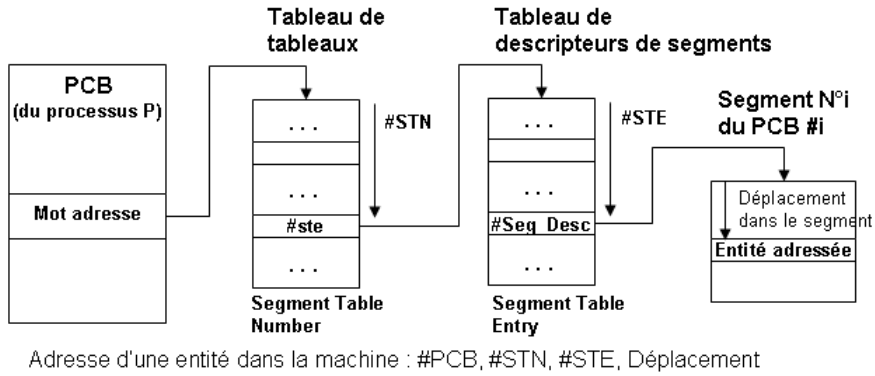


Figure 7.5 - Espace d'adressage segmenté

Le nom d'une entité, dans cet espace, est :

(J, P), STN, STE, D ; soit deux abstractions intermédiaires par rapport au précédent schéma d'adressage, auquel il faut ajouter le *segment descriptor* SD (Descripteur de Segment) qui contient les propriétés communes à toutes les pages constitutives du segment (NB : les pages du segment ne sont pas représentées sur la figure). Le SD peut être « indirect », c'est-à-dire pointer vers un autre SD.

Parmi ces propriétés, nous allons trouver des droits d'accès et des « anneaux » de protection, directement hérités des travaux effectués sur les modèles de sécurité par le MIT et MITRE corporation (un laboratoire privé fonctionnant sur fonds publics, émanation du Lincoln Laboratory à l'époque du projet SAGE[8], typique de la structure de recherche américaine).

Les droits d'accès sont directement hérités de MULTICS : REWA (*read, execute, write, append*), c'est-à-dire : droit de lecture, droit d'exécution, droit d'écriture, droit d'agrandir le segment (i.e. ajout de nouvelles pages).

Un numéro d'anneau de protection qui donne aux segments différents privilèges (NB : sur MULTICS, il y avait 64 anneaux de protection ; sur le DPS7, seulement 4). Les anneaux sont un mécanisme de protection à l'intérieur même de l'espace d'adressage du processus. La notion d'anneau généralise la distinction *mode maître*, réservé au système d'exploitation, et le *mode esclave* pour les programmes utilisateurs, devenus insuffisants avec les nouveaux besoins. L'idée des architectes est de faire coïncider la notion de couches (qui est une notion purement logique) avec celle d'anneau. À l'intérieur d'un même anneau, l'adressage est libre, mais pour passer d'un anneau à l'autre un mécanisme particulier est indispensable, au niveau du segment référencé, d'où une protection renforcée contre les accès intempestifs et les erreurs d'adressage.

8. Cf. K. Redmond, *From Whirlwind to MITRE,* déjà cité ; SAGE (*Semi-Automatic Air Ground Environment*) est l'ancêtre de tous les systèmes de contrôle aérien.

Le *segment descriptor* SD peut être direct ou indirect ce qui permet de choisir la référence au segment recherché, au dernier moment, comme l'adressage indirect avec les pointeurs.

Le fait de mettre ces mécanismes au niveau de l'interface hardware-software était la meilleure façon de stabiliser les interfaces et d'éviter les variantes qui sont une tentation permanente à laquelle succombe bon nombre de programmeurs qui ont toujours de bonnes raisons locales de changer mais qui complexifie globalement le système.

Une fois hardwarisé, il n'y avait plus de tentation, ni de querelles stériles. On évite ainsi le paradoxe présent dans de nombreux systèmes : l'optimisation locale (sur-spécification) détruit l'optimisation globale ; ou, en d'autres termes, toutes les parties sont « bonnes », il n'y a que le tout qui n'est pas bon.

C'est une propriété du hardware très utile au génie logiciel, du moins quand l'interface est parfait, ce qui était le cas. Le nom de processus #PCB, lui-même formé de deux numéros, le J (pour un groupe de processus) et le P (pour un processus du groupe de N° j1) était physiquement représenté par un entier sur 16 bit, soit 64K noms possibles.

À l'époque, ce nombre nous paraissait énorme, mais quelques années plus tard, avec les transactionnels lourds à plus de 1 000 TPR/seconde, c'était devenu insuffisant. Chaque transaction crée plusieurs processus dans la machine. En une minute de fonctionnement 60 000 transactions peuvent être initialisées, d'où un problème de gestion des noms évident (le problème bien connu des noms uniques dans les bases de données) qui suscita, beaucoup plus tard, dans les années 90, les adressages dits à 64 bits.

Les possibilités offertes par ce mécanisme, au prix d'une complexification certaine, étaient extrêmement riches, en matière de gestion du partage et du multiplexage. En jouant sur le segment *descriptor* SD on peut référencer tel ou tel segment particulier, sans toucher à l'arborescence. En jouant sur le STN, on peut basculer, en une seule opération une table vers une autre. Ceci peut permettre de commuter un contexte utilisateur vers un autre contexte, uniquement en jouant sur les tables de segments.

Tout ceci était un avantage concurrentiel décisif en matière de traitement transactionnel où de telles opérations sont courantes. Idem pour les caches de base de données qui va se traduire, pour un programme utilisateur, par l'accès à une zone de mémoire commune (1 ou plusieurs segments, voire par un certain nombre de STE, réservés de façon conventionnelle).

C'était une fort belle architecture qui permettait aux programmeurs de travailler de façon sereine, car la moindre violation des règles « hardwarisées » se traduisait par une exception machine qui permettait d'identifier immédiatement l'instruction fautive, d'où un gain de temps de diagnostic important en intégration.

Le schéma 7.6 donne un exemple de possibilité de partage.

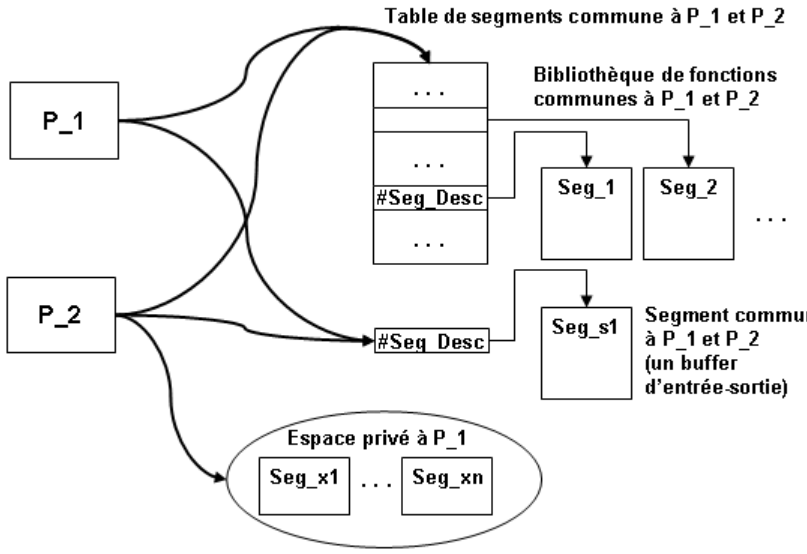


Figure 7.6 - Possibilité de partage dans un espace segmenté

Tout ceci était couplé aux mécanismes de mémoire virtuelle et d'édition de liens dynamiques, donnant à l'ensemble une très grande souplesse d'emploi côté software, sans rien concéder à la sûreté de fonctionnement qui était l'affaire du hardware. Nul doute qu'avec une telle architecture on aurait moins de virus et de failles de sécurité sur les systèmes comme WINDOWS.

Avec cette organisation, la structure d'adressage physique de la machine qui reste évidemment de type tableau, est complètement masquée par la structure logique de l'adressage. Au niveau de l'interface hardware-software, le programmeur ne connaît que la structure logique et des noms de type (J, P), STN, STE, D. La traduction de ces noms en adresse physique est faite par le hardware, qui les gère à sa façon, avec des registres associatifs (invisible du programmeur) qui conservent la correspondance (noms logiques/noms physiques).

C'est déjà du MDA, bien avant que l'OMG ne reprenne ce concept à sa façon très médiatique comme une révolution, mais dans ce cas précis, elle a déjà eu lieu, trente ans auparavant, comme d'ailleurs avec les compilateurs et les SGBD.

7.2 LES SÉMAPHORES ET LA COMMUNICATION INTER-PROCESSUS.

Avec l'asynchronisme des organes périphériques de la machine et le parallélisme des traitements, la programmation système était devenue passablement compliquée et arbitraire, sans autre logique que celle de l'équipement. C'était un ensemble de cas

particuliers, souvent astucieux, toujours coûteux et difficiles à faire évoluer (cf. le paradoxe de l'optimisation : on optimise localement et on désoptimise globalement).

Dans une machine avec des organes asynchrones, un flot d'exécution peut être interrompu de façon aléatoire à chaque instruction machine au gré de la survenue des évènements. Il fallait contrôler l'arrivée des interruptions à l'aide d'instructions indivisibles dont la plus célèbre était la `test_and_set`.

Dans le modèle producteur-consommateur, un processus produit des états à imprimer (c'est juste un exemple) et un autre processus (le consommateur) se charge de les envoyer s'imprimer, ou de les ranger dans un fichier spool. Entre les deux, il y a une mémoire tampon (i.e. un « buffer ») qui sert alternativement de zone d'écriture, puis de zone à imprimer. Il faut éviter que pendant une impression, une nouvelle écriture vienne écraser la zone d'impression, rendant le tout incohérent. Pour cela, il faut s'assurer qu'avant d'écrire dans le buffer, celui-ci est « libre ». Si c'est le cas, il faut immédiatement le déclarer « occupé » pour éviter toute autre écriture intempestive. Pour garantir ce changement d'état, on ne peut pas programmer de façon simple, comme :

IF état-buffer = « libre » THEN état-buffer = « occupé »

Car cela génère deux instructions, une comparaison (le test) et une assignation (le set) ; entre les deux, il peut y avoir interruption du flot d'exécution.

La solution la plus simple, du point de vue du hardware, était de rendre les deux instructions solidaires, en une seule opération indivisible, la test_and_set. Du point de vue du software, c'est exactement l'inverse, car toute la complexité, et surtout la complication arbitraire, est rejetée sur la programmation.

Avec la diversité des équipements, l'apparition de nouveaux équipements, les évolutions fréquentes, la situation était devenue infernale. L'auteur se souvient avoir programmé une imprimante, dont les caractéristiques physiques nécessitaient un temps de latence, puis quatre cycles d'impression. Pendant le temps de latence, on pouvait encore écrire dans la zone d'impression. Pour accélérer les temps d'impression qui, à l'époque, était un goulot d'étranglement, les programmeurs utilisaient toutes les astuces possibles, et en général, ils en étaient fiers (mais à quel coût !). Le programmeur gérait l'entrelacement du flot d'instructions permettant d'écrire dans le buffer, et le cycle d'impression, comme présenté en figure 7.7.

Pour faire ce travail de façon sûre, le programmeur devait connaître la durée d'exécution de chaque instruction et les caractéristiques temporelles de l'appareil.

En cas de nécessité, le programmeur pouvait programmer une instruction « `wait n_cycles` » ou des `nop` (`No_operation`, qui prenaient quand même quelques cycles) pour laisser l'appareil terminer son travail.

Avec l'arrivée, pour des raisons économiques, du concept de gamme d'ordinateurs et plus généralement de gammes d'équipements, avec des performances variées d'un modèle à l'autre (c'était le pari de la série 360 d'IBM[9]), une programmation de

ce type n'avait aucun avenir, car trop peu de programmeurs étaient capables de maîtriser cette complexité.

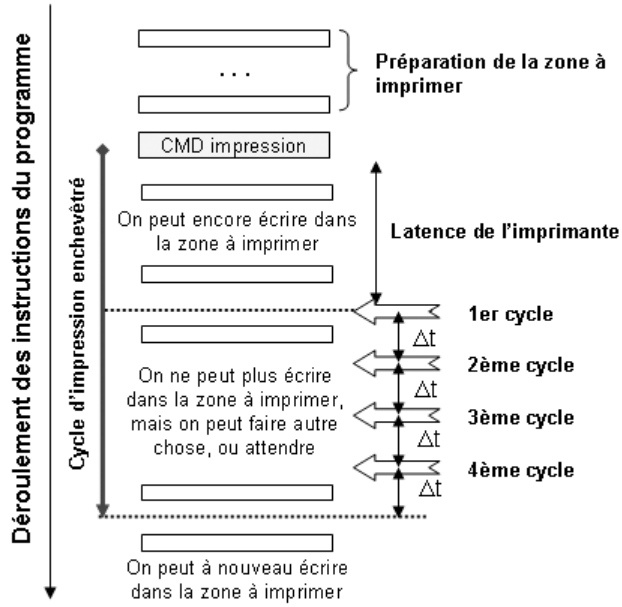


Figure 7.7 - Entrelacement de flots asynchrones

C'est dans ce contexte que E. Dijkstra, et quelques autres inventèrent le concept de sémaphores, vers la fin des années 60. E. Dijkstra, mathématicien de formation, signa un rapport technique célèbre (il travaillait alors pour Philips qui, comme beaucoup de sociétés d'électroniques, fabriquait également des ordinateurs), intitulé *Cooperating sequential processes*. Quiconque veut comprendre en profondeur le concept de processus séquentiels et leur synchronisation doit connaître ce rapport qui a fait l'objet de nombreuses publications, la plus récente dans *The origin of concurrent programming*, Per Brinch Hansen, Springer, 2002. Nous n'allons pas ici exposer le concept de sémaphores, il y a de nombreux ouvrages sur le sujet, où tout cela est expliqué en détail. Nous nous en tiendrons à l'essentiel et à la façon d'implémenter ce concept dans les machines et/ou les systèmes d'exploitation :

- soit dans le hardware (cas du DPS7),
- soit dans le software : c'est une fonction basique des systèmes d'exploitation qui peut revêtir de très nombreuses formes, et des middleware d'intégration (moniteurs transactionnels, bus logiciel type CORBA, EAI/ESB…).

9. Cf. le livre passionnant de E.Pugh, L.Johnson, J.Palmer, *IBM's 360 and early 370 systems*, MIT Press.

7.2.1 Dynamique des opérations de synchronisation

Pour comprendre la dynamique des opérations, il faut revenir à la notion d'état du processus et des transitions autorisées entre ces états, gérées par le dispatcher. Sur une machine comme le DPS7 (et sur quelques autres) le diagramme états-transitions était le suivant (en simplifiant quelque peu) :

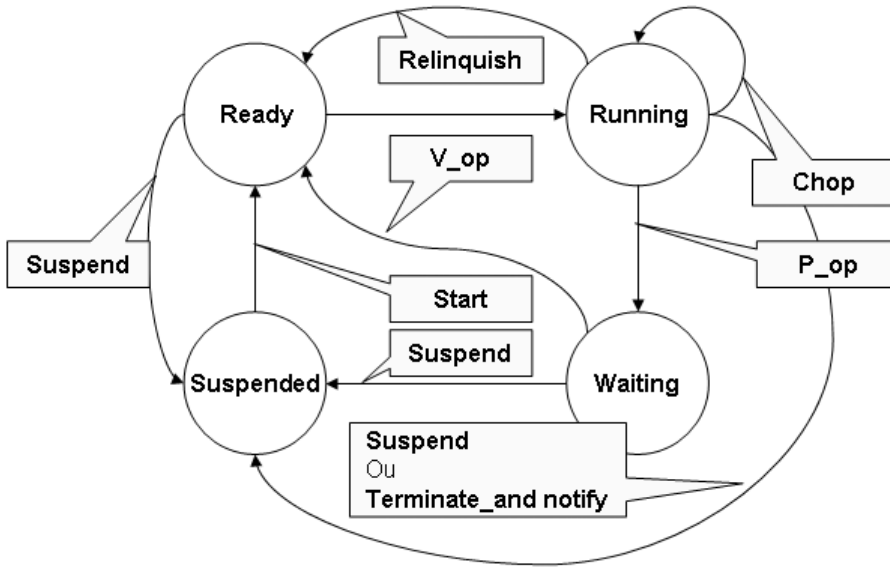


Figure 7.8 - Etats – Transition des processus

Les changements d'états sont provoqués par l'exécution d'instructions dédiées à la gestion et à la synchronisation des processus, et/ou d'évènements en provenance des autres organes de la machine (y compris la console opérateur et l'opérateur lui-même) ; le tout sous le contrôle du dispatcher, véritable pilote et chef d'orchestre de toutes les opérations.

Management des processus à l'aide de fonctions primitives

Exemple de fonctions primitives, disponibles soit dans l'interface hardware-software, soit dans le micro-noyau :

- `Démarrer_processus (nom du processus)` ; c'est le « *start* », le nom du processus est le numéro de PCB qui le définit.
- `Suspendre_processus (nom du processus)` ; fait passer le processus nommé dans l'état « *suspended* », c'est un changement de file d'attente.
- `Changer_priorité` ; change la priorité du processus courant (*chops*), ce qui peut se traduire par la perte du processeur si un processus plus prioritaire est dans la file d'attente des *ready*.

- `Relâcher_processus (nom du processus courant)`; le processus courant relâche le CPU au bénéfice d'un autre processus (*relinquish*).
- `Terminate_processus (nom du processus courant)` ; c'est la fin de vie normale d'un processus qui a terminé son travail, un message de fin est envoyé sur un sémaphore particulier (*terminate and notify*).

7.2.2 Synchronisation des processus – les sémaphores

Les sémaphores, comme leur nom l'indique, servent de mécanisme de signalisation. Ils permettent de coordonner l'exécution de tous les processus coopérant à l'accomplissement d'un certain travail, du point de vue de l'utilisateur.

Un sémaphore est une structure de données dans l'espace d'adresse de tous les processus coopérant, donc une structure partagée qui doit faire l'objet de conventions explicites entre les programmeurs des différents processus.

Un sémaphore est manipulé de façon indivisible par un jeu d'opérations ad hoc de façon telle, qu'en cours de manipulation, aucun autre processus ne pourra y accéder ; l'accès est verrouillé, y compris en cas de multi-processeurs (i.e. plusieurs processus sont dans l'état running). En conséquence, un processus qui exécute une instruction de manipulation du sémaphore opère comme s'il était seul dans la machine.

Il n'y a que deux opérations de base sur les sémaphores utilisés comme mécanisme de signalisation, appelé `V_opération`, ou `V_op`, et `P_opération`, ou `P_op`, en hommage à E. Dijkstra qui les a inventés[10].

L'opération V_op permet la notification d'un signal du processus qui l'exécute, vers un autre processus présent dans la machine. L'opération V_op peut être conçue comme une opération d'envoi d'information mise en œuvre par le processus qui a des données à expédier vers un autre processus.

L'opération V_op permet au processus d'envoyer les données, ou pour le moins un signal indiquant que les données sont prêtes, en se servant d'une structure médiane : le sémaphore.

Le sémaphore enregistre le signal jusqu'à ce qu'un autre processus soit prêt pour le retirer (NB : imaginer une boîte aux lettres). Une fois le signal expédié, l'expéditeur est libre de continuer à travailler, puisqu'il a effectivement délivré les données hors de son processus, dans une « boîte aux lettres » commune à tous (mais n'appartenant à aucun). Il peut toutefois attendre un accusé de réception.

L'opération P_op est la réception des données, par le processus destinataire qui a besoin de ces données. Pour cela, le processus examine le sémaphore et récupère les données ou le signal qui y sont attachées. Si les données sont effectivement présen-

10. C'était des mot hollandais : « Proberen » et « Vrijgeven » ; dans les systèmes IBM, on disait plutôt ENQUEUE/DEQUEUE. Dans les télécommunications et en transactionnel, c'était SEND/RECEIVE.

tes, le processus qui exécute le P_op continue à travailler. Si il n'y a rien (la boîte aux lettres est vide), le processus se met en position d'attente. Le sémaphore détient le processus récepteur, au lieu des données via une file d'attente attachée au sémaphore ; le processus reste dans l'état WAIT jusqu'à ce que des données soient envoyées sur le sémaphore par les processus émetteurs.

En d'autre termes, on peut dire que le sémaphore conserve les données jusqu'à ce qu'un autre processus les retire, ou qu'il retient, en attente, les processus récepteurs jusqu'à ce que le signal attendu se manifeste, et les libère.

La distinction signal simple et signal accompagné de données permet de distinguer deux types de sémaphores :

- le sémaphore sans message (SN),
- le sémaphore avec message (SM).

Le sémaphore SN gère une file d'attente de processus en attente d'un signal.

Le sémaphore SM gère, selon la valeur de son compteur, une file d'attente de messages (CNT > 0) ou une file d'attente de processus en attente de messages (CNT < 0, avec |CNT| = longueur de la file d'attente).

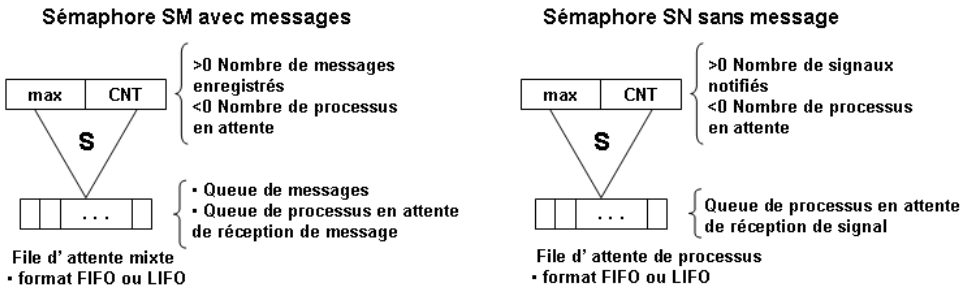


Figure 7.9 - Sémaphores avec ou sans message

À chaque envoi de signal ou de message, le compteur CNT est incrémenté de 1 : CNT CNT + 1 ; jusqu'à une valeur maximum dont le dépassement provoquera une exception.

À chaque demande de réception de signal ou de données (P_op) le compteur CNT est décrémenté de 1 : SCN SCN-1.

Une P_op sur un sémaphore dont le SCN est $\sum$ 0 provoque la mise en WAIT du processus correspondant.

Une V_op sur un sémaphore dont le SCN est < (il y a des processus en attente) provoque le passage du premier processus en attente de l'état WAIT à l'état de READY.

Dans le modèle producteur/consommateur, et plus généralement le modèle client/serveur, le ou les clients émettent des informations, à leur rythme, vers un serveur d'application qui traite les informations au fur et à mesure de leur arrivée.

Imaginons une situation où tout transite à travers un tampon centralisé unique, avant d'être pris en compte par le serveur de traitement. Le chemin de données est le suivant (figure 7.10) :

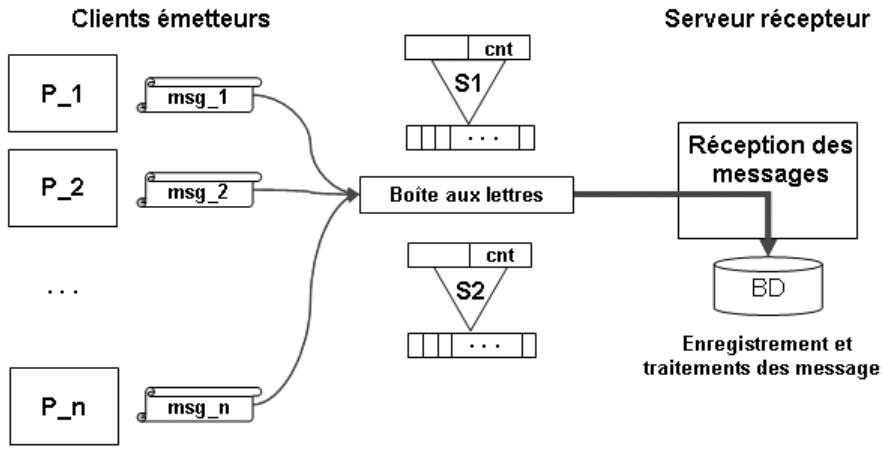


Figure 7.10 - Client-serveur avec sémaphores

Les clients P1, P2,…, Pn signalent l'envoi de messages via le sémaphore S1. Si la boîte aux lettres est vide, le message est déposé dans la boîte aux lettres qui devient occupée. Le serveur de courrier vide la boîte aux lettres chaque fois qu'elle contient un message, sinon, elle attend l'arrivée du prochain message.

Pour déposer un message, le client doit s'assurer que la boîte aux lettres est vide, sinon, il doit se mettre en attente, d'où une variable d'état nécessaire à ce contrôle : dans notre schéma, c'est le sémaphore S2.

Ebauche d'une programmation

À l'arrêt, les compteurs SCN de S1 et S2 sont à 0 (zéro). Côté client, on aura :

`P_op (S2)`/* si boîte aux lettres occupées attendre */

[construction du msg et dépôt dans la boîte aux lettres]

`V_op (S1)`/* notification, via S1, qu'un message a été déposé dans la boîte aux lettres * /

Côté serveur de courrier, on aura :

`P_op (S1)` /* si boîte aux lettres vide attendre */

[extraction du message et traitement]

`V_op (S2)` /* notification, via S2, que la boîte aux lettres a été vidée */

Dans cet exemple, le sémaphore S2 joue le rôle d'une opération `test_and_set`. On pourrait imaginer une situation où la boîte aux lettres, une fois vidée, aurait une valeur conventionnelle, partagée par tous les processus, permettant de savoir, à l'émission, si la boîte aux lettres est vide ou non ? Pour cela, il faudrait, côté émetteur, une véritable `test_and_set`.

La solution avec un sémaphore additionnel S2 est plus élégante, car plus explicite (i.e. moins conventionnelle).

D'une certaine façon, le sémaphore S2 matérialise la convention, ce qui est plus simple. Le problème permanent, avec les conventions, est que si un des acteurs ne joue pas le jeu, pour une raison quelconque (e.g. le référentiel est ambigu, ou n'a jamais été documenté correctement, ou les acteurs ne sont pas formés, ou …), la synchronisation devient incohérente ; il sera très difficile de retrouver l'origine de la défaillance.

On pourrait imaginer également que le sémaphore S1 serve également à la notification que la boîte aux lettres est vide. Le sémaphore S1 serait alors bi-directionnel. Bien que théoriquement envisageable, une telle utilisation du sémaphore S1 nécessiterait, là encore, des conventions entre toutes les parties prenantes. Pour la même raison que ci-dessus, c'est fortement déconseillé. C'est à l'architecte de s'assurer que ce qui garantit la cohérence de l'ensemble est explicite et partagé par tous.

Les sémaphores sont des objets simples (mais non simplistes), il convient de leur garder ce caractère de simplicité. Les sémaphores jouent un rôle essentiel vis-à-vis de la sphère de contrôle du processus, en filtrant tout ce qui entre et tout ce qui sort. Aucune convention « artificielle » n'est nécessaire, tout est explicité.

7.3 LES LEÇONS : LES CONTRAINTES SYSTÈMES ET LA RECHERCHE D'UN ÉQUILIBRE ÉCONOMIQUE

Cette mécanique de haute précision que nous n'avons fait qu'ébaucher, est à l'œuvre dans toutes les fonctions du type :

- READ/WRITE pour les entrées-sorties, sur tout type de mémoire externe.
- ACCEPT/DISPLAY pour la connexion avec les périphériques « lents »de types consoles graphiques, imprimantes, etc.
- SEND/RECEIVE pour l'envoi et la réception de messages à travers un réseau de type quelconque.
- PUBLISH/SUBSCRIBE pour les échanges d'information entre clients et serveurs, qui intègrent en un seul tout, les trois catégories précédentes, via les middlewares d'échanges comme les MOM ou les EAI/ESB.

Elle est le cœur des architectures client/serveur et/ou des *Service Oriented Architectures* (SOA). Elle n'est généralement vue par les concepteurs d'applications qu'à travers les paramétrages effectués sur la plate-forme d'intégration par l'architecte, mais il est essentiel pour tous de bien la comprendre si on veut garantir la sûreté de fonctionnement global du système, ou tout simplement être capable de diagnostiquer correctement les défaillances. Un défaut de synchronisation peut occasionner des défaillances redoutables car entre le moment où le code défectueux est exécuté et celui où la défaillance est constatée il peut s'écouler un temps relativement long (temps de latence défaut-défaillance). Le seul recours, dans ce cas, est la capacité à remonter l'histoire, à travers la chronologie des évènements qui déterminent les communications inter-processus. Encore faut-il que les fichiers logs correspondants aient été correctement constitués et archivés.

Les processus et l'IPC, les compilateurs et les grammaires, ont réalisé, à leur façon, cet idéal de beauté logique qui est la marque de fabrique des belles architectures, où tout est équilibre. Elles sont la matérialisation, pour l'ingénierie de l'information, du principe de parcimonie que nous a légué Guillaume d'Ockham, et de l'architecte créateur d'ordre, cher à Thomas d'Aquin.

Dans le cas des processus, il est remarquable de constater qu'au départ il ne s'agissait que de découvrir des abstractions permettant à des organes asynchrones de fonctionner indépendamment tout en communiquant de façon sûre, via des évènements gérés conjointement par les programmeurs et le système.

Les sémaphores sont des constructions logiques quasi parfaites, qui nécessitent cependant quelques aménagements quand on considère la réalité des contraintes système. Dans le chapitre 6, nous avons vu qu'à côté des sémaphores et des états des processus gravitaient un certain nombre de files d'attentes : file d'attente de processus dans l'état ready, files d'attente de processus en attente sur tel ou tel sémaphore, files d'attente de messages postés sur tel ou tel sémaphore en attente de réception par le processus destinataire. Toutes ces files d'attente consomment de la ressource mémoire qu'il faut gérer. Sur une machine comme le DPS7 la taille des cellules constitutives de ces files d'attente était de 8 octets pour les processus et de 16 octets pour les messages.

Compte tenu de la nature dynamique et aléatoire des interactions il n'est évidemment pas possible de déterminer a priori la taille de la zone mémoire nécessaire à leur stockage, mais une fois la machine configurée et initialisée, cette taille était fixée.

Du point de vue des sémaphores, et plus particulièrement des messages postés, la zone mémoire correspondante qui doit être vue de chaque côté de l'interface hardware-software, apparaît comme une ressource partagée, elle-même redevable d'une gestion à l'aide de sémaphore, d'où l'invention d'un troisième type de sémaphore, appelé dans le jargon du DPS7 le FLS (*Free Link Semaphore*) ou sémaphore de gestion des cellules des files d'attente. On retrouve ici cette situation si caractéristique des « bonnes » constructions informatiques qui sont auto-applicables à elle-même (ou récursives).

La situation, du point de vue de l'architecte, est tout à fait simple. Elle correspond au cas d'une boîte aux lettres partagées entre des expéditeurs de messages et leurs destinataires. Que faire quand la boîte aux lettres est pleine ?

La solution consistant à arrêter la machine est profondément insatisfaisante car cela revient à arrêter le service et ruiner la disponibilité, avec probablement une perte du travail effectué par les utilisateurs. Une telle situation peut très facilement apparaître. Il suffit pour cela que les processus récepteurs soient ralentis ou arrêtés pour une raison quelconque (par exemple, la dé-fragmentation d'un disque arrête temporairement toute opération sur le disque).

La bonne solution, celle qui prolonge la durée de vie du système, consiste à bloquer l'émission de messages à la source, lors de la V_op effectuée par le processus émetteur. C'est là qu'entre en scène le sémaphore FLS.

En cas de boîte aux lettres pleines, le message restera temporairement chez l'émetteur qui prendra ses responsabilités : attendre, faire autre chose, alerter l'opérateur du système, etc. Rien ne sera perdu quant à l'intégrité de l'information.

Le FLS génère une situation claire entre émetteurs et récepteurs. Si le ou les récepteurs sont momentanément ralentis, il est normal que les files d'attente s'allongent. Dès qu'ils retrouveront leur vitesse de croisière (éventuellement en modifiant temporairement la priorité des processus) le système retrouvera son cours normal.

Les outils de gestion du système (*capacity planning*, *system management*, la boucle « *autonomic computing* » de chez IBM) scrutent en permanence ces files d'attente et peuvent, de ce fait, anticiper des situations potentiellement anormales, à l'aide de seuils d'alertes.

On peut comprendre que si cette file d'attente FSL n'est pas explicitée de façon précise, la surveillance sera plus difficile, plus coûteuse en ressource, voire impossible car dispersée dans la machine.

Dans ce cas, la beauté logique est synonyme de fiabilité et de disponibilité, i.e. le R de FURPSE. Une autre propriété intéressante et particulièrement instructive des sémaphores est leur statut ontologique (pour parler comme Guillaume d'Ockham) selon qu'on les considère du point de vue hardware (i.e. système) ou du point de vue logiciel.

Du point de vue logiciel, ce sont des structures de données, faites avec les types de données de base de la machine. De ce point de vue, ils appartiennent à l'espace d'adressage du processus qui les utilise (mais pas aux autres, d'où une propriété de confinement intéressante) et sont manipulés via des instructions ad hoc (i.e. une vingtaine d'instructions en assembleur, 5 ou 6 en PL1/GPS dans le cas du DPS7/GCOS7).

Du point de vue du hardware, qui voit également les sémaphores (tous les sémaphores) les choses sont différentes. Le dispatcher (cf. figure 7.1) voit toute la mémoire, y compris la sienne propre qui n'a pas besoin d'être structurée en (J, P), STN, STE, Déplacement. Les zones mémoire où vont être stockés les sémaphores,

sont des segments système globaux, appelés pour cette raison segment G, au nombre de 255.

La structure d'adressage système se simplifie et prend la forme d'une table de segments (cf. figure 7.5), ce qui fait qu'un nom système a la forme :

G#, Déplacement /* numéro de segment G dans la table + localisation du descripteur de segment */

Au final, un sémaphore est une entité logique qui dispose de deux noms :

- un nom système, de la forme G#, D,
- un nom logiciel, de la forme (J, P), STN, STE, D,

selon que l'on se situe du côté hardware de l'interface, ou du côté software.

C'est le rôle du système d'exploitation, et plus particulièrement du noyau superviseur (i.e. le micro-noyau), de s'assurer de l'unicité de la correspondance.

Nonobstant cette correspondance fondamentale qu'il faut gérer, chacun des deux mondes peut fonctionner avec sa logique propre. Sans entrer dans le détail on peut également comprendre comment, d'un point de vue de management de projet, on va pouvoir faire fonctionner en parallèle deux équipes de grande taille qui, si ces interfaces n'existaient pas, seraient étroitement enchevêtrées.

Cette décohérence des trajectoires projets, pour utiliser une métaphore de la physique quantique, est le résultat de la standardisation rigoureuse de l'interface hardware – software qui permet de regarder de chacun des côtés, sans que l'on perturbe l'autre. L'interface H-S est le juge de paix qui met tout le monde d'accord.

Et c'est là, la deuxième grande leçon qui nous montre, comment, à partir d'un problème hautement complexe (J. von Neumann[11], dans ses conférences, parlait de « *extremely high complication* »), on peut tout de même séparer les variables et créer de l'ordre.

En prenant le point de vue du logiciel, le bilan de l' « effet » processus est encore plus instructif. À l'époque où tout cela a été inventé, les seuls logiciels de grande taille étaient les systèmes d'exploitation et les grands systèmes technologiques pour la défense et l'industrie spatiale, comme SAGE dont nous avons déjà parlé. Les applications utilisateurs étaient relativement petites, mais cela n'allait pas durer avec, dans les années 80s, l'arrivée des premiers grands logiciels de gestion, de type MRP/ERP. L'organisation des équipes de développement commençait à poser des problèmes analogues pour les mêmes raisons : nécessité d'interfaces ultra-stables pour bâtir par-dessus de façon sûre, détermination de périmètre de projet en cohérence avec les contingences de l'ingénierie de développement (au-delà de 40-50 personnes, tous les projets sont à haut-risque), intégration des composants applicatifs livrés par les équipes, etc.

11. Cf. son ouvrage posthume, *Theory of self reproducing automata*, University of Illinois Press.

Le découpage du système à réaliser, quelle qu'en soit la nature, systèmes d'information ou systèmes technologiques, en processus élémentaires, aux interfaces bien définies est rapidement apparu comme un dispositif fondamental de l'ingénierie informatique. Dans le cas de GCOS7, les choses sont claires. Le référentiel de cette architecture de processus est un ensemble de documents totalisant environ 500 pages dont environ 200 vraiment indispensables à la compréhension de la philosophie générale du système[12]. Si l'on prend en compte la durée des développements qui se sont étalés sur vingt ans, avec des hauts et des bas dans l'activité, avec le renouvellement naturel des équipes, c'est au final plusieurs milliers de personnes qui ont pu travailler de concert, en parfaite cohérence. A contrario, il est facile d'imaginer la situation chaotique qui se serait instaurée en l'absence d'un tel référentiel.

La décision d'intégrer l'architecture des processus à l'interface hardware-software du DPS7, a été une décision difficile et à haut risque, à l'époque où elle fut prise. À l'usage, elle s'est révélée excellente. Cela complexifiait la partie hardware (en fait micro-programmée) de façon certaine, mais en contrepartie, elle donnait au software, un espace de liberté permettant d'envisager des solutions économiques équilibrées aux problèmes nouveaux posés par les SGBD et les moniteurs de transactions, raisonnables en terme CQFD/FURPSE, que tout le monde voyait poindre à l'horizon.

Un seul système, à ce jour, a poussé plus loin le niveau d'abstraction de l'interface hardware-software : l'AS/400 d'IBM, qui est l'une des machines phares de ce grand constructeur. L'architecture révolutionnaire de cette machine est décrite, avec bon nombre de détails et de remarques qui recoupent les nôtres, par F.G.Soltis, qui fut le *chief scientist* de ce nouveau système : *Inside the AS/400*, News 400 Books, 1997.

Cette façon de faire donne la clé de ce qu'il faut mettre en œuvre pour maîtriser l'informatisation massive dans laquelle nous sommes désormais installés :

- Découpage du système en entités autonomes de taille raisonnable (i.e. limitée par le savoir-faire des équipes) mais communicantes via des interfaces ultra-stables ; la règle de facto qui émerge est : a) UN projet = UN processus = UNE équipe de développement ; et b) architecture des communications = interopérabilité des processus via les messages = interopérabilité des équipes via les interfaces.
- Contrôle rigoureux de la croissance de la complexité, matérialisé par les tests et les procédures d'intégration.
- Sûreté de fonctionnement, en particulier tout ce qui permet la surveillance du système et le diagnostic des pannes, sur des points d'observation explicites.
- Construction progressive du système, évolution et adaptation, sur la base d'un noyau de services qui sont les fondations du système.

Ce sera l'objet des chapitres qui vont suivre.

12. L'interface hardware-software complet totalisait environ 3.000 pages !

PARTIE 3

Architecture fonctionnelle logique

8

Principes et règles de construction des architectures fonctionnelles logiques

La traduction des modèles métiers en modèles informatiques

8.1. LES PROCESSUS DU MONDE RÉEL

Lorsqu'on analyse le fonctionnement d'une entreprise, assimilée à un organisme dans les normes ISO 9000, avec un regard de biologiste, ou celui d'un architecte au sens du chapitre 1, on constate deux choses :

1) L'entreprise est constituée d'entités élémentaires que les économistes appellent unités actives[1] (UA). Cette terminologie est plus évocatrice que celle d'agent économique également utilisée. L'UA a une capacité de transformation qui, moyennant un apport d'énergie, à partir d'un flux en entrée de l'UA produira un résultat et/ou un effet sur l'environnement. L'UA est un quantum de production de biens et/ou de service. L'énergie nécessaire à la transformation est celle développées par les collaborateurs de l'entreprise intégrés dans l'UA. Le collaborateur apporte avec lui des compétences et de l'expérience qui lui permettront de tenir différents rôles, à différents degrés d'expertise, selon les besoins et la finalité de l'UA.

Les UA peuvent s'organiser et s'intégrer pour former des UA de niveau supérieur, certaines UA jouant alors le rôle de centre de décision et de pilotage, capable d'auto-régulation, et ce jusqu'à constituer les grandes fonctions de l'entreprise. Une entreprise qui se développe, c'est d'abord un accroissement net du nombre de ses

1. Cf. F.Perroux, *Unités actives et mathématiques nouvelles*, Dunod ; plus près de nous, T. de Montbrial, *L'action et le système du monde*, PUF.

UA. Dans cette approche, l'UA est pour l'entreprise un quantum d'action disposant de ressource, que l'on peut décrire à différents niveaux d'abstraction, soit en tant que processus dans les chaînes de valeur de l'entreprise (par exemple : le processus achat), soit en tant qu'organisation des groupes humains constitutifs de l'entreprise (par exemple : une usine, un atelier, un service, une DSI, etc.) selon la précision requise pour en analyser la performance. Ceci nous amène à distinguer l'UA atomique, qui comme son nom l'indique doit être un élément stable et pérenne de l'entreprise (donc pas une organisation), de l'UA moléculaire agrégée qui peut être un regroupement conjoncturel d'UA atomiques et/ou déjà agrégées pour satisfaire un nouveau besoin nécessitant des restructurations. Cette distinction est fondamentale pour identifier les éléments stables du SI informatique. D'un point de vue de biologiste, en assimilant l'entreprise à un organisme, on aurait les éléments constitutifs de la cellule, la cellule (NB : pour un biologiste une cellule est une usine) puis les tissus, les organes fonctionnels (un œil, un membre, le cerveau, etc.), l'organisme complet, une société, etc. avec de la communication (le système nerveux, le système hormonal) et de la surveillance (le système immunitaire), et l'orchestration d'ensemble par le cerveau.

2) L'entreprise produit biens et services au moyen de chaînes de valeur qui matérialisent la finalité des activités de l'entreprise par rapport à son environnement et ses clients. Ces chaînes de valeur (cf. M.Porter, *Competitive advantage* et *Competitive strategy*) sont contraintes par l'environnement socio-économique (cf. facteurs PESTEL au chapitre 1) dans lequel l'entreprise est plongée et par les décisions stratégiques qui incombent aux directions générales des entreprises vis-à-vis de ces contraintes. On doit donc pouvoir les modifier, les adapter en fonction de ces décisions stratégiques, et ceci avec un temps de latence aussi réduit que possible (ce qu'on appelle alignement stratégique, qui nécessite agilité/flexibilité pour recomposer les UA différemment). En terme d'UA, on doit pouvoir recomposer l'organisation et le fonctionnement (contrôle-commande des UA nucléaires) de façon à former des UA moléculaires agrégées mieux adaptées à l'environnement, ou simplement plus efficaces.

Modifier les capacités des UA atomiques, ou intégrer de nouvelles UA atomiques, relève de contraintes temporelles d'une autre nature. Créer de nouvelles capacités relève d'une logique du temps long, ne serait-ce que par l'inertie des acteurs humains par rapport à leur capacité de changement, ou par les difficultés inhérentes à l'intégration de nouvelles ressources dans un environnement organisationnel et humain qui lui-même manifeste des résistances au changement. Réaménager des compétences existantes relève du temps « court », toute chose égale par ailleurs.

Pour une entreprise, il y a toujours deux façons d'intégrer de nouvelles capacités :

1. Créer effectivement la capacité au sein de l'entreprise, avec tous les aléas et les risques bien connus de la création : temps « long » et risques liés à la durée, incertitudes liées à la création. L'entreprise reste autonome.
2. Aller chercher la capacité recherchée chez des partenaires, et gérer contractuellement le partenariat avec le niveau d'interopérabilité organisationnel et

humain que cela impose ; voire acquérir l'entreprise partenaire avec les risques afférents aux acquisitions. L'entreprise fonctionne en réseau.

Chacune des deux options à ses avantages et ses inconvénients. Dans le temps court, seule 2) est envisageable. Par défaut, nous considérons que l'entreprise est un organisme ouvert qui fonctionne en réseau.

8.2 COMMENT INFORMATISER LES PROCESSUS MÉTIER

L'informatisation des fonctions de l'entreprise est tout à la fois un support aux unités actives et une capacité d'intégration et de composition qui viennent en support des chaînes de valeur, avec comme contrainte économique celle de tirer le meilleur parti des capacités de traitement des infrastructures informatiques vues ici comme une centrale d'énergie informationnelle au service des traitements à effectuer : « *business processing* de type OLTP/OLAP » ; « *scientific processing* » pour tout ce qui est calcul, « *real-time processing* » pour tout ce qui nécessite un couplage avec le temps vrai ; « *security processing* » et « *autonomic computing* » pour tout ce qui nécessite une surveillance des organes vitaux de l'entreprise (en poursuivant la métaphore biologique, c'est le système immunitaire de l'entreprise) ; « *grid computing* » pour la virtualisation. Ces différents types de traitements sont en interactions les uns avec les autres.

L'architecture informatique, matérielle et logicielle, entretient des rapports étroits et profonds avec les chaînes de valeur de l'entreprise et les UA qui les composent, ce qu'on appelle parfois architecture de l'entreprise (NB : Terminologie douteuse, selon nous, car génératrice de confusion sur le terme architecture qui est mis « à toutes les sauces » et devient une auberge espagnole, ce qui ne facilite pas la communication entre les acteurs métiers, MOA, MOE, MCO).

Les figures 1.2, 1.3 et 1.4 montrent l'articulation des processus métiers de l'entreprise et de l'informatique qui les supporte, dans le chapitre 1, section 1.3, Architecture de l'information.

Il se trouve, et ce n'est pas un hasard, que le concept de processus est commun aux deux mondes, avec bien sûr des spécificités. Cela va faciliter la mise en cohérence des deux mondes.

On peut représenter le modèle générique de processus[2] comme suit (figure 8.1) :

2. Ce modèle a été discuté en détail dans nos ouvrages antérieurs, nous le considérons ici comme un acquis. Cf. *Productivité des programmeurs* et *Ecosystème des projets informatiques*, tous deux chez Lavoisier-Hermès.

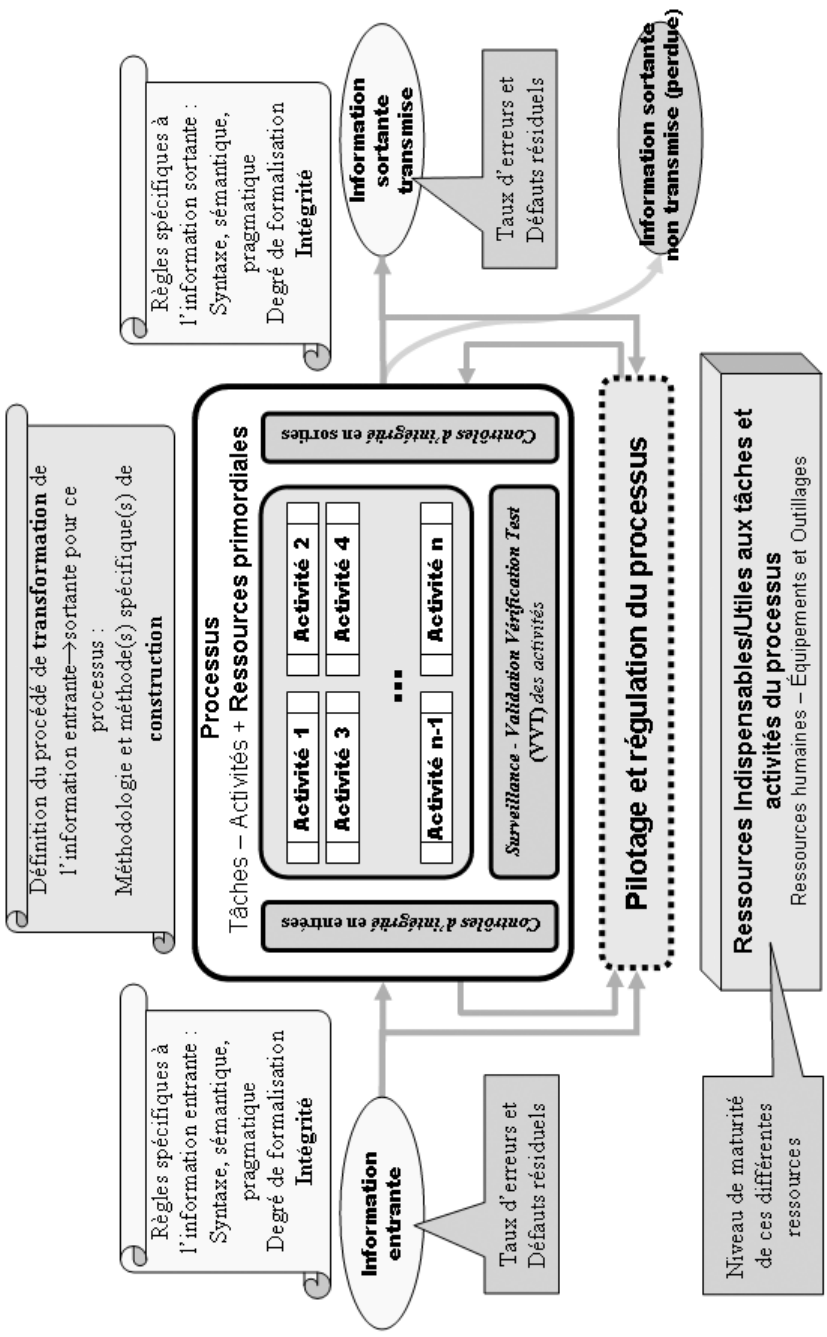


Figure 8.1 - Modèle générique de processus

Le schéma fait apparaître une structuration du processus avec trois types d'entités, correspondant à différents niveaux de granularité des UA métiers, et des modes de fonctionnement spécifiques. C'est ce que résume le schéma 8.2.

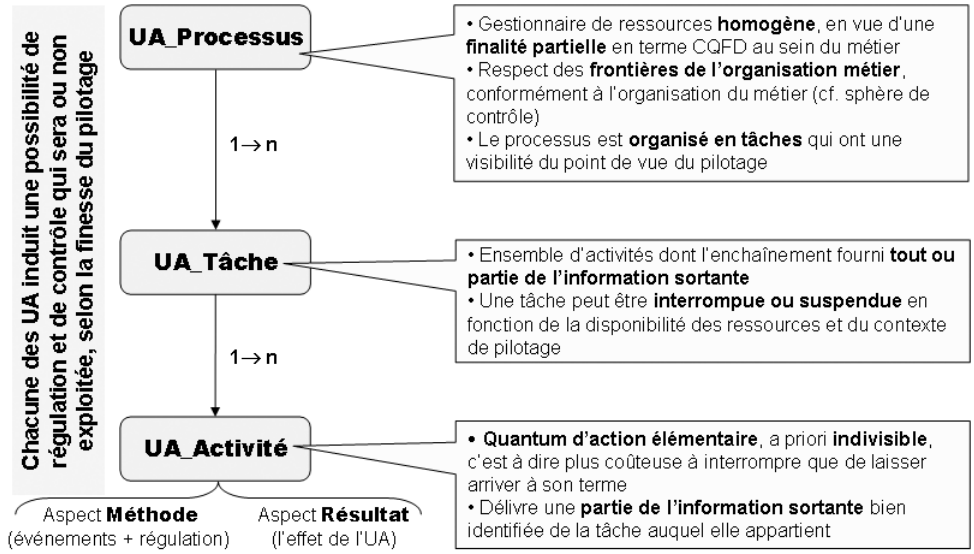


Figure 8.2 - Processus, tâches et activités métier

L'activité élémentaire est l'équivalent d'une opération précise. Chacun des niveaux est décomposable en lui-même de façon récursive, mais sans mixité des niveaux.

Ce faisant on peut représenter précisément les chaînes de valeurs de l'entreprise en agençant les différents processus qui définissent les transformations effectuées par la chaîne de valeur ainsi que les centres de décisions appelés ici PILOTE, interne ou externe, selon que le centre de décision s'occupe de la régulation d'un processus particulier ou d'un ensemble de processus.

Le schéma 8.3 montre d'autres aspects du processus.

La gestion des ressources nécessaires aux transformations effectuées par le processus se fait sur deux niveaux : 1) celles qui sont directement attribuées au processus lors de sa création et 2) celles dont il peut avoir besoin mais qui ne sont pas directement dans sa juridiction et qui nécessitent une négociation avec le gestionnaire de ressources.

Du point de vue des priorités du processus, on peut considérer qu'il y a trois catégories de ressources :

- Les ressources **primordiales** : sans elles, la création du processus n'a pas de sens.

- Les ressources **indispensables** (ou importantes) : le processus ne peut pas s'exécuter si la ressource est absente, mais elle est attribuable temporairement au processus qui en fait la demande via le pilote externe, selon des protocoles d'allocation et de restitution négociées avec le pilote interne. Selon les cas, cela peut entraîner une terminaison du processus correspondant à une défaillance (service non rendu).
- Les ressources **utiles** : celles dont l'absence dégrade la qualité de service sans pour autant nuire à l'intégrité du fonctionnement du processus.

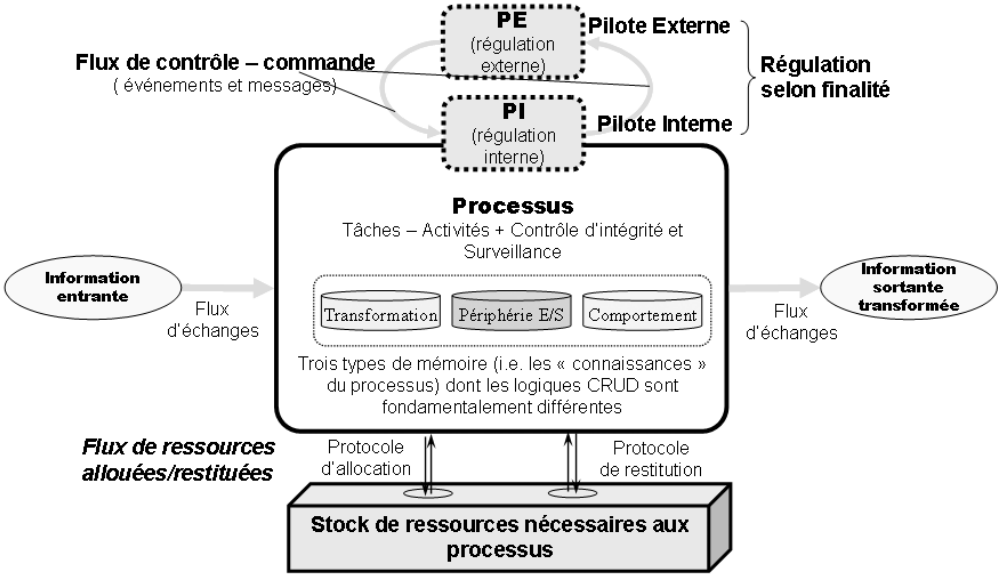


Figure 8.3 - Régulation et stockage des connaissances d'un processus

Les priorités peuvent être gérées dynamiquement.

Le schéma 8.4 montre différents états possibles d'un processus et/ou de ses tâches constitutives.

En fonction des évènements rencontrés, un processus ou une tâche peut être dans différents états. Le schéma 8.4 donne une sémantique intuitive des états possibles et des transitions entre ces sept états fondamentaux.

L'enchaînement nominal des états, dans une vie normale du processus, est :

Défini → Créé → En cours/Actif → Terminé → Relaxé

En cours d'exécution, nous avons vu que le processus peut être momentanément arrêté, entre deux activités. Selon la durée de l'arrêt résultant d'un événement particulier, la sémantique de l'arrêt peut présenter deux modalités temporelles :

- **Interrompu**, lorsque la durée de l'arrêt est bref et ne nécessite pas de relaxer les ressources ; par exemple une activité attend les résultats d'une autre acti-

vité effectuée dans une autre tâche. En informatique, une entrée sortie résultant d'un ordre de lecture interrompt le flot d'exécution du processus, en attente du résultat de la lecture.

- **Suspendu**, lorsque la durée de l'arrêt est significative et nécessite une relaxation totale ou partielle des ressources au bénéfice d'une autre tâche. En informatique, un terminal inactif se voit retirer progressivement toutes les ressources qui lui ont été attribuées.

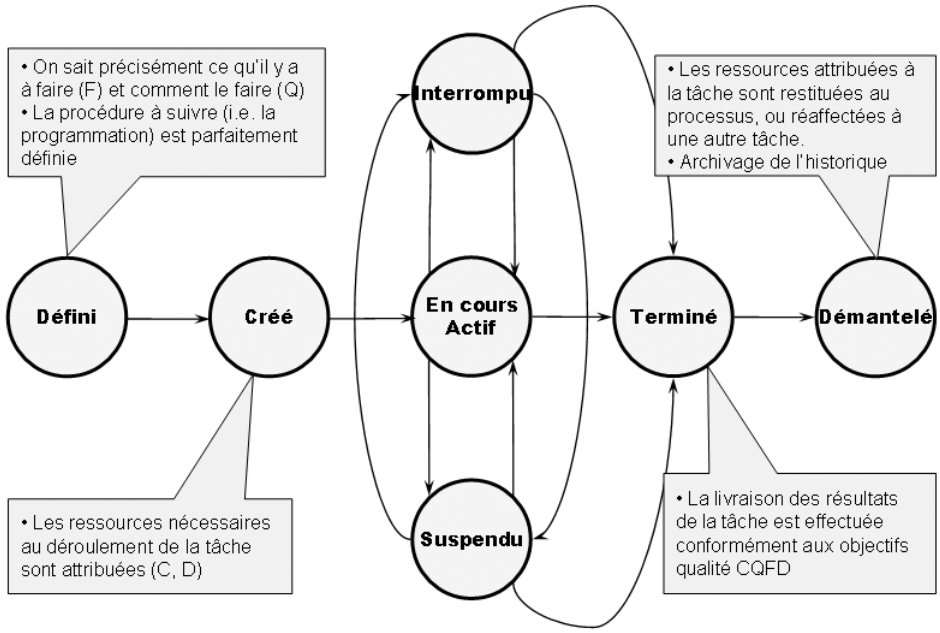


Figure 8.4 - Etats d'un processus et transitions entre états

Interruptions et suspensions résultent d'événements endogènes (ressource temporairement indisponible car affectée à un autre processus, défaillance en exécution) ou exogènes (une ressource est retirée du processus au bénéfice d'un autre processus jugé plus prioritaire suite à une décision du pilote externe). Elles ont évidemment un coût qui va dégrader la performance du/des processus impactés par ces événements.

La chronologie des changements d'états matérialise l'histoire de l'exécution du processus ; c'est un élément fondamental de compréhension du comportement du processus, en particulier en cas de défaillance lorsqu'il faut retrouver un état antérieur cohérent.

Les connaissances nécessaires au bon fonctionnement du processus constituent la mémoire du processus. Dans des processus métier, ce sont toutes les connaissances qui constituent les savoir et les savoir-faire des acteurs des UA, les manuels de procédures permettant de faire face à différentes situations ainsi que toutes les connaissances nécessaires à leur gestion, appelées méta-connaissances.

L'activité humaine est faite de connaissances immédiatement utilisables dans l'action, et de méta-connaissances permettant l'adaptation et la restructuration des connaissances utilisables à de nouvelles situations[3]. Cette distinction fondamentale qui concerne au premier chef les UA et les acteurs qui les animent, devra se retrouver sous une forme ou sous une autre dans les processus informatisés supportant les métiers, sous la forme de règles de gestion exprimées de façon non-procédurales (comme une table de décisions, ou un ordre de sélection en SQL), de paramétrages, de règles de configuration, etc.

Les trois types de mémorisation représentés sur le schéma concerne les connaissances et méta-connaissances concernant 1) les transformations effectuées (information et procédures de traitements), 2) les dispositifs d'entrée et de sortie du processus (i.e. sa périphérie) et 3) le comportement, i.e. évènements et messages qui déterminent les règles d'enchaînement des activités.

En y associant l'aspect méta-connaissance, cela nous amène à distinguer six types de connaissances, soit en fait six types de mémoire dont la gestion (via les fonctions CRUD) est différente.

Dans le schéma 8.1, les règles concernant les flux d'information entrants et sortants font partie de la périphérie. Ces règles doivent être négociées entre les processus émetteurs et récepteurs de ces flux et faire l'objet de contrats, ainsi que les règles de construction, mise à jour et évolution qui sont des méta-connaissances associées à l'information échangée.

Dans le monde réel, il faut considérer que ces flux peuvent être entachés d'erreurs, ce qui nécessite de la surveillance et du contrôle de façon à ce que les acteurs puissent identifier des incohérences possibles et effectuer les réparations éventuelles.

Bien antérieurement à l'informatique, les UA ont su élaborer et faire fonctionner des systèmes de classification et d'archivage, des guichets d'entrées et de sorties des processus organisés en files d'attentes avec des secrétariats affectés au classement du courrier, des systèmes d'audit, de contrôle et d'assurance qualité ; les procédures métier que l'UA applique constituent la « programmation » de l'UA.

La machinerie informationnelle succinctement décrite ci-dessus a toutes les apparences d'une machine logique telle que celle présentée au chapitre 1. La grande différence entre une machine logique du Monde M1 et une machine logique du monde M2 est que la première dispose de ressources qui sont des acteurs humains pourvus d'intelligence et de bon sens, capables de réagir à des situations nouvelles et à des aléas non prévus. Cette machinerie est auto-adaptative selon les capacités et l'expérience des acteurs humains ou des UA nucléaires. Un « fonctionnaire » est un acteur qui fait marcher la machine administrative conformément aux procédures administratives. Une armée en campagne est une très grosse UA dont les UA centres de commandement constituent le « système nerveux ».

3. Sur ces notions, d'un point de vue psycho-cognitif, voir J. Piaget, *L'équilibration des structures cognitives*, PUF, 1975.

8.3 LES CONTRAINTES DE L'AUTOMATISATION ET DE LA MACHINERIE INFORMATIQUE

Dans le monde M2, le système informatisé ne peut réagir qu'en fonction de sa programmation. Tout ce qui n'a pas été prévu comme un état du processus doit être diagnostiqué comme une défaillance qui nécessite une interruption, une suspension, voire une terminaison avec recours aux exploitants de l'infrastructure et aux utilisateurs métiers pour arrêter proprement le processus informatisé (i.e. une ou plusieurs applications/composants applicatifs selon notre terminologie, dans un ou plusieurs systèmes).

La surveillance dans le monde M2 est par certains côtés plus complexe que dans le monde M1, car toute défaillance non détectée à temps se traduira par une propagation d'états erronés qui, au bout du compte, détruiront l'intégrité du système (mort entropique ou de vieillissement). Les fuites de ressources, comme les « fuites mémoire », sont un bon exemple de ce type de dégénérescence. Si la traçabilité des transformations effectuées n'est pas organisée, il y aura quasi impossibilité de remonter à la cause de la défaillance du système, donc de le réparer dans de bonnes conditions CQFD. La traçabilité est un problème central du développement des architectures informatiques, a fortiori si la plate-forme d'exploitation est elle-même un système distribué.

La question fondamentale pour l'architecte système est de choisir ce qu'il doit modéliser des transformations et comportements observés dans le monde M1, afin d'en abstraire une solution d'architecture compatible avec les technologies envisagées (court terme) ou envisageables (moyen terme) durant le cycle de vie du système (cf. chapitre 5), à partir de laquelle il pourra construire un premier prototype, puis les versions successives, par capitalisation et compatibilité ascendante des versions (cf. la notion de modèle de croissance et d'adaptabilité du système, au chapitre 14). Rappelons que le cycle de vie d'un système peut être long, contrairement à ce que l'on croyait encore dans les années 80s et dont le passage de l'an 2000 fut le révélateur : 30 ans pour un système d'information (banque, administration, santé, etc.) et 40 à 50 ans pour des équipements technologiques (énergie, nucléaire, spatiale, etc.).

Le monde M1 peut être modélisé selon trois points de vue qui correspondent à des visions du monde bien différentes, que l'on peut schématiser comme suit (figure 8.5) :

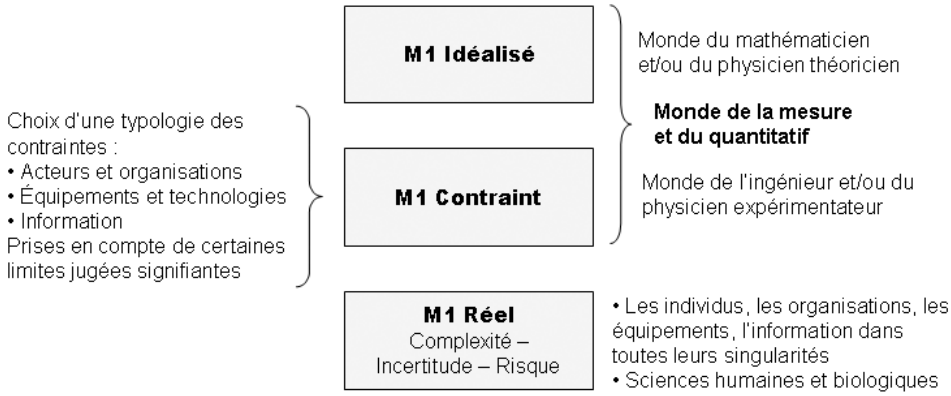


Figure 8.5 - Trois vues du système du monde M1

Le monde réel dans toute sa complexité organisationnelle et humaine est considéré comme non modélisable, dans l'état actuel de nos connaissances. Ce qui s'est fait de mieux dans ce domaine sont les modèles socio-économiques ; en particulier ceux issus de la théorie des jeux dont l'inventeur est von Neumann. Le pouvoir prédictif de ces modèles est cependant faible. Souvent, on ne peut décrire les situations qu'a posteriori, une fois les catastrophes arrivées (cf. U. Beck, *Risk Society – Towards a new modernity*, Sage publications, 1992 ; D. Vaughan, *The Challenger launch decision*, University Chicago Press, 1996, qui analyse en détail les aspects sociologiques de la catastrophe de la navette Challenger ; C. Morel, *Les décisions absurdes*, Gallimard, 2002 ; D. Kahneman, A.Tversky, *Choices, values, and frames*, Cambridge University press, 2000 ; etc.), ce qui est mieux que rien, mais tout de même frustrant pour un ingénieur.

Le monde biologique est également une bonne illustration de systèmes complexes dont on sait modéliser certains aspects mais dont on a aucune idée du fonctionnement et des régulations globales (voir le dossier de Pour la science, *La complexité, science du XXI^e siècle*, N° 314, 2003), malgré les désirs de certains biologistes qui aimeraient que la médecine devienne prédictive[4].

Le biologiste et prix Nobel F.Jacob, dans son livre célèbre, *La logique du vivant*, Gallimard, 1980, avait défini de façon qualitative une logique applicable à la biologie fondée sur une unité d'intégration appelée « intégron » qui renvoie à nos notions d'intégrat et d'UA.

À l'autre extrémité on idéalise au maximum, ce qui a comme effet de simplifier les phénomènes. La physique classique appuyée sur les mathématiques du continu nous a fourni une grande variété de modèles qui ont permis le décollage de la révolution industrielle, et la fabrication de nombreux équipements sans lesquels la vie socio-économique moderne serait impossible. La théorie des gaz parfaits donne une

4. Cf. J. Ruffié, *Naissance d'une médecine prédictive*, Odile Jacob.

vision idéalisée du comportement des gaz grâce à la loi de Mariotte $PxV=RxT$ qui relie la pression *P*, le volume *V* et la température absolue *T*. Pour l'ingénieur motoriste qui veut modéliser ce qui se passe dans une turbine ou un moteur à quatre temps, elle est insuffisante bien qu'elle fournisse un comportement limite qui ne pourra jamais être dépassé.

La position intermédiaire est celle du monde contraint, qui est celui des ingénieurs et des physiciens expérimentateurs qui se confrontent à la réalité, en lui ayant apporté des simplifications raisonnables (« simple, mais pas simpliste » aurait dit Einstein).

La logique de l'ingénieur est par nature, et par déontologie, déterministe pour la simple raison qu'il doit donner aux usagers qui utilisent ses artefacts, une garantie de service sans laquelle ses constructions n'auraient aucun intérêt. En cas de panne d'un équipement, l'ingénieur doit comprendre toute la séquence de décisions et de transformations qui ont conduit à la panne ; il doit être capable de la reproduire afin de formuler un diagnostic puis d'effectuer une correction. C'est ainsi que progressent la science et la technologie, souvent par essai-erreur, parfois avec l'aide de théories et de modèles mathématiques.

Il est surprenant de constater que nombre d'architectes fonctionnels considèrent que les performances ou la sûreté de fonctionnement ne font pas partie de leur domaine de préoccupation et que de ce fait, ce n'est pas leur affaire. Implicitement ils raisonnent dans un monde M1 idéalisé. Or le monde métier M1 est un monde contraint ; l'ignorer serait fatal à tout acteur métier, avec ou sans informatique. La fiabilité des informations qui s'échangent entre les acteurs, comme la sécurité qu'ils perçoivent, est un élément fondamental de la confiance qu'ils ont les uns dans les autres et dans l'organisation.

Toute action prend du temps, et comme dit le proverbe « on ne peut pas courir plus vite que la musique » comme peut parfois le laisser penser un volontarisme naïf ; aucune ressource n'est jamais la propriété exclusive de l'UA. D'une façon générale l'acteur métier individuel, les UA atomiques et/ou agrégées sont soumis aux caractéristiques FURPSE qui déterminent leur performance, selon des modalités propres au monde métier M1. Les systèmes qualité n'existent que pour corriger l'imperfection des actions humaines et améliorer la performance des processus métiers et optimiser les chaînes de valeur (cf. chapitre 5).

Nous considérons que la prise en compte de ces contraintes est une nécessité absolue dès la phase EB/EC si l'on veut élaborer des modèles informatiques qui reflètent fidèlement les besoins de processus et chaînes de valeur métiers. Cependant, il est indispensable que l'expression de ces contraintes soit faite selon une logique et un langage compréhensible par les métiers, et non pas en termes de solutions informatiques.

Nous pouvons compléter le tableau 2.3, états des entités architecturale, par le schéma 8.6 :

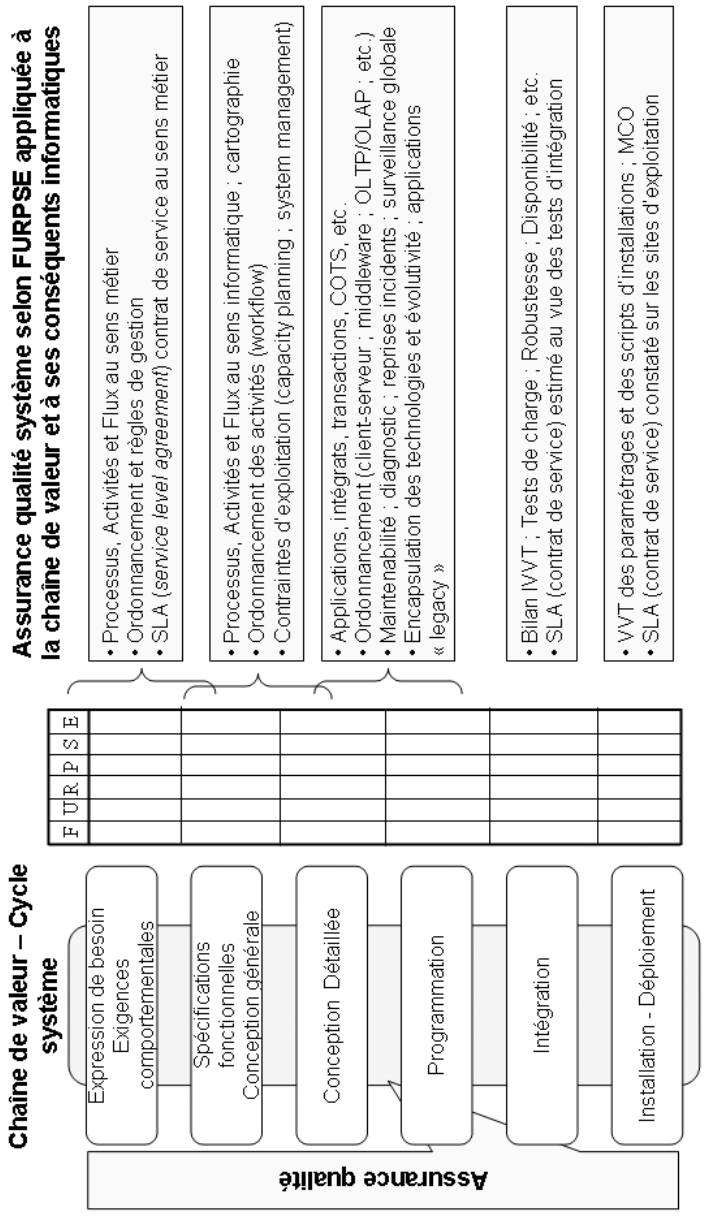


Figure 8.6 - Prise en compte des caractéristiques FURPSE dans la modélisation système

Si les caractéristiques non-fonctionnelles ne sont pas correctement exprimées dès la phase EB/EC, il est hautement improbable que les choix faits dans les phases ultérieures satisfassent comme par magie ce qui n'a pas été explicité du point de vue métier. En particulier, la traçabilité est impossible, ce qui reporte la vérification de la conformité au besoin métier en fin de cycle, c'est-à-dire beaucoup trop tard pour rétro-agir efficacement.

La position que nous défendons dans cet ouvrage, pour éviter une caricature de modélisation sans aucun intérêt, est la suivante :

> Intégrer dans la réflexion de l'ingénieur architecte les contraintes incontournables du monde réel qui font partie des fonctionnalités attendues du système, et ce dès les premières phases de la modélisation (phase EB/EC et CG).

En d'autre terme, il y a des exigences dites non fonctionnelles qui ont un sens du point de vue métier ; il faut les capter le plus en amont possible. Ce qui est fondamental dans la modélisation est la traçabilité de la correspondance entre le modèle métier et le modèle informatique qui matérialise les dépendances entre les entités modélisées. C'est ce que montre le schéma 8.7 :

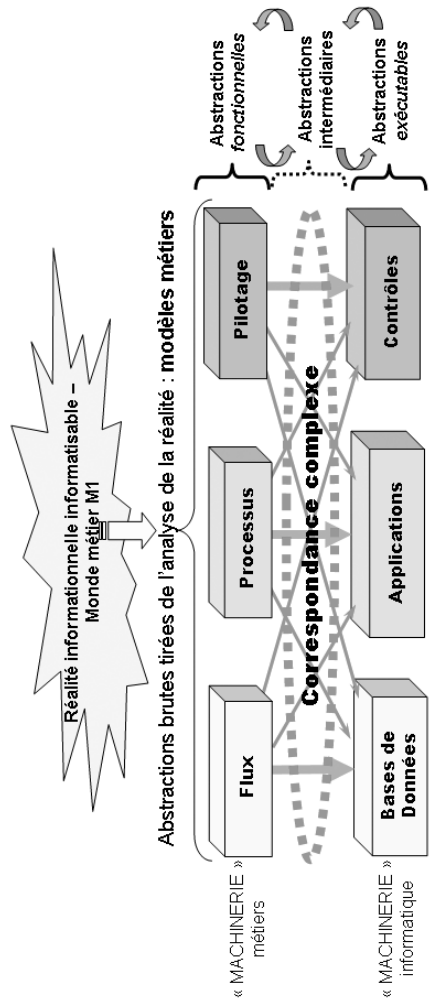


Figure 8.7 - Correspondance métier ↔ informatique.

Cette correspondance est intrinsèquement complexe car, certes il y a bien une correspondance naturelle FLUX ↔ BASE DE DONNÉES, PROCESSUS ↔ APPLICATIONS, PILOTAGE ↔ CONTRÔLES, mais cette correspondance n'épuise pas les combinaisons possibles.

Par exemple, un flux va générer des parties de modèle de données, mais il va également générer des fonctions de conformité des données portées par le flux, qui elles-mêmes vont générer des actions de contrôle.

Si l'on rajoute à ce schéma la dimension interopérabilité, tous les organes se dédoublent (voir chapitre 12), ce qui fait que au lieu d'un schéma simple à 3 + 6 flèches on obtient un schéma à 6 + 30 flèches (pour la correspondance métier ↔ informatique), auquel il faut rajouter les nouvelles correspondances issues du dédoublement du plan informatique, soit 18 flèches de plus, pour gérer les dépendances fonctionnelles potentielles entre les éléments fédérés et les éléments spécifiques du système considéré.

En terme de combinatoire statique, les combinaisons deux à deux de N entités varient comme $\frac{N(N-1)}{2}$ soit une complexité statique en $O(N^2)$; alors qu'en combinatoire dynamique où il faut intégrer tous les cheminements possibles, il faudra considérer l'ensemble des parties, soit 2^N.

NB : Les correspondances complexes qui relient les entités métier aux entités informatiques (transactions et/ou données) ne sont pas des arborescences mais probablement des treillis, voire même des catégories. D'où les grandes difficultés à les faire évoluer de façon cohérente.

L'objectif fondamental de la modélisation est de contrôler la taille de l'ensemble des correspondances possibles, de façon à ce que l'ingénierie correspondante reste dans le domaine du raisonnable, c'est-à-dire linéaire, ou localement en N^2. Toute combinatoire supérieure sera généralement fatale au projet, d'ou une règle de bon sens de l'ingénierie de l'interopérabilité.

Règles d'ingénierie :

- Ne jamais rendre interopérable ce qui n'a pas lieu de l'être.
- Appliquer systématiquement et intelligemment la règle de subsidiarité (c'est-à-dire faire baisser la complexité, sinon la subsidiarité ne sert à rien) quitte à développer quelques traducteurs supplémentaires.
- Adopter des règles de partition des ensembles de données qui intègrent la dimension dynamique des traitements de façon à rendre explicite les dépendances fonctionnelles entre les données.
- Dans tous les cas de figure, automatiser ce qui peut l'être (une combinatoire en N^2, si N n'est pas trop grand, reste gérable avec la puissance des machines dont nous disposons aujourd'hui ; une combinatoire supérieure est ingérable et mettra le projet en grand péril).

Les modèles issus de la phase de conception du système doivent être déterministes, quand bien même le comportement d'un intégrat de niveau inférieur pourrait être non déterministe.

Pour faire court, nous avons résumé ces contraintes par les sigles : PESTEL, FURPSE, CQFD et TCO. Ces contraintes font partie de la réalité dont il faut s'occuper dès le début lors du passage M1→M2, et ne pas attendre une solution hypothétique des seuls informaticiens dans un monde M2 qui n'est plus fidèle au réel. Ce travail est au centre de la relation MOA, MOE, Architecte-Urbaniste (voir chapitre 5, section 5.2).

Les organisations humaines, les entreprises (pour ne pas parler des systèmes biologiques) ont toutes des systèmes immunitaires qui garantissent leur survie, et compensent leurs défaillances, et ceci a été acquis bien avant l'invention des ordinateurs (NB : pensez aux services de renseignement intérieur et extérieur). C'est cette préoccupation de sûreté de fonctionnement du système, quelle que soit la nature du système, qu'il faut mettre au centre de la problématique de nos constructions artificielles, de part et d'autre de la sphère de contrôle.

En terme d'ingénierie et de vécu d'ingénieur, on peut énoncer le principe.

Principe de testabilité : Il est futile, pour un ingénieur-architecte de concevoir un système que :

- l'on ne saura pas tester (phase IVVT pour le MOE), d'où l'exigence de déterminisme.
- l'on ne pourra pas financer dans la durée : CQFD des projets et TCO du système, y compris le coût du retrait (le SE de FURPSE).
- l'on ne saura pas intégrer dans l'environnement et qui occasionne plus de problèmes par les effets induits qu'il n'en résout dans le processus métier initial (facteurs PESTEL).

L'erreur humaine est un phénomène naturel, relativement bien étudié[5], qui touche tous les acteurs du système sans exception. Les statistiques nous indiquent qu'un acteur humain en activité peut commettre de 5 à 10 erreurs par heure d'activité. Ne pas en tenir compte revient à se voiler la face, à fuir la réalité et à renvoyer la réponse à plus tard, selon le syndrome bien connu dit du « bâton merdeux » que l'on se repasse sans avoir le courage, ni la volonté de le nettoyer une bonne fois pour toutes.

Pour une IHM, c'est tout simplement se préparer à une catastrophe qui se produira immanquablement. Les diverses théories de l'information et de la fiabilité nous enseignent que face à l'erreur aléatoire, la seule attitude responsable est : redondance, au niveau du procédé de construction, au niveau de l'architecture du système. Comment l'organiser, la mettre en œuvre via le système qualité qui, dans l'organisa-

5. J. Reason, *L'erreur humaine*, PUF, 1990 ; F.Vanderhaegen, *Analyse et contrôle de l'erreur humaine*, Hermès, 2003.

tion de l'entreprise et des projets, a le rôle du système immunitaire (même si aujourd'hui le terme qualité est connoté négativement, cela ne retire rien à la fonction qualité qui est fondamentale ; dans les systèmes biologiques, l'AQ s'appelle système immunitaire).

Quand un acteur métier commet une erreur, son premier réflexe est de revenir à un état antérieur jugé par lui cohérent. De deux choses l'une :

- soit il dispose d'un jeu de commandes qui lui permettent de défaire « proprement » ce qu'il a fait et qu'il juge erroné.
- soit il a la possibilité de faire appel aux acteurs MCO (exploitants et/ou support) pour leur demander d'effectuer la réparation préconisée.

Dans les deux cas, il faut pouvoir défaire ce qui a été fait, ou intervenir auprès de l'UA si une action irréversible a été déclenchée. Pour cela, la journalisation des opérations est indispensable (cf. le livre de bord dans la marine ; ou le journal des modifications d'une base de données). Une ressource consommée doit pouvoir être remplacée, fusse avec un délai de reconstitution du stock.

Quand un acteur développement logiciel commet une erreur, il introduit un défaut dans le logiciel qui, s'il n'est pas découvert dans le processus IVVT du processus de développement, restera latent jusqu'à ce qu'un utilisateur le découvre, peut-être, en exploitation. Il faudra alors que l'exploitation et/ou le support puisse effectuer le diagnostic afin de corriger le défaut. Les exploitants doivent disposer d'un minium d'instrumentation pour effectuer le diagnostic. Les équipes de maintenance doivent pouvoir effectuer la correction et s'assurer de la non-régression avant de procéder à la livraison de la correction.

La technologie, et plus particulièrement la technologie informatique fortement tributaire des acteurs humains, est soumise également à la loi universelle de l'erreur (c'est l'entropie naturelle). Un équipement informatisé, outre ses limitations physiques, contient des programmes avec des défauts latents. L'informatique qui nous a parfois habitué à des « miracles », est comme toutes les autres soumise à des lois entropiques qui font que le désordre augmente inéluctablement. Ne pas en tenir compte, c'est courir droit à l'échec. Il est indispensable de les connaître et d'en suivre les évolutions pour en mesurer le risque.

Par exemple, la performance d'un équipement informatique ne se réduit pas à la performance du micro-processeur. Il y a les entrées-sorties vers les disques et/ou les réseaux qui, relativement au temps de cycle du micro-processeur, sont très lentes : 10 à 20ms. Ce qui veut dire qu'une interaction IHM dont l'exécution provoque 10 000 E/S aura un temps réponse de l'ordre de 2 à 3 minutes, ce qui est tenu pour inacceptable. La performance reste un problème central de toutes les architectures clients-serveurs, dès que le nombre de poste de travail (IHM/GUI) dépasse quelques centaines.

Le challenge du jeu des acteurs MOA, MOE, architecte-urbaniste est de trouver le meilleur compromis PESTEL/FURPSE/TCO/CQFD entre le niveau de risque

acceptable pour l'organisation cible, la performance globale et les coûts (coûts instantanés et coûts complets) :

a- les coûts de développement d'un ensemble de projets à court terme qui vont adapter le système d'information à sa finalité immédiate (alignement stratégique).

b- les coûts de MCO (Maintien en condition opérationnelle : maintenance, support et exploitation qui sont un aspect du TCO) qui garantissent le niveau de service (SLA, *Service Level Agreement*) demandé par les utilisateurs afin d'améliorer l'efficacité des chaînes de valeur métier du monde M1.

c- Le coût de possession totale (TCO) qui intègre les coûts précédents sur le long terme afin de dégager le ROI du système complet, relativement à son contexte d'emploi.

Ce jeu à trois acteurs est un jeu à somme nulle que l'on peut résumer par un dilemme qui devra être tranché par l'architecte urbaniste, en toute lucidité et humilité :

- Pour réduire les coûts (CQFD/TCO) à risque constant, la performance (FURPSE) doit être réduite.
- Pour réduire le risque à coût (CQFD/TCO) constant la performance doit être réduite.
- Pour réduire le coût à performance constante, un risque (PESTEL + Complexité, Incertitude) plus élevé doit être accepté.
- Pour réduire le risque (PESTEL + Complexité, Incertitude) à performance constante, un coût plus élevé doit être accepté.

C'est l'intégration dynamique de ces différentes catégories de contraintes qui vont permettre d'effectuer les choix architecturaux de façon explicite. La construction de la machine informationnelle est effectuée à partir de module fonctionnel (« building block ») que nous avons francisé en « INTÉGRAT ».

8.4 ORGANISATION HIÉRARCHIQUE DES INTÉGRATS – VISION STATIQUE DE LA MACHINE INFORMATIONNELLE

La machinerie informatique (cf. figure 8.7) est décomposable en un ensemble d'intégrats en interaction qui vont constituer les organes logiques de la « machine informationnelle» qui réalise le service attendu par l'organisation cible, selon les modalités contractuelles qui viennent d'être rappelées.

En prenant comme élément de construction l'intégrat tel qu'il résulte des analyses du chapitre 2 (cf. figures 2.1 et 2.9), du point de vue de l'architecture, un intégrat aura toujours l'allure topologique ci-dessous. La topologie dont il est question ici est

d'ordre « géométrique ». Elle consiste à marquer les éléments du bloc qui participent au contrôle, aux transformations et à la gestion des ressources car la logique qui fonde la construction de ces différentes composantes n'est pas la même. Dans la programmation réelle il y a enchevêtrement, mais on a vu, dans la partie 2, que dans le cas des traducteurs on peut séparer radicalement la composante contrôle sous une forme grammaticale.

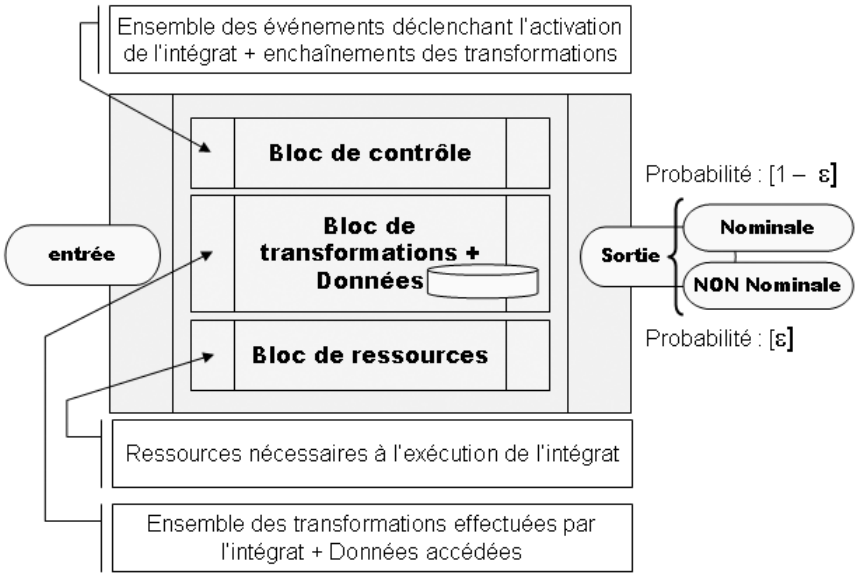


Figure 8.8 - Intégrat du point de vue de l'architecte et de l'intégrateur

Conformément au principe de décomposition hiérarchique vu au chapitre 2, on pourra selon le besoin d'intégration travailler sur des intégrats de type application, ou sur des intégrats de rang inférieur nécessaires à la mise en œuvre d'un procédé d'intégration complètement contrôlé.

Dans le monde contraint de l'ingénieur, l'intégrat contient un nombre de défauts latents (i.e. non détectés par la stratégie de test IVVT) qui est une fonction croissante de l'effort nécessaire à sa réalisation (ou à sa taille, ce qui revient au même). Si l'intégrat intègre des équipements et des progiciels, les défauts de ceux-ci viennent s'additionner aux défauts propres à l'intégrat, augmentant ainsi le niveau d'incertitude.

La performance de l'intégrat, au sens P de FURPSE (mémoire nécessaire + temps de réponse) dépend de la nature des ressources utilisées, des données partagées (hors de la sphère de contrôle de l'intégrat) et de la nature des transformations effectuées sur les entrées et les sorties.

Tous ces éléments qui interviennent dans la traçabilité de l'intégrat doivent être visibles de l'intégrateur ; ils constituent la configuration de l'intégrat. C'est ce que montre le schéma 8.9 :

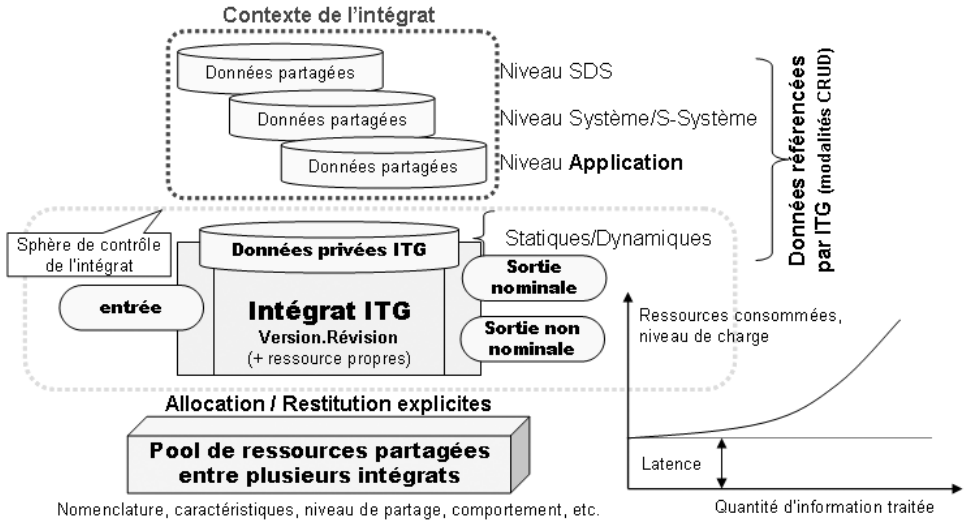


Figure 8.9 - Configuration statique et dynamique d'un intégrat

La forme de la courbe des ressources consommées en fonction de la quantité d'information traitée est une donnée fondamentale pour l'architecte. C'est la complexité algorithmique, au sens classique[6], correspondant au nombre d'opérations réalisées pour rendre le service ; c'est en quelque sorte le « prix » de la transformation effectuée par l'intégrat, eu égard aux ressources disponibles. L'« économie » des ressources est une tâche fondamentale de l'architecte ; ne pas s'en préoccuper à temps est une condition suffisante d'échec, et une grave pathologie en terme de management de projet. Cette complexité a évidemment son équivalent dans le monde métier que l'on cherche à automatiser.

La nature des données et des ressources partagées peut avoir un effet très important sur les performances. Il est fondamental pour l'architecte d'avoir une idée aussi exacte que possible, même qualitative, des performances induites : E/S rapides (disques) ou lentes (réseaux, selon capacité des réseaux).

L'histoire de l'intégrat, du point de vue du cycle de développement, telle qu'on pourrait la trouver dans un gestionnaire de configuration, est résumée par le schéma 8.10.

6. Cf. J. Printz, *Puissance et limites des systèmes informatisés*, chez Hermès-Lavoisier, avant d'attaquer des ouvrages plus complets sur le sujet.

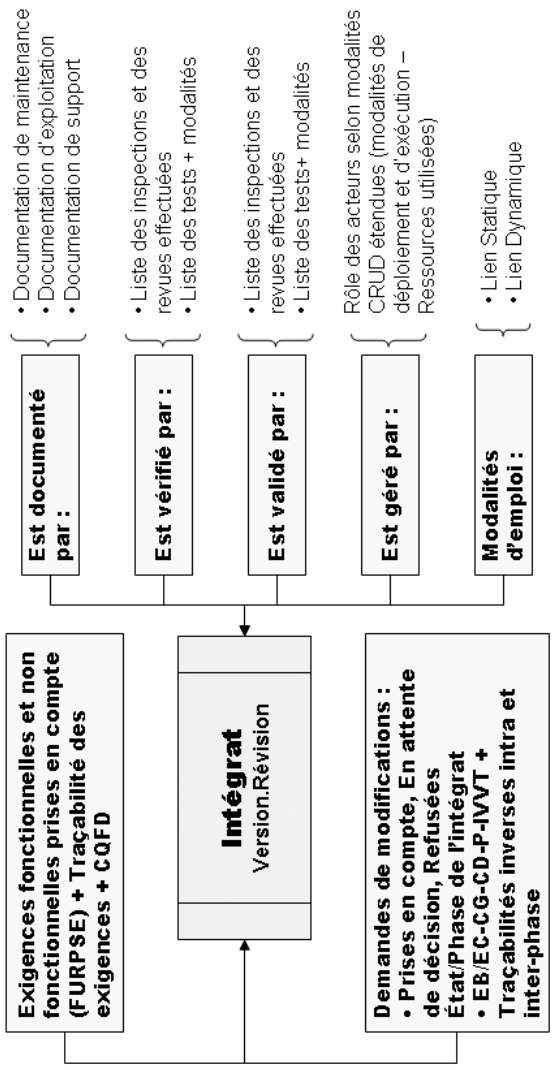


Figure 8.10 - Attributs projet indispensables de l'intégrat

Ces informations de traçabilité sont vitales pour la mise en œuvre du processus d'intégration[7].

Conformément aux principes de modularité (cf. D.Parnas, B.Meyer[8], redécouverts par J.Sassoon dans son livre sur l'urbanisation du SI) qui sont d'abord des règles

7. Cf. J. Printz, *Ecosystème des projets informatiques*, Hermès 2005.
8. Cf. bibliographie. Pour une approche graduée, on peut lire R. Orfali et al., *Client/server survival guide*, Wiley, 1999, puis P. Bernstein, E. Newcomer, *Principles of transaction processing*, Morgan Kaufmann.

de bon sens intuitivement évidentes, un intégrat a un point d'entrée et un point de sortie. Dans le schéma 8.8 nous avons fait ressortir les états de sorties nominales et non-nominales qui seront pris en compte par le mécanisme d'enchaînement des intégrats (automate d'enchaînement). L'observation de l'enchaînement peut alors se faire de façon systématique (cf. chapitre 13, Disponibilité).

Notons que la contrainte de modularité est très forte, car elle interdit un style de programmation, utilisé par défaut par beaucoup de programmeurs qui n'ont pas bien compris la mécanique des enchaînements en environnement d'exécution interactif (partage des ressources, concurrence, distribution, etc.), et dont l'effet est catastrophique pour la sûreté de fonctionnement. Le contrôle de la qualité du code par l'architecte est fondamental. La bonne mise en œuvre de la règle exige que toute exception, toute interruption, tout code retour de l'activation d'un service, toute défaillance soit analysée dans l'intégrat et passe par la case sortie. Idem pour le critère d'entrée dans le processus d'intégration.

Le schéma de principe de cette règle fondamentale est le suivant (figure 8.11) :

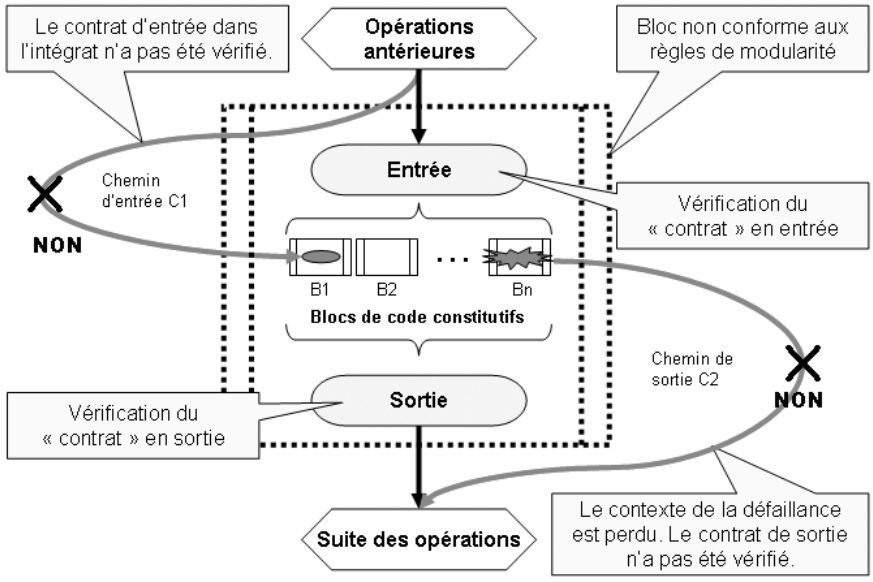


Figure 8.11 - Règle de modularité 1 entrée – 1 sortie

L'entrée dans l'intégrat par le chemin C1 viole le principe de la programmation par contrat, car l'accès se fait directement dans B1, sans passer par le point d'entrée. De même l'échappement par le chemin C2 occasionne la perte du contexte de la défaillance, ce qui fait qu'aucune réparation ne sera possible.

Le respect de la règle de modularité permet de rassembler les informations qui déterminent l'état de sortie de l'intégrat, nominal ou non nominal. Par analogie

avec ce qui se passe dans la machine, la structure logique correspondante est appelée vecteur d'état, comme suit (figure 8.12) :

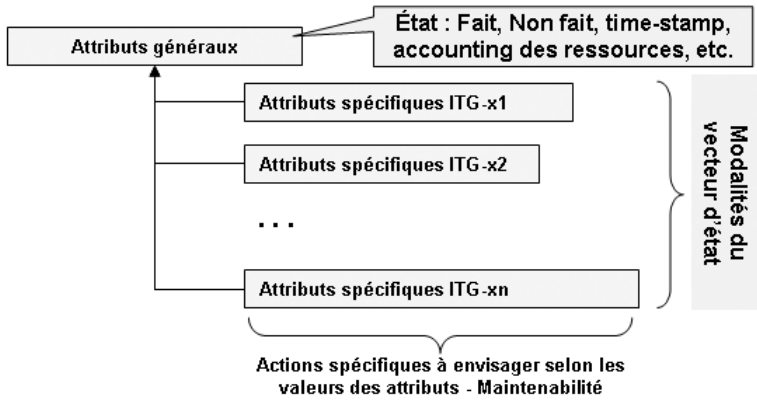


Figure 8.12 - Etat de sortie d'un intégrat

Les attributs généraux synthétisent l'historique de l'exécution de l'intégrat telle que l'architecte le souhaite. Il faut en particulier disposer de l'information permettant d'enchaîner la suite des opérations, soit en nominale, soit en non nominale. Un minimum d'accounting sur les ressources est intéressant à ce niveau, par exemple la durée en temps vrai de l'opération, le temps d'occupation CPU, la mémoire allouée, le nombre d'entrées-sorties générées, etc.

Les attributs spécifiques donnent des informations beaucoup plus fines sur les modalités de l'exécution pour la disponibilité et la maintenabilité de l'intégrat ; par exemple, tel service a été appelé trois fois avant d'être disponible, ce qui peut indiquer un niveau de saturation. Ces informations sont exploitées par le système de surveillance général de l'exécution des intégrats ; elles sont vitales pour la sûreté de fonctionnement et la disponibilité (cf. chapitre 13).

8.5 ENCHAÎNEMENT DES INTÉGRATS – VISION DYNAMIQUE DE LA MACHINE INFORMATIONNELLE

La bonne compréhension de l'architecture, telle qu'elle a été esquissée aux chapitres 2 et 3, repose sur une analyse approfondie des changements d'états opérés par les transformations résultant des enchaînements réalisés pour satisfaire les cas d'emplois issus du monde M1. L'intégrat application, au sommet de l'arbre produit (cf. la figure 2.9), a la forme générale des intégrats telle que décrite par les figures 8.8 et 8.9.

Cet intégrat est mis en exécution par le moniteur de travaux du système d'exploitation, éventuellement via un ordonnanceur. La supervision est réalisée par un opé-

rateur humain (exploitant) qui est le pilote externe et/ou par un opérateur programmé qui assiste l'exploitant ; c'est le workflow de plus haut niveau opéré par le système.

Les états possibles de l'intégrat en cours d'exécution sont conformes à ceux définis par la figure 8.4 (ligne de vie de l'intégrat).

En se concentrant sur l'aspect contrôle, l'arbre d'intégrat est doublé (c'est une forme de dualité) d'un arbre de contrôle dont les nœuds ont la structure suivante (figure 8.13) :

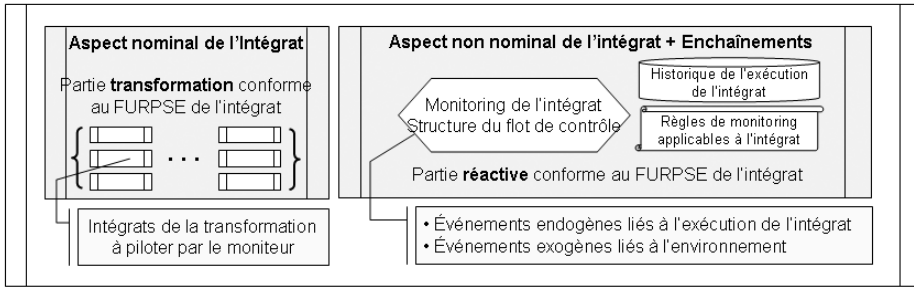


Figure 8.13 - Monitoring des intégrats

La présentation de l'intégrat sous cette forme revient à séparer le texte du programme de l'intégrat en deux parties, 1) le texte correspondant à la partie purement transformationnelle, et 2) le texte correspondant à la partie réactive que matérialise les changements d'état.

Dans la réalité de la plupart des programmes, ces deux textes sont enchevêtrés. Un programmeur débutant mal formé aura la tentation de tout mélanger, d'où un texte compliqué et arbitraire ; un programmeur expert sera très vigilant et distinguera soigneusement les deux textes.

Dans le chapitre 6, nous avons vu, dans l'exemple des compilateurs, une distinction radicale de ces deux textes avec une composante réactive programmée de façon déclarative à l'aide d'une grammaire, et une composante action programmée de façon telle que chaque action soit indépendante (c'est en fait une instruction d'une machine abstraite d'analyse syntaxique).

En matière de CQFD projet, une telle programmation, est régie par un modèle de coût dont le coefficient d'intégration peut être inférieur à 0, appelé économie d'échelle dans le modèle COCOMO II. C'est toute la différence entre un programmeur débutant et un concepteur-programmeur expert.

Le pilotage (i.e. monitoring) est effectué sur la base des évènements reçus via le canal d'entrée, qui joue le rôle du compteur ordinal de la machine, en fonction des règles de monitoring applicables à l'intégrat. La programmation correspondant à ces règles peut être de nature impérative ou de nature déclarative (par exemple, une table de décision logique), les évènements peuvent résulter de l'exécution de l'inté-

grat (évènements endogènes) ou d'actions effectuées dans l'environnement de l'intégrat (évènements exogènes).

Par exemple la détection d'une incohérence à l'exécution va générer l'édition d'un message d'erreur et le positionnement d'un code retour permettant l'enchaînement d'une action appropriée permettant la poursuite de l'analyse. Le langage de programmation utilisé dépend du niveau de profondeur de l'intégrat. Au niveau application, c'est le JCL (Job Control Language ; ou SHELL dans UNIX) du système d'exploitation.

Parmi les évènements, deux d'entre eux jouent un rôle particulier :

1. l'événement début qui déclenche l'activation de l'intégrat : DÉMARRER l'intégrat
2. l'événement fin qui déclenche la terminaison de l'intégrat et la restitution des ressources qui lui ont été attribuées : ARRÊTER l'intégrat. Cet événement peut être levé de l'intérieur (endogène) ou de l'extérieur (exogène) par un pilote de plus haut niveau.

Dans une vision couche hiérarchisée de l'intégrat, la structure de l'enchaînement de l'intégrat racine serait matérialisée comme suit (figure 8.14) :

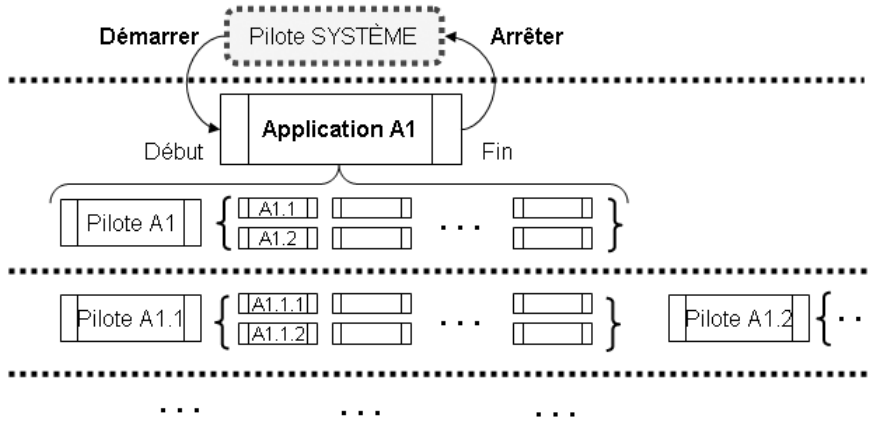


Figure 8.14 - Structure hiérarchique des enchaînements

L'ensemble des pilotes constitue l'automate d'enchaînement de l'application. On pourrait lui donner une forme déclarative de type grammaire, mais ce qui émerge sont plutôt des langages de type impératif, comme le BPML et/ou divers langages de workflow.

La matérialisation de ces niveaux de pilotage est un élément fondamental de la testabilité de l'architecture (cf. chapitre 13) et de l'adaptabilité (cf. chapitre 14).

C'est ce que dans la littérature des tests (cf. B. Beizer, *bibliographie*) on appelle un langage caché.

Idéalement, pour la simplicité de la programmation, un intégrat ne devrait avoir ni contexte, ni partage de ressources. C'était le cas des applications batch avant l'avènement des applications conversationnelles de type OLTP/OLAP, jusque dans les années 80.

Les applications correspondantes étaient simples, mais incapables de s'adapter à différents contextes (par exemple le multilinguisme) et/ou à différents profils utilisateurs (novice, expert,…). Pour les adapter, il fallait les reprogrammer, ce qui multipliait les versions et créait de la complexité. La construction se faisait de façon rigide, via l'éditeur de liens statiques (cf. figures 8.4 et 2.3).

Dans la décennie 90, avec l'arrivée des langages objets comme C++, Java, C # et quelques autres, les modalités de la construction de l'intégrat sont devenues beaucoup plus souples, donnant d'une part des possibilités d'adaptation au contexte, mais d'autre part créant de nouveaux problèmes de validation car le processus d'intégration devient dynamique ; dans ce cas, il est essentiel de laisser la technologie d'intégration à demeure dans les composants applicatifs. Les états successifs de la construction sont décrits dans la figure 8.15.

Tous ces mécanismes sont bien connus des programmeurs systèmes, depuis les années 70, via les éditeurs de liens dynamiques dont celui du système MULTICS qui a été la référence incontournable de tous ceux qui lui ont succédé. Ils sont désormais accessibles à tous les programmeurs, même s'ils n'en ont pas conscience, comme le couper-coller des applications Microsoft via l'interface OLE/DCOM.

Bien utilisés, ils donnent un très grand pouvoir d'expression à l'architecture ; mal utilisés, c'est un énorme facteur de risque d'incohérence, de violation des règles de sécurité, de chute drastique de la disponibilité et de la qualité de service.

Mais avant de les utiliser, l'architecte doit s'assurer que l'équipe de programmation à la capacité et le niveau de maturité requis pour leurs mises en œuvre sûre dans les projets.

NB : Sur cet aspect projet de l'architecture, voir le chapitre 4.5 Dynamique des processus d'ingénierie, de notre livre : *Ecosystème des projets informatiques*, chez Hermès, 2005, où le sujet est traité plus en détail.

Du point de vue de la gestion de configuration (cf. figure 8.10), il faut distinguer les articles de configuration (ACL) statiques, des ACL dynamiques susceptibles d'être utilisés durant l'exploitation.

Ces derniers doivent avoir un niveau de confiance en adéquation avec le risque qu'ils font courir à l'exploitation ; idéalement , cela devrait être explicité dans le contrat de service.

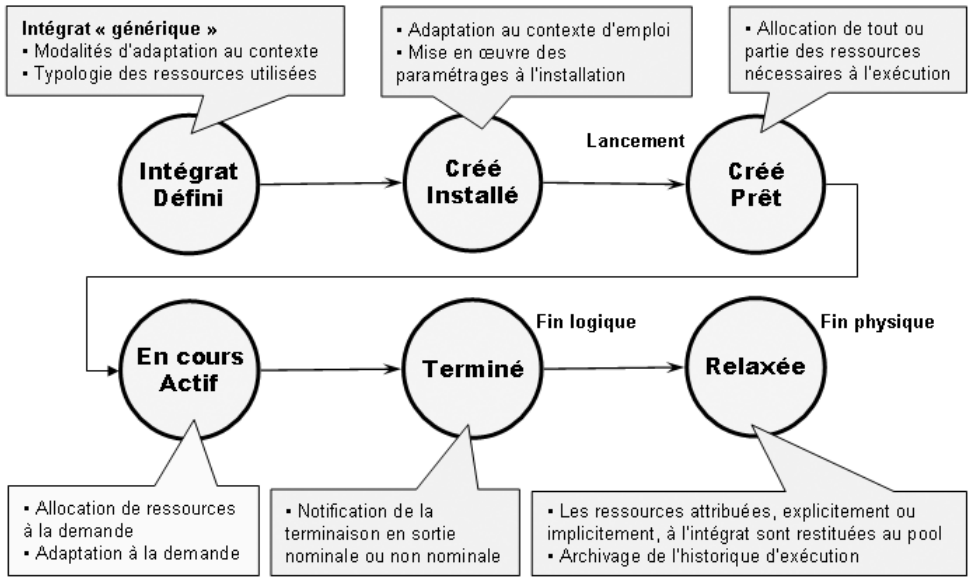


Figure 8.15 - Etat de la construction de l'intégrat

9

Propriétés sémantiques des intégrats – Transactions – Services

9.1 TRANSACTIONS

Les systèmes transactionnels de type OLTP, OLAP, puis client-serveur, développés à partir des années 70-80 ont montré l'importance cruciale de gérer de façon cohérente la relation entre les actions effectuées par les UA du Monde M1 et celles effectuées pour le compte des précédentes, dans le monde informatisé M2. Le fondement de cette mise en cohérence est le concept de transactions tel qu'il a été développé, en particulier sous l'impulsion de J. Gray, A. Reuter[1], et quelques autres, dans les années 80.

La notion de transaction informatique prend sa source dans la notion de transaction métier, beaucoup plus ancienne, telle qu'elle apparaît dès qu'il y a contrat entre plusieurs parties prenantes (*stakeholders*).

Tout agent économique, chacun d'entre nous, informaticien ou non, a une expérience intuitive et concrète de ce qu'est une transaction dans le monde M1. Réserver une place d'avion, passer une commande (via Internet ou non) met en interaction deux parties prenantes et un tiers de confiance, au minimum, dont le résultat sera la garantie de réception d'un bien pour celui qui effectue la commande, contre la remise d'une certaine somme d'argent (avec garantie que le compte de l'acheteur est effecti-

1. Cf. bibliographie. Pour une approche graduée, on peut lire R. Orfali et al., *Client/server survival guide*, Wiley, 1999, puis P. Bernstein, E. Newcomer, *Principles of transaction processing*, Morgan Kaufmann.

vement approvisionné) ou d'un bien de valeur équivalente (en cas de troc, comme cela est encore souvent le cas dans le négoce international) à la fourniture à celui qui fournit le bien ou le service ; dans ce dernier cas, c'est un échange entre deux stocks. Le modèle de transaction élémentaire le plus universel est le « débit-crédit » que tout le monde peut comprendre. En cas d'achat avec garantie, le débit du compte bancaire de l'acheteur (i.e. le client) s'accompagne simultanément du crédit de celui de la banque, qui garantira (via les mécanismes de compensation) que le vendeur (i.e. le fournisseur) pourra acheminer son bien (i.e. une baisse de son stock) avec comme contrepartie la garantie, via la banque, que son compte sera effectivement crédité. En cas de litige, c'est la banque qui garantie la régularité de la transaction.

Le modèle logique d'une telle opération peut être schématisé comme suit (figure 9.1) :

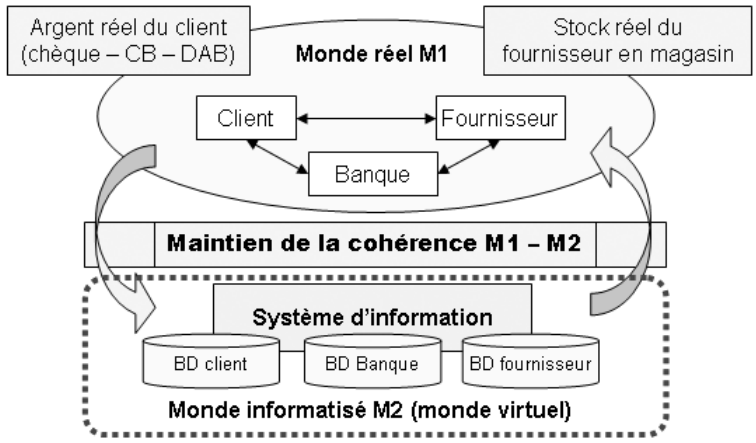


Figure 9.1 - Articulation des transactions métiers et des transactions informatiques

La banque joue le rôle de tiers de confiance. Dans le monde contraint où nous nous situons, il faut prendre en compte le fait que :

- le client et/ou le fournisseur peuvent arrêter leur transaction et revenir à un état pro ante où il ne s'est rien passé ; la transaction est purement et simplement annulée. Le client utilise le délai de garantie pour annuler la transaction, retourner le bien au fournisseur et récupérer son argent ; dans ce cas, il faut compenser. Etc.
- le système d'information peut connaître pannes et défaillances qui nécessitent des réparations informatiques, dont les usagers métiers n'ont pas à se soucier.

Dans ce cas, il faut s'assurer que la réparation effectuée maintient la relation de cohérence M1/M2. On distinguera, du point de vue M2, les actions protégées des actions non protégées qui, en cas de défaillance, laissent le système dans un état quelconque à la charge de l'administrateur.

Le maintien de cette cohérence est le problème fondamental des systèmes transactionnels OLTP/OLAP car le monde virtuel M2 est fondamentalement réversible (on peut toujours revenir à un état antérieur), du moins en théorie, que le système informatique soit déterministe à un certain niveau d'abstraction, ou non (dans ce cas, l'état restauré n'a aucun rapport avec la réalité passée, ce qui revient à truquer, ou mentir !) alors que le monde M1 ne l'est pas. Une ressource, un stock consommé, est consommée ; pour revenir en arrière, il faut reconstituer le stock, ce qui a un coût, et à condition que cela soit possible. À vrai dire, dans ce cas précis, on ne doit même pas parler de retour arrière, c'est simplement une évolution de la réalité qui restaure en partie une situation passée.

Il est clair que si la réparation, au sens informatique, n'est pas le reflet des réparations effectuées par les acteurs métiers, l'incohérence M1/M2 est inéluctable. En conséquence, toute réparation informatique n'est « vraie » que si l'acteur métier déclare qu'elle l'est effectivement. La logique applicable au monde des transactions (i.e. décider ce qui est VRAI et ce qui est FAUX) est entièrement pilotée par les acteurs métiers, seuls juges de la validité des opérations.

Dans un tel contexte métier (i.e. sémantique), que peut proposer la technologie informatique pour assurer la cohérence M1/M2, en prenant en compte la survenue de défaillances informatiques (Monde M2 contraint, et non pas M2 idéalisé).

La réponse à cette question fondamentale pour la sûreté de fonctionnement du logiciel est le concept de transaction informatique tel qu'il a été élaboré tout au long des décennies 70-80.

9.1.1 Le modèle classique des transactions ACID

La notion de transactions informatique, en abrégé **TP** ou **TPR** pour Transaction Programme Routine, généralise la notion d'instruction machine élaborée par les architectes des premiers vrais ordinateurs dans les années 50-60. La transaction est un quantum de transformation élémentaire d'un état mémoire à un autre état mémoire. Elle joue le rôle d'une instruction d'une machine abstraite à états (MAE) définie par l'architecte et programmée par le programmeur.

Rappelons que, idéalement, une instruction machine est un changement de l'état mémoire de la machine résultant de l'exécution de l'instruction. Cette exécution est indivisible du point de vue de l'interface hardware-software, la seule interface connue du programmeur d'application, alors qu'elle ne l'est pas du point de vue de la micro-machine, et a fortiori vue du microprocesseur. Pour le programmeur, l'exécution d'une instruction machine n'a que deux états possibles : FAIT ou NON FAIT ; dans ce dernier cas, la mémoire est inchangée, il ne s'est rien passé, sauf l'exécution de quelques cycles de l'unité centrale ; le compteur ordinal du processus n'a pas progressé.

La forme logique de toutes les instructions machine est basée sur le modèle à deux opérandes (ce sont des adresses mémoire) :

Type_de_l'instruction (adresse d'une donnée en mémoire de type T1) → (adresse de la mémoire où ranger le résultat, de type T2)

Soit : `TR_I (donnée entrée) → (donnée résultat)`, qui est une fonction indivisible avec deux opérandes

La généralisation d'une telle opération, du point de vue du programmeur d'application, se formalise comme suit :

`TR_I_généralisée (DE1, DE2, …, DEn) → (R1, R2, …, Rm)` , qui est une fonction partielle[2] avec $n + m$ opérandes de types quelconques.

NB : On remarquera la similitude avec la notion d'intégrat.

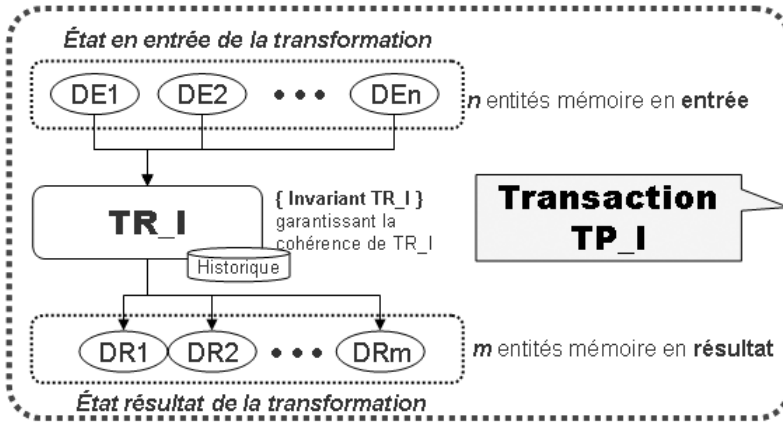


Figure 9.2 - Instruction généralisée

Pour être valide, et indépendante du contexte d'exécution, l'acquisition de l'état en entrée devrait être instantanée ; les entités mémoire relatives à cet état sont alors réservées à l'instruction en cours. Une fois la transformation opérée, le résultat est retranscrit instantanément dans les entités mémoire relatives à l'état résultat.

Dans la réalité de la machine, aucune de ces opérations n'est instantanée et à coût nul ; il faut des protocoles d'acquisition de façon à verrouiller toutes les ressources nécessaires. Notons qu'un état peut être constitué de données appartenant à un même espace d'adresse, à plusieurs espaces d'adresse sur une seule machine ou sur plusieurs machines. Idem pour la transformation. La complexité des protocoles d'acquisition dépend du degré de distribution de l'état sur les différentes ressources

2. Dans une fonction partielle, les opérandes n'ont pas obligation à être tous définis, s'ils ne sont pas utilisés dans l'exécution. Le terme a été introduit par Turing lui-même. Voir J. Vélu, *Méthodes mathématiques pour l'informatique*, Dunod et les ouvrages d'informatique théorique, par exemple l'excellent P. Denning, *Machines, languages and computation*, Prentice Hall.

utilisées et sur les équipements gérant ces ressources (voir chapitre 12, pour la complexité ainsi engendrée).

Un état mémoire, aussi complexe soit-il, est la traduction en termes informatiques, d'une situation cohérente du monde M1 (dans la logique qui est celle de M1). Le rôle des transactions informatiques est de maintenir en permanence la cohérence M1/M2. M2 n'est l'exact reflet de M1, et inversement, que si l'évolution de M1 est fidèlement modélisée par un ensemble de transformations discrètes de M2 dont l'enchaînement résulte des actions effectuées par les usagers Ui. C'est l'usager, et lui seul, qui fixe la « vérité » des transformations effectuées.

On peut schématiser la transformation comme suit (figure 9.3) :

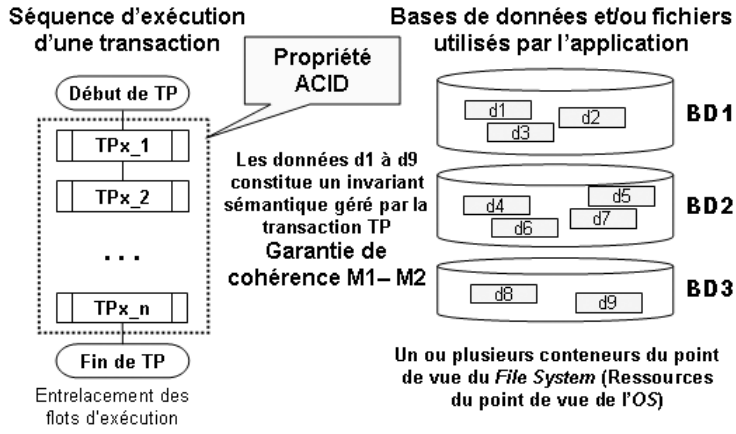


Figure 9.3 - Schéma d'exécution d'une transaction

Sur la figure, on voit que la transformation résulte d'une suite de transformations élémentaires dont le plus petit pas est l'instruction machine (NB : ce peut être une machine abstraite de haut niveau comme la JVM, ou de très haut niveau comme une machine SQL).

La décomposition de la transformation en blocs indivisibles est ce qui va permettre le multiplexage de la transformation pour le compte de différents usagers (phénomène d'entrelacement des flots d'exécution).

Pour que la transformation constitue une transaction, les quatre propriétés Atomicité, Cohérence, Isolation, Durabilité (**ACID**) doivent être respectées.

Du point de vue de l'usager Ui, tout se passe comme si la transformation opérée sur {d1, d2, …, d9} était faite instantanément, en interdisant de laisser la mémoire dans un état intermédiaire où seule une partie de { d1, d2, …, d9} aurait été transformée : c'est la propriété d'atomicité, le **A** de ACID.

Ceci veut dire que du point de vue de l'observateur Ui qui a lancé l'opération, les états intermédiaires qui ont servi à fabriquer l'état final sont inaccessibles et invisibles (ce sont des brouillons). L'opération n'a que deux statuts possibles : FAIT (tout a

été transformé), NON FAIT (rien n'a été transformé) ; mais dans les deux cas des ressources ont été consommées.

De plus, la transformation doit préserver la cohérence sémantique des données. Ceci est une propriété liée à la nature de la transformation et à la programmation qui en a été effectuée. Seul le programmeur peut donner une telle garantie. Dans le cas d'un Débit-Crédit , la somme *Débit* + *Crédit* doit être = 0. C'est le **C** de ACID.

Pour cela le programmeur devra avoir passé un contrat explicite avec l'expert métier, seul habilité à déclarer VRAIE (ou juste) la transformation effectuée. L'expert métier fixe la vérité, c'est lui qui spécifie l'invariant sémantique ; le programmeur écrit le programme conformément à la spécification de l'invariant avec les mécanismes ad hoc de la plate-forme d'exécution.

Pour éviter d'avoir à se coordonner dynamiquement avec les autres programmeurs, ce qui constituerait un accroissement de complexité considérable de la programmation, le sous-système transactionnel donne au programmeur la garantie complémentaire qu'il peut opérer sur la mémoire comme s'il était seul, selon la règle de priorité naturelle premier arrivé, premier servi. Pour cela, le programmeur doit fournir au système une information concernant les ressources qui lui sont nécessaires, de façon à ce que le Moniteur TP, (en abrégé TPM *Transaction Programme Monitor*) puisse temporairement réserver ces ressources au bénéfice de la TP demanderesse (verrouillage temporaire). En fin d'opération les ressources sont déverrouillées et remises à la disposition de la communauté des usagers. En cas de panne, il faut pouvoir relaxer les ressources au bout d'un certain temps (*time-out*) pour éviter le blocage progressif du système. Cet ensemble d'opérations constitue la propriété d'isolation, le **I** de ACID.

Du point de vue du modèle d'estimation COCOMO, la propriété d'isolation a un double impact sur l'équation d'effort du modèle : $Eff=k(KLS)^{1+\alpha}$.

Le fait de se croire seul permet de choisir un k faible (on programme plus vite) et surtout un coefficient d'intégration α voisin de 0. En effet, toute la complexité des interactions entre les transformations est gérée par le moniteur de transactions TPM. C'est ce qui a permis aux programmeurs COBOL de s'adapter sans grande difficulté à la programmation OLTP/OLAP.

Le schéma 9.4 donne une vue générale de la situation :

Du point de vue des usagers Ui, tout se passe comme si les transformations avaient été sérialisées et exécutées l'une après l'autre, dans l'ordre d'arrivée.

Le rôle du moniteur TPM est de maximiser le parallélisme latent de l'application, tant du point de vue des blocs de transformations élémentaires {TPx_1, TPx_2, …, TPx_n} de chaque TPR, des zones mémoire Z_Ui attribuées aux usagers, que des données élémentaires constitutives des états mémoire M2 en cohérence avec M1 dans les serveurs de données.On remarquera qu'il y a autant de zone de travail que de postes de travail connectés à l'instant t, soit pour 10 000 postes avec 100K de mémoire chacun (NB : c'est faible), un espace mémoire de 1 Giga est nécessaire ; si sur ces 10 000 postes, 1 000 sont effectivement actifs, la zone active en mémoire

centrale sera de 100 Méga. Avec des applications Web, la zone de travail par poste peut être beaucoup plus importante, ce qui permet d'imaginer la complexité de la gestion mémoire d'une telle application.

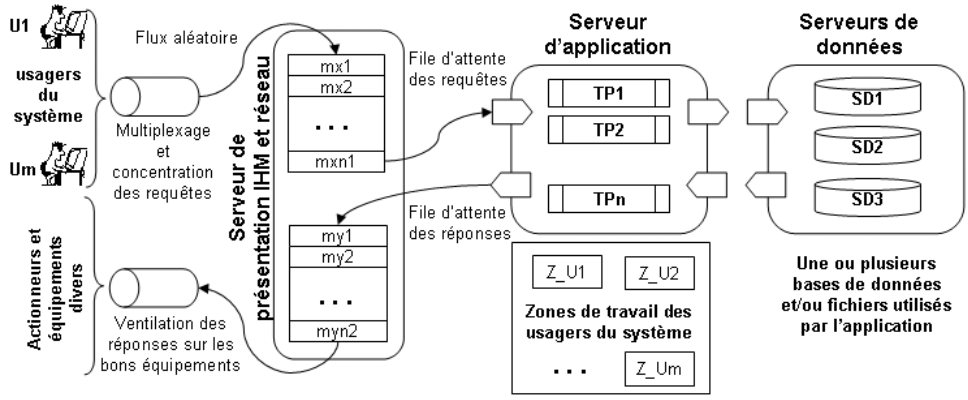


Figure 9.4 - Architecture générique d'une application transactionnelle

En cas de défaillance de la TPR, pour une raison quelconque (y compris une erreur de programmation) le moniteur TPM doit pouvoir récupérer les ressources empruntées temporairement et les rendre aux gestionnaires de ressources (SGBD, mémoire/virtuelle, sessions réseaux…).

Le mécanisme de sérialisation peut être schématisé comme suit (figure 9.5).

Le gain est d'autant plus important que le niveau de multiplexage est élevé, jusqu'à un certain point, car comme on l'a vu au chapitre 7, la commutation de processus est un mécanisme coûteux quand il y a beaucoup de processus en concurrence.

Avec le mécanisme de sérialisation, le moniteur TPM donne une vision macroscopique « simple » (mais non simpliste) de ce qui se passe dans le système, ce qui permet aux programmeurs et à l'architecte de l'application de comprendre ce qu'ils font, du point de vue de la sémantique métier de M1 (le C de ACID), en leur évitant de plonger dans le détail microscopique du middleware transactionnel et/ou du système d'exploitation.

Tout se passe comme si la répartition des travaux entre le programmeur d'application et le middleware TPM était la suivante :

- Les transactions s'exécutent comme elles peuvent, en fonctions des ressources disponibles au niveau du moniteur TPM (et de la programmation effectuée par le programmeur de transactions) seul maître de la gestion de la concurrence qui est de fait totalement retirée des problèmes à la charge du programmeur de TPR, pour autant que le mécanisme TPM garantisse la sérialisation des changements d'état de la mémoire.

- Le TPM prend en charge la gestion de tout l'environnement de l'application, i.e. la sphère de contrôle, qui garantit aux usagers que tout reste cohérent quoiqu'il advienne (pannes, destruction, arrêt, etc.).

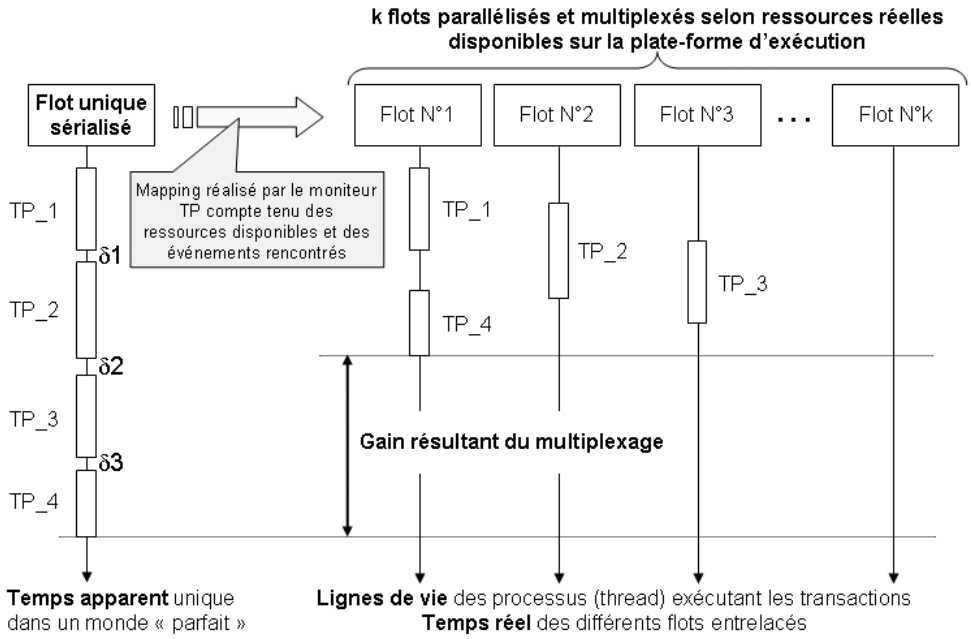


Figure 9.5 - Sérialisation des transactions

La sérialisation garantit un niveau de déterminisme du point de vue du métier M1 (à la durée des opérations prêt), ce qui permet à l'ingénieur de travailler effectivement.

Du point de vue des mécanismes du moniteur TPM, les choses sont très différentes, car c'est lui qui gère le choc du non déterminisme inhérent à la gestion des ressources (zones de travail Z_Ui des TPR) et à la concurrence, et la cohérence de l'entrelacement des différents flots d'exécution (lignes de vie des transactions), en fonction des directives données par l'architecte-concepteur de programmes.

C'est le mécanisme TPM, et lui seul, qui encaisse le choc de la complexité de l'exécution, au niveau du détail des opérations.

Du point de vue projet, cette séparation est fondamentale car c'est ce qui va permettre aux programmeurs-architectes d'applications de fonctionner dans un modèle CQFD de type S selon COCOMO, alors que le programmeur-architecte système qui réalise le TPM fonctionnera dans un modèle CQFD de type P et/ou E (algorithme complexe de gestion mémoire, gestion de tous les évènements des équipements fournissant les ressources et ceux du système d'exploitation).

Si la séparation est « poreuse », le niveau M1 est contaminé par un modèle CQFD de type P et/ou E, d'où une catastrophe économique certaine pour l'application et le projet de réalisation.

Le niveau TPM ne voit de la mémoire métier M1 que la partie qui intéresse TP1. On peut considérer que la mémoire M1, {d1, d2, …, d9} dans notre exemple, est simultanément dupliquée dans la zone de travail ZU_1 de TP1 et verrouillée pour toutes les autres transactions, jusqu'à la terminaison normale ou anormale de TP1. Cette double réalité, réalité M1 et réalité M2, peut être schématisée comme suit (figure 9.6) :

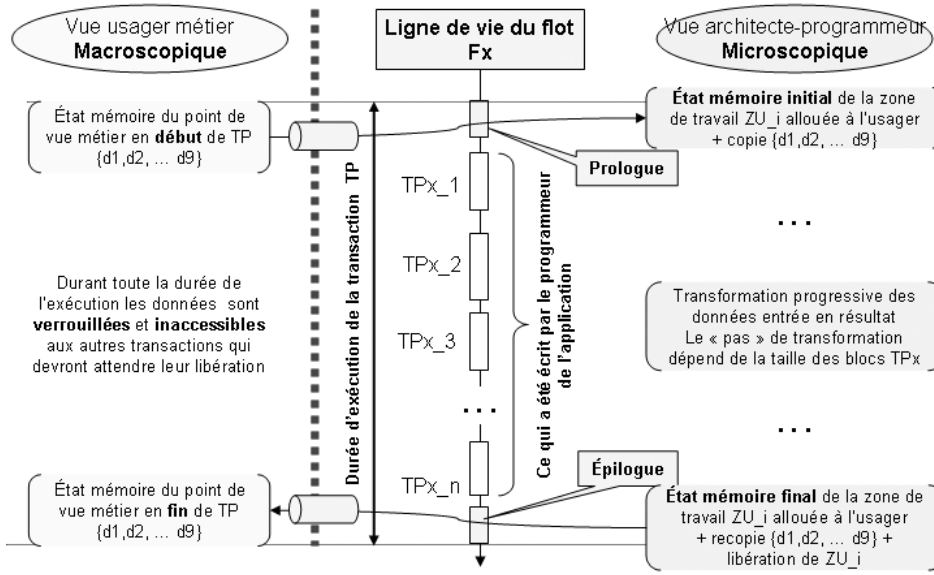


Figure 9.6 - Dualité des points de vue M1/M2.

Tout l'art de l'architecte consiste à faire en sorte que la frontière M1/M2, (i.e. la notion de sphère de contrôle) reste entièrement sous son contrôle, et qu'il n'y ait aucune fuite d'état du monde M2 vers le monde M1.

Plus, et mieux, grâce à la programmation effectuée, les flots d'exécution sont finement entrelacés, meilleur est le débit en transactions par seconde, mais plus complexe est l'état microscopique, donc plus coûteux à mettre au point ; ceci milite pour des middleware ayant une maturité suffisante.

La capacité de multiplexage dépend de la stratégie de verrouillage et du découpage de la transformation en blocs élémentaires indivisibles qui dépend du style de programmation adopté par les programmeurs, conformément aux règles édictées par l'architecte. La complexité résultante ne peut être maîtrisée que par une construction hiérarchique de l'application, depuis le niveau des instructions élémentaires jusqu'à l'intégrat final ; on en reparlera au chapitre 12.

NB : Dans une application batch, le grain de verrouillage est le fichier (ou toute ou partie d'une base de données) ; l'application entière est vue comme un bloc indivisible.

La propriété de durabilité répond à une exigence de l'usager qui est de ne jamais lui demander de refaire ce qu'il a déjà fait, pour des raisons spécifiques à l'informatique. Ce qui a été saisi est durablement saisi ; ce qui a été modifié dans les BD est durablement modifié (ordre `commit` du transactionnel). Pour cela, le système doit gérer une chronologie des interactions IHM sur une mémoire stable (journal des interactions) et une chronologie des modifications effectuées sur les données, dans une ou plusieurs bases (journal des modifications et des mises à jour). Il s'agit de se prémunir contre d'éventuelles pannes informatiques, qui bien que rares, se produiront inéluctablement.

Indépendamment de ce qui se passe sur le poste de travail de l'usager, une fois qu'une requête a été enregistrée dans la file d'attente des requêtes (cf. figure 9.4), le système doit considérer cette opération comme durable. En cas d'arrêt du serveur de présentation-réseau la file d'attente doit être sauvegardée de façon ad hoc. Idem pour l'interface entre le serveur d'application et le serveur de données. Les sauvegardes doivent être gérées sur des équipements différents de ceux qui contiennent les données dont la durabilité doit être garantie, faute de quoi une panne de l'équipement entraîne également la perte des sauvegardes.

Les mécanismes correspondants sont très délicats à gérer car ils nécessitent une synchronisation fine entre ce qui a été notifié aux usagers et utilisateurs du système dans le monde M1, et l'état sauvegardé. Ceci est particulièrement vrai avec la file d'attente des réponses qui va déclencher des actions potentiellement irréversibles vers les actionneurs et équipements divers (cf. figure 9.4) qui font partie de M1.

Dans un monde parfait, où les machines fonctionnent sans erreur à vitesse infinie, la plage d'incertitude serait nulle ! Dans la réalité elle ne l'est pas.

Moralité : si l'on veut appliquer la bonne sauvegarde, et re-émettre les bons messages, il faut aller consulter les MIB des équipements pour savoir exactement ce qui a été fait ou non fait. Faute de quoi, il y aura possibilité de créditer deux fois le même compte (ce qui n'est pas très gênant pour l'heureux bénéficiaire), mais également la possibilité de débiter deux fois le même compte, ce qui est fort désagréable. L'architecture doit donner les moyens de réduire la plage d'incertitude à zéro (propriété dite « *Fail – Fast* ») et, dans tous les cas, d'identifier tous les équipements à contrôler (cf. notion extensive de sphère de contrôle). La propriété de durabilité contribue à la propriété de cohérence, et de ce fait indissociable du C de ACID.

On remarquera que le fait qu'un message ait été correctement traité par l'équipement destinataire est, vis-à-vis de la mémoire de sauvegarde l'équivalent d'une transaction. Si le message d'acquittement de l'opération sur EQi n'a pas été réceptionné, il y a doute ! ce qui ne veut pas dire que le travail n'a pas été fait (c'est le problème des incertitudes liées au réseau). En cas de ré-émission d'un message non acquitté, il

y a lieu de se poser la question de l'idempotence (en mathématique, une opération comme A+A=A est idempotente). Dans un ordinateur, on peut demander deux fois la lecture d'un même enregistrement, il y a idempotence ; par contre un ordre d'écriture n'est pas idempotent. Pour traiter correctement l'idempotence, il faut que toutes les requêtes aient un identifiant unique pour distinguer, au niveau de l'équipement, un ordre qui a déjà été pris en compte (journal de bord de l'équipement).

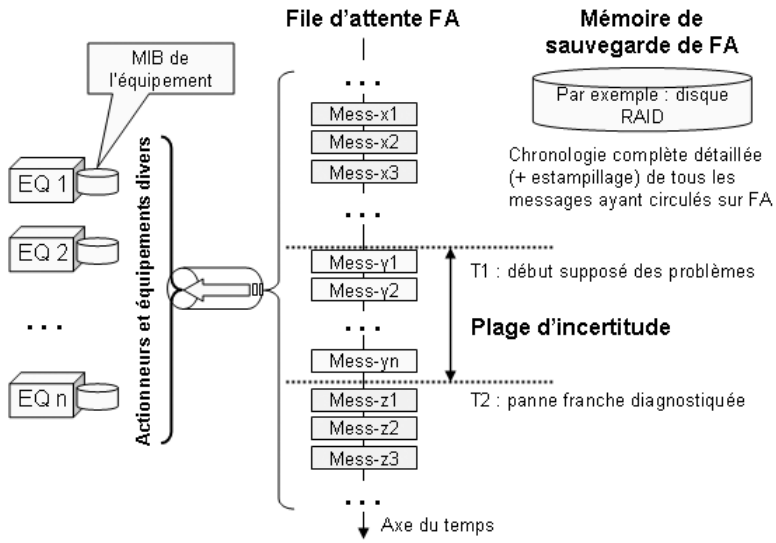


Figure 9.7 - Sauvegarde et propriété de durabilité.

La notion de transaction vaut donc également pour les équipements dont l'état chronologique est corrélé avec les informations de sauvegardes. Ce type de transaction est essentiel si l'on veut automatiser l'administration et l'exploitation des plates-formes (cf. la notion d'« *autonomic computing* », introduite par IBM).

9.1.2 Transactions longues – Compensation

Le schéma 9.8 montre un exemple de transaction dite longue car elle est formée de plusieurs transactions élémentaires. Il s'agit de réserver un certain nombre de ressources de façon coordonnée pour que le client de l'agence de voyage puisse avoir l'ensemble des réservations nécessaires à un séjour serein. La sphère de contrôle d'une transaction longue est l'ensemble des bases de données qu'il faut manipuler pour obtenir les confirmations de réservations. D'un point de vue statique, c'est la somme des sphères de contrôles des transactions élémentaires. D'un point de vue dynamique, le problème se pose de façon très différente, car en fonction de la progression de la réservation (c'est un processus), le client peut faire varier ses choix

et changer d'avis. Il est donc impossible de verrouiller l'ensemble des ressources pour opérer comme si c'était une transaction élémentaire. Le verrouillage ne sera que progressif.

Qui plus est, il peut se présenter des situations où les difficultés de réservation rencontrées à l'étape En oblige à revenir sur des décisions antérieures. Il faut alors défaire proprement, à l'aide de nouvelles transactions ce qui avait été considéré comme définitivement acquis. C'est le rôle des transactions de compensation.

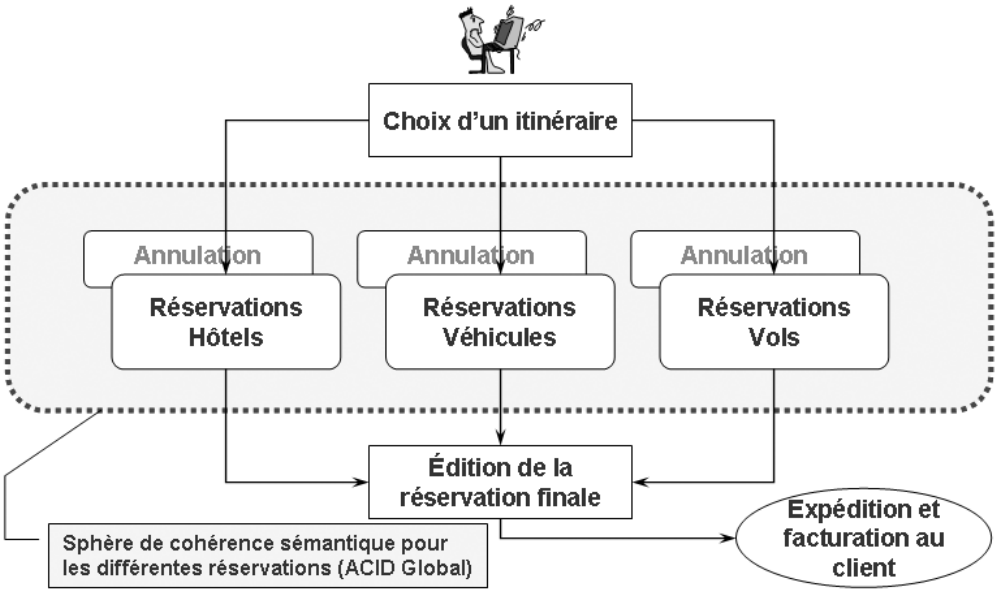


Figure 9.8 - Transaction longue

Le problème est beaucoup plus délicat qu'il n'y paraît à prime abord. Dans une logique de transaction élémentaire, toute réservation confirmée doit donner naissance à un paiement. Comme on ne peut confirmer qu'à la fin de la réservation globale, il faut introduire dans la logique transactionnelle de nouveaux états, comme :

- Option de réservation, pour une durée limitée, sans aucun engagement contractuel.
- Réserver/Non confirmé (avec éventuellement un time-out), ce qui veut dire qu'à épuisement du time-out, la ressource retournera à l'état libre.
- Réserver/Confirmé, mais l'adresse de facturation sera celle de l'agence de voyage et non pas celle du particulier qui a fait la réservation, s'il s'agit d'un package.

De nombreuses variantes sont possibles, mais cette simple séquence montre qu'à chaque ressource il faudra associer un diagramme d'état qui dépend du gestionnaire

de la ressource et de la sémantique de réservation, qui peut être différente d'une ressource à l'autre.

La propriété ACID globale est un changement synchronisé de tous ces diagrammes d'états qui devront garder le souvenir de ce changement synchronisé, ne serait-ce que pour avertir l'agence et le client en cas de difficulté : par exemple annulation d'un vol intermédiaire qui risque de désynchroniser l'ensemble.

La transaction de compensation que l'on aurait pu introduire d'entrée de jeu avec les transactions élémentaires devient primordiale. Une transaction élémentaire confirmée est une modification définitive de la réalité.

Ceci étant, le client a le droit d'annuler une réservation (généralement sous certaines conditions). L'annulation est un autre changement de la réalité. Le siège réservé redevient libre, le paiement effectué doit être remboursé : c'est exactement l'opération inverse.

Notons qu'avec des transactions élémentaires on pourrait procéder de façon beaucoup plus brutale en restaurant l'état de la base de données juste avant la réservation, en utilisant le journal des modifications.

Le travail de compensation est effectué par l'administrateur de la base, à la requête d'un tiers de confiance. Une telle mécanique n'est pas généralisable dans le cas des transactions longues, ni même pour certaines transactions courtes qui modifient plusieurs bases de données.

Sans entrer dans les détails on peut représenter cette situation de la façon suivante (figure 9.9) :

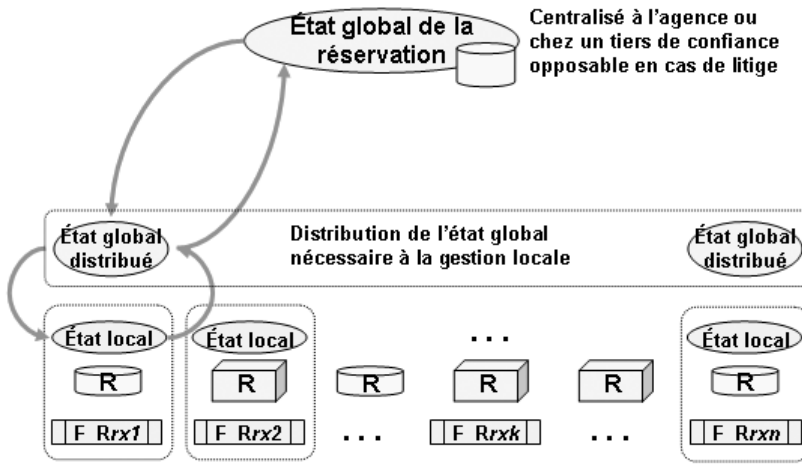


Figure 9.9 - Etat global d'une transaction longue

Le rôle du gestionnaire de transaction longue est de s'assurer de la cohérence de l'état global, en particulier pour le sous-ensemble qui a été distribué localement sur chacune des ressources ayant fait l'objet d'une réservation.

Les événements associés sont gérés par le moniteur de transaction longue. L'agence et/ou le tiers de confiance doit voir remonter tout événement susceptible de compromettre la réservation effectuée par le client.

Tout manquement à cette logique se traduira par des désagréments pour le client (par exemple l'« overbooking ») ou des désagréments pour le gestionnaire de ressource (fuite de ressource).

9.2 FONCTIONS DE SERVICES – FONCTIONS PRIMITIVES

9.2.1 Généralités

La notion de fonction de service fait partie de l'histoire des langages de programmation de haut niveau, dès l'arrivée de langages comme COBOL et FORTRAN. Appelée également fonction primitive, ou tout simplement primitive, dans la littérature sur les protocoles et les architectures en couches dans les réseaux, elle étend le langage de programmation hôte avec des « instructions » que l'on ne peut pas trouver à l'état natif dans l'interface hardware-software (cas du tri en COBOL et des fonctions mathématiques en FORTRAN).

Certaines de ces fonctions sont considérées comme partie intégrante du langage de programmation et fonctionnent exactement comme des instructions du langage ; dans ce cas le compilateur peut vérifier les modalités d'appels et le respect des règles syntaxiques et sémantiques. Les bibliothèques correspondantes font partie du système. On peut parler, dans ce cas, de machine « étendue ». Par exemple, le langage Ada 95 (malheureusement non utilisé, suite aux déboires d'Ada 83) dispose d'une machine étendue qui intègre un mécanisme de communication inter-processus particulièrement bien conçu que l'on peut utiliser comme modèle pédagogique. Certains langages comme Java et C# juxtaposent différents mécanismes, parfois incompatibles, dont le maniement sûr nécessite d'avoir parfaitement compris la logique très particulière de l'IPC.

Pour des usages plus spécifiques à certains problèmes, des bibliothèques additionnelles peuvent être fournies par des tierces parties ; dans ce cas, le compilateur ne peut fournir qu'un service minimal, conformément aux mécanismes d'extensibilité du langage s'ils existent (méta-compilation).

Le bon emploi de ce type de bibliothèques ne peut venir que de l'architecte (guide de programmation permettant de faire les inspections et revues de code) et du programmeur. Dans cette philosophie, un SGBD est un « gros » service qui fournit des moyens d'accès (i.e. l'interface CRUD) aux données qu'il gère. Pour les SGBD standardisés, les modalités d'appel font partie des langages de programmation, via ce

qu'on appelle des « binding » (cf. ce qui est proposé comme interface langage pour le SQL du modèle relationnel).

Du point de vue de l'environnement de conception-développement une telle fonction doit être gérée en configuration car selon les exigences elle peut être dupliquée /répartie dans l'intégrat qui l'utilise, ou rester dans un processus qui les centralisera.

Une fonction/service répartie (i.e. intégrée dans l'espace d'adressage de l'appelant) est invoquée par les mécanismes standards d'appel de procédure fournis par l'interface hardware-software.

Une fonction/service centralisée est invoquée directement ou indirectement par les mécanismes standards de la communication entre processus.

La distinction répartie/centralisée est d'ordre sémantique. Il n'y a aucun problème à répartir une fonction stabilisée comme le calcul d'un sinus ou d'un cosinus. Pour une fonction susceptible d'évolutions fréquentes, c'est un vrai problème pour l'architecte. Dans ce cas, une solution centralisée peut être préférable, car la mise à jour, en cas d'évolution du service, se fera en un point unique.

Dans les années 80 une innovation majeure a été proposée par les architectes de systèmes d'exploitation, consécutivement à l'arrivée des architectures distribuées, le RPC (*Remote Procedure Call*) dont SUN a fait un usage massif dans son *file system* distribué NFS (*Network File System*) que les mauvaises langues (souvent avec raison) qualifiaient de *No File System*.

Pour le programmeur, la facilité offerte par le mécanisme RPC est évidente, car elle lui permet d' «oublier» le réseau (provisoirement, quand tout se passe bien !).

Pour l'architecte, cette facilité a un coût :

- En terme de performance (à la fois CPU, mais surtout en entrées-sorties sur le réseau).
- En sûreté de fonctionnement, car si le réseau sature ou ne répond plus (par exemple : épuisement de « time-out »), cela va occasionner des remontées d'évènements que le programmeur devra gérer correctement. En local, le programmeur ne verra jamais de tels événements.

Pour utiliser de façon sûre les mécanismes RPC, l'architecture doit être conçue en conséquence.

NB : Notons que cette exigence ne viendra jamais des utilisateurs et de la MOA métier qui ignorent (et doivent ignorer) comment sont organisées physiquement les plates-formes d'exploitation. C'est un choix de l'architecte pour satisfaire des contraintes projet CQFD/FURPSE, qui nécessitent un accord explicite avec les exploitants (exigence primordiale pour le *capacity planning* et le *system management*).

L'un des résultats visibles du travail de l'architecte est la mise à disposition de bibliothèques de services, avec des interfaces standardisées. La constitution de ces bibliothèques est un préalable à l'architecture des composants applicatifs (i.e. les intégrats) qui les utilisent.

Du point de vue technologique, l'architecte doit être conscient, dans ses choix, de l'usage, par les progiciels métiers et/ou systèmes (i.e. les middleware, de ces mécanismes et du degré de standardisation proposé par le fournisseur. Les standards internationaux ou les standards de facto (i.e. ceux du « libre » ou de l'OPEN SOURCE) doivent être préférés à tout autre.

Pour les données, deux cas sont à considérer :

- Cas 1 : les données sont dans des fichiers et/ou des bases de données. Elles sont accédées via les fonctions d'accès CRUD du SGF ou du SGBD.
- Cas 2 : les données sont dans des mémoires locales aux composants applicatifs. Elles sont accédées via les mécanismes offerts par le langage de programmation et/ou les environnements de programmation.

D'un point de vue architecture, l'architecte doit être vigilant sur les formats des données en fichiers et/ou bases de données qui seraient dupliqués de façon « sauvage » dans les mémoires locales des composants applicatifs, car cela crée des dépendances cachées entre ces différentes données. Les dictionnaires de données ont été inventés, dans les années 80s, pour gérer ce problème. Aujourd'hui tout cela est résorbé dans les outils de gestion de configuration qui généralisent le concept de dictionnaire. Il faut néanmoins être très vigilant car ce type de dépendance peut résulter d'erreur d'inattention, ou de mauvaise documentation du référentiel.

En toute logique aucunes données communes, partagées entre différents acteurs, ne devraient être accédées, autrement que par un interface d'accès, soit à la compilation, soit à l'exécution, (c'est l'encapsulation des données). Ce doit être la pratique de conception-programmation par défaut, ce qui permet d'organiser les inspections et les revues en conséquence.

9.2.2 Synthèse simplifiée des mécanismes d'appels des services

Le détail de ces mécanismes qui doivent être maîtrisés par l'architecte et les concepteurs-programmeurs peut être trouvé dans les ouvrages sur les systèmes d'exploitation, ou dans la documentation des environnements de programmation.

Modalité de l'appel du service

L'appel du service peut être séquentiel, via CALL et EXIT, ou concurrent via SEND et RECEIVE).

Dans le cas séquentiel, le service appelé doit être disponible ; l'appelant s'arrête et attend le résultat, ce qui peut conduire à des situations de blocage si le service n'est pas disponible (cas du service distant).

Modalité de répartition (aspect local ou distant)

Le service appelé peut être local, dans l'espace d'adresse du processus appelant, ou distant, dans l'espace d'adresse d'un autre processus. Dans le cas d'un appel local les données peuvent être partagées ; dans le cas d'un appel distant, le contexte de l'appel doit être transmis à l'appelant. Les données contextuelles peuvent être communiquées via les modalités propres aux données.

Il faut distinguer les quatre cas {séquentiel, concurrent}×{local, distant} car les sémantiques et les caractéristiques non fonctionnelles sont totalement différentes dans chacun des cas.

Tableau 9.1 - Mécanismes d'appel

	Appel local	**Appel distant**
Appel séquentiel	C'est l'appel de procédure classique. Les modalités de passage des paramètres (référence, valeur, etc.) doivent être précisées.	Nécessite un mécanisme RPC dans le socle. C'est un protocole. Cf. SOAP, ou des protocoles type transactionnel (middleware).
Appel concurrent (en parallèle)	Est effectué via les mécanismes IPC disponibles (mémoire partagée) dans le socle selon l'environnement choisi (cf. par exemple : DOT.NET ou J2EE)	Est effectué via les mécanismes d'interopérabilité du socle avec message + boîte aux lettres (de type messagerie, MOM, CORBA, EAI, ESB, etc.)

Dans chacun des quatre cas, il faut préciser les modalités du retour : {FAIT, NON FAIT} ; dans le cas de NON FAIT, il faut préciser les causes de l'échec du service en distinguant la ressource insuffisante (on peut attendre et réessayer plus tard), de la ressource indisponible (inutile de réessayer, c'est la panne !).

Appel séquentiel

Le schéma de principe, avec un diagramme de séquence, est le suivant (figure 9.10) :

Le mode nominal de ce type d'appel est le retour à l'appelant. Si le service I2/SV est distant, la sémantique du retour est à préciser avec précision.

En cas d'anomalie de l'exécution de I2, le point de retour peut être différent de I1.

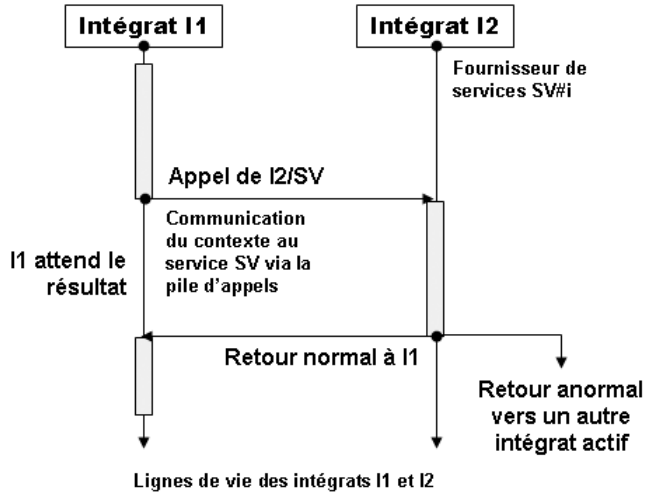


Figure 9.10 - Retours sur appel séquentiel

Appel concurrent

Les messages qui s'échangent sont de type requêtes via des ordres type SEND / RECEIVE. Le retour peut être sur l'appelant ou sur un autre intégrat (par exemple, la machine de surveillance).

Cas 1 : Notifications entre deux services. Le schéma de principe est le suivant (figure 9.11) :

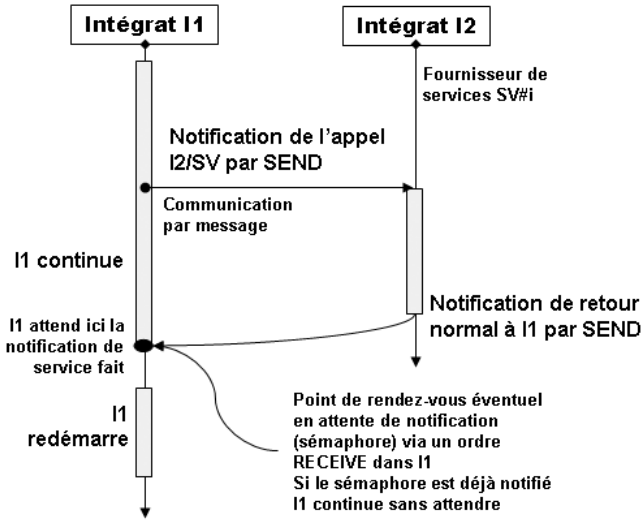


Figure 9.11 - Retour sur appel concurrent

Cas 2 : Notifications entre trois intégrats.

L'intégrat I1 envoie sa requête pour invoquer le service SV à I2 qui l'exécute ; I3 récupère le résultat . Le schéma de principe est le suivant (figure 9.12) :

Dans les deux cas, les anomalies rencontrées durant l'exécution de I2 doivent être prises en compte pour spécifier les modalités de retour (service FAIT, NON FAIT, etc.).

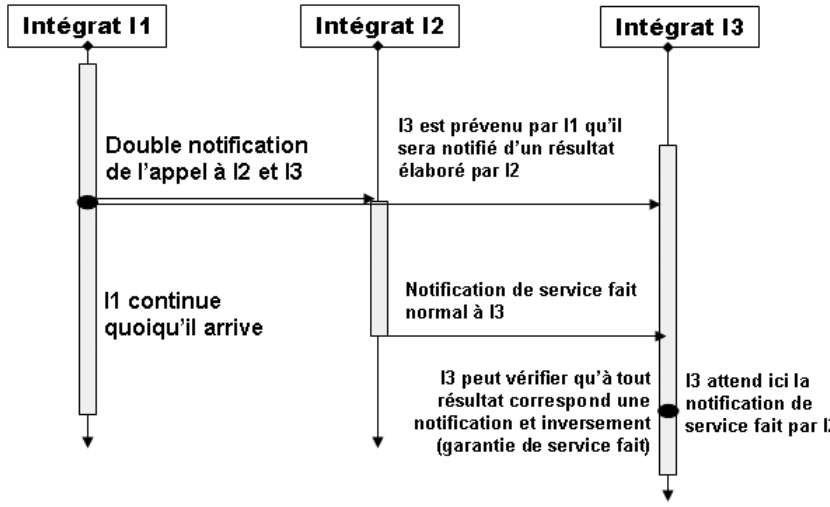


Figure 9.12 - Notification de service sur deux niveaux

9.3 SÉMANTIQUE DE COUPLAGES ET DES INTERACTIONS ENTRE LES INTÉGRATS

On dit que deux entités, ou plus, sont couplées si l'exécution de l'une influe sur l'exécution de l'autre, et vice versa. La plupart des couplages sont explicites et résultent des décisions et choix de l'architecte ; d'autres couplages sont plus insidieux, car implicites et non documentés, ce qui fait qu'une modification d'une entité provoquera des défaillances chez les autres, alors que la gestion de configuration n'aura rien enregistré. Le crash du premier tir d'ARIANE 5 a mis en évidence un couplage implicite, non documenté, entre la centrale inertielle et le comportement du lanceur ; le biais horizontal sur ARIANE 5 était différent du biais ARIANE 4, ce qui fait qu'un comportement nominal sur ARIANE 5 a été faussement interprété comme une panne (ce qui aurait été le cas sur ARIANE 4 où la centrale avait fonctionné sans la moindre anomalie sur plus de 40 tirs). L'auteur se souvient de l'angoisse des architectes lorsque le système d'exploitation GCOS7 était adapté à une machine plus puissante, ce qui avait comme premier effet de démasquer des

défaillances restées latentes, particulièrement difficiles à diagnostiquer. Ce sont des exemples de couplage.

Les couplages résultent toujours de décisions de programmation faites par les programmeurs qui doivent parfaitement connaître le référentiel système qui définit le contexte dans lequel ils programment. Si le référentiel est mal documenté, il est inévitable que des couplages implicites apparaissent. D'où l'importance de la qualité (précision et rigueur) de ce référentiel et de la connaissance que doivent en avoir les équipes d'ingénierie. C'est ce que E. Dijkstra[3] avait appelé, il y a bien longtemps, la discipline de programmation.

9.3.1 Nature des couplages

Il est commode de distinguer quatre niveaux de couplage, d'intensité ou de visibilité décroissante, qui peuvent exister entre les intégrats : le couplage par les traitements, le couplage par les données, le couplage par l'ordonnancement et les événements, et enfin le couplage par l'environnement résultant des facteurs PESTEL.

Chacun de ces couplages peut prendre différents aspects, ce qui permet de graduer les niveaux de dépendance fonctionnelle des services.

Le meilleur moyen d'identifier les couplages est de construire les matrices N2 des services résultant des différentes traçabilités[4] telles qu'elles apparaissent dans le référentiel.

Le couplage par l'ordonnancement est la relation minimum qui puisse exister entre les services. Les seules informations à considérer sont les évènements qui déterminent l'ordonnancement possible des services compte tenu de la façon dont sont récupérées les exceptions par les différents moniteurs qui pilotent l'exécution des intégrats.

Couplage par les traitements

Ce type de couplage concerne l'enchaînement des opérations. Il est généralement explicite via le mécanisme d'appel, facilement repérable dans le texte des programmes. Cependant, la levée d'exceptions, programmées ou inopinées, peut occasionner des couplages d'enchaînement implicites si les drivers réceptionnant ces exceptions sont mal documentés et/ou programmés (NB : le driver reçoit un message d'exception dont il ne sait que faire).

La matrice N2 des services permet l'identification immédiate de ce type de couplage.

Règle d'ingénierie : Ce type de couplage implique un mécanisme unifié de gestion des codes retour et d'émission de messages d'erreurs (via un dictionnaire de messages centralisé dans le référentiel).

3. Cf. *A discipline of programming*, Prentice Hall.
4. Cf. JP. Meinadier, *Ingénierie et intégration des systèmes*, Hermès.

Traçabilité des codes retour et des messages d'erreurs émanant des traitements

On réalise une matrice {liste des services}×{liste des codes retour/messages d'erreur} en indiquant dans chaque case l'état du traitement {gravité de l'erreur, modalités de reprise, dialogue opérateur ou avec un automate d'administration, etc.}.

9.3.2 Couplage par les données

Deux services S1 et S2, ou deux composants, ou deux systèmes, sont dans une relation de couplage par les données si, et uniquement si, la seule chose qu'ils ont en commun est des formats de données. Les flots de contrôle dans lesquels ces services sont utilisés non aucun élément commun (i.e. les graphes d'appels des applications qui les utilisent sont non connexes) ni directement, ni indirectement.

Description des données

Il y a des données structurées (par exemple à la XML) et des données typées avec les mécanismes de typage des langages de programmation (ADT en objet, classes en UML, mappings IDL, etc.).

Utilisation des données

Les formats de données peuvent être utilisés de deux façons :

1. Utilisation pour référencer des données : Les services utilisent des descriptions qui doivent être identiques pour garantir leur intégrité. Ce cas correspond à des informations physiquement dupliquées dans les systèmes, via les mécanismes d'édition.
2. Utilisation pour accéder aux données : Ce cas correspond à des informations partagées. Les accès doivent être encapsulés, soit implicitement par les méthodes d'accès du SGF et/ou des SGBD utilisés, soit explicitement par des objets et les méthodes qui leurs sont associés (ce que nous avons appelé CRUD). Les règles de partages sont celles du SGF ou du SGBD ; en cas d'accès en mode transactionnel (respect des propriétés ACID) il faut un moniteur de transactions qui peut être soit un service informatique, soit un arbitre humain, soit un mélange des deux (cas du travail collaboratif).

Concernant les données accédées, il faut distinguer le cas ou le service à la vision directe des données (via le schéma de la base de données), du cas ou il a une vision indirecte et partielle des données (mécanismes de sous-schéma et de vues dérivées dans les SGBD, ou via une encapsulation ad hoc avec des méta-données).

Au total, il y a trois modes d'utilisation de la données, auxquels nous pouvons associer deux types de représentation : a) une représentation textuelle structurée à la XML (c'est une syntaxe concrète de la donnée) et b) une représentation typée au sens de la programmation telle qu'elle sera rencontrée dans les programmes. Soit six

cas de couplage, a minima. Les données textuelles structurées à la XML sont les moins contraignantes, au problème de performance près.

Dans un serveur de données (voir le modèle CRUD, ci-dessous, figure 10.15), les données sont accédées de façon indirecte via les méthodes d'accès CRUD associées au service. La restitution de la donnée peut être soit en XML (par défaut), soit typée dans le type de l'appelant, si le service dispose du convertisseur ad hoc ; dans ce cas l'impact en terme de dépendance est plus important, le couplage est plus fort (le coût de l'évolution est plus important).

NB : on pourrait imaginer une description de type texte libre (comme pour les moteurs de recherche) qui n'aurait d'intérêt que pour les parties purement textuelles des échanges.

Problème de traçabilité

On réalise une matrice {liste des services}×{liste des données} en indiquant dans chaque case l'état de la données du point de vue de ses modes d'accès CRUD {input, output, input-output}.

9.3.3 Couplage par l'ordonnancement et les événements

Ce couplage, faible dans les serveurs centralisés mais potentiellement fort en distribué, concerne l'ordre ou la chronologie dans lequel les services doivent ou peuvent être exécutés. L'ordonnancement est réalisé via les évènements générés par les services et/ou par l'horloge des systèmes.

Exemple : le service S1 est réalisé (événement FIN NORMAL de S1) ; S2 peut démarrer. S1 est « planté » (événement FIN ANORMAL) ; le service de nettoyage doit être activé. S1 et S2 sont tous deux terminés (quel que soit l'ordre entre S1 et S2) ; S3 peut démarrer. Etc.

L'identification de ce type de couplage nécessite d'avoir une nomenclature précise des évènements générés par les services et les composants applicatifs, ainsi que les drivers d'évènements correspondant. Ce type de description est possible dans des langages comme SDL (le standard UIT) ou BPML (*Business Process Modeling Language*) qui est un projet de norme OMG/W3C pour le workflow.

Types de dépendance temporelle

Le schéma 9.13 donne les différents cas d'ordonnancement possibles entre deux tâches.

Légende : F = FIN de la tâche ; D = DEBUT de la tâche ; δ = intervalle temporel.

Les relations de dépendance par l'ordonnancement sont décrites soit à l'aide de diagrammes d'activités, soit à l'aide de diagrammes d'états, comme pour les workflow.

Problème de traçabilité

On réalise une matrice {liste des services}×{liste des évènements} en indiquant dans chaque case l'état de l'évènement {producteur, consommateur}. Un service consommateur d'événements doit disposer de files d'attentes ad hoc et des drivers associés qui sont des types de données (ADT) bien particulier qu'il faut identifier comme tel.

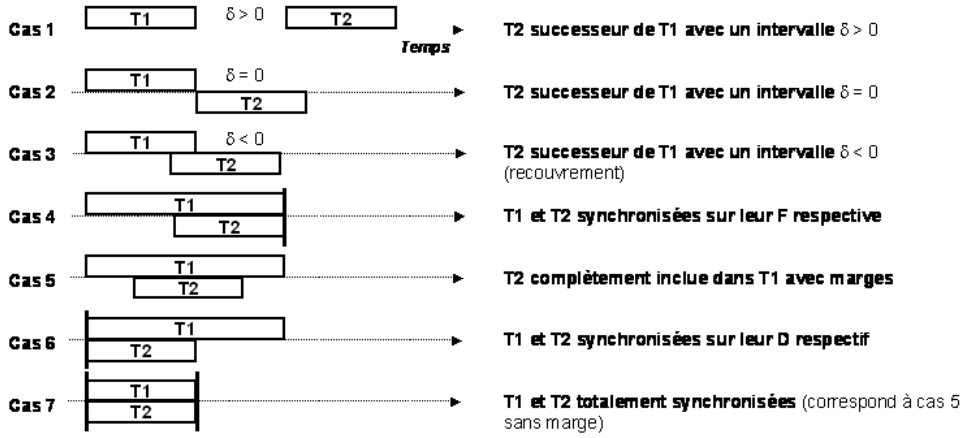


Figure 9.13 - les différents cas d'ordonnancement possibles entre deux tâches

9.3.4 Couplage par l'environnement

Ce sont les plus délicats car il n'y a aucune relation explicite entre les entités, sauf celle d'être hébergées sur une même plate-forme. Une entité qui surconsomme de la mémoire parce qu'elle est mal programmée peut induire un comportement anormal dans une autre entité utilisant la même mémoire.

La consommation désordonnée de mémoire via les primitives d'allocation dynamique est gérée dans une zone commune à tous (appelé « tas » ou « *heap* », en jargon franglais) provoque ce type de couplage. D'une façon générale, tout multiplexage, peut potentiellement créer ce type de couplage.

Cela se traduit par des phénomènes bien connus des intégrateurs et des exploitants où le système, dans un environnement d'intégration « propre », fonctionne correctement et ne fonctionne plus dans l'environnement d'exploitation réel. C'est la raison pour laquelle il faut avoir la nomenclature des ressources utilisées par le composant applicatif ; cf. le chapitre 8). Le seul recours dans ce type de situation est la mise en œuvre de techniques défensives qui constitue le système immunitaire du composant applicatif qui voit son coût augmenter.

On voit sur ce cas particulier un nouveau piège des architectures distribuées, car le contrôle de charge (le « load leveling » en franglais) que de nombreux systèmes d'exploitation ont proposé pour surveiller les ressources centralisées, ne fonctionne

plus de façon aussi simple en distribué, d'où les concepts nouveaux de *system management* et d'« autonomic computing» chez IBM.

10

Quelques modèles d'architectures

Dans ce chapitre nous allons décrire trois grands types de modèles d'architecture, et leurs principales propriétés, qui permettent de donner un cadre général à quasiment tous les systèmes informatisés connus.

Ce que nous avons appelé machine informationnelle est une architecture logique, du type architecture de von Neumann, appliquée à des entités de plus haut niveau comme : a) les flux de messages qui entrent et sortent dans la machine, b) les transformations, que l'on peut concevoir comme des instructions généralisées (i.e. des transactions), et c) la mémoire qui, dans ce cas, est constituée par un ensemble de bases de données persistantes et/ou volatiles, de structures diverses mais explicites. La cohérence d'ensemble est assurée par une machine spéciale, dite de surveillance, qui vient se brancher à des endroits ad hoc, et qui s'assure du bon fonctionnement de l'ensemble. C'est une machine dans la machine, de même que le processeur de service de l'ordinateur est un ordinateur dans l'ordinateur que voit l'utilisateur tout en lui étant invisible (NB : en thermodynamique, c'est un démon de Maxwell qui contribue à créer de l'ordre, i.e. faire baisser l'entropie naturelle). La nature récursive de l'architecture n'est que le reflet de la nature interprétative des différents niveaux d'abstraction de l'information les uns dans les autres. L'architecte peut ainsi travailler sur ces différents niveaux, sans les enchevêtrer. C'est un dispositif fondamental pour créer la simplicité nécessaire à l'action de l'ingénieur-architecte dans des projets « sains ».

10.1 NOTION DE MACHINES INFORMATIONNELLES – INTÉGRATION DE L'INFORMATION

10.1.1 Le puits, ou Machine Réceptrice (MR)

La première machine que nous considérons est la machine « puits » d'information. Le rôle d'un puits est de capter l'information jugée utile par le concepteur de la machine, de la valider avec l'aide d'un opérateur humain (ou d'un automate), de la structurer, de l'archiver dans un format ad hoc pour la mettre à disposition de clients qui feront des demandes d'extraction.

Le schéma de principe d'une telle machine est le suivant (figure 10.1) :

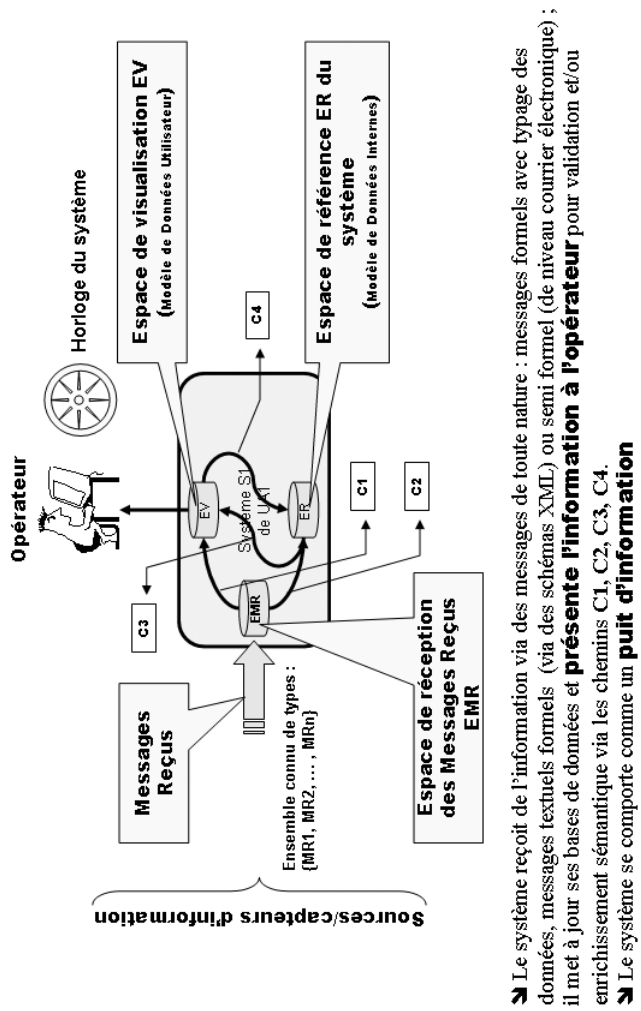


Figure 10.1 - Structure d'une machine puits / réception

L'information entrante dans la machine provient de sources de nature variée. Du point de vue informatique c'est un flux de messages dont les types sont connus. La conformité en entrée est l'appartenance à un ensemble de types, dont la syntaxe et la sémantique doivent être spécifiées. Les messages peuvent avoir entre eux des relations de toute nature. L'ensemble des messages connus peut être décrit avec le langage du modèle ERA.

Les messages se présentent à la machine dans un certain ordre qui correspond à une chronologie dont l'horloge est celle de la machine. À chaque message reçu on peut attribuer une estampille qui est un identifiant unique local à la machine. À l'entrée, les messages sont réceptionnés dans un espace EMR (espace de stockage des messages reçus) qui contient une zone d'échappement pour les messages reçus non reconnus. Pour des raisons de sûreté, il est primordial de conserver ces messages. Après réception, les messages peuvent être aiguillés dans deux directions, par les chemins C1 ou C2.

Par C1 le message va dans un espace de visualisation EV qui permet à l'opérateur de le valider ; une fois validé, le message est dirigé vers l'espace de référence ER de la machine, dans le modèle de données de cet espace. ER gère des données dont la qualité est garantie (traçabilité indispensable).

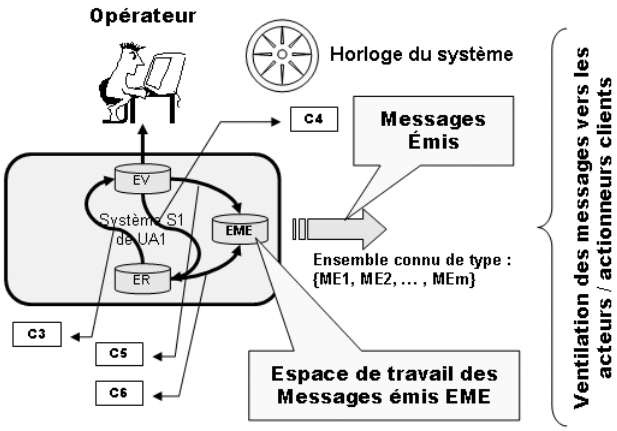
Par C2 le message est immédiatement dirigé vers l'espace de référence (ou un sas, dans cet espace). Dans cette architecture, la validation peut être a) différée (l'opérateur validera plus tard), ou b) omise, ce qui revient à donner une confiance absolue à la source. Les messages non validés vont à l'opérateur via le chemin C3 et reviennent dans ER via C4.

10.1.2 La source, ou Machine Emettrice (ME)

C'est la machine symétrique de la précédente. Les espaces EV et ER sont identiques. L'espace ER est complété par une zone donnant la liste des messages prêts à être émis, selon ce que l'opérateur souhaite faire. Le schéma permet une émission totalement automatique sans intervention de l'opérateur, (via le chemin C6) ou une émission contrôlée, avec ou sans temporisation, via les chemins C4 ou C5. L'événement déclenchant l'émission peut être un événement interne, ou un événement généré par l'opérateur.

La ventilation des messages émis se fait par publication, par exemple via un annuaire de type LDAP, ou tout autre moyen jugé bon par l'architecte. L'essentiel est de garantir la qualité des données émises, en particulier leur traçabilité.

La structure de la machine ME est la suivante (figure 10.2) :



➘ Le système émet de l'information via des messages de toute nature ; il propage l'information qu'il contrôle vers ses correspondants destinataires via les chemins **C3, C4, C5, C6** ; la qualité et la traçabilité des données propagées est primordiale pour l'intégrité de l'ensemble des systèmes intégrés. En cas d'action erronée il est primordial de pouvoir reconstituer l'histoire des transformations opérées.
➘ Le système se comporte comme une **source d'information**

Figure 10.2 - Structure d'une machine source/émission

10.1.3 La traduction, ou Machine transductrice (MT)

La machine transductrice est une composition des deux machines précédentes. La transduction est un terme plus fort que la simple traduction qui n'en est qu'un cas particulier. La machine transductrice permet un couplage de la machine avec la réalité (monde M1) via les capteurs et actionneurs qui font partie des deux mondes M1 et M2. La machine elle-même est purement M2.

Notons que la simple réunion de MR et ME ne produit pas une MT ; on a simplement accolé deux fonctions, pour mutualiser des ressources, comme par exemple dans les opérations publish/subscribe.

Avec une MT l'opérateur peut agir dans la réalité, il peut faire décoller des avions (contrôle aérien) ou négocier des ordres de bourses (salles de marchés), qui vont, à leur tour, rétroagir, via les capteurs, avec le système.

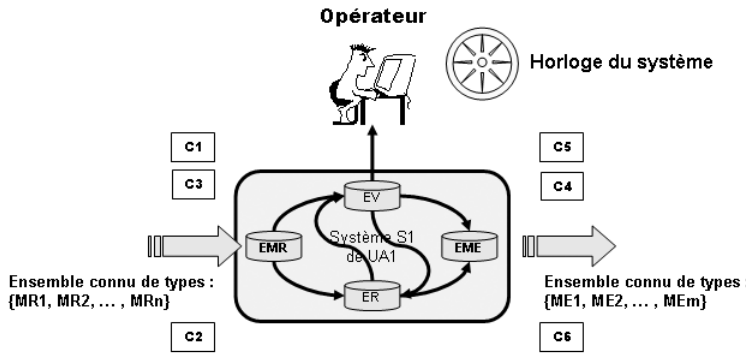
Les six chemins de données, C1 à C6, sont actifs simultanément.

Une MT est ce qui modélise le mieux le comportement d'une UA. On peut considérer deux grandes catégories de MT :

- Classe 1 : l'opérateur décide (cas du contrôle aérien), la machine surveille les opérations.
- Classe 2 : la machine décide, l'opérateur surveille.

En classe 1, le temps de réponse est celui de l'opérateur. Dans ce cas, il faut que les processus du monde M1 soient compatibles avec les réactions et la disponibilité de l'opérateur. La plupart des systèmes de gestion automatisés sont dans cette catégorie, sachant qu'un opérateur humain bien entraîné peut réagir au 1/10 de seconde, mais pas au delà, dans le meilleur des cas. En classe 2, le temps de réponse est celui de la machine et de sa programmation, ce qui permet de descendre le temps de réaction à la milliseconde, voire un peu moins (s'il y a du logiciel dans la boucle) ; au delà, c'est un système purement électronique qui peut être digitalisé (cf. les équipements de compression dans les modems).

Une MT de classe 1 peut être représentée comme suit (figure 10.3) :



➘ Le système reçoit et émet simultanément de l'information via des messages de toute nature ; il propage l'information qu'il contrôle vers ses correspondants destinataires ; le temps de latence réception→émission est court ; le risque de défaut d'intégrité est beaucoup important. La qualité globale {**données traitements contrôles**} est primordiale.
➘ Le système se comporte comme un **transducteur** ; les messages reçus et émis constituent des langages formels (automatisme pur) ou semi-formel (si intervention de l'opérateur) ; tous les chemins peuvent être activés.

Figure 10.3 - Machine traductrice de classe 1

Dans une telle machine, l'organe critique est l'IHM (cf. figure 1.4) ; son architecture doit être très soignée, car c'est elle qui organise la rétroaction dont l'opérateur est le pilote.

Une MT de classe 2 peut être représentée comme suit (figure 10.4) :

Dans une telle machine, l'organe critique est l'opérateur programmé. C'est lui qui supervise l'ensemble des opérations. Il présente à l'opérateur des vues de synthèses qui permettent à ce dernier d'agir conformément à son mandat (comme par exemple tout arrêter si danger, ou reprise d'un mode manuel si certains organes sont défaillants).

Un système de clase 2 est plus complexe qu'un système de classe 1 car il faut organiser l'interface, entre les tâches dévolues à l'opérateur programmé et celles qui restent sous la responsabilité de l'opérateur humain (cf. systèmes de type C3I/C4ISR, ci-dessous).

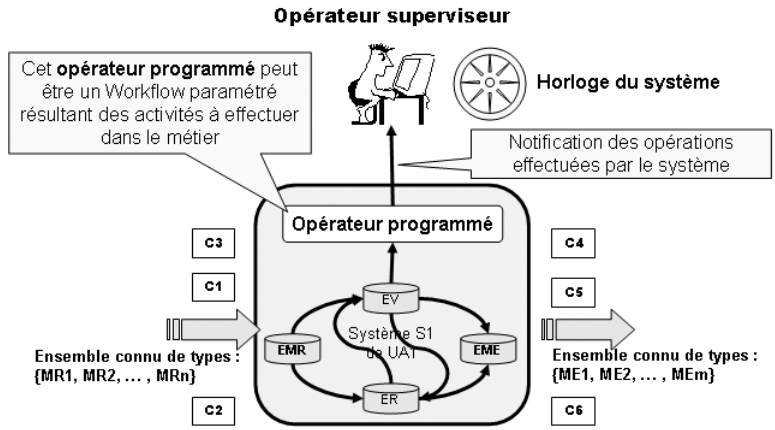


Figure 10.4 - Machine transductrice de classe 2

Cela revient à organiser très finement le FURPSE système global entre a) le FURPSE automatisé et b) le FURPSE de la composante humaine du système ; et faire en sorte que tout se complète, sans se contredire. Une MT de classe 2 sans opérateur humain permet un couplage étroit avec la réalité, par exemple sur le temps vrai des phénomènes physiques. Dans ce cas, la machine réalise un système temps réel.

10.1.4 Intégration de l'information – Couplage des machines

À l'aide des trois modèles MR, ME et MT nous pouvons désormais composer des ensembles plus vastes sur le modèle suivant (figure 10.5) :

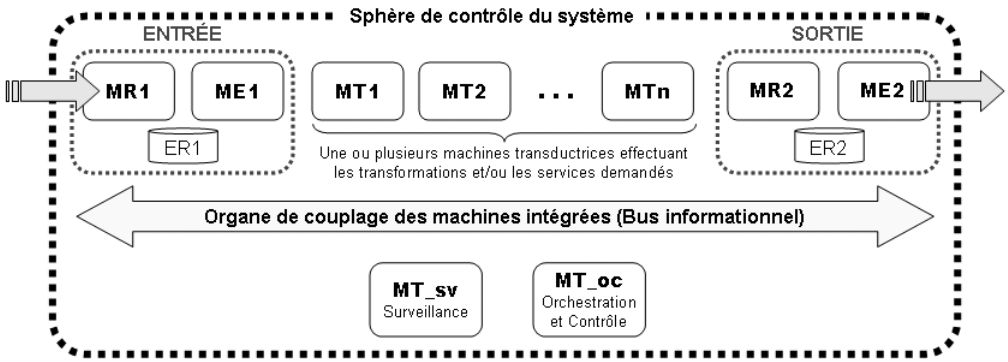


Figure 10.5 - Intégration et couplage des machines

Dans ce schéma nous retrouvons les organes fondamentaux de la machine abstraite succinctement décrite au chapitre 1. Les blocs entrée et sortie sont parfaitement symétriques. Les espaces de références sont partagés, de façon à bien séparer la partie réception de la partie émission. Aucune hypothèse n'est faite sur la structure de ER1 et ER2 qui peuvent fonctionner comme des files d'attentes, comme de simples fichiers ou des bases de données, ou comme des fichiers « spool » selon la nature des exigences.

Sur le schéma, nous avons explicité les deux moniteurs qui assurent 1) l'orchestration et le contrôle d'enchaînement des opérations et 2) la surveillance ; ce sont des machines de classe 2.

Par défaut, ces fonctions peuvent être effectuées par les moniteurs disponibles sur les plates-formes, en s'assurant que ceux-ci permettent d'assurer la conformité aux exigences formulées. Le bloc MT1, MT2, …, MTn matérialise le corps de la transformation effectuée par la machine intégrée. Le monitoring des opérations peut être assuré par un moniteur TP qui implémente les services transactionnels.

L'organe de couplage permet l'échange d'information entre les différentes machines, typiquement un bus logiciel qui peut être de la mémoire partagée par toutes les machines et/ou un réseau. Nous appellerons cet organe : bus informationnel. Les MT#i ont leurs espaces en propre, mais les espaces de références peuvent être partagés ainsi que les EME. Les espaces EV sont nécessairement privés (espaces de confinement) car la qualité des données qu'ils détiennent est en cours de validation.

Les espaces EMR sont privés par défaut, car pouvant faire l'objet d'une validation s'il y a doute sur la qualité des données entrantes.

L'organisation d'un système comme un ensemble de machines informationnelles intégrées est une technique puissante qui permet de spécialiser et/ou de confiner les différentes opérations à effectuer dans des sphères de contrôles indépendantes que l'on peut surveiller avec des techniques ad hoc. Tout ce qui entre et sort des machines passe par le bus informationnel qui constitue un lieu d'observation central des interactions entre les machines, via MT_sv et MT_oc.

10.2 ARCHITECTURE EN COUCHE

La notion d'architecture en couche est aussi ancienne que la programmation. Elle est un classique du génie Logiciel dès les années 60s (cf. E.Dijkstra, *THE multi programming system*, 1968 ; les conférences du NATO Science Committee, 1968 et 69 ; etc.). C'est la base de l'architecture des protocoles de communication où elle trouve son expression quasi parfaite (cf. IUT/CCITT, Recommandation X210, *Conventions relatives à la définition de service des couches de l'interconnexion des systèmes ouverts* ; Recommandation X200, *Modèle de référence pour l'interconnexion des systèmes ouverts pour les applications du* CCITT).

10.2.1 Les deux types de couches

On peut envisager la notion de couche de deux façons.

1er cas - Séparation des entités/objets métiers et des entités/objets techniques

Pour bien comprendre la distinction, reprenons la figure 1.4, *La frontière IHM entre le réel et l'informatique*, du chapitre 1. Le bloc « Automatisation des processus et tâches métiers » peut être envisagé selon deux points de vue :

- Si l'on regarde côté IHM/GUI on a des fonctions de services à vocation métier en interaction directe avec les opérateurs humains. La programmation correspondante est tributaire des commandes CRUDE nécessaires à leur travail.
- Si l'on regarde côté capteurs et actionneurs, on a des fonctions de services qui interfacent les équipements en périphérie du système. La programmation correspondante dépend de la nature de ces interfaces et de la variété des équipements utilisés.

Il est évident que pour des raisons de simplicité on a intérêt à bien séparer les deux types de programmation, de façon à pouvoir faire évoluer les objets métiers indépendamment des objets techniques et inversement.

Entre les deux, il faudra une couche d'intermédiation pour adapter les uns aux autres ; c'est ce qu'on appelle les API (*Application Programming Interface*) dont le rôle est fondamental (cf. Chapitre 15, Interfaces).

Dans la partie métier, on peut distinguer ce qui est à programmer de ce qui peut être acquis « sur étagère » (ce qu'on appelle les produits COTS, *Commercial Off-the-Shelf*) ou en libre service (« free » software).

Dans la partie technique, on peut réutiliser les fonctions de services correspondantes (c'est le patrimoine réutilisable de la DSI) via les interfaces d'accès à ces services.

Le schéma fait ressortir trois niveaux d'interfaces qui matérialisent trois couches fondamentales que l'on retrouve de façon plus ou moins explicite dans tous les systèmes.

Conformément au bon usage du concept de couche, la couche N-1 offre les services nécessaires à l'exécution de la couche N, à travers son interface. En application du principe de modularité, tout accès à N-1 passe par l'interface d'accès à la couche.

Ceci conduit à une vision hiérarchique des couches métiers/techniques comme suit (figure 10.6) :

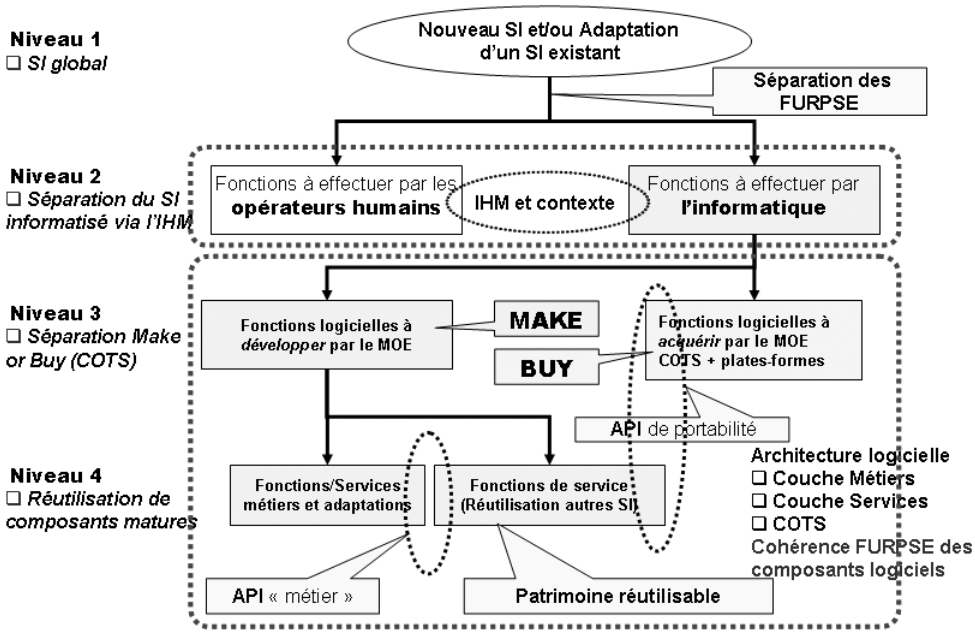


Figure 10.6 - Couches métiers – Couches techniques

En retravaillant la figure 1.4, nous avons le schéma 10.7 :

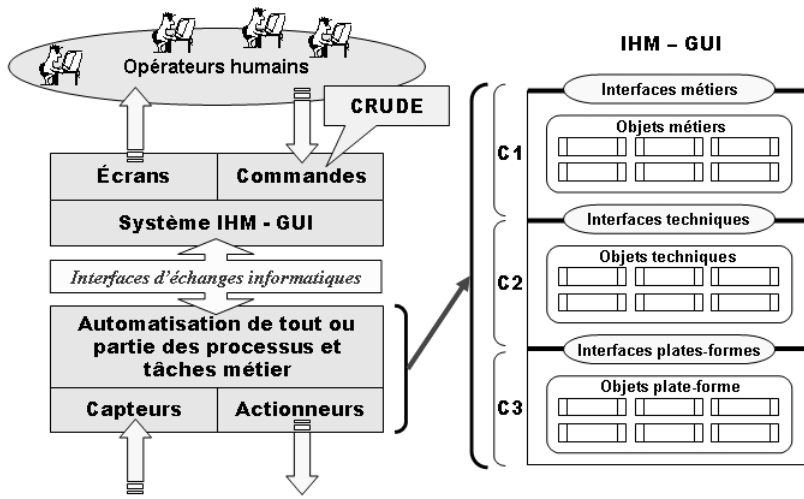


Figure 10.7 - Les trois couches fondamentales

Dans cette architecture, chaque couche peut évoluer indépendamment ; elle n'est connue de l'extérieur que par son interface qui assure la stabilité de l'ensemble. C'est l'essence même de l'architecture. Concernant les objets plates-formes, qui sont

parfois très divers, tout en étant apparentés : équipements de même nature, versions différentes d'un même équipement, équipement dont on veut restreindre l'emploi de certaines fonctions, etc., on a souvent intérêt, pour créer de la simplicité, à prendre un niveau d'abstraction supplémentaire. En génie logiciel, c'est une encapsulation. Ceci permet de contrôler les dépendances du système par rapport à ces équipements, en parfaite conformité avec les principes de l'architecture en couche. La structure logique de l'interface est comme suit (figure 10.8) :

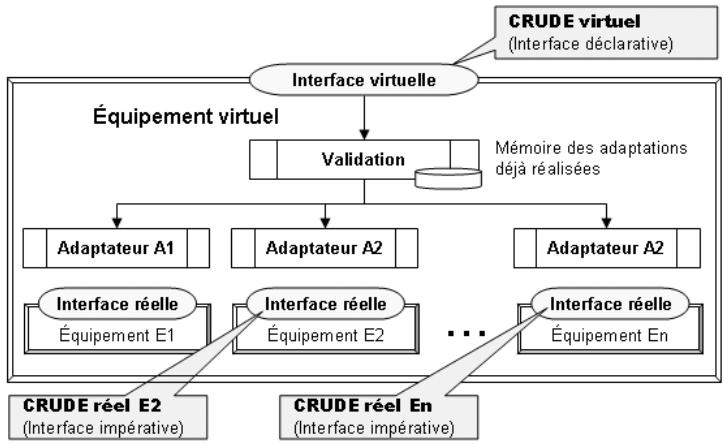


Figure 10.8 - Equipement virtuel

Le même principe pourrait s'appliquer à la fabrication d'un équipement virtuel résultant de l'intégration de plusieurs équipements réels. L'ensemble est vu par le reste du système comme une parfaite boîte noire qui n'est connue que par son interface.

L'interface virtuelle est validée en entrée, puis traduite dans le langage de l'interface réelle de l'équipement qui assure la fonction. Le résultat de la traduction peut être stocké dans la mémoire des adaptations déjà réalisées, ce qui peut améliorer les performances (effet « cache »).

Du point de vue de l'appelant, une interface déclarative est toujours plus simple, mais ce n'est pas une obligation et ce n'est pas toujours possible. Le mode d'appel par défaut est impératif (i.e. un langage de commande).

2ème Cas - Fractionnement d'une couche en plusieurs sous couches

Il existe aujourd'hui de nombreux systèmes ou les transformations effectuées sont telles qu'elles induisent des projets de très grande taille dont la complexité organisationnelle va poser problème. Les principes de modularité préconisés par le génie logiciel conduisent à organiser la transformation en une série d'étapes qui progressivement vont fabriquer le résultat attendu.

Le schéma logique résultant de ce travail d'abstraction peut se schématiser comme suit (figure 10.9) :

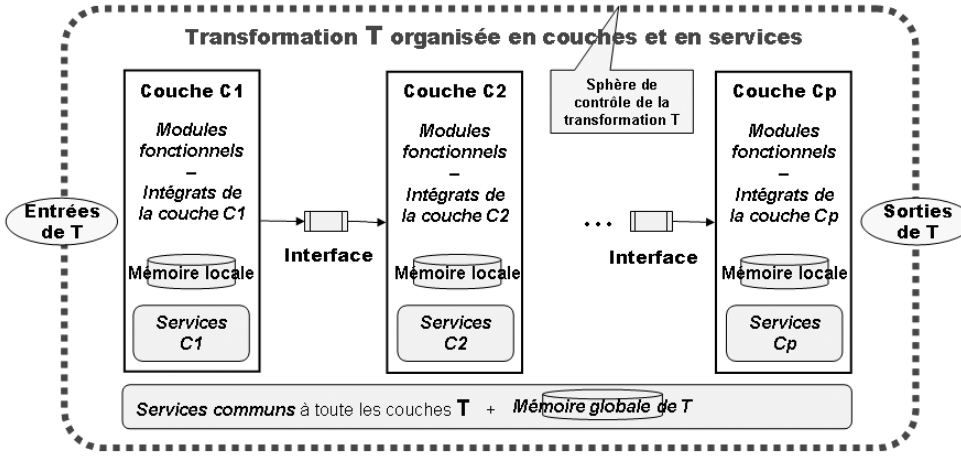


Figure 10.9 - Modèle d'architecture générique en couches

Dans ce modèle, chaque couche à sa mémoire locale et des services qui lui sont propres. L'ensemble de la transformation dispose de services communs à toutes les couches, ainsi qu'une mémoire globale de la transformation T. La communication entre les couches est effectuée via les structures interfaces.

Dans un tel modèle, il y a deux niveaux d'abstraction :

- Niveau 1 : les services communs, la mémoire globale et les interfaces.
- Niveau 2 : les services Ci et la mémoire locale à Ci.

Les modules/intégrats utilisent librement ces deux niveaux de services.

Pour que le modèle soit efficace, et résiste aux évolutions inéluctables (on emploie parfois le terme de résilience), il faut que les éléments constitutifs des niveaux 1 et 2 soient beaucoup plus stables que les modules fonctionnels puisque ces derniers s'appuient sur les premiers.

Les traducteurs de langage, les SGBD, les piles de services utilisés dans les protocoles de télécommunication sont de bons exemples de ce type de structuration.

10.2.2 Identification des fonctions de services et des interfaces entre couches

La difficulté de la mise en œuvre des architectures en couches est la découverte des abstractions constitutives des deux niveaux. Ces abstractions sont spécifiques des

domaines de problèmes concernés par la transformation T. L'architecte regroupe la complexité dans les services de façon à créer de la simplicité au niveau des modules. C'est évidemment plus facile à dire qu'à faire, mais c'est cette capacité qui est le signe de reconnaissance des vrais architectes.

Un projet de réalisation d'une transformation T dont l'arbre produit correspondrait au modèle générique serait naturellement structuré en un ensemble de sous-projets correspondant aux couches et en un projet d'intégration dont le rôle est de définir et faire réaliser les éléments constitutifs des niveaux 1 et 2. C'est le cœur du référentiel projet.

NB : On remarquera l'analogie de ce procédé avec celui ayant conduit à la découverte des différentes catégories de machines informationnelles (cf. figure 10.5).

Pour mutualiser les services et les mémoires associés, il faut d'abord les identifier, c'est-à-dire les percevoir, au sens phénoménologique. L'histoire des sciences et des techniques nous enseigne que cette perception est un art difficile, d'où la nécessité pour les architectes d'avoir une grande culture de leur domaine métier et des technologies, et des retours d'expériences abondants pour en connaître les limites. Nous allons ébaucher un procédé d'analyse qui permet cette identification, tels que les architectes des systèmes informatisés la pratiquent. Les mécanismes psycho-cognitifs à l'œuvre sont ceux de l'abstraction tels que décrit par J. Piajet (voir, dans l'encyclopédie Pleïade, son ouvrage collectif, *Logique et connaissance scientifique*, qui contient toutes les références utiles).

Premier stade : Inventaire

Au premier stade de l'analyse, la transformation à effectuer est identifiée par la nature des entrées et sorties, des contraintes environnementales et des ressources nécessaires, conformément au schéma 10.10 :

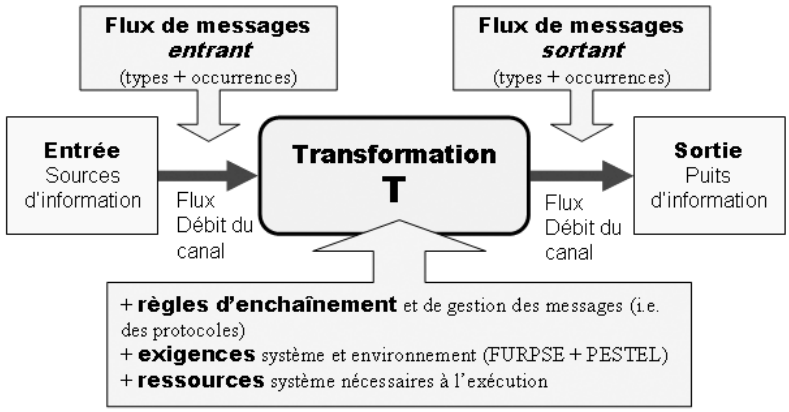


Figure 10.10 - Contexte de la transformation

NB : On remarquera que le contexte de la transformation correspond à ce que dans la théorie des ensembles on appellerait une représentation en extension de la transformation, ce qui peut être décrit avec les diagrammes de collaboration et les cas d'emplois du langage UML. Le concept de représentation en extension est logiquement cohérent, ce qui n'est pas le cas de la notion de cas d'emploi où l'on peut tout mélanger, extension + intention. Le risque de la conception par cas d'emploi a été souligné par de nombreux auteurs.

On fait l'hypothèse que ce qui entre et sort est discrétisé sous forme de messages, ce qui est toujours le cas en informatique, même quand ce qui entre est le résultat d'un capteur analogique dont le signal a été digitalisé.

Deuxième stade : Organes nécessaires

Au deuxième stade, nous faisons apparaître les organes et les mécanismes logiques indispensables à la transformation, conformément à notre approche par machine informationnelle. Le schéma de contexte devient (figure 10.11) :

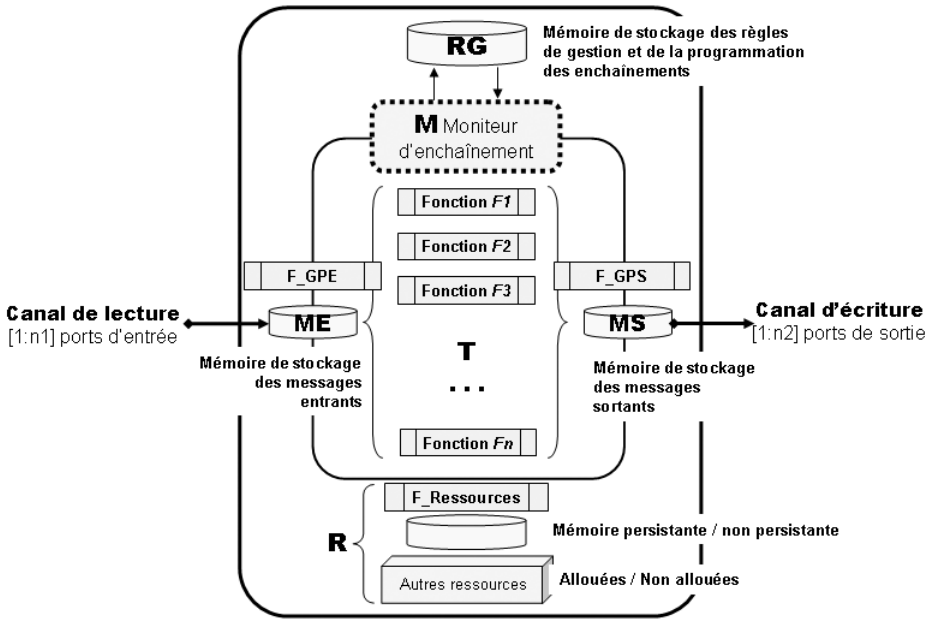


Figure 10.11 - Organes fonctionnels logiques de la transformation

Autour du bloc fonctionnel qui assure le cœur de la transformation T, nous avons fait apparaître quatre organes fonctionnels indispensables qui caractérisent le contexte d'exécution des fonctions du bloc fonctionnel. Tous ces organes ont des caractéristiques qui leur sont propres ; il convient de bien les identifier, pour mutualisation ultérieure avec d'autres transformations.

ME et MS, et les fonctions de gestion des ports associées F_GPE et F_GPS sont dépendantes de la structure de ces ports quels qu'ils soient, ce qui n'est pas le cas des fonctions transformatrices F1, F2, …, Fn. Il faut les séparer de l'environnement afin d'identifier les interfaces.

Idem pour les ressources R et la fonction de gestion F_ressources associée.

Le moniteur d'enchaînement M met en exécution les règles de gestion RG telles qu'elles résultent de l'analyse métier. La programmation des règles peut être impérative, ou déclarative (automates d'enchaînement, grammaire). Elles peuvent faire l'objet de paramétrages, donc il faut les identifier avec précision, pour satisfaire, par exemple, une contrainte d'évolutivité.

Troisième stade : Sémantique des messages

Le cœur sémantique de la transformation T matérialisé par le bloc fonctionnel B_F#i est indépendant de la nature des ressources utilisées pour effectuer la transformation. C'est pourquoi nous avons décomposé la fonction F_Ressource du schéma précédent en un ensemble plus fin prenant en compte les différentes catégories de ressources nécessaires. Le pilotage de l'accès à ces différentes ressources se fait via le moniteur de gestion de ressources G_R, conformément au schéma 10.12.

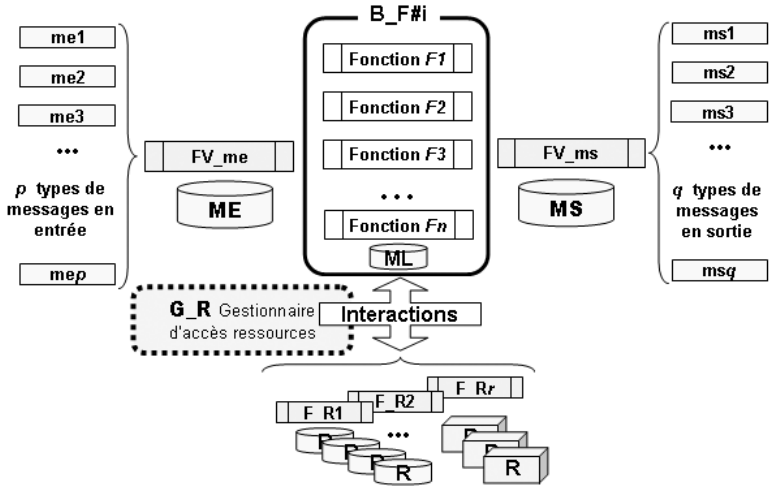


Figure 10.12 - Identifier la sémantique des messages

Le résultat de ces fonctions d'accès est une vue logique de l'information, ce qui permet : a) aux fonctions F#i d'être indépendante de la structure physique des différentes ressources, et b) d'isoler rigoureusement la sémantique. Cette distinction est classique dans les bases de données où il est recommandé de distinguer le modèle logique du modèle physique (cf. figures du chapitre 2, section 2.3, indépendance des données et des programmes).

Les fonctions FV_me et FV_ms valident et vérifient ce qui entre et ce qui sort du bloc fonctionnel. Ce qui permet au bloc B_F#i de s'abstraire de tout travail de validation. En cas d'anomalies de fonctionnement de B_F#i, la défaillance ne peut provenir que de défauts internes aux fonctions F#i.

Nous pouvons maintenant nous concentrer sur la transformation elle-même des p messages en entrée en q messages en sortie.

Quatrième stade : distribution des contraintes

Tout message en sortie est construit à partir des informations véhiculées par les messages en entrée et l'état interne de ML. On peut représenter cette correspondance par un produit cartésien entre les deux ensembles de messages {ME_p × MS_q}.

Si l'on ne considère que les caractéristiques fonctionnelles résultant des analyses métier, le produit cartésien peut se représenter à l'aide d'un plan unique. Ce plan caractérise une vision de la transformation opérée dans M1 idéalisé où il n'y aurait ni erreur ni limitation de ressources (cf. paragraphe 3.1 ci-dessus).

Si l'on considère les contraintes résultant des caractéristiques non fonctionnelles (FURPSE) et de l'environnement (PESTEL), le plan fonctionnel idéal, se dédouble en autant de plan qu'il y a de contrainte à prendre en compte pour effectivement caractériser la transformation dans M1 contraint. Le schéma de synthèse de la transformation peut se représenter comme suit (figure 10.13) :

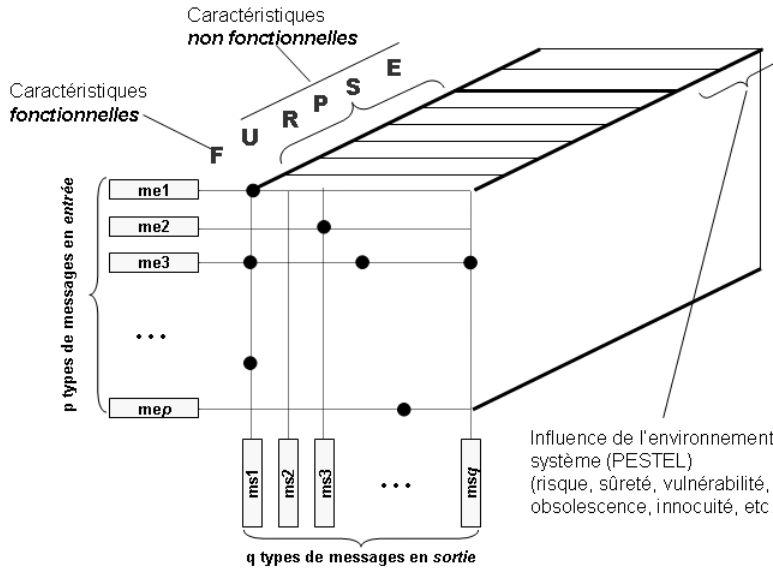


Figure 10.13 - Distribution des contraintes

Cette analyse permet de comprendre l'importance qu'il faut attacher à la prise en compte des exigences non fonctionnelles et à leur bonne gestion (NB : c'est le problème fondamental de l'expression de besoin et de la traçabilité). Ne pas en tenir compte peut conduire à mutualiser des fonctions qui pourraient être différentes du point de vue du comportement.

Inversement, une sur-spécification ou une sur-détermination de ces exigences pourrait conduire à des fonctions différentes alors que si le MOA avait été plus raisonnable dans les exigences, l'architecte aurait pu mutualiser, donc réaliser des économies d'échelle dans le projet.

Cette analyse est obligatoire si l'on veut mettre en évidence un optimum raisonnable CQFD/FURPSE/PESTEL, en coût complet. Ne pas la faire est un facteur d'échec important des projets informatiques (cf. le rapport du Standish group, *Chaos chronicles*, version 3).

Notons que le jeu de fonctions ainsi identifiées réalise, selon les valeurs respectives de p et q, une fusion d'information (cas $p > q$), c'est-à-dire un codage, ou une fission (cas $p < q$), c'est-à-dire un décodage .

Dans le premier cas, la quantité d'information (au sens de C. Shannon) baisse, dans le second cas elle augmente.

La nature des fonctions informatiques nous donne une méthode générale d'analyse en fonctions élémentaires que l'on va pouvoir regrouper par famille sémantique. Il y a cinq catégories de fonctions élémentaires qui peuvent être plus ou moins complexes selon les situations rencontrées : adresser, lire, écrire, adapter-traduire (changement de syntaxe), transformer (i.e. l'opérateur de la transformation, le cœur sémantique, qui peut être un algorithme plus ou moins complexe, voire très complexe comme les opérateurs de data-mining ou de maillage d'une surface en 3D).

Le tableau ci-dessous résume l'enchaînement des six étapes de la transformation T (NB : on remarquera l'analogie avec la boucle d'exécution des instructions dans l'architecture de von Neumann). À chaque étape, le vecteur d'état évolue, en fonction des situations rencontrées.

Tableau 10.1 - Les opérations élémentaires d'une transformation T

Etape	Entrée	Opérations	Sortie	Etat	Commentaires
E1	×	Accès aux données E, S	×	×	Possibilité de conflit d'accès lors de l'adressage des données – Verrouillage éventuel des mémoires E et S
E2	×	Lecture des données E		×	Recopie dans une mémoire de travail
E3	×	Adaptation au format de T		×	Les données lues sont adaptées au format de l'algorithme choisi

Etape	Entrée	Opérations	Sortie	Etat	Commentaires
E4		Exécution de T		×	Algorithme de la transformation T (plus ou moins complexe) dans la mémoire de travail ML de B_F#i
E5		Adaptation au format de S	×	×	Le résultat de la transformation T est adapté au format de sortie
E6		Ecriture dans la mémoire S	×	×	Possibilité de conflits d'accès

On remarquera que dans cette approche, ce sont les données, au sens large, qui pilotent l'identification des fonctions élémentaires. La notion de chemin de données est une notion centrale des architectures applicatives, comme elle l'est pour l'architecture des unités centrales, pour les mêmes raisons. Nous en donnerons une justification au chapitre 12, Simplicité.

Les données permettent toujours une représentation en extension de l'opérateur de transformation (cf. les tables numériques avant l'invention des calculettes de poche) ; la représentation en intention sous forme d'algorithmes vient plus tard (cf. les fonctions analytiques calculées à l'aide de séries). Pour illustrer ces notions un peu abstraites, par nécessité, donnons quelques exemples de modèles génériques d'architecture en couches, rencontrés dans de nombreuses situations.

10.2.3 Modèle ETL (*Extract, Transform, Load*)

C'est le modèle de la fusion de données que l'on rencontre tout aussi bien en data-mining, qu'en gestion de multi-capteurs homogènes et/ou hétérogènes. Le schéma de principe du modèle ETL est le suivant (figure 10.14) :

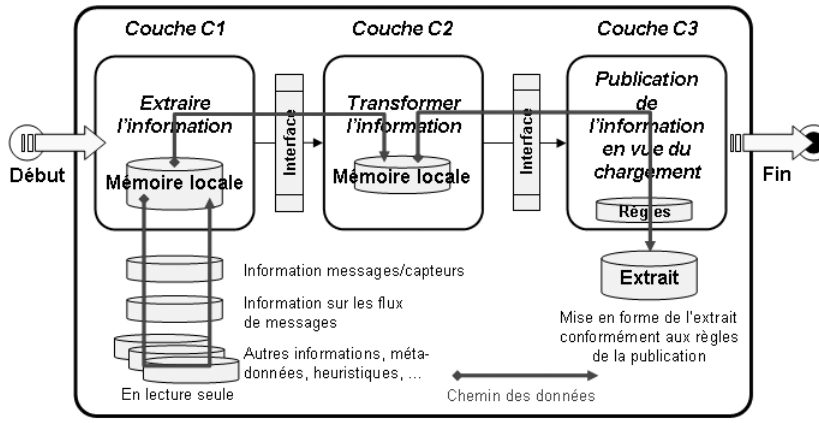


Figure 10.14 - Modèle d'extraction de données ETL

Ce qui est caractéristique du modèle ETL est la multiplicité et l'hétérogénéité des sources d'information, et la fragmentation de l'information.

Pensons aux caisses d'une chaîne de grands magasins qui fournissent une information brute du comportement des acheteurs (pour autant qu'ils payent avec des cartes de crédits, car en plus ils sont alors identifiés) ; ou aux capteurs nécessaires à la gestion du réseau de transport de l'énergie électrique ; ou aux capteurs (Radar, Satellites, Balises,…) nécessaires au contrôle aérien, ou à ceux nécessaires à la mise en œuvre du concept stratégique du « *Network Centric Warfare* » qui nécessite une surveillance mondiale ; ou aux commerciaux de terrain qui renseignent leur progiciel CRM et/ou PGI en temps réel.

Le premier problème est la gestion de la masse d'information et des flux aléatoires qui remontent via les réseaux de données. Le second est celui de la qualité des données brutes à partir desquelles il faudra élaborer des données de synthèse pour pouvoir décider. Le troisième est l'opérateur de transformation proprement dit qui met en œuvre des techniques statistiques souvent très élaborées. La publication n'est pas un véritable problème, quand bien même, quantitativement parlant, elle puisse nécessiter des programmes de très grande taille. En terme de quantité d'informations, on passe d'un ensemble fragmenté et chaotique (entropie très élevée) à un ensemble très structuré et organisé (entropie faible) ; l'analogie thermodynamique serait intéressante à développer.

Signalons, entre autre, le problème de l'indexation dans les bases de données de très grande taille, le géo-référencement spatio-temporel, etc., qui n'ont pas de solution générale et nécessitent des adaptations au cas par cas.

L'intégration et l'exploitation de tels systèmes, surtout lorsqu'ils fonctionnent en temps réel, est un problème extrême, car en cas de défaillance et/ou de doute sur la qualité de ce qui est transmis aux décideurs, il faut pouvoir remonter la traçabilité vers les sources initiales.

Le seul élément simplificateur de ce modèle est que les données sont en lecture seule. Il y a trois couches logiques, qui pourraient elles-mêmes se scinder en plusieurs sous-couches.

10.2.4 Modèle CRUD (*Create, Retrieve, Update, Delete*)

C'est le modèle générique de la gestion de données, tel qu'il est implanté dans les SGBD. Pour bien comprendre la problématique de ce modèle fondamental, il est utile de se cultiver le plus possible sur le problème des données qui est le cœur de la technologie de l'information. Sans gestion de données (data management, data intégration), il n'y a pas de traitement de l'information.

Il existe de nombreux ouvrages, facilement accessibles, sur ce sujet ; ceux de J.D. Ullman et T. Teorey (voir bibliographie) sont excellents.

Dans le modèle CRUD, un serveur client fait une requête au serveur de gestion des données, qui la traite, et fournit en réponse l'ensemble des données qui satisfont la requête, et l'état de l'opération effectuée (fait, non fait, re-essayer,…).

Le schéma de principe du modèle CRUD est le suivant (figure 10.15) :

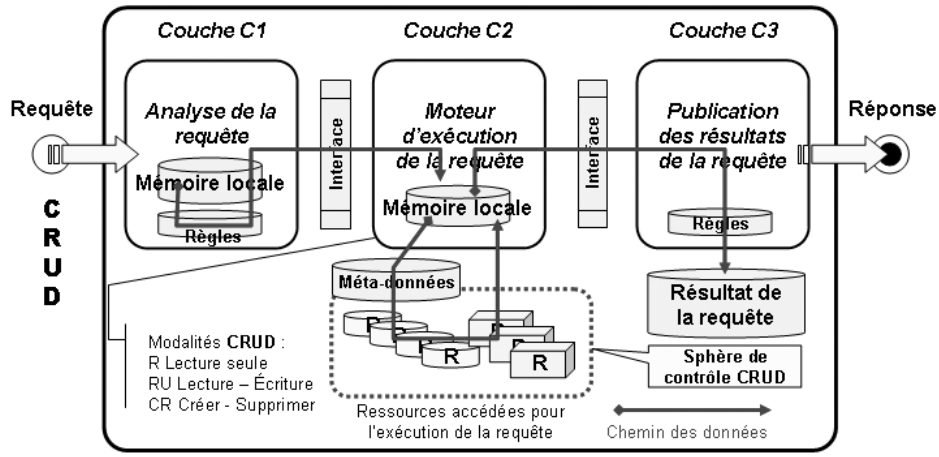


Figure 10.15 - Modèle de gestion de données CRUD

Dans le modèle CRUD, il n'y a que quatre types de requêtes qui peuvent cependant avoir de nombreuses variantes.

Il y a trois modalités d'exécution principales :

1. La **lecture seule** (le R de CRUD), qui est le cas simple ; aucun risque de détérioration de l'information.
2. La **lecture-écriture** (le RU de CRUD) qui amène le problème de la mise à jour des donnés et des risques associés (cohérence des données dupliquées, cohérence des indexes, cohérence des caches, etc.). Sur ces problèmes, voir en particulier :
   – G.Kowalski, M.Maybury, *Information storage and retrieval systems*, Kluwer, 2000.
   – B. Mc Nutt, *The fractal structure of data reference*, Kluwer, 2000.

   La mise à jour propre nécessite de mettre en œuvre la notion de transaction vue précédemment.
3. La **création-suppression** (de CD de CRUD), qui sont des opérations complexes mais rares, car elles nécessitent l'accès à des ressources centralisées (fiabilité, performance). La suppression intempestive d'entités peut faire courir de très graves risques d'incohérence si les adresses de l'entité n'ont pas été convenablement gérées (cf. la duplication anarchique de pointeurs).

Tous les problèmes liés à la gestion de données sont correctement gérés au niveau du SGBD, mais tout réapparaît lorsque l'on manipule des données dans plusieurs bases, de surcroît hétérogènes, et éventuellement distribuées. Dans ce cas, la plus

grande prudence s'impose et il faut gérer l'ensemble de ces données comme si c'était une base unique. C'est le rôle du modèle CRUD global de structurer cette gestion, en s'appuyant sur les CRUD spécifiques des SGBD utilisés.

En environnement distribué ouvert (en particulier Internet), le U et a fortiori le CD de CRUD, posent des problèmes extrêmes, car c'est la porte ouverte à toutes les malveillances si l'authentification des acteurs ayant ce type de droit est mal gérée. En tout état de cause, il convient de distinguer les requêtes CRUD qui proviennent de l'intérieur du système, de celles qui proviennent de l'extérieur. La notion de sphère de contrôle (i.e. la « peau » du système) trouve ici une nouvelle fois toute son importance pour assurer la sûreté de fonctionnement de l'ensemble.

La création-suppression d'entités détruit inexorablement la structure topologique de la mémoire de stockage qui se fragmente en morceau de toute taille, créant progressivement une structure fractale faite de trous qu'il faut périodiquement tasser. L'opération correspondante est appelée « garbage collection » (ou ramasse-miettes) ; elle nécessite l'arrêt complet de toutes les opérations sur la base pendant la dé-fragmentation.

L'effet le plus spectaculaire de la fragmentation progressive de l'espace de stockage est une baisse concomitante mais aléatoire des performances d'accès à l'information, qui elle-même engendrera des phénomènes de saturation, ailleurs dans le système (taille des files d'attente, time-out sur certaines ressources, …).

Si aucune action préventive n'est effectuée, la panne aléatoire est inéluctable. Elle se produira à un endroit qui n'a en général rien à voir avec la cause réelle de la défaillance, d'où un diagnostic difficile.

Le cœur du modèle CRUD est le moteur d'exécution des requêtes. C'est une machine abstraite dont les commandes permettent de naviguer dans la structure de données (c'est un MCD) jusqu'à identification de la donnée recherchée. Pour se faire une idée précise de ce type de machine, voir par exemple le modèle de navigation des bases de données en réseau (le langage NDL, *Network Data Langage*).

Dans le modèle relationnel, le moteur d'exécution est complètement masqué par le langage SQL qui est déclaratif (ou fonctionnel). Un ordre SQL select identifie en intention un sous-ensemble de données (NB : c'est une fonction propositionnelle qui répond VRAI chaque fois qu'un élément satisfait à la condition de sélection) ; l'exécution de cet ordre (c'est une navigation dans des structures arborescentes faites de B-tree) calculera l'extension de cet ensemble. La couche C1 du modèle est un compilateur de requêtes conforme à ce qui a été décrit dans la partie 2.

Le texte généré est conforme aux règles de la machine abstraite d'exécution des requêtes.

Le texte initial de la requête peut se présenter sous différentes formes : format binaire, format textuel (éventuellement XML_isé), format graphique avec icônes, etc.

Le résultat de la traduction est stocké dans une mémoire locale au cas ou une requête similaire viendrait à se présenter à nouveau. Ce type de mémorisation a été

exploité « à fond » par les architectes des SGBD relationnels pour résoudre progressivement les problèmes de performance inhérents au style déclaratif du langage SQL.

L'extension de l'ensemble constituant le résultat est publié conformément aux règles de présentation adoptées (couche C3).

La mémoire de la machine abstraite CRUD est décrite par un modèle conceptuel (meta-données) qui permet d'effectuer les traductions des commandes CRUD génériques vers les commandes spécifiques des ressources utilisées. Si ces ressources sont des SGBD, la machine abstraite CRUD pourra utiliser tous les mécanismes disponibles, pour simplifier le travail à effectuer.

Pour effectuer des mises à jour synchronisées de façon sûre entre plusieurs ressources, il faut un gestionnaire d'accès concurrent généralisé, ce qui revient à implémenter les propriétés ACID au niveau du moteur d'exécution. En cas de lecture seule, c'est évidemment beaucoup plus simple.

On remarquera que le modèle ETL implique un CRUD simplifié pour gérer l'information dont est extraite la partie intéressante.

10.2.5 Modèle Client – Serveur/Services

Ce modèle compose librement différents modèles ETL et/ou CRUD, en ajoutant un nouvel organe qui est la bibliothèque de services et l'éditeur correspondant qui permet la gestion de cette bibliothèque.

Les services disponibles sont décrits via les méta-données qui leur sont associées. Ce méta-modèle doit fournir au client toutes les informations nécessaires pour invoquer le service correspondant et en récupérer le résultat.

Le moteur d'exécution des services a comme fonction celle de dialoguer avec le client, via un jeu de commandes (requêtes et réponses) permettant de collecter toute l'information nécessaire à l'exécution du service, puis d'activer le service pour le compte du client. Le moteur d'exécution supervise toutes les opérations et les interfaçages avec les ETL et les CRUD utilisés par le service (NB : cela fait partie des méta-données gérées par le modèle conceptuel des services via un langage de définition de services ; cf. comme exemple, le WSDL de l'OMG).

Le schéma de principe du modèle Client-Services est le suivant (figure 10.16) :

Dans ce modèle, une question de fond est celle de la mise à disposition de l'éditeur de services. Du point de vue du client, deux cas sont à considérer.

- 1er Cas : Le client utilise les services qui lui sont proposés. Il est tributaire du fournisseur de services pour obtenir des aménagements ou de nouveaux services.
- 2ème Cas : le client souhaite définir ses propres services et les faire héberger via le fournisseur de services, pour éventuellement les valoriser auprès d'autres

clients. Dans ce cas, le client initial doit disposer de l'éditeur de services et en assurer les risques de l'ingénierie (évolutions, nouveaux interfaces, etc.).

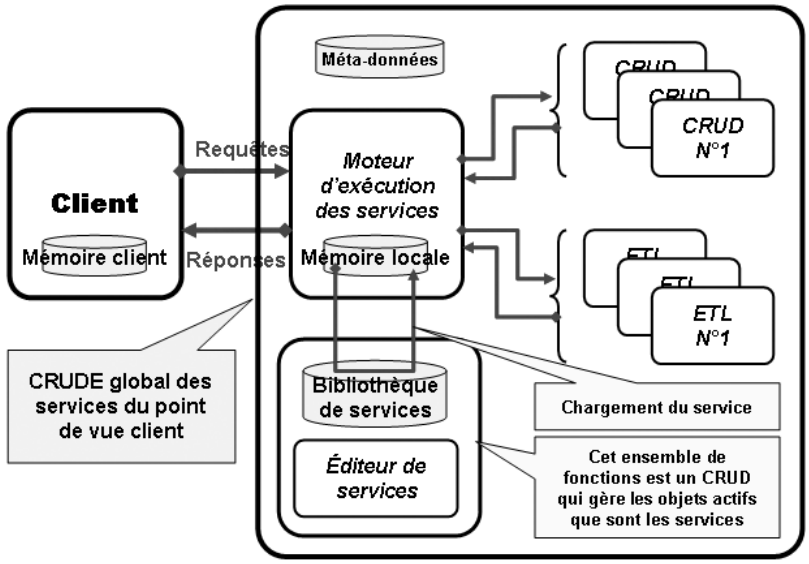


Figure 10.16 - Modèle Client-Services

L'ensemble, du point de vue du client, peut être assimilé à un équipement virtuel dématérialisé, un CRUDE, avec un E pour EXÉCUTER. Les données qui lui appartiennent pourraient être hébergées dans un CRUD qui resterait sa propriété. Ne resterait sur le poste client que sa mémoire locale sous la forme de documents, de tableaux EXCEL, d'un navigateur.

10.3 LE MODÈLE GÉNÉRIQUE TRADUCTEUR-TRANSDUCTEUR TT

Avec l'exemple des compilateurs décrit au chapitre 6, nous avons vu avec un relatif détail, un aperçu du modèle de type TT. L'intérêt capital de ce modèle est sa généralité et sa facilité de compréhension compte tenu des capacités linguistiques qui sont l'une des spécificités de l'espèce humaine. Tout le monde a une intuition et un sens inné du langage, comme N.Chomsky l'avait souligné dans ses travaux fondamentaux. Le concept traducteur est accessible au très grand nombre des informaticiens. La syntaxe et la sémantique sont décrites de façon précise, avec des formalismes ad hoc comme les grammaires et/ou les automates.

Avec un peu de recul, on peut analyser la fonction traduction de la façon suivante (figure 10.17) :

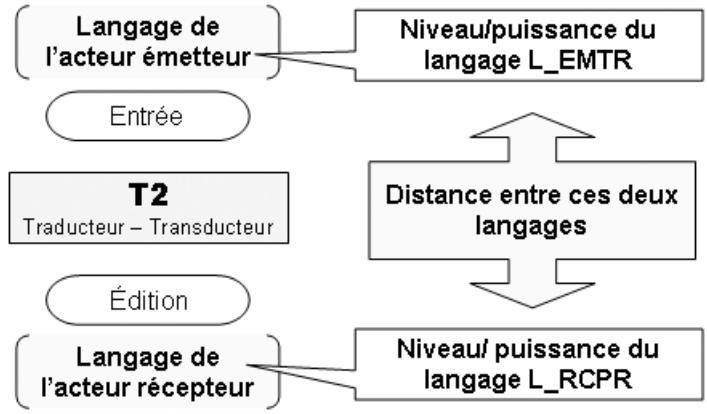


Figure 10.17 - Modèle générique Traducteur-Transducteur

Trois cas sont à considérer :

- Cas 1 : Les langages sont de même niveau ou de niveau très voisin ; par exemple un traducteur de format de données (schéma XML vers différents DDL).
- Cas 2 : Le langage de l'acteur émetteur est plus puissant que le langage de l'acteur récepteur (i.e. plus concis), soit : L_EMTR > L_RCPR ; c'est la compilation, au sens habituel, où l'on va du général au particulier.
- Cas 3 : C'est l'inverse, soit : L_EMTR < L_RCPR ; c'est un décompilateur, ou compilateur inverse, qui partirait d'un texte binaire pour fabriquer du C++ ou du Java.

Il est intéressant d'exprimer le niveau du langage en terme de quantité d'informations. Chacun s'accorde à dire qu'un programme en assembleur est plus « complexe » que le même écrit en C++ ou Java. En moyenne la taille du texte en nombre d'instructions est multipliée par 5 ou 6.

Il y a beaucoup plus de symboles utilisés dans l'un que dans l'autre. Si l'on assimile le texte du programme à un message, au sens de la théorie de l'information, on peut calculer la quantité d'information (au sens de Shannon) du programme. La valeur de cette quantité est liée à la fréquence d'emploi des symboles et à la longueur du texte.

On n'entrera pas ici dans une discussion fine de la pertinence de cette mesure, mais on se contentera de remarquer que la quantité d'information, au sens de Shannon, du texte assembleur est beaucoup plus importante que celle du texte C++/Java, ce qui traduit correctement l'intuition de la complexité textuelle.

Un programmeur n'aura pas trop de difficulté à mémoriser un texte de 1 000 LS Java, mais calerait sur l'équivalent assembleur qui en ferait 5 000 ; la traduction fait

passer, en gros, le texte de 20 pages à 100 pages. Chacun peut comprendre que l'effort de mémorisation n'a rien à voir dans chacun des cas. La différence prend en compte le fait que le programmeur Java est débarrassé de la connaissance de la machine (provisoirement) sur laquelle son programme sera exécuté.

Le programme assembleur, quant à lui, intègre en son sein la complexité de la machine sous-jacente.

La situation des 3 cas se résume par le tableau suivant :

Tableau 10.2 - Typologie des traductions

Type N°1 – TT1	Type N°2 – TT2	Type N°3 – TT3
L_EMTR ≈ L_RCPR (puissance comparée)	L_EMTR > L_RCPR (puissance comparée)	L_EMTR < L_RCPR (puissance comparée)
La traduction est un simple changement de format qui se réduit à des manipulations syntaxiques et des règles de transformations simples.	La traduction est une compilation qui enrichit le texte traduit de la complexité de la machine cible par injection massive d'information.	La traduction est une recherche de structures et de régularités dans le texte initial (recherche de « patterns » fonctionnels) pour synthétiser l'information.
La taille du texte, en nombre de symboles, est stable.	La taille du texte augmente fortement.	La taille du texte diminue.

La situation la plus difficile est le type 3 : Cas des langages idéographiques (hiéroglyphes égyptiens, les caractères chinois), ou du décodage des génomes. Sans modèle fonctionnel cible le cas 3 n'a pas d'issue.

C'est d'ailleurs la raison pour laquelle l'accès libre au code source d'un système d'exploitation comme LINUX ne fait courir aucun risque, car quel utilisateur un peu sensé irait s'amuser à le modifier, si tant est qu'il le comprenne !

Dans ce cas, la meilleure protection est la quantité d'information du texte source et sa complexité.

D'une certaine façon, dans le modèle ETL vu précédemment, l'extraction d'information est un exemple de traduction de type 3, mais l'on sait exactement ce que l'on cherche sous la forme de tel ou tel pattern (régularité statistique) dans le cas ETL du data-mining.

NB : La puissance du langage est un paramètre important des modèles d'estimation car il corrèle l'effort et la taille du texte programmé[1].

1. Cf. J. Printz, *Productivité des programmes*, chez Hermès-Lavoisier

Le modèle PTE - Présentation, Transformation, Édition

C'est le modèle le plus fréquent car il correspond à la démarche méthodologique classique qui consiste à passer d'un modèle abstrait général à un ou des modèles concrets, spécifiques à telles ou telles situations.

Le schéma de principe de ce modèle est le suivant (figure 10.18) :

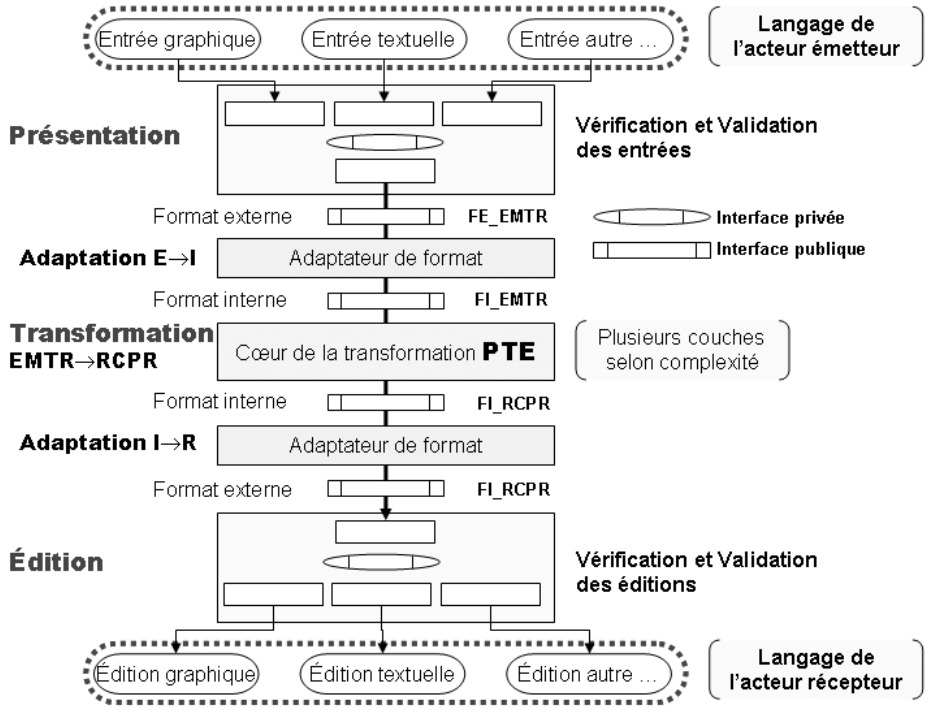


Figure 10.18 - Modèle générique PTE – Présentation, Transformation, Edition.

C'est un modèle à trois couches.

La couche présentation peut être alimentée par différentes entrées selon les caractéristiques de l'acteur du langage émetteur. Toutes ces entrées sont informatisées sous la forme d'un langage privée à la couche présentation de façon à leur faire subir un même traitement générique (NB : c'est la distinction syntaxe concrète, syntaxe abstraite en compilation ; cf. chapitre 6). Le texte résultant constitue le format externe de la couche présentation FE_EMTR.

Ce format externe est adapté au format d'entrée du cœur de la transformation PTE par l'adaptateur E→I. Cet adaptateur joue un rôle très important car c'est lui qui sépare le format de la présentation du format interne. Les modules PRÉSENTATION et TRANSFORMATION doivent pouvoir évoluer indépendamment l'un de l'autre ; c'est l'adaptation E→I qui sera modifié si nécessaire.

L'absence de cet adaptateur provoque inéluctablement un enchevêtrement des formats FE_EMTR et FI_EMTR dans chacun des deux modules. La complexité augmente et les équipes de réalisation de chacun des modules deviennent interdépendantes (projet plus complexe à manager).

Selon les modes de fonctionnement du modèle PTE, soit traduction interactive en mode conversation, soit traduction par lots différés, l'adaptateur peut comporter une fonction de monitoring pour aiguiller le message correspondant vers la bonne fonction du traducteur. Dans ce cas, nous retrouvons le pattern MVC bien connu depuis le langage Smalltalk et les *design patterns*[1].

En traduction interactive, la transformation PTE est alimentée à la volée, par des messages successifs. Dans ce mode de fonctionnement, PTE peut interagir l'acteur émetteur qui peut, par exemple, corriger à la volée, ce qui est plus convivial.

En traduction par lots différés, les messages sont stockés dans des fichiers gérés par les interfaces FE_EMTR et FI_EMTR. L'interaction se fait en différé par l'édition d'un rapport de transformation qui donnera à l'acteur émetteur toutes les informations nécessaires à la poursuite des traitements.

La couche édition est l'exacte symétrique de la couche présentation.

Le modèle PTE qui a largement fait ses preuves depuis plus de vingt ans, est d'une grande richesse adaptative et permet l'évolution progressive des composants applicatifs qui y sont conformes.

Notons que le langage de l'acteur récepteur peut être le langage de commande d'un équipement complexe (ou d'un ensemble d'équipements) dont on souhaiterait masquer la complexité pour en assurer une meilleure gestion.

Validation de la cohérence des messages émis et/ou reçus

Le contexte de cette validation est celui des modèles généraux des systèmes puits et source vus au chapitre 10.

Le schéma de principe du contrôle de la cohérence des messages reçus est comme suit (figure 10.19) :

1. Cf. E. Gamma, *Design Patterns*, Addison Wesley.

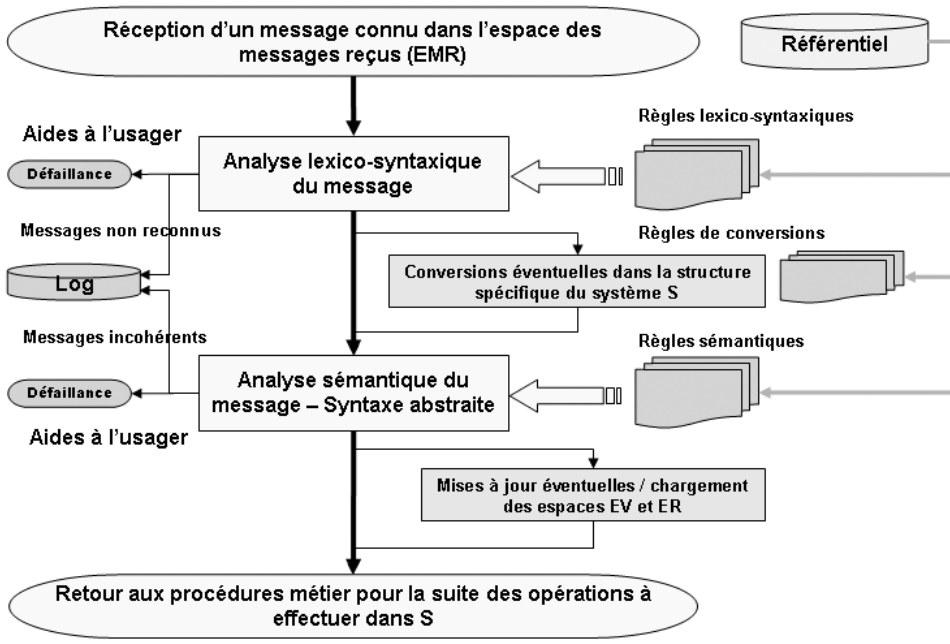


Figure 10.19 - Cohérence des messages reçus

La validation de cette cohérence distingue trois types de règles qui peuvent faire l'objet de paramétrage (cf. Chapitre 14, Adaptabilité).

Les règles lexico-syntaxiques permettent la vérification de la structure du message, conformément aux règles du langage de l'acteur émetteur.

Les règles de conversion permettent de générer le langage générique de la couche présentation, ce qui permet d'effectuer l'analyse sémantique du message, sous sa forme canonique (domaines sémantiques des différents attributs et relations entre ces domaines), conformément aux règles sémantiques, indépendamment de la présentation.

Toutes ces règles peuvent être prises en compte statiquement à l'intégration ou dynamiquement à l'exécution (on a alors un paramétrage à l'installation du logiciel) pour garantir la cohérence globale des messages. Ne pas les appliquer revient à prendre le risque de propager des incohérences et/ou de faire une confiance aveugle à l'émetteur des messages.

En émission, le schéma de principe est l'inverse du précédent, soit le schéma (figure 10.20) :

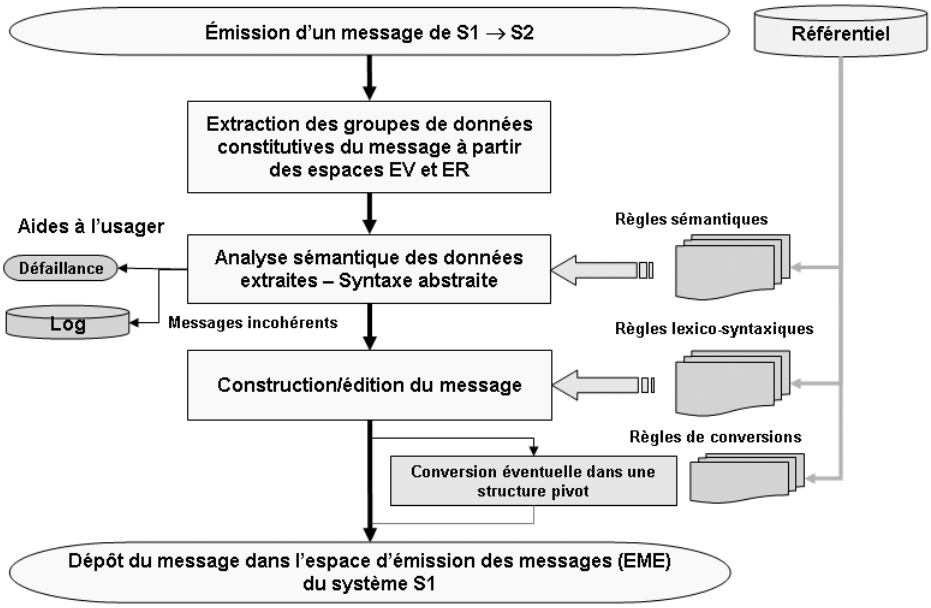


Figure 10.20 - Cohérence des messages émis

Du point de vue du référentiel, tout message existe sous deux formes :

- Une forme conceptuelle, indépendant de toute représentation sur laquelle vont être attachées les propriétés sémantiques (domaines sémantiques des attributs du message + relation entre les attributs et les messages).
- Une forme extérieure, qui est une vue du point de vue de l'acteur concerné par le message. On notera ici l'équivalence du modèle à trois schémas du rapport ANSI-SPARC et du modèle des langages avec les syntaxes concrètes et abstraites qui dénotent exactement la même nuance.

Toutes ces règles sont exprimables dans la notation XML qui est une notation de type grammatical, comme chacun sait, que l'on pourra traduire (traduction de type TT1) dans des notations spécifiques aux langages de la plate-forme d'exécution.

Emploi du modèle PTE pour l'interopérabilité des systèmes

Au chapitre 1, section 1.3, Architecture de l'information, nous avons présenté une approche très générale des interactions entre système, caractéristique de l'interopérabilité d'un ensemble de systèmes. Nous illustrons ici le concept d'adaptateur à partir du modèle PTE que nous venons de présenter.

Schéma de principe d'une architecture d'interopérabilité à l'aide d'un modèle pivot (figure 10.21).

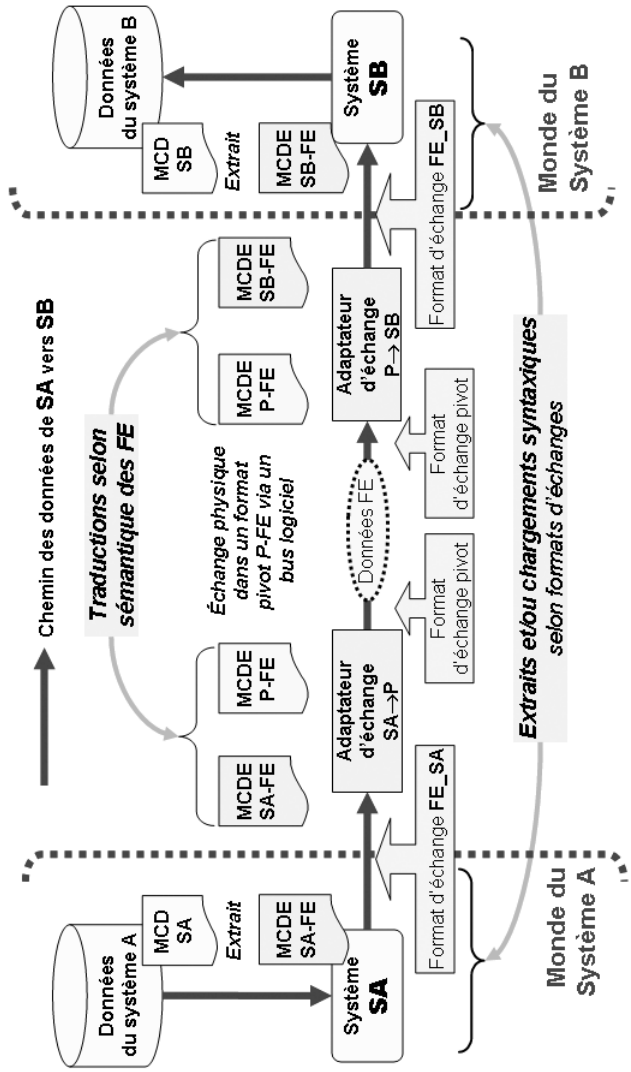


Figure 10.21 - Traduction et interopérabilité

Les systèmes SA et SB ont en commun des données, des procédures métiers, des évènements ainsi qu'un environnement PESTEL. L'interaction entre SA et SB se fait via un ensemble de messages, qui constitue le modèle conceptuel des formats d'échanges MCDE.

Conformément à la logique du modèle PTE, ce format d'échange est défini dans un format pivot unique MCDE/P_FE et dans différents formats spécifiques des systèmes en interaction, soit MCDE/SA_FE et MCDE/SB_FE.

Ces modèles conceptuels des données échangées sont eux-mêmes des sous ensembles des données des référentiels des systèmes SA et SB.

Le chemin de données détaillé de SA → SB est donc le suivant :

Données de SA (MCD_SA) → extraire (via MCDE/SA_FE) → adaptation (SA_FE → P-FE) → émission/réception via le bus → adaptation (P_FE → SB_FE) → charger (via MCDE/SB_FE) → Données(MCD_SB)

Chacun des systèmes reste libre d'évoluer à son rythme ; les projets d'adaptation de chacun d'eux sont indépendants. La logique d'évolution et de compatibilité ascendante des interfaces est entièrement gérée par les adaptateurs d'échanges.

Le rôle du format d'échange pivot P_FE est double :

- Il évite la multiplicité des traducteurs ; si N systèmes sont en interaction, le nombre de traducteurs est égal à N×(N-1) alors qu'il n'en faut que N avec le pivot (mais il faut pour cela faire une architecture pivot qui a un coût).
- Il fournit un point de surveillance unique pour l'observation et le contrôle des échanges, ce qui est une condition nécessaire à la maîtrise de la sûreté de fonctionnement.

Dans les deux cas, l'architecture est plus simple, ce qui se matérialise par un nombre de test et un effort d'intégration moindre.

10.4 MODÈLE GÉNÉRIQUE D'UN MONITEUR SYSTÈME

L'architecture des processus et de la communication inter-processus vue dans la partie 2, nous a donné une idée assez précise de la nature des travaux qui incombent au moniteur de processus, i.e. le dispatcher de niveau système. Le pilote externe du système, pour utiliser la terminologie introduite au chapitre 1, est un opérateur qui dispose d'un certain nombre de commandes avec lesquelles il va administrer le système en fonction des ressources disponibles et des travaux à effectuer.

Le moniteur système est fondamentalement un organe de planification et d'ordonnancement de travaux, avec une fonction de maximisation du contrat de service passé entre les usagers et les opérateurs du système, ce qui exige une bonne connaissance du métier supporté par le système et des comportements que le métier induit dans la partie informatisée (prise en compte du FURPSE métier).

Le diagramme de contexte d'un moniteur a toujours l'allure suivante (figure 10.22) :

Selon la nature des travaux à effectuer et de la disponibilité des ressources, différents types de moniteurs peuvent être définis (en fait, ce sont des familles de moniteurs), conformément aux exigences FURPSE/PESTEL du système et de son environnement. Un moniteur transactionnel comme CICS d'IBM sera très différent d'un moniteur temps réel utilisé pour des applications où il y a une exigence de déterminisme (avionique, nucléaire). Dans les plates-forme de développement des constructeurs, on trouve aujourd'hui des boîtes à outils permettant de réaliser des

moniteurs mais il est recommandé de ne s'y aventurer que si l'architecte connaît parfaitement la problématique.

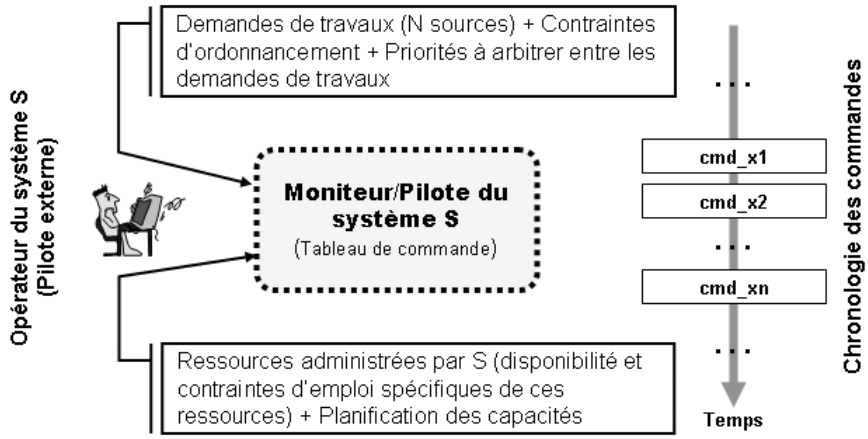


Figure 10.22 - Diagramme de contexte d'un moniteur système.

Moniteur de contrôle-commande, type C2

C'est à la fois le plus simple et le plus ancien. Il correspond à des systèmes fonctionnant sur un mode réflexe STIMULUS – RÉPONSE automatique, avec une mémoire court terme qui est une ressource critique.

Le système reconnaît un certain nombre de stimulus, matérialisés par des messages ou de simples signaux. À chacun d'eux , il sait associer une réponse appropriée avec une logique très simple de type table de décision :

```
Si msg_x et si [expression_logique_1] émettre_réponse_1 ;
si [expression_logique_2] émettre_réponse_2 ;
. . .
sinon émettre_réponse_erreur ;
```

et ceci pour chacun des messages et/ou signaux connus du système.

La structure d'un tel système nécessite deux processus (ou deux types/classes de processus) :

1. Un processus qui reçoit les messages et/ou les signaux, et les analyse selon la logique ci-dessus.
2. Un processus qui reçoit les réponses du précédent et actionne les équipements nécessaires à leur mise en œuvre.

Le pilote externe dispose d'un jeu de commandes rudimentaires du type DÉMARRER / ARRÊTER le système.

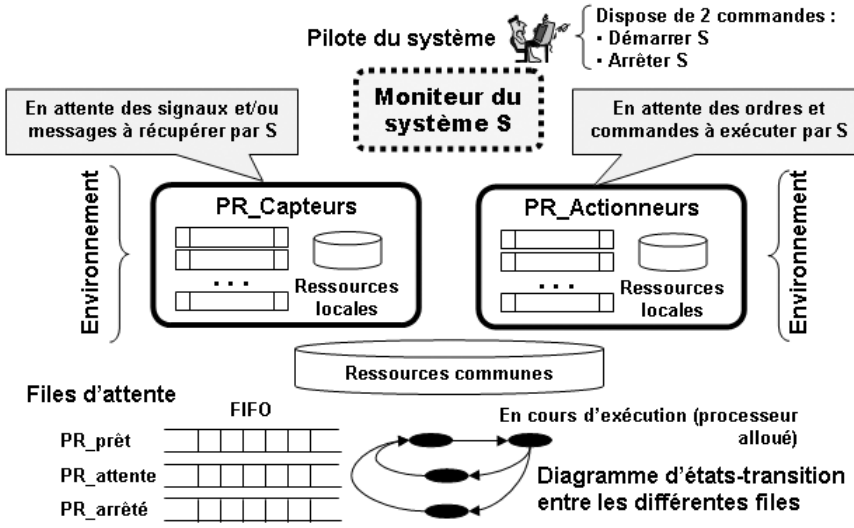


Figure 10.23 - Moniteur de type C2

Les messages qui arrivent sur PR_capteurs ont des priorités différentes. À chaque arrivée de messages, PR_capteur analyse les processus en cours d'exécution et positionne le traitement du nouveau message avec la priorité ad hoc en mettant en attente, éventuellement, tel ou tel processus. Les ordres envoyés sur les équipements peuvent éventuellement mettre à jour les ressources communes, qui seront utilisées lors de la réception de nouveaux messages.

Chaque processus gère un ensemble d'activités concurrentes exécutées en parallèle qui sont des sous-processus des processus pères, PR_capteurs et PR_actionneurs.

Si le temps écoulé entre la réception d'un message et la réponse est contraint, quelle que soit la durée de la contrainte, le système est dit temps réel, ce qui signifie qu'il est asservi sur le temps vrai de l'environnement.

Le moniteur du système effectue l'ordonnancement des tâches, en prenant en compte la contrainte temporelle ce qui nécessite une gestion beaucoup plus fine des tranches de temps allouées aux différents processus à l'aide des « timer » de la machine et/ou d'une horloge externe (NB : le fonctionnement d'une constellation de satellites comme le GPS ou GALILEO nécessite des horloges atomiques de haute précision, avec correction relativiste).

Dans les systèmes C2 ayant des contraintes temps réel fortes, toute la mémoire active est en RAM ; la mémoire persistante de type disque ne sert que pour gérer les

logs, les sauvegardes, qui peuvent être différées hors de la boucle temps réel. Pour satisfaire la contrainte de performance, la gestion mémoire doit rester simple, en général complètement statique.

Le schéma montre une gestion des processus avec trois files d'attente, du type de celles vues au chapitre 7, permettant de compenser les vitesses respectives des différentes tâches et de gérer les priorités.

NB : au démarrage du système, tous les processus sont en attente. L'arrivée du premier message déclenche l'animation du système qui sera orchestré automatiquement. Le pilote n'est là que pour surveiller.

Moniteur de contrôle – commande de type C3I/C4ISR

Les systèmes C3I/C4ISR (*Communication, Command, Control, Computers, Intelligence, Surveillance, Reconnaissance*) pour reprendre la terminologie du DOD, intègre en parallèle avec la boucle C2, une boucle de « réflexion » (on parle alors de systèmes en temps réfléchi, car l'homme est dans la boucle) qui permet d'optimiser les réponses fournies par le système.

Il se trouve que, compte tenu de la puissance des ordinateurs actuels, il est tout à fait possible d'intégrer des traitements sophistiqués en préservant les contraintes temporelles, pour autant que le nombre d'entrées – sorties nécessaires aux traitements reste sous contrôle.

Le schéma de principe d'un système C4ISR est le suivant (figure 10.24) :

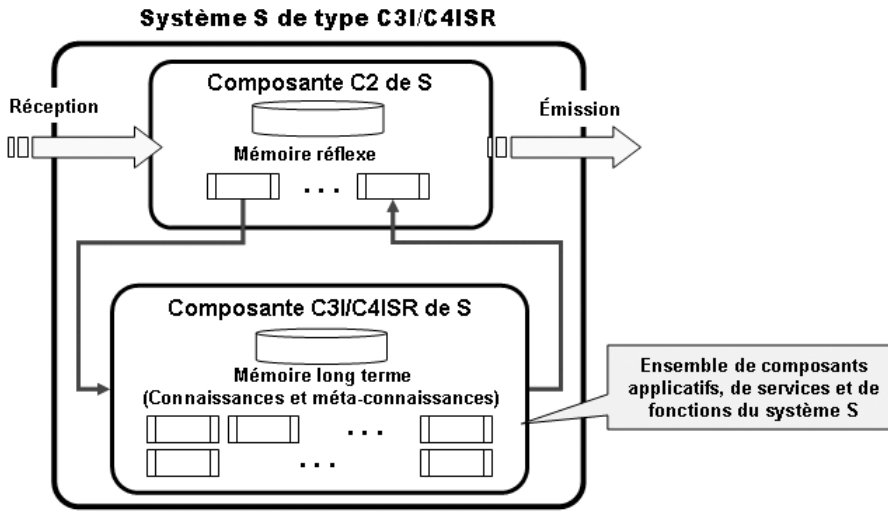


Figure 10.24 - Moniteur générique de type C4ISR

Ce qui est caractéristique des systèmes C3I/C4ISR est l'hétérogénéité à tous les niveaux :

- Hétérogénéité temporelle, qui mélange des messages à haute priorité avec des messages sans contraintes autres que celle de la cohérence logique (de type transactionnelle).
- Hétérogénéité des traitements, avec, à côté des tâches temps réel, des tâches de fond de type « batch » pouvant monopoliser beaucoup de ressource, mais avec un niveau de priorité bas.
- Hétérogénéité et variété des ressources.
- Hétérogénéité des messages, dont certains peuvent intégrer une grande quantité de données, et d'autres véhiculer des ordres à prendre en compte par le système.
- Hétérogénéité des caractéristiques qualité non fonctionnelles nécessitant la mise en œuvre de techniques d'ingénierie diversifiées, pour éviter la sur-spécification (impact CQFD fort) ou la sous- spécification (SLA pénalisé pour certaines fonctionnalités).
- Hétérogénéité des IHM (variété de profils utilisateurs) et sophistication des GUI (cf. les écrans de visualisation dans le contrôle aérien et/ou l'affichage des cours de bourses dans les systèmes de trading/salles de marché), avec prise en compte de l'erreur humaine et du stress.

Pour toutes ces raisons, l'architecture des systèmes C3I/C4ISR est extrêmement complexe, ainsi que les projets de développement correspondant qui peuvent mobiliser des centaines d'ingénieurs. Pour paraphraser von Neumann, on pourrait à bon droit parler d'« ultra haute complexité ».

Certains de ces systèmes doivent avoir des capacités de programmation pour des requêtes inopinées, non prévisibles à l'avance, ce qui nécessite de disposer d'outils de développement intégrés au système. D'un point de vue purement architectural, ceci nécessite la définition explicite du méta-modèle du système, qui sert de fondation aux outils de développement.

Le fonctionnement d'un système de type C3I/C4ISR nécessite beaucoup plus que deux processus, car à côté des processus gérant les capteurs et les actionneurs, il y a tous les processus correspondant aux composants applicatifs, avec au moins un processus par composant.

L'un de ces processus joue le rôle de dispatcher de commandes, (c'est le « *shell* » du système), les commandes étant les interfaces d'accès aux composants applicatifs. C'est le chef d'orchestre du système. Le pilote externe du système dispose d'un jeu de commandes riches, permettant aux utilisateurs d'effectuer toutes les actions autorisées par le système (NB : plusieurs centaines pour un grand système).

La structure d'un tel système est la suivante (figure 10.25):

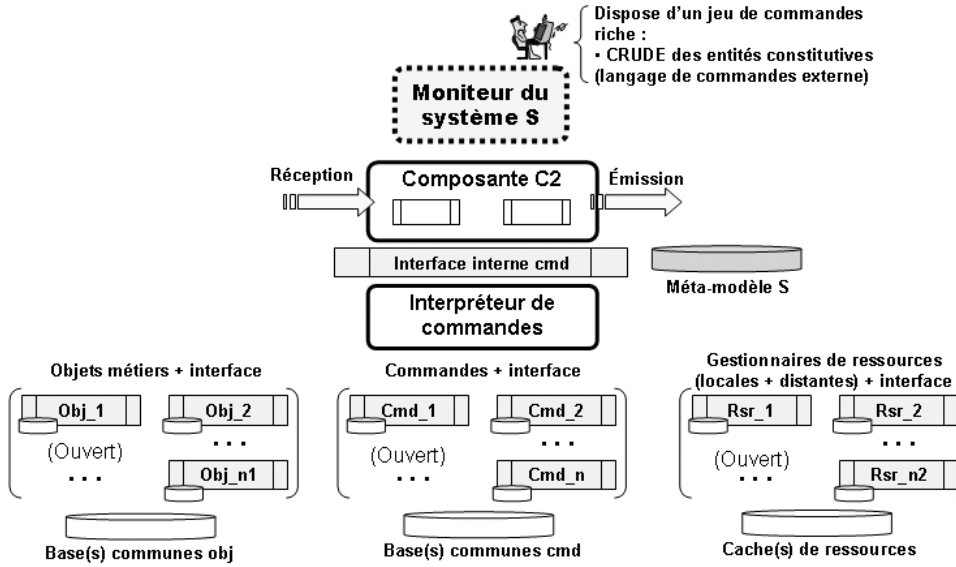


Figure 10.25 - Moniteur système de type C3I/C4ISR

La colonne vertébrale d'un tel système est l'interpréteur de commandes, donnant accès aux fonctions et objets métiers.

Ce système comporte trois types de processus : les processus actionneurs et capteurs, comme pour le C2, et un nouveau processus PR_interpréteur_commandes qui actionne les commandes mises à disposition des usagers.

Parmi les commandes essentielles, il y en a deux qui précisent les modes de croissance des capacités fonctionnelles du système.

Un premier mode de croissance consiste à ajouter une commande au système, de façon à étendre ses capacités fonctionnelles. Le mécanisme d'ajout de commandes matérialise la propriété d'ouverture du système, ce qui nécessite dès ce niveau d'avoir un méta-modèle des interfaces commandes qui doivent être standardisées. Ce mode de croissance peut être qualifié d'horizontal. L'interpréteur de commande est alimenté par une file d'attente de travaux à effectuer qui proviennent, soit des actions de l'opérateur, soit de fichiers contenant des enchaînements pré-programmés de commandes. Un ensemble de travaux cohérents pré-programmés constitue un « *workflow* » qui peut faire lui-même l'objet d'un pilotage ad hoc, avec des règles qui lui seraient spécifiques (cf. des langages de workflow comme le BPML – *Business process Modeling Language*).

Un second mode de croissance, que l'on pourrait qualifier de vertical, est la propriété d'autoréférence dont peut être doté ce type de système. Cette propriété correspond à la notion de sous-système qui a la même structure que le système père. Pour cela il faut que le méta-modèle du système auto-référant soit explicite, ce qui permet d'instancier un sous-système avec des adaptations compatibles avec ce que permet le

méta-modèle. Historiquement, le premier système à avoir développé cette capacité fut le système MULTICS dons nous avons déjà parlé. Les systèmes avec moniteur de machines virtuelles ont également cette capacité : c'est très commode pour partitionner les ressources, confiner certaines fonctions critiques pour la sécurité, administrer automatiquement tout ou partie du système via une « machine d'administration » comme dans le concept d'« autonomic computing » d'IBM. etc.

Au final, le système dispose de deux modalités d'extension, via son méta-modèle :

- `Create_commande` [arguments de paramétrage] ;
- `Create_sub-system` [arguments de paramétrage] ;

Exprimé en terme de langages, on peut dire que le premier mode correspond à l'ajout d'instructions et/ou de fonctions (i.e. « *built-in function* » dans un langage donné, ce qui est une opération assez simple, car en terme de méta-modèle, il suffit de standardiser les règles de passage des paramètres (par référence, par valeur, etc.) pour les aspects dynamiques et de rajouter les règles de grammaire et de traduction pour les aspects statiques (i.e. c'est la fonction méta-compilateur).

Le second mode correspond à l'intégration et à l'interopérabilité de différents langages au sein d'un même environnement de programmation, ce qui est une opération beaucoup plus complexe que la précédente, tant au niveau statique qu'au niveau dynamique. Le méta-modèle doit être une abstraction des différents langages susceptibles de cohabiter de façon à expliciter le noyau sémantique commun à tous. Le degré de standardisation des interfaces porté sur tous les aspects nécessaires:

- d'une part, à la construction (i.e. compile time) des entités exécutables par les éditeurs de liens statiques et dynamiques : format de données et règle de correspondance, format de la pile, format des unités de compilation, format des exceptions et des messages d'erreurs, etc.,
- d'autre part, à l'exécution cohérente des différentes entités, et à leurs interactions (sémaphores, boîtes aux lettres, divers drivers pour adapter la sémantique du langage à celle de la machine et du système d'exploitation sous-jacent, etc.).

De tels mécanismes sont à l'œuvre dans tous les systèmes d'exploitation modernes, depuis MULTICS qui les a inventés. L'un des systèmes où le concept a été poussé le plus loin est l'AS400 d'IBM, avec l'interface ILE (*Integrated Language Environment*).

Dans son aboutissement le plus parfait, le langage de commande est conçu comme un langage complet (au sens de Church), ce qui donne à l'usager d'un tel système un langage de programmation orienté métier qu'il peut façonner conformément à ses besoins, pour autant que l'administrateur du système ait bien compris tous ces mécanismes.

Comme indiqué précédemment, le système MULTICS a été le premier à utiliser à fond ce type de mécanisme d'extensibilité, dans un substrat PL1 qui avait été choisi comme langage pivot du système.

Parmi les systèmes les plus récents, c'est l'AS400 qui est à l'avant garde, et dans une certaine mesure, le MAC-OS des machines Apples, également fortement inspiré de MULTICS.

Dans le domaine des SGBD, les implémentations du modèle relationnel (ORACLE, IBM/DB2, SYBASE, etc.) ont également largement tiré parti de ces mécanismes.

NB : Dans les implémentations du modèle relationnel, il est de bon ton de représenter le méta-modèle du SGBD sous la forme d'un schéma relationnel, ce qui permet de manipuler le méta-modèle avec le même langage que le modèle (i.e. le DDL et le SQL) ce qui est d'une grande élégance et logiquement parfait. Le « prix » à payer est un traducteur complexe qui transforme ce langage déclaratif en langage de navigation (celui des B-tree), invisible à l'usager. Avec l'amélioration des performances des CPU (cf. loi de Moore) et des tailles mémoire, ce qui était encore un vrai problème dans la décennie 80, est devenu un avantage concurrentiel évident, qu'une société comme ORACLE avait complètement intégré dans sa stratégie dès sa fondation, en 1981.

L'auteur se souvient du premier partage d'ORACLE en environnement DPS7, qui avait nécessité le développement d'environ 30 KLS de PL1/GPL pour porter le méta-modèle d'ORACLE sur GCOS7, ce qui était une preuve absolue de la qualité du travail des architectes d'ORACLE, et permettait d'installer l'ensemble du code du noyau ORACLE (> 500 KLS) sur GCOS7.

Application au cas de systèmes avec mémoire « cache »

Le « cache » est un organe logique fondamental qui permet de maîtriser à la fois les performances et la complexité, et ce à tous les niveaux d'abstraction que l'on trouve dans nos systèmes les plus modernes.

Le premier « cache » a sans doute été celui de la machine de von Neumann, à l'IAS à Princeton, qui avait déjà un mécanisme de pipe-line (à quatre étages). Très rapidement, dès les années 70, c'est devenu également un mécanisme irremplaçable des SGBD et des SGF. Sans cache, la loi de Moore n'aurait qu'un intérêt très limité.

Le mécanisme de cache est un aménagement de la figure 10.24, comme suit (figure 10.26) :

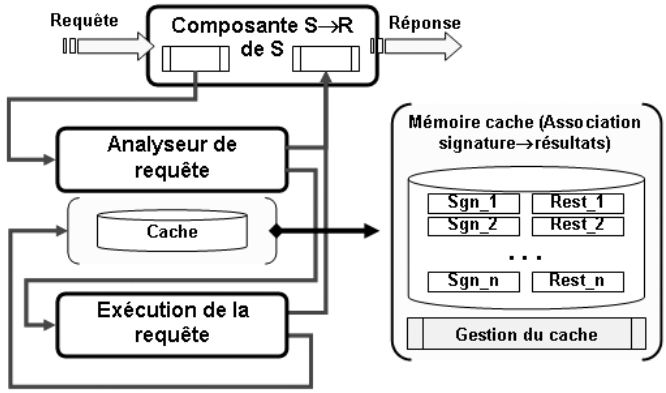


Figure 10.26 - Un exemple de C3I, le « cache »

L'effet cache est fondé sur la nature répétitive des traitements effectués par l'ordinateur. L'analyse statistique des comportements montre que des opérations, souvent complexes, sont effectuées des dizaines, voire des centaines de fois, dans des laps de temps très courts.

Le cache fonctionne comme une mémoire associative qui, à partir d'une signature, permet de retrouver rapidement (NB : la rapidité est impérative, faute de quoi le cache n'a aucun intérêt) le résultat associé à la signature. En cas d'entrées-sorties ou d'accès réseau, c'est particulièrement payant compte tenu de la lenteur relative des opérations d'E/S par rapport à la vitesse du CPU.

- Si la signature est présente dans le cache, il n'y a qu'à récupérer le résultat permettant d'élaborer la réponse.
- Si la signature est absente, le processus d'exécution continue jusqu'à son terme. En fin de travail le résultat est archivé dans le cache. Le cache est doté d'un algorithme de gestion du vieillissement qui permet la libération des entrées qui ne sont plus utilisées.

Comme effet indirect, on peut remarquer que le cache décharge complètement l'émetteur des requêtes du problème de l'optimisation des performances. Le mécanisme d'émission est beaucoup plus simple, donc moins coûteux à développer. La complexité est concentrée dans le cache, ce qui permet de mieux gérer la compétence des équipes. C'est un nouvel exemple de l'impact de l'architecture sur la maîtrise des coûts CQFD/FURPSE/PESTEL.

La nécessité d'un cache ayant été mise en évidence par les études d'architecture, son développement peut être différé, en fonction des comportements observés. Les caches peuvent être plus ou moins sophistiqués, mais ce sont toujours des algorithmes difficiles dont la maîtrise requiert une véritable compétence et beaucoup d'expérience.

11

Clients et serveurs
De la machine logique à une machine physique

Depuis son poste de travail, l'usager voit les composants applicatifs comme une « machine informationnelle » MI qu'il peut actionner via les commandes qui lui sont offertes pour effectuer différents types de tâches. La machine informationnelle doit être construite sur le concept de machine abstraite MA, indépendamment de la plate-forme qui interprètera les commandes, i.e. les « instructions » de la MA (dans le langage MDA, cette machine serait un PIM, *Platform Independent Model*) ; c'est la meilleure garantie d'évolutivité.

11.1 MACHINE INFORMATIONNELLE BASÉE SUR LE PATTERN MVC

Le concept de machine abstraite a été présenté au chapitre 1. Les machines abstraites vont permette de spécifier un univers logique où l'architecte peut rigoureusement filtrer ce qui entre et ce qui sort via les ports de la machine, ce que nous avons appelé une sphère de contrôle. Les commandes sont organisées en séparant la couche métier (programmation à partir d'objets métiers élémentaires) de la couche technique (cf. Figure 10.6).

Les MI ébauchées dans les schémas qui suivent ont une programmation conforme au pattern *Model View Controler* (MVC[1]) dont l'usage est relativement généralisé pour les applications clients-serveurs.

Le premier niveau de description fait apparaître trois types de serveur, chacun ayant un rôle et des caractéristiques bien spécifiques (capacité d'enchaîner des

1. Cf. le livre de E. Gamma et al, *Design patterns*, Addison Wesley ; le pattern MVC remonte au langage SMALLTALK. C'est, en fait, un pattern général de traduction.

actions, mémoire de travail, mémoire persistante durable quels que soient les aléas rencontrés) :

- Le serveur d'IHM. Il prend en compte la facilité d'emploi requise pour telle ou telle catégorie d'acteurs et les différents rôles qui leur sont attribués (authentification, droits, profil d'utilisation, etc.).
- Le serveur d'applications. Il prend en compte la logique métier, ce qui peut conduire à considérer différents serveurs chacun spécialisé sur un thème métier particulier.
- Le serveur de données (gestion de la persistance). Là encore, on peut les organiser en fonction de la nature des données traitées (données vivantes à évolution rapide, données réglementaires à évolution lente, etc.). Un serveur de données peut gérer une ou plusieurs bases de données.

La machine informationnelle gère toutes les ressources qui lui sont nécessaires. Elle peut héberger un ou plusieurs composants applicatifs et/ou systèmes d'information selon le besoin de confinement requis. C'est une entité homogène d'administration et d'allocation de ressource qui peut être placée sous la responsabilité d'un pilote qui arbitrera les allocations de ressources conformément au contrat de service (SLA) en fonction de la situation de saturation (capacity planning) et des aléas résultant de l'environnement des systèmes (pertes de ressources, reconfiguration, etc.). La machine administre toutes les ressources qui lui ont été attribuées via un configurateur dans lequel toutes les ressources disponibles sont décrites.

La structure de la MI MVC est la suivante (figure 11.1).

En environnement distribué, il faut faire apparaître les réseaux, et les aléas qu'ils introduisent de part leur existence. Du point de vue de la logique applicative cela se traduira par de l'interopérabilité entre composants applicatifs, à différents niveaux d'intégration. Le niveau de modélisation de la MI MVC, bien que fonctionnellement complet, est insuffisant car en aucun cas la ressource communication ne peut être considérée comme banalisée ; elle doit être explicitée. Utiliser les services de La Poste, de DHL, ou la valise diplomatique est fondamentalement différent du point de vue des caractéristiques non fonctionnelles.

Une première raison est un impératif de regroupement sémantique des serveurs logiques assurant une même fonction ou des fonctions voisines pour faciliter l'administration des serveurs.

Une seconde raison est de nature plus technologique en fonction du type de réseau et/ou de middleware d'intégration utilisé, avec la prise en compte de caractéristiques qualité fonctionnellement significatives pour le contrat de service (sécurité, fiabilité, performance, etc.).

D'un point de vue administration, il est indispensable de faire apparaître, a minima, deux niveaux d'administration :

Une administration de ressources attribuées en propre à une machine informationnelle MI. La MI administre ses ressources en optimisant sa capacité de survie,

comme pourrait le faire un moniteur de machines virtuelles dans les systèmes d'exploitation qui possède cette fonctionnalité[2].

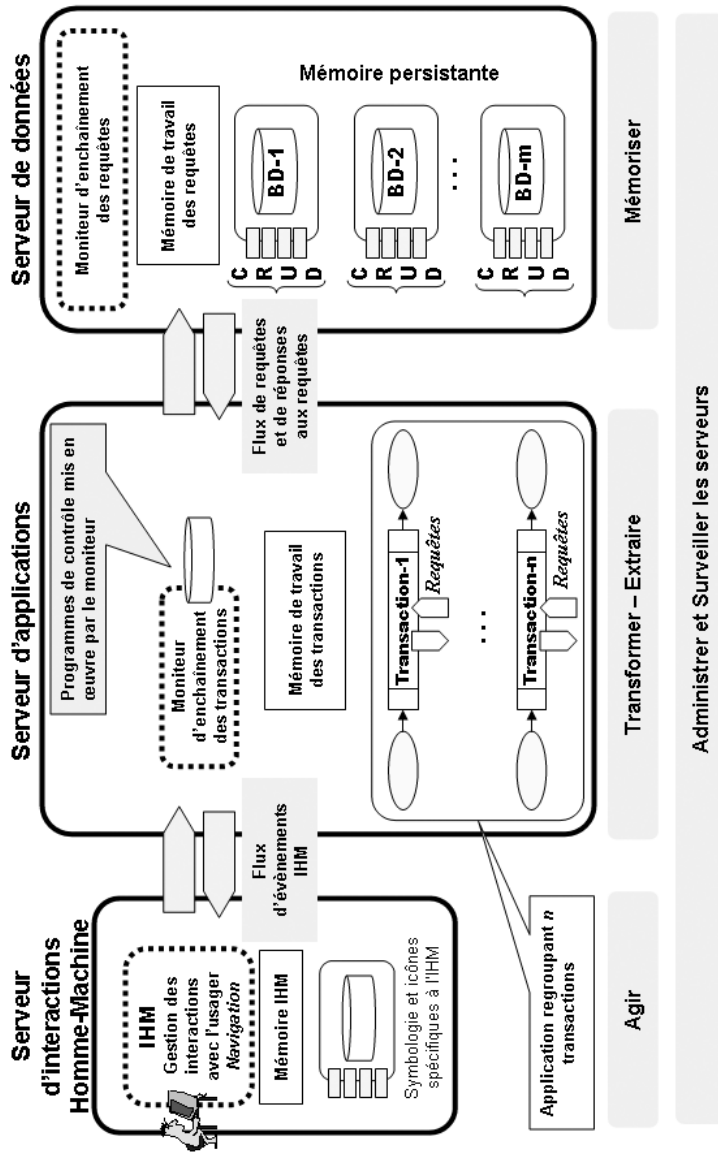


Figure 11.1 - Modèle de machine informationnelle MVC générique

2. Assez rare, et toujours en solution propriétaire ; voir par exemple chez IBM l'offre « *business on demand* » ex VM-CMS.

Une administration de ressources mise en commun, au profit des différentes MI qui inter-opèrent. Les ressources correspondantes sont gérées de façon ad hoc, à la demande, avec comme critère d'optimisation le maintien de la capacité d'interopérabilité des différentes MI.

-Dans le premier cas la programmation peut s'appuyer sur des mécanismes du type CORBA, OLE/DCOM, ou des plates-formes complètes comme J2EE, Websphere/Eclipse, .NET, etc. selon les adhérences admises pour tel ou tel fournisseur.

-Dans le second cas, il faut une infrastructure d'échange gérée comme telle, comme on en trouve dans le réseau de télé-compensation des banques, ou celui des billetteries des compagnies aériennes, etc. Une MI qui souhaite interopérer se connecte à un point d'accès au réseau commun, selon les modalités techniques de raccordement prévues.

L'infrastructure d'échanges entre les systèmes fédérés peut être elle-même vue comme une machine informationnelle à part entière, bien que spécialisée sur un domaine technologique lui-même très complexe, dont la mise en œuvre peut elle-même nécessiter plusieurs MA spécialisées.

11.2 MACHINE INFORMATIONNELLE MVC EN ARCHITECTURE DISTRIBUÉE

Il faut distinguer les serveurs locaux et les serveurs distants car les sémantiques d'appels et les couplages sont différents, comme indiquées dans le schéma 11.2 :

NB : les → indiquent quelques uns des chemins de données utilisés depuis les serveurs de données locaux et/ou distants, jusqu'au poste de travail de l'usager. Ceci est essentiel pour la traçabilité des différentes traductions effectuées tout au long de ces chemins. La mémoire de travail peut contenir des information locales et des informations qui sont des copies de données distantes (par exemple un cache).

En toute logique, aucune fonction critique de SA1 ne doit être accédée de façon distante car on ne peut jamais être sûr d'une disponibilité de 100% du réseau de communication. La répartition des ressources doit être faite selon les critères de disponibilité des différentes applications des différents systèmes.

Dans tous les cas de figure, il faut une gestion de configuration et une administration rigoureuse[3], basée sur un état précis des installations d'équipements matériels et logiciels sur les plates-formes (cf. figure 2.1), compte tenu de la nature des services demandés aux systèmes. Ceci permettra d'effectuer l'intégration dans des conditions optimales.

3. Voir l'approche *Autonomic Computig*, préconisée par IBM.

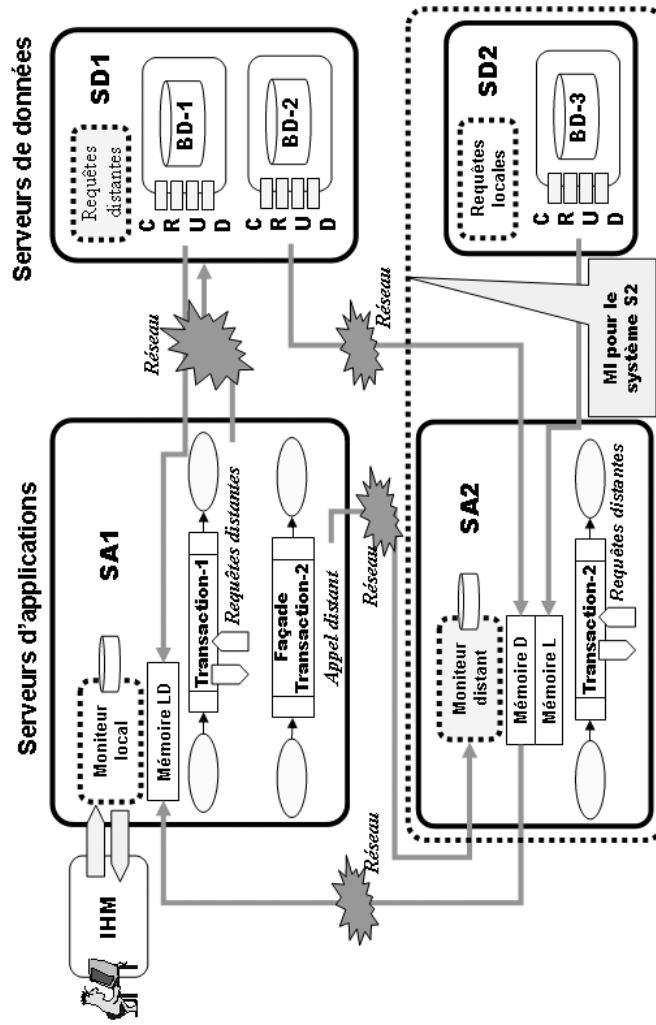


Figure 11.2 - Modèle de machine MVC généralisée – Actions locales et distantes

11.3 STRUCTURE DES ORGANES DE LA MACHINE INFORMATIONNELLE

Pour que la machine MI MVC soit complètement définie, il faut spécifier ses organes principaux : La structure de contrôle ; La structure actions-opérations (i.e. les services rendus par la machine) ; La nomenclature des types reconnus par la machine ; La structure de surveillance (invariants, règles d'intégrité, etc.)

Commençons par la structure actions opérations.

11.3.1 Structure Actions-Opérations

La structure Actions-Opérations est un type abstrait de données (TAD, ou ADT en anglais) au sens strict du terme. C'est un ensemble de services basiques (i.e. les « instructions » de la machine) complètement défini : tous les services sont identifiés, ainsi que leurs conditions d'appels (cf. la notion de contrat) ; la nature des types manipulés est connue, ainsi que l'état de l'exécution du service (l'aspect « PERFORM » de la sémantique du service, i.e. « ce que ça fait » réellement), i.e. FAIT, NON-FAIT, et toutes les modalités intermédiaires : c'est le statut de l'opération (i.e. le vecteur d'état ; cf. figure 8.12). Cette structure est le « jeu d'instructions » accessibles à chacun des systèmes constituant la fédération. Le jeu d'instructions définit ce que la machine sait faire (aspect fonctionnel) et comment elle va le faire (aspects non-fonctionnels et modalités de l'exécution).

La structure externe fonctionnelle de chacun de ces services peut être schématisée comme suit (figure 11.3) :

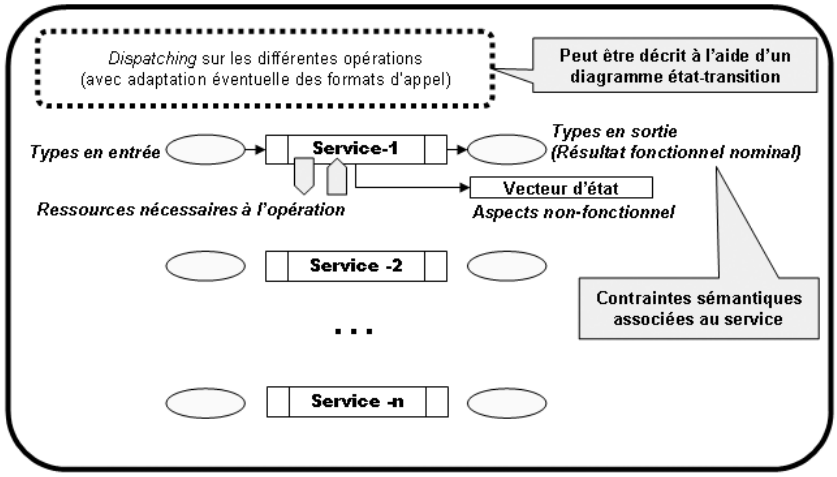


Figure 11.3 - Structure fonctionnelle des actions-opérations.

À ce niveau, on ne fait aucune hypothèse sur le déploiement des services ; ils peuvent être locaux ou distants selon les contraintes spécifiques à chacun des systèmes. Les ressources nécessaires peuvent être gérées de façon autonome par le service lui-même, ou fournies par l'appelant du service.

11.3.2 La nomenclature des types de données

La nomenclature répertorie tous les types de données qui peuvent être utilisés par la structure actions-opérations. Pour des raisons d'extensibilité de la structure action-opérations, il faut se donner la possibilité de travailler sur deux niveaux de typage :

Le niveau N0 constitue les types primitifs ; ils correspondent aux opérations atomiques de la machine abstraite correspondant à la granularité la plus fine de la sémantique qu'il faut modéliser. Sur ce niveau, la machine doit être déterministe.

Le niveau N1 qui est un niveau construit sur le niveau N0, avec par exemple comme objectif de définir la sémantique orientée métier que l'on souhaite définir en tant que service pour l'ensemble de la fédération.

D'où une construction à deux niveaux de la structure actions-opérations, et la mise en évidence d'un niveau de paramétrage de première importance. On trouve des structures de ce type à tous les niveaux d'abstraction des systèmes, par exemple dans toutes les UC avec la distinction machine/micro-machine, ou dans les convertisseurs d'adresses dans les caches[4] de la machine ou ceux des SGBD, ou dans la distinction entre composants métiers et composants techniques. Cette distinction est primordiale pour une mise en œuvre des recommandations de l'architecture MDA préconisée par l'OMG.

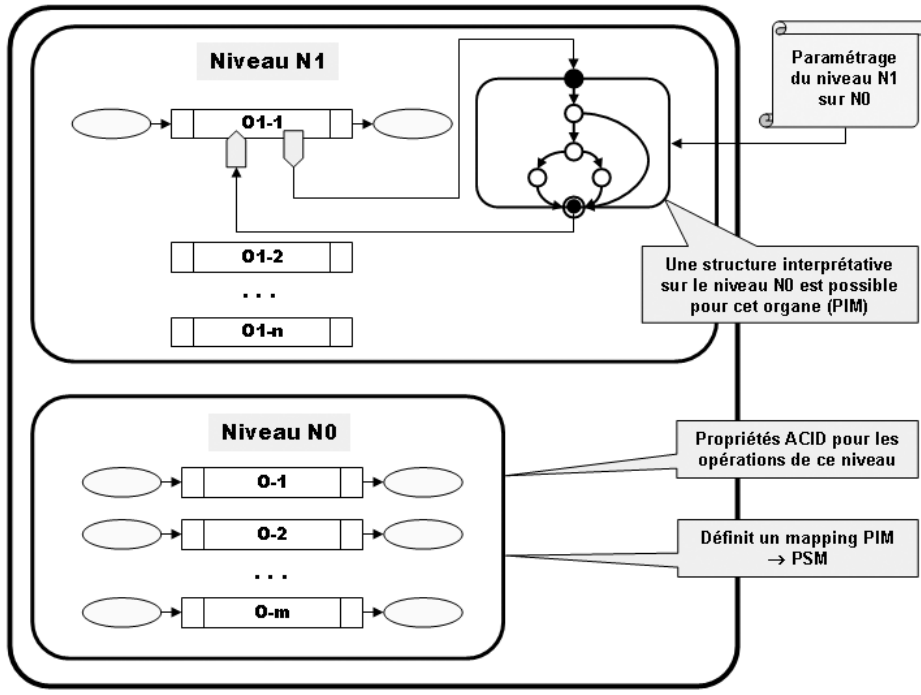


Figure 11.4 - Organisation de la structure actions-opérations.

D'un point de vue sémantique, le niveau N0 sera spécialisé sur la plate-forme sous-jacente. Ce niveau N0 pourrait implémenter la correspondance PIM vers PSM, d'où la nécessité absolue du déterminisme du niveau N0, sans lequel il n'y a pas de

4. Cf. J. Hennesy, D. Patterson, *dans la bibliographie*.

traduction automatique possible. Le niveau N1 implémente les objets métiers. C'est ce niveau qui doit être doté des facilités de paramétrage les plus puissantes (c'est-à-dire de programmation), car c'est lui qui permet l'adaptation aux évolutions de la réalité (figure 11.4).

Notons que la structure interprétative du niveau N0 sur N1 peut être décrite de façon concrète sous une forme déclarative (et non impérative) comme cela a été fait dans les SGBD entre les niveaux conceptuels et logiques dans les années 70-80s, puis dans les langages de 4ème génération au milieu des années 80s.

À la fin d'une opération, qu'elle soit de niveau N0, ou de niveau N1, l'opération restitue un résultat et un vecteur d'état précisant les modalités d'obtention de ce résultat.

La standardisation du vecteur d'état est une nécessité absolue si l'on veut spécifier rigoureusement la sémantique comportementale des opérations effectuées car elle va déclencher l'appel de telle ou telle fonction de surveillance et/ou d'intégrité.

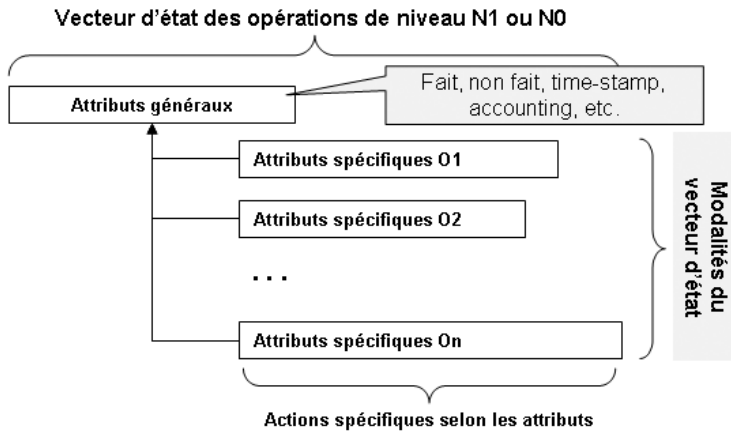


Figure 11.5 - Standardisation du vecteur d'état.

Parmi les propriétés générales véhiculées par le vecteur d'état on aura : l'état {FAIT, NON FAIT}, la date universelle (TIME-STAMP, ou estampille temporelle) et la durée de l'opération, les performances, les anomalies rencontrées (conflits d'accès, etc.), le nombre d'opérations intermédiaires faites au niveau N0, etc.

Parmi les attributs spécifiques, on aura : la liste détaillée de toutes les opérations intermédiaires (identification dans l'historique de l'exécution du service), le détail des modalités de l'échec si on est dans l'état NON-FAIT, les ressources inaccessibles, opération interrompue et/ou suspendue, etc.

Tout ou partie du vecteur d'état est transmis à la structure de contrôle pour exploitation.

11.3.3 La structure de contrôle.

La structure de contrôle de la machine abstraite d'interopérabilité a un triple rôle :

1. Lancer l'exécution requise par le programme de contrôle qui a été chargé dans la mémoire de la structure de contrôle.
2. Recueillir les résultats des opérations effectuées et transmettre ces résultats à l'environnement appelant.
3. Analyser le vecteur d'état et les messages de contrôle ayant pu être transmis à la machine pendant la durée de l'opération de façon à sélectionner la prochaine opération à effectuer.

À tout moment, la structure de contrôle doit pouvoir mettre fin aux opérations en cours, si des évènements prioritaires qui le nécessitent sont survenus. Dans ce cas la machine doit être arrêtée dans un état cohérent qui ne met pas en cause son intégrité

La structure simplifiée d'un cycle de contrôle peut être schématisée comme suit (figure 11.6) :

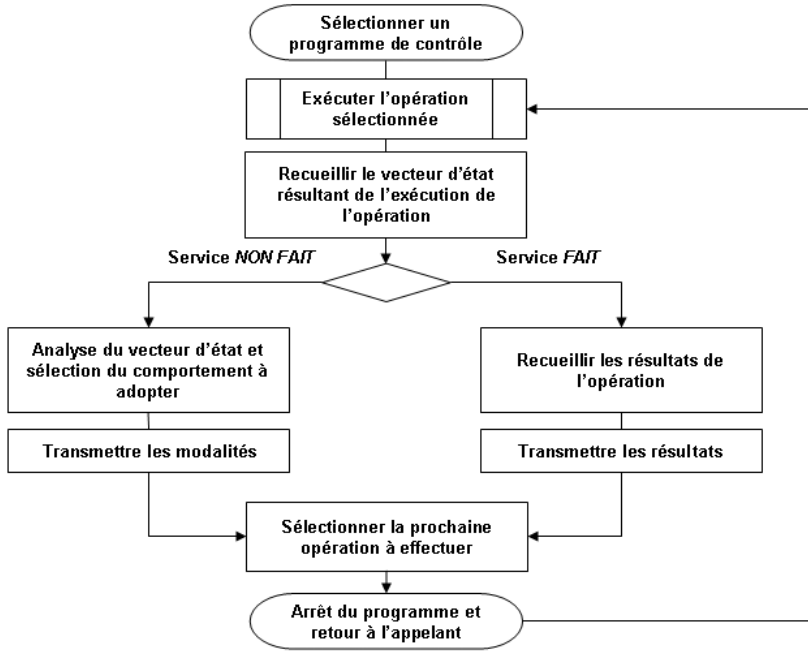


Figure 11.6 - Structure d'un cycle de contrôle.

Il faut prendre en compte le fait que le service effectué est soit local, soit distant car la sémantique du contrôle sera différente dans chaque cas.

11.3.4 La structure de surveillance.

Cette structure gère tout ce qui est susceptible d'altérer le bon déroulement des opérations. Elle construit le vecteur d'état. En fonction des observations effectuées sur les différents organes de la machine, y compris sur elle-même, elle peut notifier à la structure de contrôle d'interrompre ou suspendre les opérations en cours.

En cas de panne générale de la machine abstraite, elle doit disposer d'un canal de secours permettant de notifier à l'environnement que la machine n'est plus en état de rendre les services attendus ; ce canal doit être distinct du canal nominal pour des raisons de sûreté.

La structure de surveillance gère l'ensemble des tests permettant de s'assurer que toutes les opérations sont vérifiées et validées, ainsi que toutes les sondes nécessaires à l'observation des organes. Le dispositif d'observation est lui-même paramétrable en fonction du niveau de maturité de la machine (cf. chapitre 12 et 13).

PARTIE 4

Propriétés d'une bonne architecture

12

Simplicité – Complexité

12.1 FONDEMENTS DES MESURES DE COMPLEXITÉ TEXTUELLE

12.1.1 Complexité et/ou complication

Le terme simplicité est généralement préféré au terme complexité, devenu au fil des ans une auberge espagnole sociologique où l'on range tout ce qui nous cause problème ou nous paraît incompréhensible, et ce dans une grande confusion. La complexité est partout, c'est-à-dire nulle part, tandis que la simplicité est perçue comme un attribut positif. Le mot complexité déplaît ; il fait irrémédiablement penser à « usine à gaz ».

L'un des douze principes agiles définit la simplicité comme l'art de maximiser le travail à ne pas faire (principe N° 10). Les Anglo-Saxons, qui aiment les acronymes, parlent du principe KISS, « *Keep it simple, Stupid* » (i.e. Garder cela simple, bon sang !). C'est incantatoire, mais pas vraiment constructif.

Le problème n'est pas de donner un nom à la chose (cf. le rasoir d'Ockham), mais de bien comprendre ce que cela veut dire, et surtout, pour des ingénieurs, de savoir comment s'y prendre pour simplifier, où pour éviter de créer de la complexité inutile, donc nuisible et coûteuse, tout en sachant que nous avons une formidable propension à créer de la complexité, du désordre, etc. et que c'est même une évolution inéluctable de l'univers si l'on en croit le second principe de la thermodynamique. Pourquoi avoir peur de ce qui est notre destinée !

Au sens littéral, est **complexe**, ce qui est composé d'entités sémantiquement distinctes, reliées entre elles, ce qui nous amène à nous intéresser : a) au nombre d'entités et à leur variété, c'est-à-dire le nombre de types perçus comme distincts (i.e. les abstractions fonctionnelles), et b) à ce qui les relient, c'est-à-dire les relations de toute nature qu'elles entretiennent entre elles.

Selon cette définition, la complexité est un être trinitaire, un Janus à trois visages :

- le nombre de **types d'entités**, i.e. la variété de l'ensemble des entités (cf. le débat des architectures CISC et RISC pour les CPU),
- le nombre d'**entités**, dans chacun des types, et leur nombre total d'occurrences, par exemple le nombre de « pas » de calcul nécessaires pour effectuer une transformation de l'état mémoire,
- le nombre de **relations** que les types et les occurrences entretiennent entre eux.

Il suffit de compter, mais c'est là que les choses se compliquent, car arrive immédiatement la question du quoi et du comment, suivi de celle du sens : qu'est ce que le ou les nombres obtenus veulent dire dans le contexte des projets que nous réalisons ? C'est là qu'il faut être simple et drastique.

Tout projet est défini par le quadruplet CQFD qui donne les caractéristiques économiques du projet. Tout projet produit (i.e. délivre, en anglais « release ») quatre types de texte de nature très différente :

1. Le texte « **programme** », c'est-à-dire tout ce qui est destiné à être exécuté sur les plates-formes d'exploitation, y compris les scripts de paramétrage. Ce texte existe sous deux formes : une forme source, compréhensible par le programmeur, une forme binaire, ou exécutable, comprise par la machine.
2. Le texte « **tests** du programme », c'est-à-dire tout ce qui permet de vérifier et de valider que ce qui a été programmé est correct, et permet d'installer le programme conformément au contrat de service négocié avec les utilisateurs ou leur représentant, maître d'ouvrage. Ce texte matérialise l'assurance qualité, i.e. le **Q** ce CQFD.
3. Le texte « **documentation** du programme », c'est-à-dire tout ce qui permet aux acteurs ayant affaire avec le programme de s'en servir correctement (usagers, support, maintenance, exploitation). Le volume de ce texte, les menus, les bulles d'aides en lignes, etc. dépend de la variété des acteurs, surtout usagers, et de leur compétence moyenne. Un système grand public comme des bornes de réservation de billets de trains nécessitera un volume de documentation important accessible en ligne, alors qu'un système de contrôle aérien, avec des techniciens très bien formés se contentera de l'essentiel, dans un manuel de référence.
4. Le texte « **référentiel** d'ingénierie », c'est-à-dire tout ce qui permet aux acteurs d'effectuer correctement les travaux pour construire, exploiter, maintenir le programme en état de marche tout au long de la vie du système dans lequel il est intégré, y compris la phase de retrait. C'est un méta-texte qui fixe les règles grammaticales que tous les acteurs ingénierie doivent suivre pour « parler la même langue ». Le référentiel peut contenir des programmes non livrés mais indispensables aux équipes d'ingénierie.

D'une façon ou d'une autre, toute action de simplification doit se matérialiser en terme de CQFD et dans les différents textes.

Faire simple, c'est minimiser chacun de ces textes dont on sait qu'ils ont des coûts spécifiques directs ou indirects (cas d'une mauvaise documentation du point de vue de l'usager) mais avec des pondérations bien différentes. On se doute que le support textuel, i.e. le ou les langages utilisés pour matérialiser tous ces textes, et la compétence des acteurs concernés à les manier correctement va jouer un rôle important quant à la taille de ces différents textes. Ceci nous permet de raffiner le sens du mot complexité et de distinguer ce que l'on va appeler complication.

Un même programme peut être écrit en assembleur ou en C++, en JAVA, en C#. Les deux programmes ont la même complexité, du point de vue de l'utilisateur ; ils font exactement la même chose. Chacun s'accordera à dire que le texte assembleur est plus compliqué que le texte Java ; et très concrètement, il sera en moyenne 5 à 6 fois plus volumineux en nombre de lignes sources. Notons que le volume de tests devrait être identique dans les deux cas, car le compilateur Java fabrique, in fine, un texte en assembleur qui sera testé, selon l'adage « *What you prove is what you execute* » car c'est ce qui s'exécute qui est la référence ultime.

Ceci nous permet de dire que :

- La complexité est intrinsèque au problème, et ne dépend que de l'expression de besoin et des exigences à satisfaire ; dans notre terminologie, c'est FURPSE + PESTEL. C'est ce que captent, grossièrement, les mesures fonctionnelles en Points de Fonctions.
- La complication dépend des langages utilisés, et de la compétence des acteurs à bien les utiliser. C'est ce que captent, en plus des aspects fonctionnels, les mesures en volume de code dans des modèles d'estimation comme COCOMO.

D'où une première conséquence :

Simplifier un problème n'a rien à voir avec choisir un langage, ou choisir une technologie. Ceci étant, pour matérialiser et décrire le programme, nous devons choisir a) des technologies et/ou des langages associés, et b) des personnels plus ou moins compétents et expérimentés.

D'où une deuxième conséquence :

Pour simplifier l'expression du problème, a-t-on choisi la bonne technologie et/ou a-t-on choisi les bons acteurs pour l'ingénierie, en étant assez lucide pour savoir qu'une technologie, aussi bonne soit-elle, ne sera jamais un remède à l'incompétence.

C'est la raison pour laquelle, il fut décidé dès la fin des années 60, sous l'impulsion du MIT, de réaliser les systèmes d'exploitation en langage de haut niveau, 3 à 4 fois plus compacts que l'assembleur, et de n'embaucher comme programmeurs que des ingénieurs, avec les compléments de formation[1] ad hoc pour monter leur niveau de compétence.

Aujourd'hui, on ne peut que constater les ravages occasionnés par ce contre sens absolu consistant à dire : choisissez la technologie objet, et le reste vous sera donné de surcroît. Idem pour le relationnel, avec les bases de données. Avec une telle approche, il ne faut pas s'étonner des difficultés rencontrées dans les projets.

Pour la partie programme, l'ensemble **complexité** (FURPSE + PESTEL) + **complication** (technologies + maturité des acteurs ingénierie) va conduire à un texte dont on pourra compter le nombre d'instructions ; c'est le fondement des modèles d'estimation, que nous avons traité en détail dans d'autres ouvrages (cf. *Productivité des programmeurs*, et *Coût et durée des projets informatiques*, voir la bibliographie).

Le cas des compilateurs, au chapitre 6, est un bon exemple du rapport entre complexité du langage (règles de grammaire, sémantique, etc.) et complication du compilateur. Le nombre d'instructions source du programme est clairement un indicateur de complexité/complication conforme à la définition. En est-il une mesure ? La réponse est non, car nous n'avons pas considéré les relations explicites et/ou implicites présentes, dans le texte programme. Autrement dit, la taille du texte source ne définit pas une relation d'ordre (un ordre total ou partiel) par rapport à la complexité/complication que l'on cherche à caractériser.

Pour améliorer l'indicateur, il faut considérer la partie « Tests du programme » et analyser les relations qui peuvent exister entre la taille du programme et le volume de tests nécessaire à sa validation /vérification. La question est : comment varie le volume de tests par rapport à la taille du programme ? sachant que le coût de chacun de ces textes est intuitivement une fonction monotone croissante de sa taille.

Existe-t-il des situations où une diminution de la taille du programme se traduirait par une augmentation du volume de tests ? ce qui ruinerait définitivement la taille du programme comme mesure de la complexité.

On sait depuis toujours que tout programme peut être représenté à l'aide d'un graphe de contrôle qui matérialise tous les chemins possibles dans le programme. Le nombre de chemins indépendants est caractérisé par le nombre cyclomatique associé au graphe, lui-même dépendant du nombre d'instructions de contrôle du type `IF…THEN…ELSE`, `GOTO`, etc. présentes dans le programme (cf. la littérature sur les tests pour plus de détails : B. Beizer, et R. Binder, *dans la bibliographie*).

Il doit être clair pour tout le monde que le graphe de contrôle qui résulte directement de la programmation (et de la compétence des programmeurs) est une approximation, somme toute grossière, et un minorant, des capacités fonctionnelles du programme du point de vue des acteurs métiers. Le graphe de contrôle est la traduction du comment cela a été construit du point de vue de l'équipe d'ingénierie et de ses limitations (maturité de l'équipe, expérience de l'architecte). Certaines contraintes et/ou usages métiers peuvent ne pas y figurer. Pour compenser ces limita-

1. Les premier cursus informatiques sont tout à fait intéressants. Les fondements de la discipline y tenait, toutes choses égales par ailleurs, une place beaucoup plus importante qu'aujourd'hui. Avec les tests, dans les bons ouvrages, il est indispensable de « rafraîchir » ses connaissances en théorie des graphes, en logique, … juste retour du boomerang.

tions, il faut élaborer des tests qui résultent de la logique métier (tests de validation, i.e. boîte noire) matérialisée par des cas d'emplois.

L'ensemble programme + tests peut-être représenté comme suit (figure 12.1) :

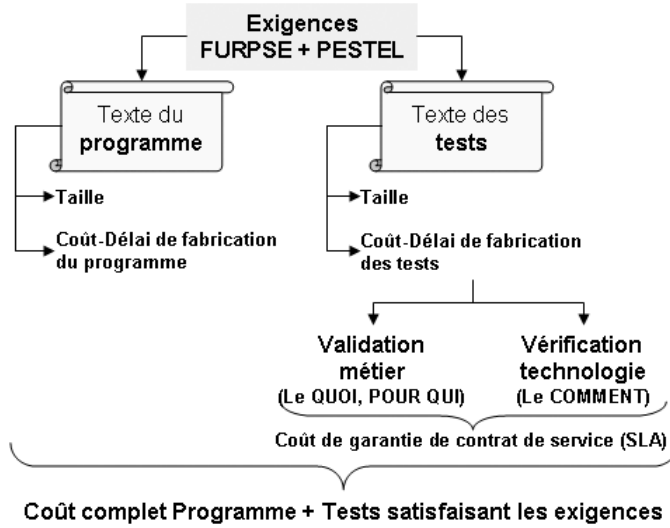


Figure 12.1 - Indicateur de complexité programme/tests

La combinatoire des tests de validation métier résulte des exigences métiers que l'on peut matérialiser à l'aide de scénarios ou de diagrammes de cas d'emplois (cf. les « use cases » en UML). Cette combinatoire est exponentielle comme le montre la figure 12.2 :

L'exponentielle provient de la cardinalité de l'ensemble des flèches qui matérialisent la correspondance entre ce que permettent la spécification et les résultats attendus. La partie robustesse concerne les cas d'emploi correspondant à des utilisations erronées qui doivent se traduire par des messages d'erreur. Aucun cas erroné ne doit donner des résultats jugés corrects, et inversement. La cardinalité de l'espace fonctionnel correspondant est une formule classique de la théorie des ensembles : $\boldsymbol{R^E}$ dans laquelle E est la cardinalité des entrées possibles (chaque paramètre d'entrée est une possibilité, ce qui en ne considérant que le cas VRAI/FAUX conduit à $\mathbf{2^N}$ possibilités de tests) et R est la cardinalité des résultats (soit, en ne considérant que VRAI/FAUX : $(\boldsymbol{2})^{2^N}$ scénarios possibles).

L'intérêt de ces tests est qu'ils sont indépendants de toute implémentation ; ils ne dépendent que des interfaces. Leur inconvénient est que leur nombre est exponentiel. Toute défaillance constatée en exploitation doit donner naissance à un test de validation correspondant, ce qui fait que la suite de tests de validation s'enrichit progressivement tout au long de la vie du système ; il est fondamental de ne pas perdre cette information.

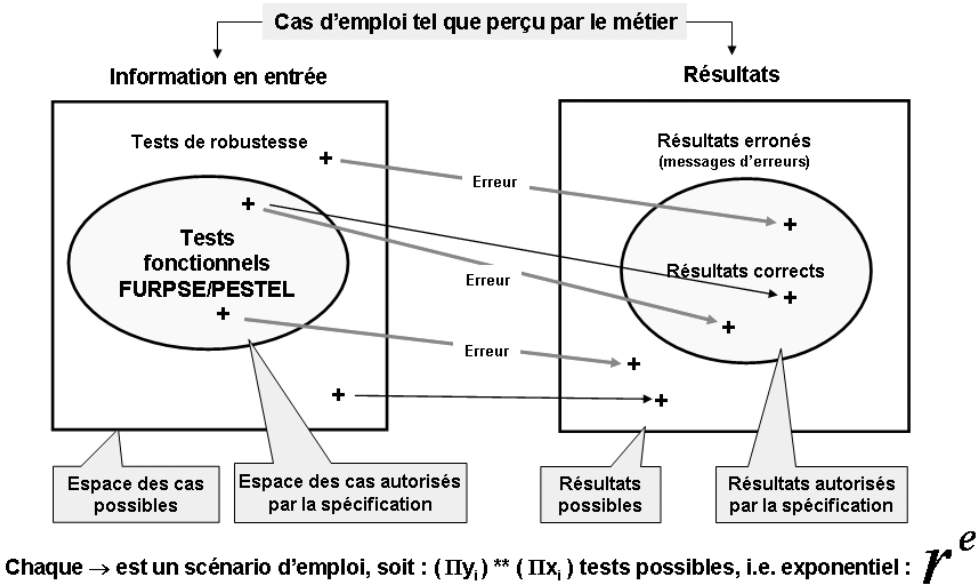


Figure 12.2 - Combinatoire validation métier boîte noire

La combinatoire des tests de vérification, fait par l'ingénierie, dépendant du graphe de contrôle, est polynomiale, mais le degré du polynôme peut être élevé, en particulier dans le cas d'une programmation événementielle.

Ce style de programmation a toujours suscité les plus grandes réserves de la part des architectes système compte tenu des problèmes de vérification induits : ils n'y ont eu recours que contraint et forcé, quand il était impossible de faire autrement. La programmation par événements implique de dresser une nomenclature très précise :

- Des instructions qui sont susceptibles de lever l'événement considéré.
- Du/des drivers susceptibles de réceptionner l'événement et d'effectuer la contre-mesure correspondante, faute de quoi l'événement se perd quelque part dans le système.

La nomenclature des événements devient un élément critique du référentiel. Inexistante dans une architecture centralisée, elle est obligatoire en architecture clients/serveurs distribuée.

On « voit » sur le schéma que la factorisation des traitements, lorsque les contre-mesures sont correctement effectuées, est optimale mais que cela a un coût matérialisé par le graphe de contrôle, avec les arcs implicites, qu'il faudra cependant tester. Toute instruction, ou séquence d'instructions, susceptible de lever l'événement sera déroutée vers un driver (c'est une composante connexe du graphe de contrôle, sans relation explicite) qui effectuera la contre-mesure.

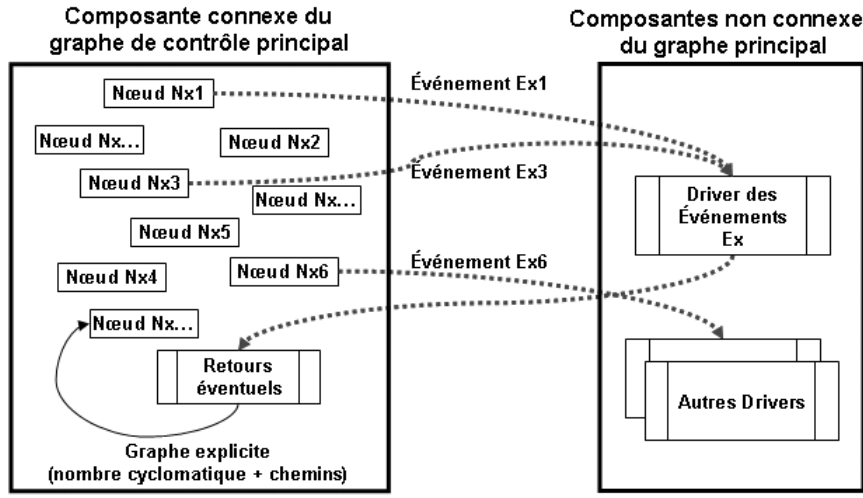


Figure 12.3 - Graphe de contrôle événementiel

Outre le problème de performance, puisqu'il y a passage obligé par le système d'exploitation (ce qui n'est peut-être plus un problème, tant qu'il n'y a pas d'entrées-sorties, avec les processeurs actuels), il faut que le driver appelé identifie précisément l'origine de l'événement pour différencier ce qui est une réparation factorisée, de ce qui est une vraie défaillance relevant d'une autre logique de réparation.

Dans ce cas, le graphe de contrôle explicite, via les instructions de contrôle, se voit doubler d'un graphe implicite résultant de la logique événementielle. On voit, sur la figure, que ce graphe est beaucoup plus complexe car les points de levée d'un événement, ainsi que les points de retour, rajoutent des arcs au graphe ce qui fait augmenter d'autant le nombre cyclomatique et créé des chemins supplémentaires qu'il faudra vérifier et valider (par exemple si la transition donne naissance à un message d'anomalie).

NB : Rappelons la formule qui donne le nombre cyclomatique d'un graphe connexe, soit : N_{cyclo}=*Nombre_d'arcs–Nombre_de_nœuds+2*. Ce nombre ne reflète pas la complexité dynamique liée aux différentes façons de parcourir le graphe.

Au final, le volume de programmation aura peut-être baissé, mais le volume de tests aura fortement augmenté. Intuitivement, on sent bien qu'un indicateur qui combinerait la taille du programme et le volume de tests serait plus satisfaisant que la taille du programme seule.

La difficulté à effectuer ce comptage vient des tests eux-mêmes. Hors certains domaines spécifiques comme les télécommunications (langage TTCN, associé à SDL) ou l'avionique (langage ATLAS, normalisé par l'IEEE), il n'y a pas de langage de tests généraux. Il y a bien eu des efforts faits dans ce sens, mais cela n'a jamais

débouché au plan industriel, car le langage de test finissait par ressembler étrangement au langage du programme à tester. Il est facile de comprendre pourquoi, comme le montre le schéma 12.4.

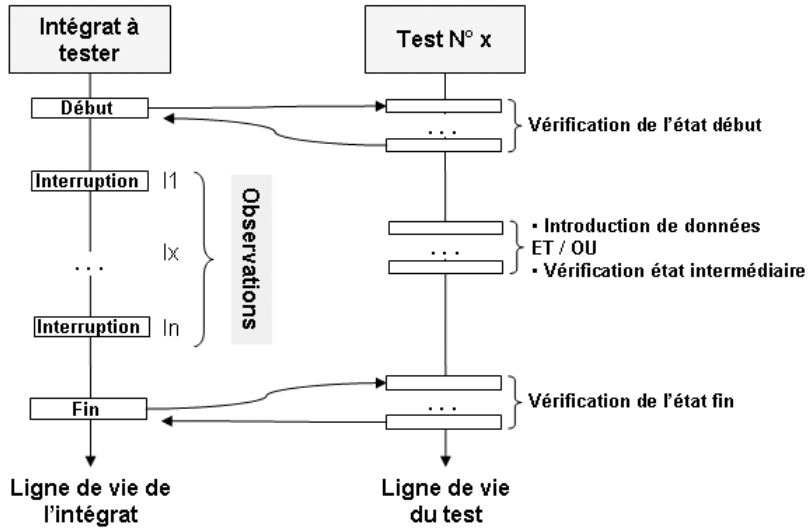


Figure 12.4 - Relation entre programme et test

Dans ce schéma, le test apparaît comme une forme duale de l'intégrat à tester. Le test appartient à la représentation en extension de la transformation effectuée, car un cas de test n'est jamais qu'un cas d'emploi de l'intégrat qui valide la transformation. On pourrait même dire que c'est un élément de sa démonstration[2].

12.1.2 Indicateur de complexité/complication

On « voit » sur le schéma que le nombre de tests et le volume des tests dépendent de la dynamique du programme. Un « petit » programme à forte dynamique pourra engendrer un volume de tests important. L'injection et/ou la vérification peuvent être soit impératives, lorsque l'instruction ou le bloc programme N° Ix est exécuté, soit conditionnelles, si la survenue d'un événement est constatée.

Compte tenu de la nature répétitive des tests, l'architecte recommandera (via le guide de programmation du projet qui est un élément fondamental du référentiel) aux programmeurs de les incorporer au texte programme lui-même ce qui, en toute logique, fait partie du test. On dit alors que le programme est **instrumenté**, i.e. prêt à être testé.

2. Cf. les textes éclairant de J-Y. Girard, *Constructivité : vers une dualité moniste* et *La logique comme géométrie du cognitif*.

Le schéma permet également de bien comprendre la différence entre les tests boîtes noires (on ne connaît que les interfaces) et les tests boîtes blanches (on connaît la structure interne).

En « boîtes noires », le testeur ne peut intervenir qu'à l'entrée et à la sortie du module/intégrat, ce qui implique que les intégrats appliquent rigoureusement les règles de modularité rappelées au chapitre 8. Si ces règles ne sont pas respectées, le test ne sera que partiel.

En synthèse, moyennant certaines précautions liées aux bonnes pratiques architecturales (modularité, entre autres) et aux bonnes pratiques de programmation (instrumentation du programme), on voit que l'on peut élaborer un indicateur qui fonctionne presque comme une mesure.

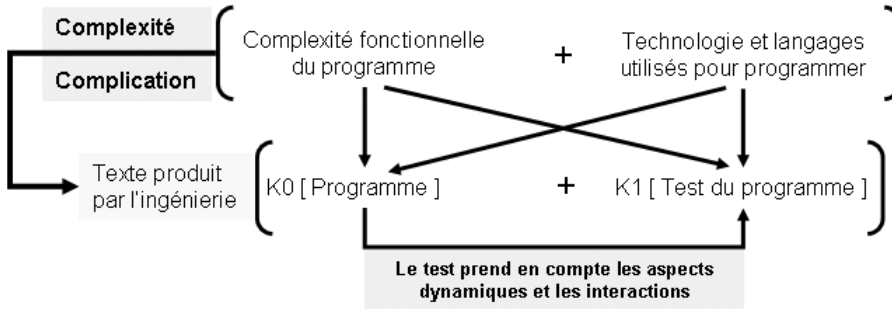


Figure 12.5 - Indicateur de complexité-complication

La combinatoire engendrée par la dynamique du programme et de ses interactions avec l'environnement se traduit par une augmentation du volume de tests à produire.

K0 et K1 sont des coefficients de normalisation. Si on adopte la logique des modèles d'estimation comme COCOMO ou les points de fonctions, tout est ramené à la taille fonctionnelle ou aux lignes de code source, K0 vaut 1 par convention. Si on considère que l'activité de test est une forme de programmation, dont le volume dépend, in fine, des caractéristiques dynamiques du système, on peut, à bon droit, penser que l'indicateur qui nous intéresse est une certaine fonction *F* du programme lui-même, soit, en la développant en série :

$$ICC(ProgrammeP)=C0[Prog]+C1[Prog]^{\alpha 1}+C1[Prog]^{\alpha 2}+K$$

ou encore :

$$ICC(ProgrammeP)=C0[Prog]\times\left[1+\frac{C1[Prog]^{\alpha 1}}{C0[Prog]}+\frac{C1[Prog]^{\alpha 2}}{C0[Prog]}\right]+K$$

NB : c'est la même logique que celle donnant la forme de l'équation d'effort du modèle COCOMO (cf. l'annexe 1 de *Productivité des programmeurs*, déjà cité). Le second terme caractérise l'influence des tests.]

Pour valider la forme de l'indicateur, appliquons-le à des situations d'interopérabilité et/ou d'intégration comme celles décrites aux chapitres 2 et 9. L'intégration de plusieurs intégrats pour former un intégrat de rang supérieur ne change pas le volume de programmation. On a :

Taille (M_{20}) = Taille (M_{11}) + Taille (M_{12}) + … + Taille (M_{1n}).

Par contre le volume de tests va nécessairement augmenter, car compte tenu de la rétroaction du programme sur les tests, il va falloir produire des tests pour vérifier/valider a) les combinaisons 2 à 2, 3 à 3,…, n à n des intégrats M_{11}, M_{12}, …, M_{1n} (ce qui peut ramener une exponentielle, car la somme de ces combinaison est $\mathbf{2^n}$) et b) les nouveaux chemins résultants des interactions. L'augmentation du volume de tests est directement corrélée à la combinatoire qui existe entre les modules, laquelle dépend des modes opératoires du système et de la sémantique des interactions autorisées (appelées chaînes de liaisons dans le langage des métiers).

Comme on l'a vu aux chapitres 2 et 9, les intégrats sont associés aux activités des processus des chaînes de valeur métier du monde M1 (cf. figures 8.1, 8.2 et 8.3). La combinatoire des intégrats dépendra des workflows autorisés par les métiers. Le volume de tests dépendra de la façon dont l'architecture a géré cette combinatoire. Dans le cas d'une architecture en couche et/ou pivot, la situation sera la suivante (figure 12.6) :

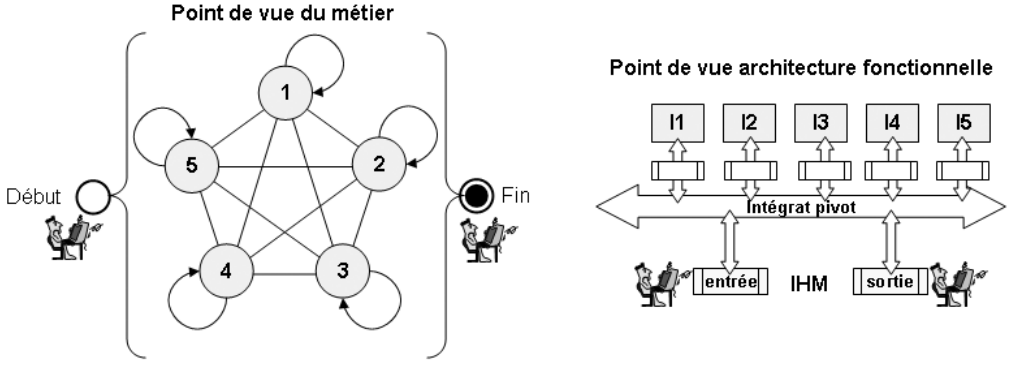


Figure 12.6 - Combinatoire métier versus combinatoire fonctionnelle.

Côté métier, la combinatoire est a minima en $\mathbf{O(n^2)}$ alors que du côté architecture fonctionnelle la combinatoire est linéaire, nonobstant le pivot. Du point de vue statique, la combinatoire métier a été cassée. Qu'en est-il de la combinatoire dynamique ?

Si les intégrats I1, …, I5 n'ont aucune dépendance fonctionnelle les uns avec les autres autrement que par les interfaces d'accès en entrée et sortie des intégrats (strict respect du principe de modularité), la combinatoire dynamique est équivalente à la combinatoire statique. Le terme combinatoire de notre indicateur ICC dégénère en un terme linéaire qui prend en compte les tests de l'intégrat pivot et les tests des adaptateurs interfaces (entrée + sortie) d'accès au bus pivot.

Cette propriété fondamentale n'est pas donnée comme par magie. Elle résulte du travail effectué par l'architecte et de la vérification par celui-ci (via les équipes qualité) que toutes les règles architecturales ont été effectivement respectées. Par rapport à la figure 8.9 cela exige qu'il n'y ait ni contexte commun aux intégrats, ni ressources partagées. En d'autres termes, que l'intégrat se comporte vis-à-vis de l'environnement comme une transaction (en particulier le **I** de ACID).

On voit ici le rôle fondamental du concept de transaction pour ce qui concerne la testabilité du système. Si ce n'est pas le cas, il faudra considérer l'ensemble des parties ; le volume de tests pourra devenir exponentiel par rapport à la taille du programme.

D'un point de vue qualitatif, la modélisation des processus métiers (i.e. les modèles métiers) permet de se faire une première idée de la combinatoire à laquelle l'intégration système va être confrontée, et ceci bien avant que la moindre ligne de code n'ait été programmée.

On aura noté, au passage, la similitude de forme de l'indicateur ICC et de l'équation d'effort telle qu'on la trouve dans les modèles d'estimation comme COCOMO. On pourrait également utiliser cette équation comme indicateur de complexité. Simplifier un système revient à diminuer son coût, en terme CQFD/TCO. Si ce n'est pas le cas, simplifier ne veut rien dire. En développant l'équation d'effort en série, on retrouve bien la forme indiquée ci-dessus (figure 12.7).

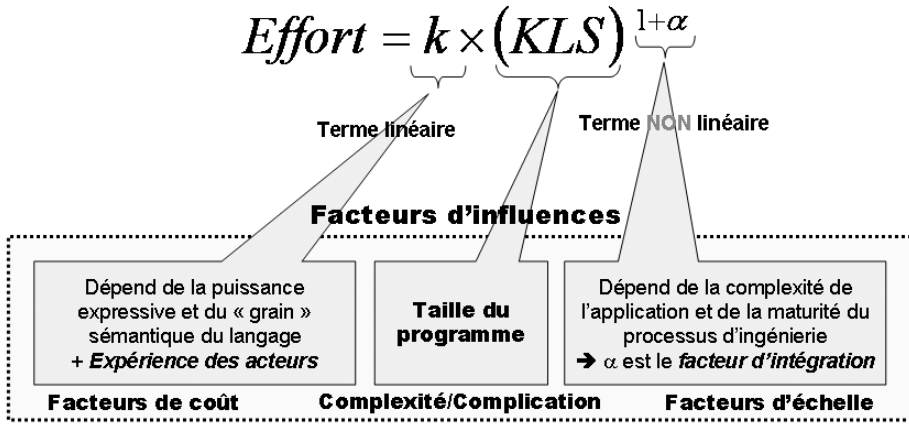


Figure 12.7 - Simplifier versus équation d'effort

Avec cette approche, simplifier un système revient à minimiser les trois termes {k, KLS, α}. Pour cela, il faut prendre tous les facteurs de coût, un par un, et regarder ceux sur lesquels les décisions de l'architecte et du chef de projet peuvent influer.

Un langage bien adapté permettra de minimiser le nombre de KLS (comme d'ailleurs la mise en œuvre d'une politique de réutilisation et/ou d'intégration de COTS).

Une architecture modulaire (avec de vrais modules) permettra tout à la fois de réduire la combinatoire dynamique et de minimiser les interactions entre les équipes réalisant les modules, ce qui se traduira par un coefficient d'intégration α voisin de 0.

À propos de langage, on a vu, dans le cas des compilateurs présenté au chapitre 6, l'intérêt d'un langage déclaratif permettant de spécifier la grammaire du langage. Par rapport à un langage comme C, le formalisme grammatical permet un gain de compacité de l'ordre 20 à 30, ce qui fait qu'une grammaire de 500 règles (langages de complexité type PL1/Ada) équivaudrait à un texte C de l'ordre de 15 KLS ; d'où l'intérêt du formalisme grammatical qui est indépendant du contexte, donc vrai dans tous les mondes exécutables possibles (c'est une propriété fondamentale du point de vue de la testabilité). A contrario, pour créer de tels langages, il faut maîtriser la théorie des langages, ce qui a un coût en terme de compétence. C'est également le cas pour les générateurs de code source utilisés pour certaines classes d'applications comme les IHM/GUI, mais pour que le générateur soit véritablement utile, en terme de simplicité, il ne faut surtout pas avoir à retoucher le code généré car dans ce cas, c'est le texte généré qui dénote la complexité et non pas le texte source. L'approche MDA[3], préconisée par l'OMG, présuppose, pour ceux qui veulent la mettre en œuvre, une bonne connaissance de la théorie des langages et des techniques de compilation du type de celles présentées au chapitre 6.

Le reproche que l'on pourrait faire à l'utilisation de l'équation d'effort comme indicateur de complexité/simplicité est qu'elle agrège des éléments projets (c'est son but) qui n'ont rien à voir avec la complexité du programme, comme par exemple les considérations sur le délai de réalisation où sur la compétence des acteurs, le nombre de défauts résiduels. Pour éliminer ces éléments projets « parasites », on pourrait définir des acteurs architectes programmeurs « parfaits » (un peu comme dans la théorie des gaz parfaits[4]) qui ne feraient pas d'erreur, qui identifieraient immédiatement les bonnes abstractions et les factorisations. Cela ne les dispenseraient pas d'écrire les tests, mais la différence avec les acteurs réels serait que les tests, eux-mêmes parfaits, ne révèleraient aucune erreur et ne seraient exécutés qu'une seule fois.

---

3. Voir IBM Systems journal, Vol 45, N°3, July/Sept. 2006, consacré au Model-driven software development, l'article de H. Hailpern, P. Tarr, *Model-driven development : the good, the bad, and the ugly.*

4. Cf. notre modèle du programmeur isolé parfait pour justifier la forme de l'équation d'effort dans notre ouvrage *Productivité des programmeurs*, déjà cité.

Dans ce monde « parfait » l'équation d'effort serait effectivement une bonne approximation de l'indicateur *ICC*, car l'effort ne serait que l'effort nécessaire à l'écriture du programme et de ses tests.

Notons que dans la logique d'un modèle d'estimation comme COCOMO, on peut faire apparaître une unité d'œuvre théorique correspondant à l'effort de développement d'un module étalon de taille P. La taille du programme réel, composé de *n* modules est de taille ***nxP***, ce qui d'une certaine façon donne une justification à l'approche points de fonctions.

$$\frac{Eff(nxP)}{Eff(P)} = n^{\alpha}$$

Le rapport ne dépend que du nombre de modules et de leurs interactions. Il pourrait jouer le rôle de notre indicateur *ICC*. Tout le problème serait alors de trouver la valeur du coefficient. Les modèles d'estimation comme COCOMO donne des heuristiques permettant d'estimer ce coefficient. Pour plus de détail, voir nos deux ouvrages : *Puissance et limites des systèmes informatisés* et *Productivité des programmeurs*.

12.2 AVANTAGES ET INCONVÉNIENTS DES MESURES TEXTUELLES

L'intérêt des mesures textuelles est qu'elles sont intuitives. Tout le monde admet qu'un texte de grande taille (en nombre de caractères, de mots, de phrases, etc.) est plus complexe qu'un texte court. D'une certaine façon, la taille matérialise la difficulté de mémorisation et donc de compréhension, même si le texte est enrichi d'une table des matières (structure hiérarchique) et de divers indexes qui permettent de naviguer dans le texte (accès direct, thésaurus, etc.) Cette intuition est aujourd'hui étayée par des apports récents de l'informatique théorique qui permettent de comprendre le pourquoi de la forme de l'équation d'effort alors que l'on sait comment s'en servir (sous certaines réserves) depuis plus de vingt ans ; cf. la théorie algorithmique de l'information, développée par A. Kolmogorov, G. Chaitin, et quelques autres. Les travaux de J.Y. Girard (cf. son manifeste : « la logique comme géométrie du cognitif ») sur la démonstration, la logique intuitionniste, le constructivisme mettent en évidence la relation de dualité entre a) le texte du programme vu comme une démonstration des transformations effectuées dans le monde réel Monde M1 par des procédures du monde M2 et b) le texte de la preuve de validité du programme que constitue le texte des tests.

Là encore, cela ne fait que traduire l'intuition des bons architectes qui ont compris l'importance cruciale des architectures dite testables, c'est-à-dire des architectures conçues pour faciliter le travail de validation, et minimiser le volume de test. Paradoxalement, ce sont les architectes hardware qui en ont perçu les premiers l'importance ; la notion de testabilité, dans la conception du matériel est un lieu

commun qui remonte aux années 60. En hardware, il est fondamental de valider la logique du circuit indépendamment de sa matérialisation dans le silicium, pour séparer les erreurs de logique des aléas de la matière et de l'environnement. C'est une sensibilité que la plupart des informaticiens n'ont pas, ou n'ont plus, mais qui est indispensable avec la montée de la complexité des plates-formes systèmes sur lesquelles s'exécutent les programmes. Pour le logiciel, les aléas proviennent des comportements des acteurs métiers et de l'environnement plate-forme, qui, du point de vue du logiciel, ne sont que des observables, et encore, si le logiciel est correctement programmé.

Il faut avoir le courage de dire que le texte des tests est, d'une certaine façon, plus important que le programme lui-même. C'est d'ailleurs ce que disent, dans leur style péremptoire et provoquant, les promoteurs de l'X-programming et du *Test Driven Development* ; ils ont ici la justification de leurs intuitions.

Nous ne développerons pas plus avant cet aspect de la complexité. Il faut savoir que la fondation théorique existe mais qu'il y a encore du travail pédagogique à effectuer pour que les ingénieurs informaticiens puissent se l'approprier[5].

L'inconvénient des mesures textuelles est qu'elles ne prennent en compte, par définition, que ce qui est explicite dans la structure du texte. Or de nombreux aspects qui influent sur la complexité ne sont pas visibles directement dans le texte sans l'aide d'un méta-modèle (i.e. le référentiel qui a permis la fabrication du programme et des tests) qui donne les clés de l'interprétation.

Comme on l'a vu dans les parties 1 et 3, le texte programmé n'est ni homogène ni uniforme, ce qui, dans l'équation d'effort, se traduit par des coefficients k et α différents (et d'ailleurs difficiles à déterminer sans une solide expérience, avec les règles d'emploi du modèle COCOMO). Le texte déclaratif (i.e. les données manipulées par les instructions) décrit la mémoire ; le texte impératif décrit les transformations effectuées. Avec les bases de données, le texte déclaratif (i.e. le DDL) est carrément sorti du programme avec les langages ad hoc des modèles sous-jacents (navigationnel, relationnel, objet). En toute logique tout pourrait se décrire avec un langage unique (du moins en théorie) qui pourrait être XML.

Le texte impératif écrit dans tel ou tel langage de programmation est enrichi avec la notion de fonctions primitives et/ou de « services » qui jouent le rôle d'instructions étendues de la machine abstraite qui effectue la transformation. Une instruction comme `y = sin(x)` compte pour une instruction alors qu'elle donne accès à un algorithme de calcul numérique complexe de plusieurs milliers de LS. Ceci revient à définir un langage spécialisé pour le calcul numérique (cf. Mathlab). On peut ainsi définir des langages spécialisés pour d'autres champs de connaissances et « habiller » les services de façon agréables à l'acteur métier (aspect pragmatique du formalisme adopté, dont on sait l'importance pour la performance des acteurs) ; c'est utile, mais

5. C'est l'un des enjeux des travaux entrepris autour de la thématique des systèmes complexes, en particulier à l'école Polytechnique et au CNAM, par l'auteur et D. Krob.

pas nécessaire. Il est intéressant de rapporter la taille du texte correspondant à la taille du programme, soit :

$$\%\ \text{textuel des fonctions de service} = \frac{\text{Tailletexte}_{service}}{\text{Tailleprogramme}}$$

De même, nous avons souligné l'importance sémantique des textes correspondant aux automates d'enchaînements (i.e. aspect contrôle) et des textes dédiés à la surveillance générale du système intégrés au programme, d'où de nouveaux ratios :

% textuel des fonctions de surveillance/instrumentation =

$$\frac{\text{Tailletextes}_{surveillance\ +\ instrumentation}}{\text{Tailleprogramme}}$$

Les tests incorporés directement au programme (i.e. « *on line test and diagnostic* », OLTD) sont le meilleur investissement quant à la pérennité de l'investissement test, d'où un autre ratio intéressant :

$$\%\ \text{texte OLTD} = \frac{\text{texte}_{OLTD}}{\text{Tailleprogramme}}$$

Par homogénéité avec les argumentations développées dans les parties 1 et 3, il faudrait distinguer, la nature des différents textes pour :

- Le contrôle (lié à la gestion des événements) ; NB : le nombre de types d'événements conditionne la taille de l'automate de contrôle.
- Les transformations (liées aux modèles de données) en distinguant les transformations simples du type règles de trois des transformations qui nécessitent la mise en œuvre d'un algorithme.
- Les entrées-sorties (liées à la structure des « ports » du système, et dépendant des plates-formes)
- Les ressources utilisées pour effectuer les transformations (liées à la structure des ressources)
- Les scripts de paramétrage du programme, dépendant de la structure du programme, qui matérialise la capacité d'adaptation du programme (i.e. le méta-modèle du programme).

Du point de vue de la complexité/complication le texte réellement significatif est :

Taille programme[Texte *services* + Texte *surveillance* + Texte *OLTD*]

qui peut être notablement inférieur à la taille du programme (si toutefois les services se comportent comme de réels services, i.e. équivalent à un appel de fonction comme le `sin(x)`, auquel on peut faire une totale confiance). Ceci revient à définir une typologie de programmes comme celle qui avait été définie dans la version 81 du modèle COCOMO (typologie en S, P, E ; cf. notre ouvrage : *Coûts et durée des projets informatiques*) et améliorée dans la version 2 du modèle (cf. B. Bohem, *Software cost*

estimation with COCOMO *II*, Prentice Hall). Le modèle montre la nécessité de définir des intégrats homogènes du point de vue de la typologie, en particulier de séparer les intégrats métiers des intégrats techniques (cf. figure 10.6). Pour ce qui concerne la validité de notre indicateur ICC, nous devons nous poser deux questions.

- Q1 : Quelle est la légitimité de la simplification opérée par un décompte brutal ?
- Q2 : Si légitimité il y a, où sont les limites de l'approximation au-delà desquelles la simplification perd son sens ?

12.2.1 Légitimité de la simplification

Il s'agit de la simplification permettant de choisir un coefficient α compris entre [0 ; 0,2] (cf. équation d'effort COCOMO).

Nous avons établi ailleurs que cette plage de valeurs n'était possible que dans le cas de structures hiérarchiques strictes (cf. également l'ouvrage de H. Simon, *Science of the artificial*, chapitre 8, Hierarchical structures, MIT Press, déjà cité). Pour cela, il faut matérialiser l'organisation du programme avec les méthodes et les outils ad hoc que sont la gestion de configuration et les matrices N2. Le but de la gestion de configuration est de matérialiser la traçabilité des exigences FURPSE/PESTEL depuis l'expression du besoin des acteurs métiers jusqu'à l'exploitation, en passant par les différentes phases du processus d'ingénierie. Cette traçabilité idéale nécessite la mise en œuvre de cinq bases de données logiques, comme indiqué ci-dessous (figure 12.8) :

Si tout ou partie des relations sont dans la tête des acteurs ingénierie plutôt que dans les BD du schéma, on peut comprendre les dégâts occasionnés par un turnover excessif dans les équipes. Comme indiqué sur le schéma, le point de départ de la traçabilité prend racine dans les processus métiers qui définissent les chaînes de valeur de l'entreprise. In fine, tout ce qui est fait par l'informatique a sa justification dans le métier (en raisonnant en coût complet, TCO et non pas en CQFD projet). Les modèles métiers issus de l'urbanisation, desquels seront extraits les scénarios de validation du point de vue métiers, définissent les critères de simplification en termes CQFD/TCO/FURPSE/PESTEL.

Si une fonction informatique a un impact quasi nul en terme de performance métier, mieux vaut ne pas la réaliser. La formalisation des modèles métiers est une innovation récente, actuellement en plein essor avec des langages spécialisés comme le BPML.

Les autres bases de données figurées sur le schéma sont plus classiques, étant entendu que les acteurs ingénierie qui ont les premiers perçu l'importance de la gestion de configuration ont été, pour des raisons évidentes, les acteurs intégration. Le développement s'est longtemps contenté des références croisées générées par les compilateurs, mais celles-ci se sont avérées insuffisantes avec la mise en œuvre des architectures clients/serveurs en distribué, d'où les dictionnaires de données des premiers AGL.

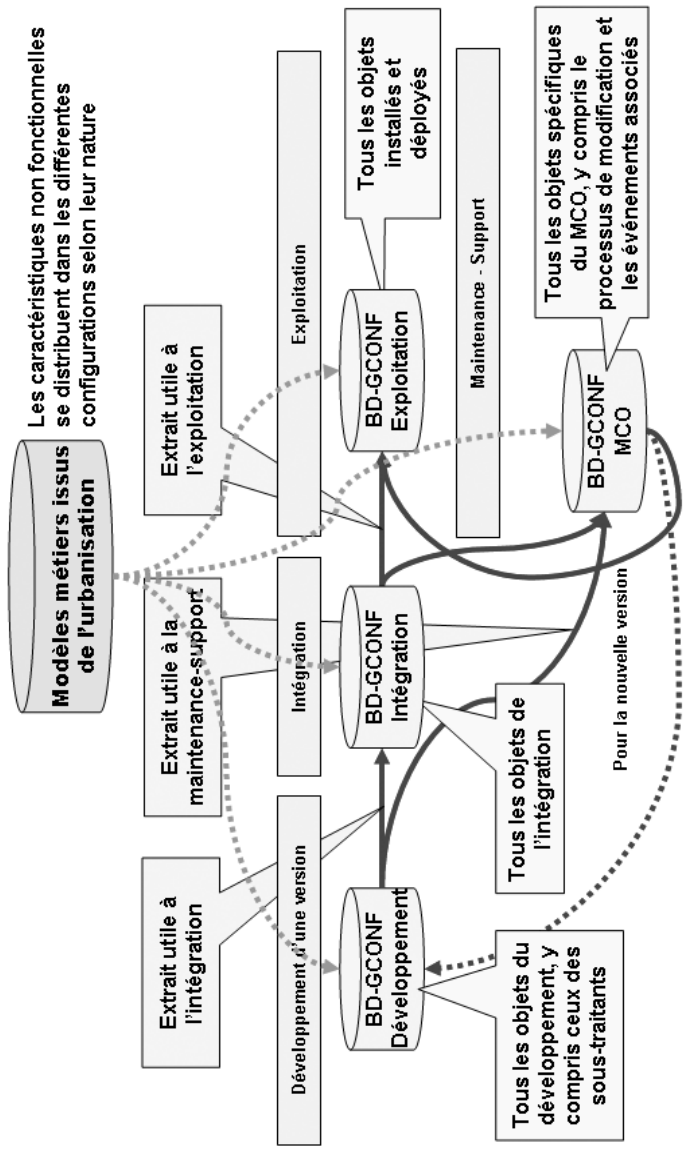


Figure 12.8 - Traçabilité de bout en bout des entités informationnelles

Encore plus récemment, avec la montée en puissance de la complexité des plates-formes d'exploitation, la gestion de configuration de l'exploitation est devenue une condition nécessaire du management du contrat de service (SLA) comme on peut s'en convaincre en analysant les préconisations de la démarche ITIL.

Pour ce qui concerne la configuration du déploiement, il faut prendre en compte et documenter les spécificités de chacun des équipements, comme suit (figure 12.9) :

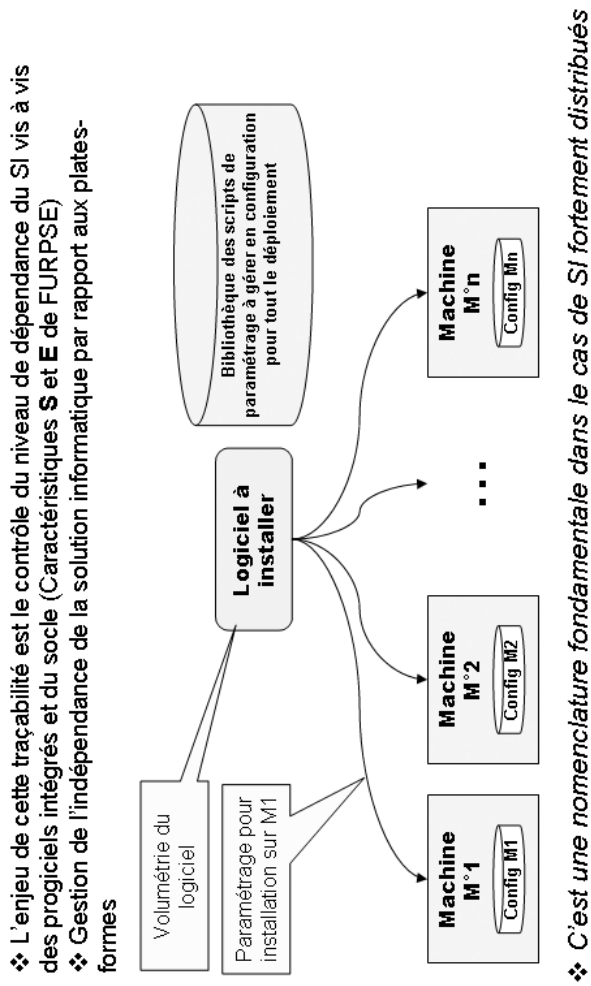


Figure 12.9 - Configuration du déploiement des logiciels

L'enjeu de cette dernière traçabilité est le contrôle du niveau de dépendance du SI vis-à-vis des progiciels métiers et système (i.e. middleware) installé sur les équipements. Elle matérialise la complexité liée aux caractéristiques **S** et **E** de FURPSE. On a ici la « preuve » de la nécessité économique de la banalisation des plates-formes, fusse au prix de la définition d'interfaces virtuelles comme celles préconisées par les approches J2EE ou logiciel libre.

Pour ce qui concerne le développement proprement dit, nous avons vu au chapitre 2 différents aspects des constituants des intégrats : les données, les fonctions, les contrôles (via les événements) et les ressources. Au chapitre 10, nous avons raffiné cette vision en introduisant les hiérarchies d'intégrats, les transactions, les services et les différents types de couplages et d'interactions entre ces différentes entités.

Tout ceci peut être matérialisé et visualisé à l'aide de matrices N2 dont nous avons déjà vu quelques exemples au chapitre 2.

Pour être complet et précis, il faut distinguer :

A) **Les couplages d'intégrats entre eux** (selon la granularité et la profondeur à laquelle on se place) qui sont des représentations matricielles des graphes de contrôles de l'enchaînement des intégrats. Ces graphes permettent d'analyser la couverture, les chemins, la fréquence d'emploi des différents intégrats, selon leur profondeur logique.

B) **Les couplages Intégrats/Données** vues par l'intégrat en faisant apparaître la granularité des données. Rappelons ici que selon le modèle universel entité/attribut/relation, nous pouvons travailler a) sur les attributs des entités (i.e. le niveau le plus fin de la modélisation des données) ; b) sur les entités elles-mêmes selon les niveaux d'agrégation.

Les agrégations à considérer en architecture des données sont :

- Enregistrements (ou t-uples du relationnel) qui regroupent des attributs sémantiquement apparentés (on pourrait parler d'une « géométrie » du sens[6]).
- Fichiers (ou tables du relationnel) qui regroupent toutes les occurrences de tel ou tel type d'enregistrements (une classe ou un ensemble de classes apparentées).
- Bases de données, selon schéma de la base.
- Fédération de bases de données (cf. la notion de système de systèmes et d'interopérabilité, vue au chapitre 1).

Une telle matrice à la structure suivante (tableau 12.1) :

Tableau 12.1 - Couplage Intégrats/Entités-Attributs

Identification de l'intégrat	Identification de l'entité ou de l'attribut				
	E1/A1		...	...	Em/Ak
ITG N°1					
ITG N°2		CRUD			
...				CRUD	
ITG N°n			CRUD		CRUD

À l'intersection on indique les modalités CRUD de l'entité agrégée ou de l'attribut élémentaire, selon les droits des différents intégrats. Le droit de créer/supprimer

6. Cf. un vieux livre de K. Lewin, *Principles of topological psychology*, que R. Thom a utilisé dans ses ouvrages *Stabilité structurelle et morphogenèse* et *Esquisse d'une sémiophysique*.

doit être fléché comme tel, vu son importance en matière de sûreté de fonctionnement (intégrité des données). En cas d'incohérence des données, il est indispensable de pouvoir recenser tous les intégrats qui ont un droit d'écriture (U) sur la donnée, soit de façon directe, soit de façon indirecte via un mécanisme de pointeur. Dans ce dernier cas, il est important de savoir distinguer la classe des entités pointées pour faciliter le diagnostic. Un exemple type est le serveur de données CRUD (cf. chapitre 10, section 10.2).

C) **Les couplages Intégrats/Evènements** en distinguant l'intégrat émetteur de l'événement (ordre SEND) de l'intégrat récepteur (ordre RECEIVE) qui contient le driver correspondant. Cette matrice doit faire apparaître l'intégrat « ramasse miette » qui récupère tous les événements qui n'ont pas été explicitement pris en compte. Cet intégrat doit être le plus haut dans la hiérarchie des intégrats. En environnement distribué, il est indispensable d'identifier rigoureusement tous les événements car contrairement à un environnement centralisé, il n'y a pas de récepteur d'événements par défaut.

D) **Les couplages Intégrats/Ressources** dont on a vu différents exemples au chapitre 9. Une ressource est vue comme un intégrat particulier qui nécessite un protocole d'accès spécifique à la nature de la ressource. Une ressource est un équipement particulier (cf. figures 2.1 et 10.8) dont l'accès nécessite la mise en œuvre par l'appelant d'un protocole d'accès qui est le « langage » d'accès à cette ressource, i.e. le CRUDE virtuel. L'intégrat appelant est donc dépendant de cette interface et doit, de ce fait, être distingué comme tel, d'où la structure de la matrice suivante (tableau 12.2) :

Tableau 12.2 - Couplage Intégrats/Ressources

Identification de l'intégrat	Identification de la ressource			
	R1	R2	...	RI
ITG N°1				CRUDE
ITG N°2		CRUDE		
...			CRUDE	
ITG N°n				

Le caractère distinctif d'une ressource est sa capacité d'exécution (i.e. le **E** de CRUDE) qui nécessite la définition d'un jeu de commandes complémentaires qui matérialisent les fonctions capacitaires de la ressource. Ce peut être un progiciel, comme SAP, dont on veut confiner les interactions avec le reste du SI de l'entreprise.

Cette dernière matrice permet d'étudier les couplages par l'environnement dans la mesure où la ressource partage le même équipement avec d'autres ressources (cf.

figure 2.1). Par exemple, un équipement peut héberger plusieurs serveurs de données logiques CRUD, ce qui fait que si l'un des serveurs sature l'équipement, cela induira nécessairement des défaillances chez les autres (cf. les approches AMDEC/FMEA[7]), ou des ruptures de contrat de service.

L'ensemble de ces différentes matrices permet de ce faire une bonne idée de la complexité de l'écosystème et de l'impact de cette complexité sur le contrat de service (SLA). Ces matrices sont des éléments importants du référentiel système. Si elles ne sont pas explicitées, on peut facilement imaginer le chaos engendré par des interprétations divergentes, résultant de « bricolage » local. L'étude de la complexité est aujourd'hui un monde en soi, dont nous avons donné un aperçu dans notre ouvrage *Puissance et limites des systèmes informatisés*, , qui va bien au-delà de l'informatique. Cf. les ouvrages de J-P. Delahaye, *Information, complexité et hasard*, Hermès, 1994 ; R. May, *Stability and complexity in model ecosystems*, Princeton University Press, 2001; R. Axelrod, *The complexity of cooperation*, Princeton University Press, 1997, pour des discussions éclairantes sur la complexité dans d'autres domaines que l'informatique.

12.2.2 Les limites de l'approximation hiérarchique

Le modèle de l'architecture en couche du chapitre 9 est une structure hiérarchique dans la mesure où les programmeurs ont strictement respecté les règles du référentiel. Si l'architecte, et les ingénieurs qualité qui s'assurent de la bonne compréhension et application des règles, n'ont pas une vigilance de tous les instants, il est inéluctable que ces règles seront violées et que ce qui a les apparences d'une structure hiérarchique (i.e. un modèle de classe, dans la terminologie UML) est en fait une structure beaucoup plus complexe. Plutôt que de faire de la théorie, prenons le cas d'un intégrat conforme à la définition qui en a été donné au chapitre 8, section 8.4 et 8.5.

Dans l'intégrat générique de la figure 8.9, la complexité cachée peut venir des données privées et/ou des données partagées. Notons qu'une transaction est une entité informatique indépendante du contexte d'exécution et de ce fait n'a ni données partagées autrement que via une ressource base de données, ni données privées statiques ; en fait une transaction n'a pas mémoire de ses états antérieurs (cf. la propriété fondamentale d'isolation, le **I** de ACID).

Dans le cas d'un intégrat avec mémoire statique et/ou mémoire partagée il n'en est pas de même, ce qui fait que le comportement de l'intégrat dépend de l'état de cette mémoire. D'un point de vue ontologique, tout se passe comme si l'intégrat générique, en fait une classe d'intégrats, pouvait manifester autant de comportements que d'états mémoire résultant de situations rencontrées durant sa « vie » d'intégrat.

7. Cf. E. Entley, H. Kumamoto, *Probabilistic risk assessment*, IEEE Press, 1992.

Le schéma 12.10 résume la situation :

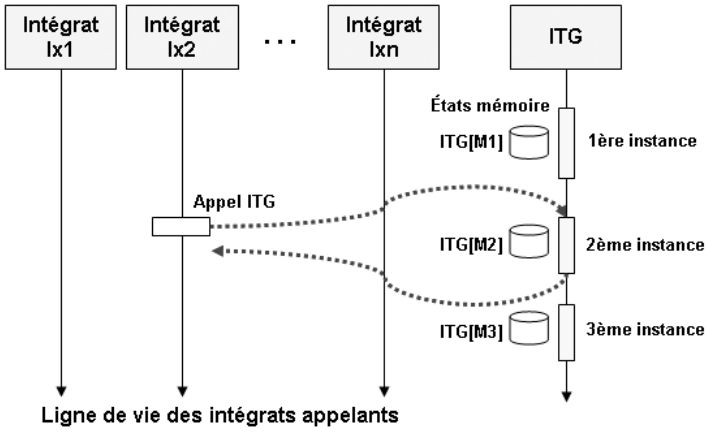


Figure 12.10 - Intégrat génériques et instanciations

L'état mémoire de l'ITG évolue de façon aléatoire en fonctions des appels résultant des lignes de vie de I1, I2, …, In. Du point de vue de la complexité, il s'agit de statuer sur le fait que ITG est une entité unique bien définie (i.e. un nom propre dans le langage de G. d'Ockham ou un nom substitut pour une famille d'entités dérivées du modèle générique ITG qui se comporte alors comme une classe artificielle, en violation du principe dit du « rasoir d'Ockham » rappelé au chapitre 1. Dans ce dernier cas il n'y a pas une entité, mais autant d'entités que d'états mémoire résultant de la structure de la mémoire statique et/ou partagée. En termes concrets on croit avoir affaire à une seule entité, alors qu'en fait on en manipule potentiellement un très grand nombre (un nombre non fini) ; cela revient à confondre la classe et ses éléments, ce qui est une faute logique caractérisée. En conséquence le coefficient d'intégration α peut être beaucoup plus grand que ce que le simple décompte des intégrats pourrait laisser penser.

Avec les données, il faut être particulièrement méfiant car la combinatoire exponentielle n'est jamais très loin, si on se laisse un tant soit peu aller du point de vue de la rigueur. Par définition, les données partagées sont vues par d'autres intégrats, soit pour chaque intégrat ITGx, des données partagées DPx. Si l'architecte n'y prend pas garde, ces données partagées peuvent donner lieu à des combinaisons 2 par 2, 3 par 3, etc. entre les différents intégrats en interaction, ce qui nous amène à une complexité exponentielle. On a vu, avec la notion d'architecture pivot (cf. figure 12.6) comment « casser » cette combinatoire. Si l'architecte n'a pas mis en œuvre cette architecture, la complexité exponentielle devient inévitable.

Il y a pourtant une situation ou ce type de mémorisation est particulièrement utile et nécessaire ; il s'agit des fonctions caches abordées au chapitre 11 (cf. figure 10.26). C'est la raison pour laquelle, il est fortement recommandé de se satisfaire de ce que fournissent les middlewares systèmes (SGBD, moniteur TP, MOM, EAI, etc.)

et de ne jamais s'aventurer à développer ce genre de fonction, sauf à avoir une expérience personnelle directe de ce type de développement que par ailleurs il faudra transmettre aux équipes de maintenance, donc très grande prudence requise.

En terme de simplicité, on doit toujours recommander de minimiser le nombre de données partagées, voire les interdire, ce qui permet à l'architecte de lever l'interdiction au cas par cas, si tant est que cela soit explicitement justifié en terme CQFD et TCO. Notons que ceci va à l'encontre de ce que font beaucoup de programmeurs qui invoquent des raisons de performances pour justifier le partage, ce qui est parfois légitime, mais qui a toujours un coût en terme de complexité. Une donnée partagée doit toujours être vue comme une ressource et accédée via un protocole d'accès ad hoc (serveur de données partagées).

12.3 LA COMPLEXITÉ DANS LE QUOTIDIEN DES PROJETS

Notre propension à fabriquer de la complexité/complication est à l'image de l'optimisme naturel des informaticiens : très grande. De fait, nous baignons dans un univers complexe, i.e. la complexité de la nature, et notre cerveau, Darwin oblige, est un organe extraordinaire, produit de l'évolution de la lignée homo sapiens sur des millions d'années, qui d'une certaine façon utilise cette complexité en nous donnant un avantage concurrentiel, peut-être temporaire, sur les autres espèces avec lesquelles nous partageons notre écosystème terrestre.

Qu'en est-il de la complexité « artificielle » des systèmes que nous créons de toutes pièces, qui eux ne sont pas soumis aux lois de l'évolution mais plutôt à celle de l'économie (quel est le juste prix d'un programme immatériel totalisant *x* milliers de lignes source et/ou de points de fonctions ?) et/ou des organisations humaines (et de leur bureaucratie !). L'informatique que nous fabriquons est, d'une certaine façon, le produit de notre système éducatif, et de notre système économique, c'est-à-dire un équilibre subtil entre : savoir et savoir-faire, compétence/expérience, juste prix, incompétence/inexpérience, illettrisme technique. Nous allons examiner quelques situations caractéristiques rencontrées dans les projets qui, si elles ne sont pas bien appréhendées par les architectes conduisent mécaniquement à des catastrophes économiques.

12.3.1 Complexité des données, fonctions, événements

Nous avons vu, dans les parties 1 et 3 que le travail de l'architecte consiste à réaliser une traduction fidèle du monde M1→ monde M2. M2 est fait de données, fonctions et événements, les trois constituants de la « matière » informationnelle. La question est : dans quel ordre faut-il s'y prendre ? Le savoir-faire empirique des architectes, depuis les années 60, nous dit : d'abord se préoccuper des données et des modèles de

données, puis des fonctions de transformations et en dernier lieu des événements associés aux processus.

Comment faut-il comprendre cette règle empirique ? Certainement pas de façon rigide, car chacun sait qu'il y a des interactions entre données et fonctions, entre fonctions et événements, etc. En fait, un architecte sérieux visitera l'ensemble des parties de nos trois constituants, c'est-à-dire : données, fonctions, événements pris individuellement, puis deux à deux (3 couples), puis trois à trois (1 triplet), soit au total $2^3-1=7$ combinaisons. Mais toujours il reviendra aux données, car les données sont le constituant le plus stable de l'édifice.

Examinons ce que donne la règle dans une situation d'interopérabilité (cf. chapitre 2). Il faut distinguer dans les trois constituants ce qui s'échange, soit un dédoublement des constituants. En terme d'ensemble des parties, on passe de 2^3-1 à 2^6-1, soit 63 combinaisons ! D'où le juste prix d'un projet d'interopérabilité et les nombreux échecs des projets d'intégration où le travail n'a pas été fait sérieusement. D'où une autre règle empirique :

Règle : Ne pas rendre interopérable ce qui n'a pas lieu de l'être ; surtout si l'on ne veut pas payer le juste prix, i.e. valider la combinatoire.

Une dernière remarque sur l'ordre DONNÉES → FONCTIONS → ÉVÉNEMENTS. Les fonctions transformatrices utilisent les données comme arguments des transformations, ce qui fait que la cardinalité de l'ensemble des transformations $||T||$ est $||S||$ à la puissance $||E||$, selon une formule classique de la théorie des ensembles. Il y a plus de choix possibles dans l'ensemble des transformations T que dans l'ensemble des données D ; où en terme de théorie de l'information : T contient plus d'information que D, ce qui est intuitivement évident, alors que le caractère exponentiel de $||T||$ ne l'est pas !

Moralité, pour le chef de projet et pour l'architecte, il est plus facile de se mettre d'accord sur l'organisation de l'ensemble D qui est plus simple, que sur celle de l'ensemble T.

La situation à laquelle l'architecte doit faire face est résumée dans la figure 12.11 :

Si les données transformées sont tantôt en entrée, tantôt en sortie, la cardinalité induite est $||D||$ à la puissance $||D||$, ce qui reste une exponentielle.

On a ici une explication théorique du risque de l'emploi de progiciels de façon débridée car précisément un progiciel est un ensemble de services offerts aux métiers. Si le chef de projet et l'architecte ne sont pas d'une extrême vigilance, le modèle de données du progiciel contaminera progressivement l'ensemble du SI de l'entreprise qui deviendra dépendant du progiciel.

Le même raisonnement s'applique à la cardinalité de l'ensemble des événements $||E||$ et des fonctions de contrôle /surveillance qui leur sont associés. Le risque est cette fois d'obtenir un système qui débride les interactions entre les acteurs métiers mais qui plombe définitivement les acteurs en charge de l'intégration et de l'exploitation du SI (cf. les trois populations d'acteurs dans le chapitre 1). L'indicateur de

complexité/complication est le volume des automates de contrôle qu'il faut mettre en face du besoin exprimé en matière d'interactivité de façon à garantir la disponibilité et la sûreté de fonctionnement.

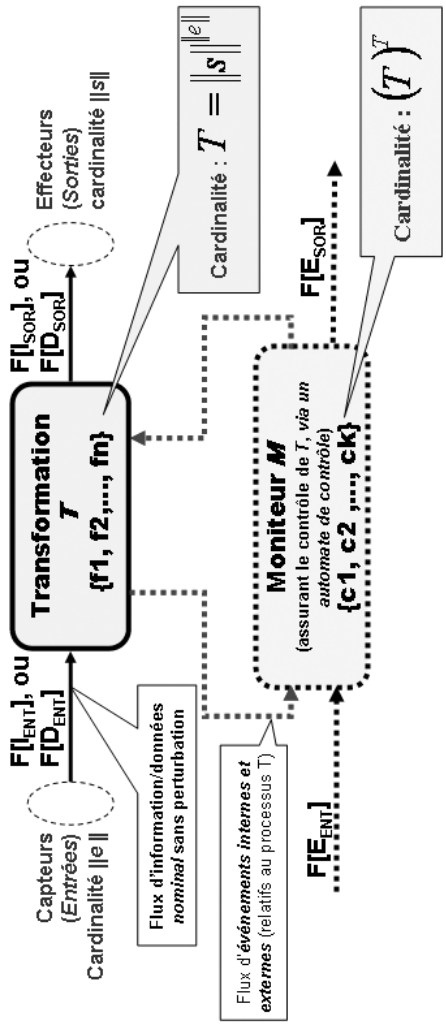


Figure 12.11 - Cardinalité des ensembles données, fonctions, événements

12.3.2 Complexité de la construction des modèles de données

Dans la littérature abondante sur la modélisation de données, on trouve sans difficulté tout ce qu'il faut connaître pour obtenir des schémas et des vues ayant de bonnes propriétés, avec la théorie des formes normales, les dépendances fonctionnelles entre les données, la gestion des indexes, etc.

Il y a cependant une étape initiale à cette construction qui n'est quasiment jamais abordée, bien qu'elle joue un rôle critique dans la pertinence de la relation [information du monde M1] / [modèles de données du monde M2] du point de vue des métiers. Elle correspond à ce qu'en statistique on appelle analyse des données (Cf. le Que sais-je N° 1854, de J.M. Bouroche, G. Saporta, *L'analyse des données*, aux PUF).

Pour étudier le comportement d'une population et comprendre ce qu'elle fait (N.B. c'est l'essence de la sémantique), on construit des tableaux individus×caractères. En langage informatique cela générera une modélisation où l'on parlera plutôt d'acteurs/rôles et de données agrégées avec différents niveaux, de grains (données scalaires, agrégats au sens programmation, agrégats au sens BD [i.e. un t-uple relationnel], un fichier [i.e. une table relationnelle], etc. Dans le langage du modèle ERA, il s'agit des entités et des attributs qui les définissent.

Le problème fondamental pour l'architecte de données est la transformation CARACTÈRES → DONNÉES ± AGRÉGÉES ce qui revient à ranger les caractères dans des « boîtes » elles-mêmes organisées en hiérarchie. Il y a un double problème à résoudre : 1) le problème de la collecte des caractères (d'où l'importance des modèles métiers faits avec les acteurs métiers) et 2) le problème de la classification des caractères dont le résultat sera un texte dans le langage du modèle ERA (ou d'une variante de ce langage universel).

On retrouve à nouveau des problèmes de partitionnement qui font que l'architecte des données manipule l'ensemble des parties, consciemment ou inconsciemment. Tout l'art de l'architecte consiste à fabriquer des classes disjointes permettant de définir une classification qui réalise un ordre partiel strict, i.e. aucun élément n'est dans deux classes ! Tout cela est bien connu, et fait (ou devrait faire !) partie de la culture basique de tout informaticien sérieux.

Pour l'architecte de données, dans son travail quotidien, la question pratique est de savoir comment ranger N caractères individuels (i.e. N attributs) dans les classes correspondant aux agrégations de données (i.e. les entités). Le nombre de combinaisons possibles d'un tel rangement est donné par la formule :

$\frac{k^n}{k!}$ qui est, à nouveaux, une exponentielle.

NB. : intuitivement, on peut « comprendre » la formule, bien qu'elle ne soit pas évidente à démontrer).

En terme projet, cela veut dire que si chacun est libre de classer selon ses propres critères, il n'y a aucune chance de tomber d'accord : la combinatoire est trop vaste. La pire des erreurs est de laisser les informaticiens choisir selon leurs critères issus des contraintes du monde M2, car ils classeront en fonction de ce qui les arrange ! Il y aura deux populations victimes : les usagers métiers et les usagers exploitants.

Si l'on veut un modèle de données informatiques qui reflète les exigences du monde réel M1, il est essentiel de spécifier les critères d'agrégation qui ne peuvent venir que des usagers métiers + exploitants, c'est-à-dire de la sémantique. À charge

des informaticiens d'effectuer la mise en forme, en privilégiant d'ailleurs d'autres critères plus liés à l'ingénierie comme la performance, fiabilité, évolutivité, etc. .

Ce recueil de contraintes est à la charge de la maîtrise d'ouvrage qui doit veiller à l'équilibre général de l'économie globale du SI de l'entreprise. Cet équilibre est dynamique et doit prendre en compte les intérêts de chacune des populations concernées, sans en privilégier aucune.

On peut comprendre par ces considérations pourquoi il n'y a pas de méthode générale pour résoudre ce genre de problème, mais plutôt un ensemble des situations avec lesquelles l'architecte doit jongler pour trouver une réponse équilibrée à ce qui constitue sa situation particulière : c'est le « *problem solving* » dans toute sa splendeur, ce qui requiert des architectes de grands talents, respectueux de la maturité des acteurs individuels et organisationnels avec lesquels l'architecte devra composer.

Dans une « matière » aussi subtile que l'information, les affirmations péremptoires, la brutalité, le volontarisme niais (cf. « ça va marcher ! sans qu'on sache ni pourquoi, ni comment !) n'auront pas prises : seul l'échec sera au rendez-vous. La seule méthode est l'intelligence et la compréhension apportées par les modèles.

NB : On remarquera que les données et les agrégats de données sont la matière de base du modèle d'estimation par la méthode des points de fonction (cf. la notion de groupe de données, caractéristique de ce modèle). Ceci montre, s'il en était encore besoin, la connexion très forte qu'il y a entre architecture et modèle d'estimation.

Si l'on reprend l'arborescence des intégrats, telle que présentée dans le chapitre 8, on peut se poser la question du nombre d'intégrats agrégés au-dessus des intégrats de rang 0 (Cf. figures 2.9 et 8.14).

Selon le critère d'agrégation retenu (par 2, par 3, etc.) le nombre d'agrégats abstraits, pour le besoin de l'intégration, est de l'ordre de **O[log(N)]** qui est à l'origine du coefficient <1 dans l'équation d'effort du modèle COCOMO. S'il y a des dépendances cachées entre les intégrats de rang 0, ce calcul n'a pas de sens, comme nous l'avons établi dans notre ouvrage *Productivité des programmeurs*, en annexe de l'ouvrage, déjà cité.

En restructurant la figure 2.9, on obtient la figure 12.12 :

On voit sur la figure que l'effort pour développer le système est de l'ordre de **O[*largeur*×*hauteur*]**, c'est à dire l'équation générale de l'effort COCOMO : [Taille moyenne de l'intégrat de rang 0]×[largeur de l'arbre d'intégration]×[coefficient d'intégration], ce qui revient à écrire l'équation d'effort sous la forme :

$$Eff = k(N \times ITG)^{1+\alpha} = k(N \times ITG) \times (N \times ITG)^{\alpha} = k'(N \times ITG) \times (N)^{\alpha}$$

avec $k' = k(ITG)^{\alpha}$ qui est une constante.

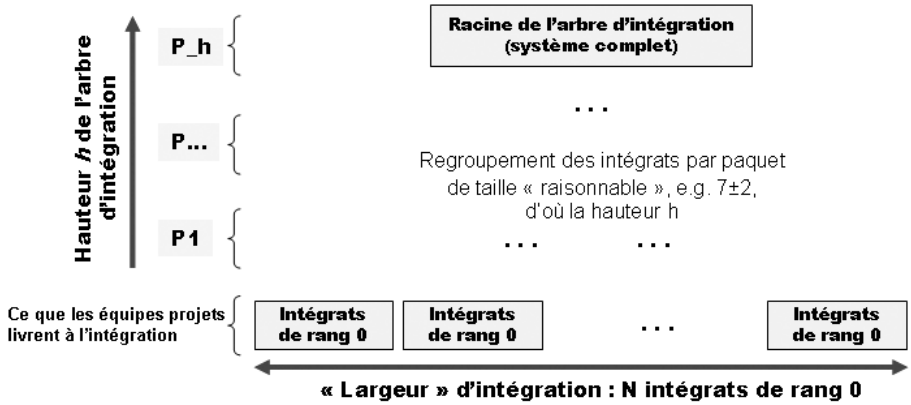


Figure 12.12 - Largeur et profondeur d'une classification

Si les intégrats sont indépendants, cela signifie que la hauteur de l'arbre est égale à 0, donc α=0. C'est le cas des transactions satisfaisant aux propriétés ACID, ce qui, là encore, nous ramène au modèle des points de fonctions qui apparaît comme un cas limite d'un modèle combinatoire plus général dont COCOMO donne un bon aperçu.

12.3.3 Complexité dynamique, interactions et couplages

Dans la réalité quotidienne, le SI gère des flux d'événements, i.e. de messages, éminemment variables qui doivent être pris en compte par les intégrats/transactions du (ou des) systèmes constitutifs du SI de l'entreprise. En termes informatiques, cela se traduit par des files d'attente d'événements plus ou moins longues, compte tenu des capacités de traitements des différents serveurs. Si l'on considère une tranche de vie du SI d'une durée D = TFIN-TDÉBUT, deux cas sont à considérer :

Cas N° 1 : On traite d'un coup une tranche de chronologie des messages enregistrés sur une durée D.

Cas N° 2 : Multiplexage des traitements qui peuvent traiter des messages de même nature, pour le compte d'usagers différents, de façon interactive.

Dans le cas N° 1, la différence entre un traitement par lot (i.e. le bon vieux « batch ») et un traitement interactif réside dans l'ordonnancement des événements. En batch, on trie, ce qui revient à créer un ordre artificiel sur un critère quelconque alors qu'en interactif on traite à la volée. En terme de combinatoire, la différence est de l'ordre d'une factorielle : n événements qui arrivent dans un ordre quelconque génèrent une combinatoire de l'ordre de n!, soit à nouveau une exponentielle si on applique la formule de Stirling :

$$n! = \left(\frac{n}{e}\right)^n \sqrt{2\pi n}$$

Pour que cette combinatoire s'annule, du point de vue informatique, il est fondamental qu'il n'y ait **aucune** relation de contexte du traitement d'un message au suivant. C'est ce qui est garanti par le **I** de ACID des moniteurs transactionnels, d'où l'importance du concept de transaction vu dans la partie 3 du point de vue de la combinatoire.

Dans le cas N° 2. il s'agit du multiplexage de flux de traitements et de la synchronisation des différents flux. Du point de vue du concepteur/programmeur d'une telle application, la situation est la suivante (figure 12.13) :

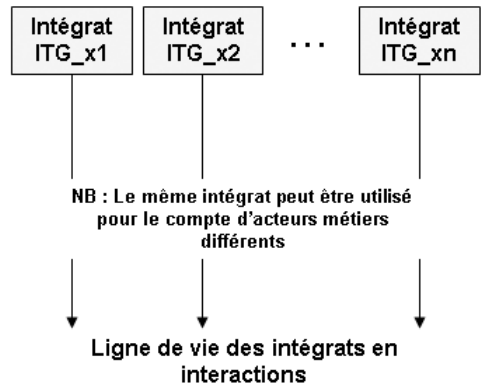


Figure 12.13 - Combinatoire induite par le multiplexage

En l'absence de moniteur de flux, c'est l'architecte/programmeur qui doit s'assurer de l'orchestration/chorégraphie, pour utiliser le vocabulaire présentement à la mode, mais qui ne résout pas les problèmes. L'architecte doit s'assurer que le partage des ressources entre les différentes lignes de vie est équitable et que chacun se comporte conformément aux règles établies. À nouveau, nous allons retrouver l'ensemble des parties. En effet, si N flux sont en interactions, le concepteur/programmeur doit s'assurer que toutes les combinaisons qui peuvent se présenter restent cohérentes, soit, compte tenu du multiplexage les combinaisons deux à deux, trois à trois, …, soit à nouveau 2^N-1.

Une règle de pouce des architectes, en particulier dans les systèmes temps réel, dit que les programmeurs les plus doués (vérifiée de nombreuses fois par l'auteur) sont capables de gérer au mieux 4 ou 5 flux parallèles, ce qui, en termes d'ensemble des parties restreint aux combinaisons 2 à 2 revient à gérer $\frac{n(n-1)}{2}$ cas d'interactions, soit de 6 à 10 combinaisons, où l'on retrouve la règle bien connue du nombre « magique » 7±2.

Si le système est effectivement transactionnel (mais encore faut-il le prouver), il y a indépendance effective des flux, le seul problème du concepteur/programmeur est de gérer les points de reprises automatiques des transactions via les ordres de commande du transactionnel : COMMIT, ABORT, ROLLBACK, soit 2 flux, ce qui reste à la

portée de la plupart des programmeurs pour autant que ceux-ci soient assistés par un moniteur qui garantisse les propriétés **ACID** dans leur totalité, et non pas seulement le I. On a vu au chapitre 9 que dans le cas des transactions longues, l'arbitrage ne peut être que sémantique, d'où une complexité additionnelle à la charge de l'architecte du système.

La fluctuation des flux de messages se traduira, du point de vue informatique, par des fluctuations de l'espace mémoire nécessaire à la bonne gestion des files d'attente correspondantes. Si les contraintes de sûreté de fonctionnement sont élevées, par exemple aucune perte d'information à la saisie, les files d'attente correspondantes doivent être gérées en mémoire persistante (i.e. le **D** de ACID), ce qui est une nouvelle complexité additionnelle.

À titre de dernière remarque, mais non pas de dernier problème, considérons des situations de couplage entre intégrats qui sont fréquentes en matière de transactions longues (cf. la figure 9.8). Chacun des intégrats ITG1 et ITG2 s'est terminé de façon nominale, i.e. V ou non nominale, i.e. F. Qu'en est-il du couplage ? la situation est donnée par le schéma 12.14 :

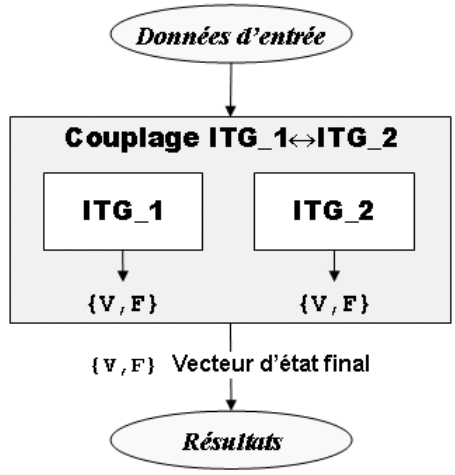


Figure 12-14 - Combinatoire induite par le couplage des intégrats

Pour deux intégrats ITG_1 et ITG_2 dont chacun peut sortir en {V, F}, il y a 16 fonctions logiques booléennes possibles pour le mapping, soit : ({V, F}, {V, F}) → {V, F}.

Si l'on considère n intégrats, ce nombre croît de façon exponentielle, soit $2^{2\alpha}$. Si l'on considère la modalité « indéfini, soit {V, F, I}, ce nombre devient $3^{3\alpha}$.

Moralité : il suffit de changer le mode de couplage, sans modifier la programmation, pour observer des comportements différents en cas de défaillances.

Typiquement : Deux composants applicatifs sont regroupés sur un même serveur (i.e. couplage par les ressources partagées). Si l'un des ITG perturbe l'environnement (blocage et/ou saturation d'une ressource), l'ensemble des ITG regroupés sur le serveur ne fonctionne plus correctement !

Du point de vue de l'usager et/ou de l'exploitant du système, le comportement observé va se traduire par des temps de réponse, des temps d'attente, des débits, des refus de service si les capacités du système ne sont plus à même de soutenir la charge de traitements demandée au système (ces informations doivent être consignées dans la partie spécifique du vecteur d'état ; cf. chapitre 8).

Le niveau de charge demandé va faire émerger des comportements dynamiques au sens que le mot a dans la théorie des systèmes dynamiques. L'architecte est face à un nouveau type de complexité qui dépend de la nature des couplages qui existent entre les intégrats et de l'état des ressources qui fournissent « l'énergie » informationnelle nécessaire aux transformations. D'une certaine façon, comme dans un réseau de transport d'énergie, il faut assurer un équilibre entre l'offre de capacité de traitement et la demande des usagers connectés au système.

La gestion des ressources devient un élément critique du système. Si un ITG gaspille les ressources (par exemple en s'allouant plus de mémoire que nécessaire) c'est l'ensemble du système qui en pâtira. D'où l'importance du concept de surveillance généralisée que nous préconisons ; c'est la clé du concept d'« *automic computing* » promu par IBM.

On peut comprendre le comportement global du système en analysant la structure de la ligne de vie de chacun des intégrats vis-à-vis des ressources utilisées. Le schéma de principe est le suivant (figure 12.15) :

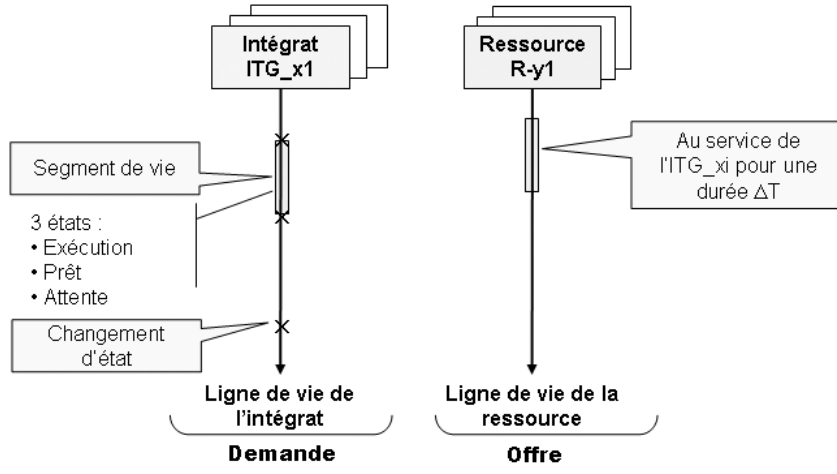


Figure 12.15 - Topologie de la ligne de vie d'un intégrat

L'intégrat qui s'exécute pour le compte d'un ou plusieurs usagers peut être dans 3 états (cf. chapitre 15) :

- En EXÉCUTION, il dispose d'une ressource (un CPU, un canal d'entrée-sortie, i.e. un contrôleur, etc.).
- PRÊT, il attend qu'une ressource se libère.
- ATTENTE, il est temporairement arrêté et attend le résultat d'une opération (par exemple une entrée-sortie, une information fournie par l'usager via un poste de travail, etc.).

La succession normale des états est : EXÉCUTION → ATTENTE → PRÊT, ce qui définit trois types de segment de vie. La × représente un changement d'état sur la ligne de vie, ce qui suppose une action du système d'exploitation. La densité de × sur la ligne de vie est un indicateur de niveau de sollicitation du système d'exploitation et/ou des services système (SGBD, réseau, etc.). Un système dont les ressources ne sont pas saturées se manifestera par des durées très brèves dans l'état PRÊT. Dès qu'une attente se termine, on repasse rapidement dans l'état en EXÉCUTION. Un système proche de la saturation se manifestera par des durées importantes dans l'état PRÊT : aucune ressource n'est disponible, l'intégrat ne peut pas redémarrer. Si le système d'exploitation intervient pour interrompre un exécution, cela augmentera le nombre de ×, donc la consommation d'énergie destinée à la gestion des conflits, c'est à dire une baisse de rendement.

Du côté des ressources, les choses sont plus simples : ou bien la ressource travaille pour le compte de l'intégrat ITG_xi, ou bien elle ne travaille pas (état « *idle* »). Une ressource proche de la saturation n'est jamais dans l'état idle ; il peut même se former une file d'attente d'intégrats qui sont tous dans l'état ATTENTE. Dans ce cas, le débit du système s'aligne sur le débit de la ressource (phénomène dit de convoi), ce qui peut engendrer d'autres phénomènes d'attente sur d'autres ressources. Le système s'engorge progressivement, voir très rapidement si la ressource où se manifeste le phénomène de convoi est une ressource critique fréquemment sollicitée.

Dans un système engorgé, l'équilibrage offre/demande peut devenir chaotique, car pour rétablir l'équilibre, il faut que le système d'exploitation intervienne, ce qui aura comme premier effet de prélever des ressources, donc d'augmenter le niveau de saturation. Ces phénomènes de non linéarité sont apparus pour la première fois avec les systèmes dits « *time sharing* » et avec la mémoire virtuelle qui n'est rien d'autre qu'une gestion du partage de la ressource mémoire, en permettant aux usagers de consommer de la mémoire comme s'ils étaient seuls sur la machine. Avec l'accroissement des tailles mémoire, cela a occasionné un nouveau phénomène dit des « fuites de mémoire », et plus généralement de « fuites de ressources » dus à de mauvaises habitudes de programmation que l'architecte doit vigoureusement combattre. Dans un système distribué, il n'y a pas, par définition, de gestion globale des ressources. Chaque système doit s'auto-discipliner pour ne pas nuire à l'équilibre global. Si le programmeur d'un intégrat ne restitue pas explicitement au système d'exploitation les ressources dont il n'a plus l'usage (très fréquent avec les ordres d'allocation dyna-

mique de mémoire), il y a risque d'engorgement artificiel car le système d'exploitation croira que les ressources en question restent utilisées alors qu'elles ne le sont plus. Idem avec un fichier resté ouvert sans raison.

Moralité : Dans un système distribué, il est essentiel de disposer d'un modèle même qualitatif et rudimentaire, de gestion des ressources et de mettre en place les sondes d'observation qui permettent d'observer l'état des ressources.

La complexité dynamique, avec la virtualisation, des services et des ressources qui créent du confort pour les usagers, est là pour nous rappeler que la puissance des équipements n'est jamais illimitée et que les ressources disponibles doivent être gérées avec rigueur. Éviter les accidents d'exploitation à cause de phénomènes de « pompage » des ressources est l'une des tâches nouvelles dont doit s'occuper l'architecte, en prenant au sérieux les besoins et exigences des équipes d'exploitation, afin de mettre en œuvre de véritables mécanismes d'autorégulation et de contrôle de charge. Tout ceci concourt à la robustesse du système (en anglais « *resilience* ») qui ne peut résulter que de la bonne connaissance des ressources nécessaires à l'exécution du contrat de service demandé par les usagers. Un progiciel qui ne précise pas sa consommation de ressources doit être géré comme un risque, du point de vue de l'exploitation.

13

Disponibilité – Sûreté de fonctionnement

13.1 INTRODUCTION

La disponibilité d'un système, ou d'un équipement, se mesure par le rapport $Dispo = \frac{MTTF}{MTTF + MTTR}$ selon une définition classique dans lequel :

- MTTF (*Mean Time To Failure*) est le temps moyen de bon fonctionnement de l'équipement.
- MTTR (*Mean Time To Repair*) est le temps moyen de réparation de l'équipement.

Ce sont des mesures statistiques qui résultent de comportements moyens, observés sur de longues périodes de temps, pour que la mesure soit significative. Si l'équipement intègre des acteurs humains, ceux-ci font partie de la boucle de mesure, avec les incertitudes que cette présence humaine impose.

D s'exprime en %. Une disponibilité de 99,99% sur une durée nominale de 1 000 heures de bon fonctionnement signifie que l'équipement ne doit pas être en panne plus de 6 minutes. Un équipement téléphonique devant fonctionner 24H/24, 7J/7 pour lequel on accepte une minute de panne par an devra avoir une disponibilité de 99,998% . La disponibilité permet de mesurer ce que les Anglo-Saxons appellent le « *How good is good enough ?* » en évitant la sur-qualité[1].

Dans un monde parfait où il n'y aurait jamais d'erreur, la disponibilité vaut toujours 1. Dans un monde imparfait, mais où la réparation serait instantanée, c'est-à-

1. Cf. IEEE Software, Vol 23, N°5, Sept./Oct. 2006, L. Huang, B. Boehm, *How much software quality investment is enough : a value based approach.*

dire MTTR = 0, la disponibilité vaut également 1. Dans le cas des systèmes à tolérance de pannes (i.e. certaines pannes comme ce que tolèrent les disques RAID) la réparation est effectuée automatiquement par le système, ce qui peut conduire à un MTTR ≈ 0, donc à D ≈ 1.

Le MTTF est lié à la fiabilité du système que l'on définit comme suit (c'est le R de FURPSE) :

Fiabilité (*Reliability*) : probabilité de fonctionnement sans panne, pour une durée donnée, sous certaines conditions environnementales explicitement spécifiées.

Pour qu'une mesure de fiabilité soit valide elle doit être faite conformément aux plages de fonctionnement définies ; il faut de plus s'assurer que les conditions d'emploi ont été respectées, ce qui vaut a) pour l'exploitation qui doit effectuer certaines opérations, b) pour l'environnement qui doit respecter les normes convenues, et c) pour les différentes catégories d'utilisateurs qui doivent être correctement formés et se comporter conformément aux procédures contractuelles.

Exemples d'opérations à effectuer par l'équipe d'exploitation : ré-initialiser les machines régulièrement, dé-fragmenter tel ou tel disque, sauvegarder certains fichiers, changer les filtres à poussière de tel disque, vérifier l'état électromagnétique des locaux, etc.

Jusqu'au début des années 80, la disponibilité du matériel était le problème N°1 de la fiabilité des équipements informatiques. Ce problème a magistralement été résolu, après vingt ans d'effort, grâce à la théorie des codes correcteurs d'erreurs[2] et à l'architecture des machines qui a permis de déterminer les placements optimaux des redondances matérielles. Avec le logiciel, la nature des pannes est différente. La fiabilité du logiciel est fonction : a) de la densité de défauts résiduels qui s'exprime en nombre de défauts par KLS ; b) de la fréquence d'exécution relative des intégrats ; c) de la durée des sessions ; d) de l'état capacitaire des ressources ; e) du comportement des usagers. Il faut être capable de mesurer cette fréquence d'emploi, sur une granularité donnée du système. L'organisation hiérarchique du programme est déterminante dans l'obtention d'une fiabilité garantissant le niveau de service exigé par les usagers métiers et la MOA.

Dans la figure 3.5 nous avons fait apparaître deux types de programmation appelés : 1) code fonctionnel nominal et 2) code fonctionnel non nominal. Par définition, la fréquence d'emploi du fonctionnel nominal est beaucoup plus grande que le non-nominal, car ce code correspond au mode de fonctionnement normal du système. Ce n'est pas pour autant que le non-nominal est moins important, surtout s'il participe à la sûreté de fonctionnement du système. La distinction des deux codes fait que le code nominal va atteindre beaucoup plus vite son palier de maturité, et que surtout on a les moyens de le garantir : il suffit d'en mesurer la fréquence d'emploi par un simple comptage des entrées et des sorties de chaque intégrat. Si tout est enchevêtré et faiblement structuré, c'est beaucoup plus difficile, voire

2. Cf. IBM Journal of research and development, Vol 28, N°2, March 84, C. Chen, M. Hsiao, *Error-correcting codes for semiconductor memory applications : a state-of-the-art review.*

impossible. Le code non nominal, par définition, est activé en cas d'erreur. L'environnement est incertain, ce qui requiert une programmation défensive afin d'éviter la double panne ; il doit être identifié comme tel.

Notons que par rapport à la définition, la fiabilité logicielle est une notion qui n'a pas un sens évident, comme c'est le cas pour la partie matérielle de l'équipement. Il faut éviter l'erreur méthodologique consistant à appliquer l'algèbre des probabilités à des coefficients de « confiance », fussent-ils compris entre 0 et 1, qui ne sont pas des probabilités. Sans entrer dans une étude détaillée qui n'a pas sa place dans cet ouvrage, on peut représenter la situation à prendre en compte pour effectuer un calcul correct par le schéma 13.1 :

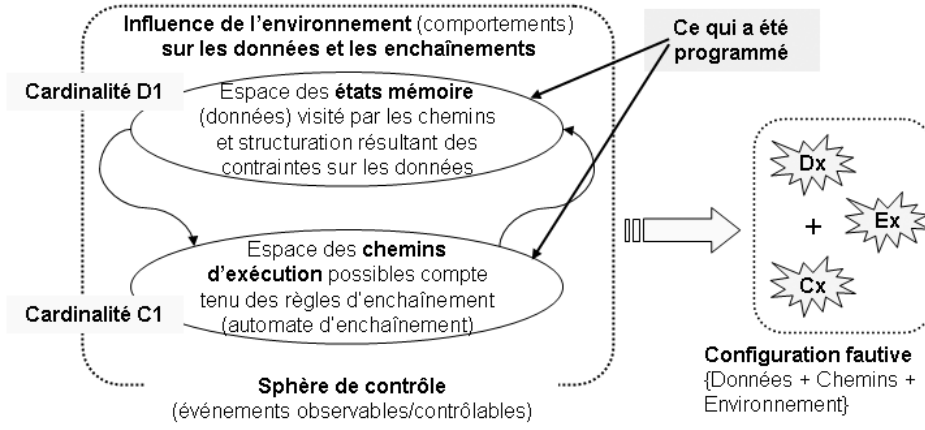


Figure 13.1 - Structure de l'ensemble des configurations fautives.

Le schéma montre les ensembles à partir desquels il faudrait effectuer des mesures pour en déduire des coefficients qui seraient des probabilités. Notons que l'environnement est un système ouvert dont la cardinalité n'est pas évidente. Le système programmé « voit » l'environnement par les événements qui lui remontent et qu'il sait traiter. Pour conserver la propriété de prédicativité, cet ensemble ne doit être manipulé qu'en extension, de façon explicite (programmation avec « gardes » ou par contrat). Comme exemple d'événements, on peut citer : a) un disque non dépoussiéré va occasionner des erreurs de lecture des pistes du disque, ce qui va engendrer un taux de relecture (appelé « *retry* » dans le jargon) donc une chute des performances visible au niveau des fonctions d'accès aux information sur le disque que le système d'exploitation peut analyser (cf. les outils de *capacity planning* et de *system management*) ; b) une ligne de télécommunication perturbée pour une raison quelconque (champs magnétiques parasites, effets d'antennes, longueur excessive, effets mécanique de pression et/ou torsion, etc.) va détecter des erreurs de parité occasionnant des ré-émissions de messages, d'où chute du débit, encombrement aléatoire des

files d'attente de messages, avec comme symptôme final le dépassement d'un « time-out » également visible au niveau du système d'exploitation ; etc.

Comme on le sait, tout défaut logiciel est toujours le résultat d'une erreur humaine. C'est ce que rappelle le schéma 13.2 :

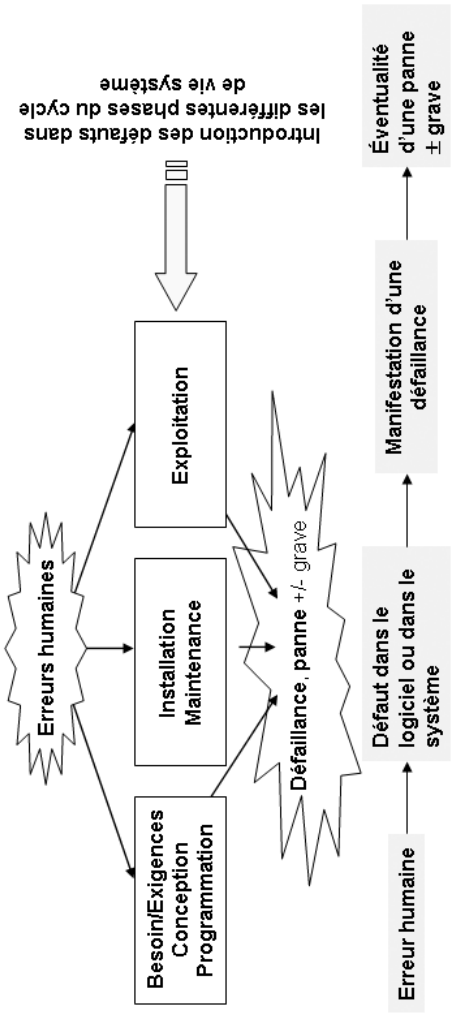


Figure 13.2 - Introduction des défauts

Un défaut est une défaillance potentielle qui ne se manifestera que sous certaines conditions d'exécution et d'environnement. Il se peut que le défaut reste latent très longtemps (cas de l'échec d'Ariane 501), ou ne se manifeste jamais. Dans la perspective d'une mesure de la disponibilité *D*, il faut considérer les actions effectuées dans le temps, ce qui donne le schéma 13.3 :

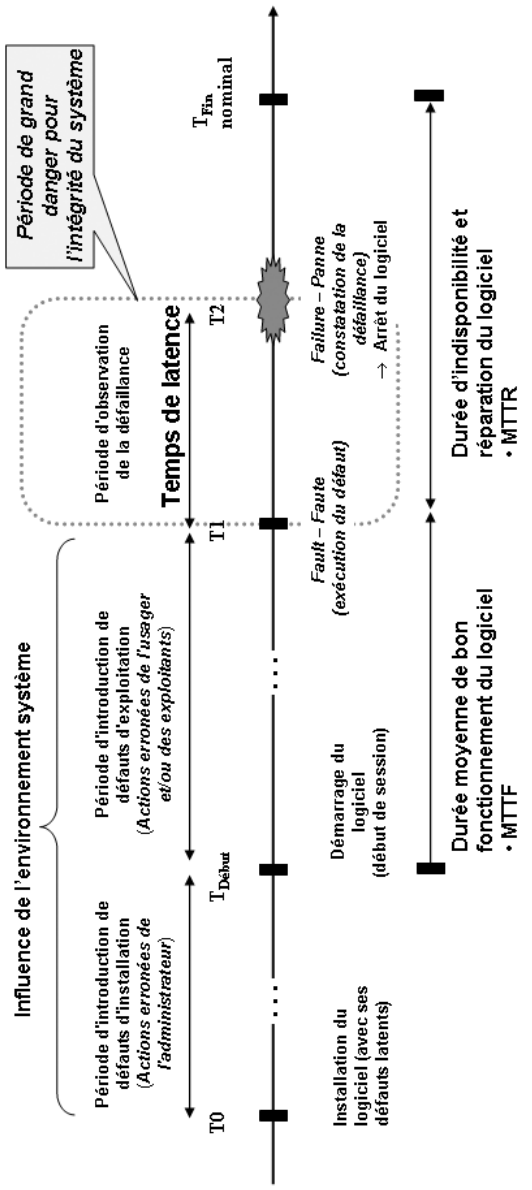


Figure 13.3 - Faute et défaillance – MTTF/MTTR

L'axe des temps est la ligne de vie de l'intégrat considéré. Par rapport à la figure 8.15, l'état en-cours/actif peut dégénérer en état fautif nécessitant un arrêt temporaire du service.

La défaillance constatée à l'instant T2 résulte d'une action erronée antérieure dont il faudra retrouver la trace afin d'effectuer les corrections. Le schéma logique de la détection de la faute est le suivant (figure 13.4) :

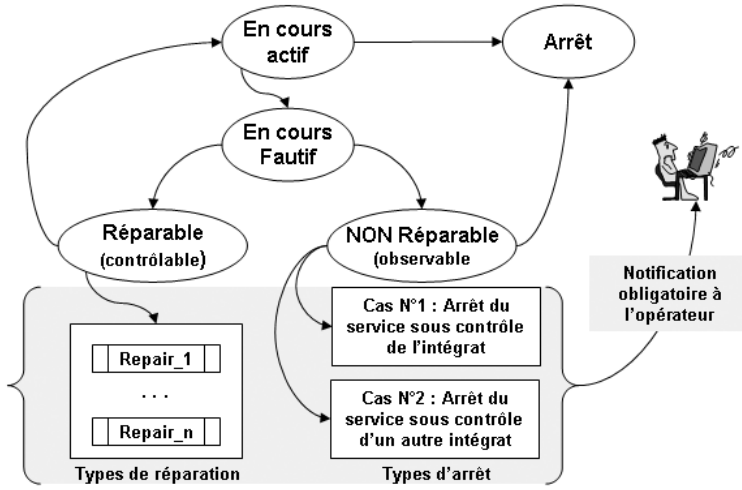


Figure 13.4 - Détection des défaillances

Du point de vue de la testabilité, l'intégrat peut être dans trois états différents :

- La faute est réparable, en temps réel (pas d'arrêt du service) ou hors temps réel (il y a arrêt du service et intervention de l'opérateur qui dispose d'une contre-mesure pour redémarrer).
- La faute n'est pas réparable mais on peut arrêter proprement le service en minimisant les pertes de travaux du point de vue métier.
- La faute est observable, via les sondes qui ont été pré-positionnées ; c'est la vraie panne.

Le problème de la disponibilité est en fait celui de la bonne utilisation de l'effort consenti pour les tests et de la stratégie d'emploi de cet effort pour les différentes catégories de test. En quoi l'architecture logicielle peut-elle contribuer à une meilleure utilisation de cet effort : c'est ce que nous appelons architecture testable.

Du point de vue des acteurs test, la situation est donnée par le schéma 13.5.

Chaque niveau a sa courbe de maturité. Si un niveau laisse passer des erreurs qui le concernent, elles lui reviendront tôt ou tard, avec un coefficient d'amplification qui dégradera la rentabilité de l'effort de test.

Tous les acteurs test ont besoin, pour effectuer les diagnostics, d'informations précises : c'est ce à quoi l'architecture peut fortement contribuer.

Pour comprendre ce qu'est, ou devrait être, une architecture testable, il est essentiel de comprendre la nature des problèmes auxquels l'acteur test est confronté lorsqu'une défaillance se manifeste. C'est ce que montre le schéma 13.6.

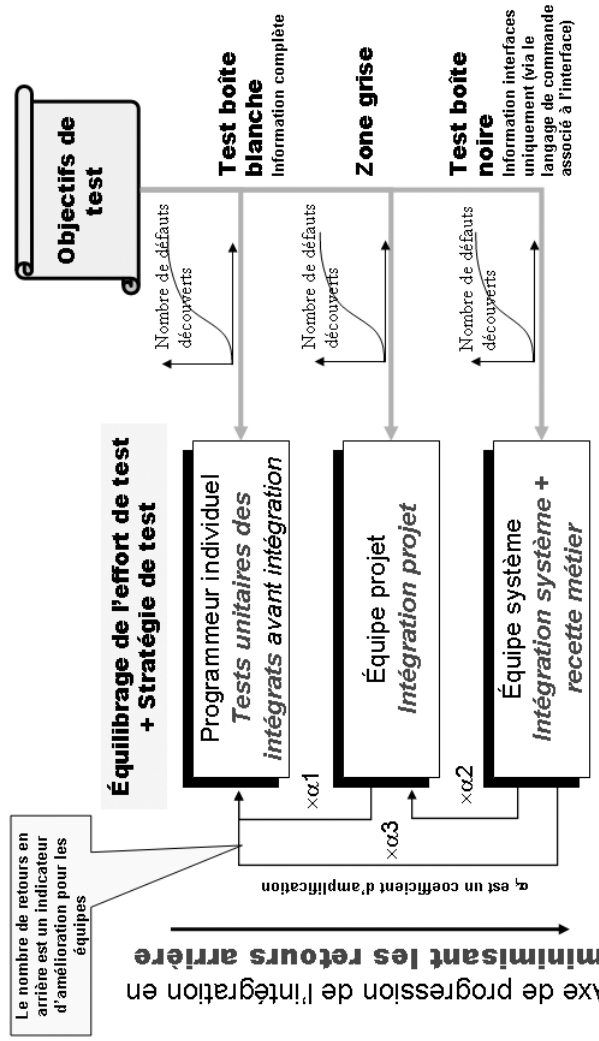


Figure 13.5 - Stratégie de test et découvertes des défauts

Lors d'une défaillance il faut une image aussi précise que possible du contexte de la défaillance de façon à pouvoir remonter de façon sûre à la faute qui est le contexte d'exécution du défaut. Le temps de latence (cf. figure 13.3) qui s'écoule entre ces deux instants est une durée critique durant laquelle il est primordial de disposer d'observations qui faciliteront le diagnostic.

Plus le temps de latence est long, plus le risque de ne pas disposer de la bonne information est élevé, ou de disposer d'une masse d'information dont seule une très faible partie concerne la défaillance (i.e. information non pertinente), ce qui revient au même. Pour l'acteur test, le problème N°1, en situation de défaillance est de disposer d'une information complète, structurée par la logique d'enchaînement, et pertinente.

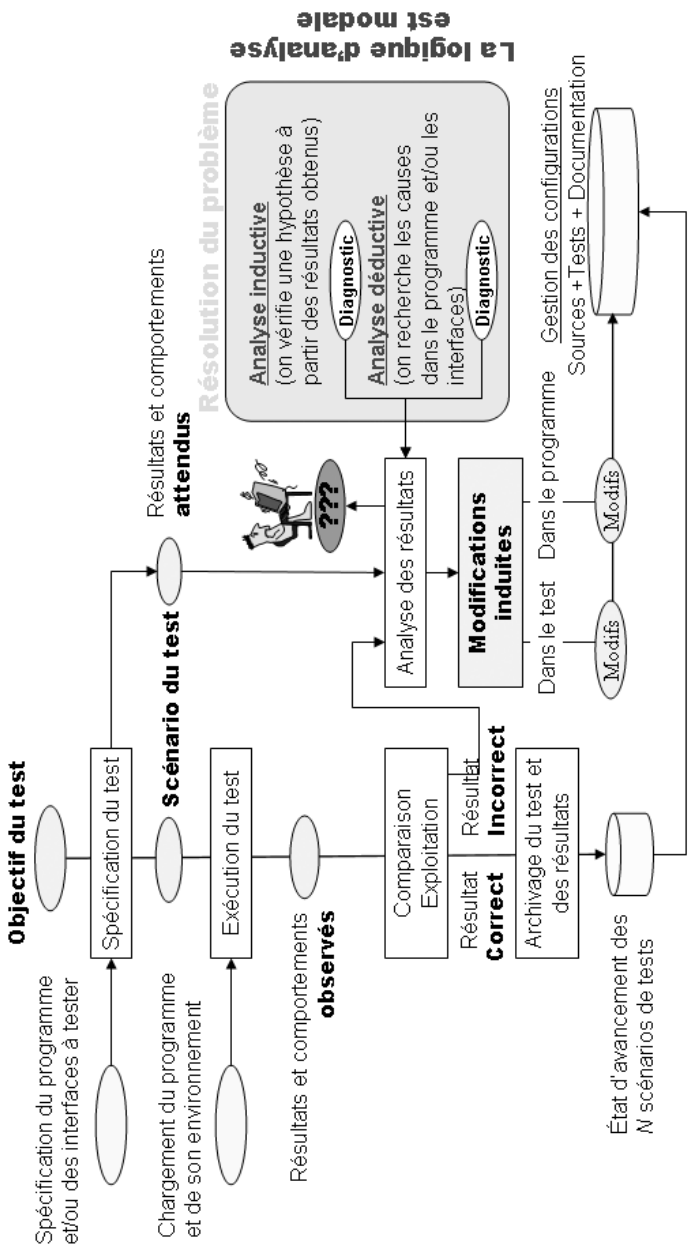


Figure 13.6 - Assistance de l'architecture au travail de test.

À partir de là, il pourra formuler un diagnostic dont la thérapie sera une ou plusieurs modifications dans des intégrats qu'il faudra reconstruire afin d'effectuer un nouvel essai. Au préalable, il faudra s'assurer que les intégrats reconstruits ne sont pas en régression par rapport aux tests déjà acquis (opération de non-régression). Si le logiciel est de grande taille, la structuration résultant de l'architecture doit mini-

miser le nombre de tests devant être exécutés de nouveau, surtout si ces tests obligent à la présence d'acteurs test pour surveiller leur déroulement.

Le processus de test décrit par la figure 13.6 ne se termine pas toujours par un diagnostic et des corrections. Il y a des cas où le contexte de la défaillance reste incompréhensible et ne permet pas de faire un diagnostic. C'est le phénomène connu depuis les années 70 des défaillances-pannes non reproductibles.

Donnons un exemple de telles défaillances.

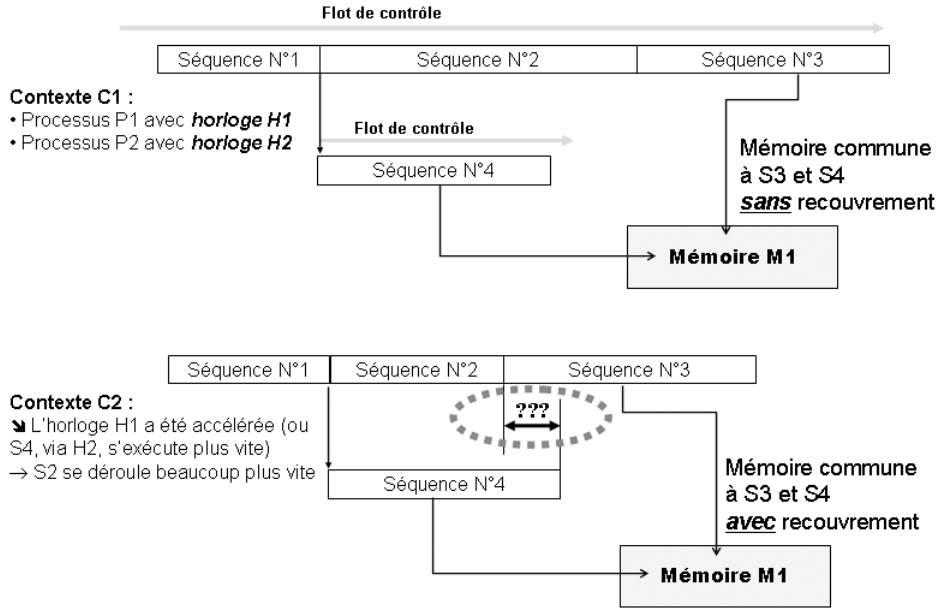


Figure 13.7 - Conflit d'accès sur la mémoire.

Le schéma montre l'émergence d'un conflit d'accès résultant d'une variation de vitesse relative des séquences de code S2 et S4 qui révèle un conflit d'accès potentiel entre les séquences S3 et S4. Si le programmeur de S3 ne sait pas qu'il peut entrer en conflit d'accès avec S4, il n'a pas de raison de se prémunir contre quelque chose qu'il ignore ; idem pour le programmeur de S4. Supposons que S4 fasse des entrées-sorties disques et/ou réseau dont la performance se dégrade avec le temps (cf. les phénomènes d'« usure », dans *Puissance et limites des systèmes informatisés*), par exemple à cause d'une saturation temporaire. Conséquence, les données M1 deviennent incohérentes. Si on réexécute avec un nouveau contexte où la saturation a disparu, l'incohérence des données aura également disparu. La défaillance n'est pas reproductible, sauf à l'observer « in-vivo » lors de la première exécution, avec des moyens d'instrumentation et d'historisation ad hoc.

Les architectes qui ont fait les premiers systèmes conversationnels (time-sharing, puis OLTP + SGBD), confrontés à des problèmes de partage de ressources massif,

avaient appelé ces types de défaillances redoutées des «heisenbugs », en référence aux relations d'incertitudes et de non-observabilité que le physicien W.Heisenberg avait découvertes dans les années 20s. Les autres défaillances se voyant qualifiées de « bohrbugs », en référence à l'atome de Bohr, beaucoup plus sympathique, présenté comme un petit système planétaire où tout est déterministe.

Avec le développement des systèmes distribués, du traitement en ligne (OLTP, OLAP, Webservices, SOA, etc.), de l'interoperabilité (EAI, ESB/Entreprise Service Bus, etc.), associés à de mauvaises pratiques de programmation, les défaillances de ce type ont aujourd'hui tendance à proliférer, ce qui fait que la courbe de maturité d'un intégrat ou d'un système doit être présentée comme suit (figure 13.8) :

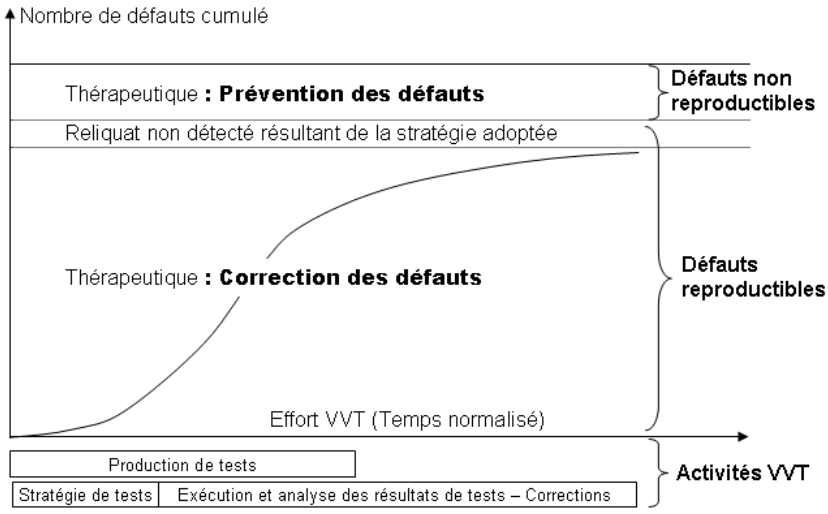


Figure 13.8 - Courbe de maturité et défauts résiduels

Cette courbe de maturité est la somme des trois courbes de la figure 13.5. Ce qui est important dans cette somme, c'est l'asymptote qui est la somme exacte des trois asymptotes résultant de la stratégie de test adoptée. Toute stratégie laisse un reliquat de défauts que ne peut pas révéler la stratégie adoptée et qui correspond à ce que B. Beizer qualifie de : « Test suites wear out »[3] que l'on peut traduire par : la capacité de découverte de défauts d'un ensemble de test s'épuise ; et le corollaire : changer de stratégie dès que l'on approche l'asymptote.

Les tests unitaires des intégrats ne peuvent pas, par définition, trouver les défauts résultant d'interprétations divergentes des interfaces entre les intégrats. Du point de vue des développeurs, la relation d'un intégrat ITG_x avec son environnement appelant-appelé peut être représenté comme suit (figure 13.9) :

3. Cf. page 58, in *Software testing techniques*.

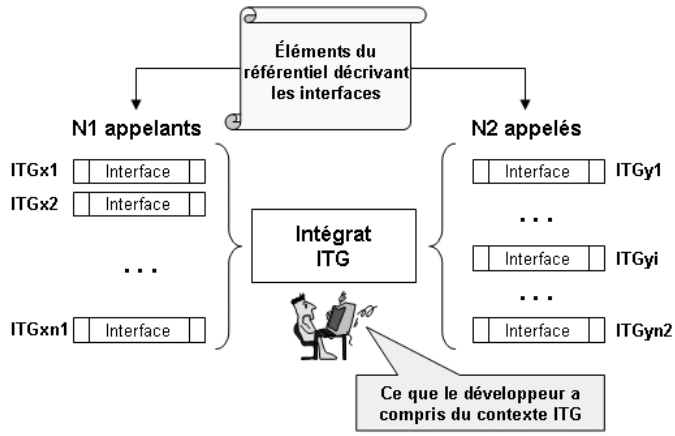


Figure 13.9 - Relation Appelant – Appelé

Les interfaces appelant-appelé sont des éléments du référentiel d'architecture qui matérialisent le contrat passé entre les intégrats. Le degré de formalisation du contrat, conformément à l'état de l'intégrat dans le cycle de développement du système (cf. Tableau 2.3) est un facteur critique pour l'efficacité de l'acteur test.

La stratégie de test unitaire de l'intégrat (qui peut lui-même résulter d'une intégration préalable d'intégrats de rang inférieur), concerne les structures internes de l'intégrat ITG (et de ses interfaces internes). La stratégie d'intégration de ITG dans son contexte appelant-appelé concerne la VVT de tous les contrats d'interfaces appelant-appelé, résultant de la dynamique des enchaînements des intégrats prédécesseurs et successeurs de ITG dans les chaînes d'appels.

À tous les niveaux du processus de test, l'architecture est omniprésente. Pour remonter dans le temps, l'acteur test et le programmeur doivent disposer de fonctions inverses de celles qui ont effectué la transformation. De deux choses l'une :

- ou bien tout est fait « à la main » sur la base des observations disponibles,
- ou bien l'architecture fournit une assistante pour les calculer.

Dans le premier cas, on s'en remet à la compétence et à l'expérience du couple programmeur/testeur. Dans le second cas, c'est l'architecte qui vient au secours de ses deux collègues, via les consignes de programmation résultant de l'architecture adoptée. Dans le chapitre 2, section 2.2, Nature sémantique des constructions informatiques, nous avons explicité différents niveaux de vision résultant de l'architecture adoptée. Le rôle de l'architecture est ici parfaitement clair. C'est l'architecture qui permet d'élaborer un cadre de travail, via l'organigramme des tâches à effectuer, cohérent avec la nature du système et des équipements qui le constituent. C'est grâce à ce travail que la complexité à gérer peut être ramenée à un niveau raisonnable, Voir, dans le chapitre 12, Simplicité, la partie consacrée aux architectures hiérarchiques.

13.2 NOTION D'INTÉGRAT TESTABLE - TESTABILITÉ

En référence à la notion d'intégrat introduite au chapitre 8, nous dirons qu'un intégrat est testable si sa structure et son organisation améliorent la rentabilité de l'effort de test, c'est-à-dire une capacité à :

- minimiser le volume de test (i.e. moins de test ; cf. l'indicateur de complexité ICC),
- observer et détecter au plus près du défaut résultant d'une erreur humaine, l'occurrence de la défaillance (minimiser le temps de latence),
- corriger et réparer l'intégrat fautif, en minimisant l'impact sur les intégrats appelants/appelés et sur le volume de test nécessaire à la non-régression de l'intégrat (cf. la gestion de configuration).

Notons qu'un intégrat testable ne devrait jamais produire de situations du type de celles de la figure 13.7, génératrices de défaillances non reproductibles (i.e. un temps de réparation infini !), ce qui oblige l'architecte à une vigilance accrue pour tout ce qui concerne la gestion des ressources nécessaires à l'exécution de l'intégrat.

En intégrant les figures 13.3 et 13.6 on peut présenter le schéma d'amélioration de la disponibilité comme suit (figure 13.10) :

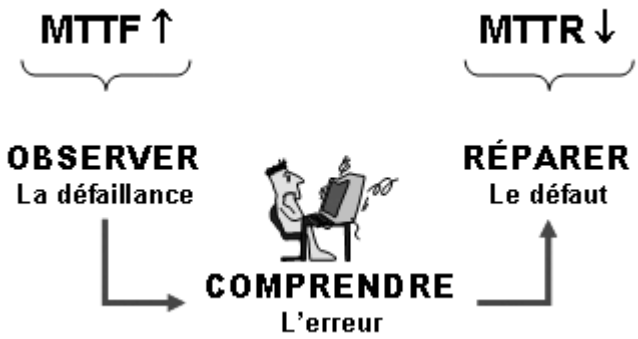


Figure 13.10 - Améliorer la disponibilité

La qualité de l'observation et la pertinence des informations résultant de l'observation sont l'élément clé de ce qu'IBM, dans son livre blanc sur l'*Autonomic Computing*, appelle la boucle de calcul autonome.

En appliquant aux intégrats schématisés dans les figures 8.8, 8.9 et 8.12, les principes qualité du modèle VEST (cf. figure 8.1) on peut représenter un intégrat testable comme suit (figure 13.11) :

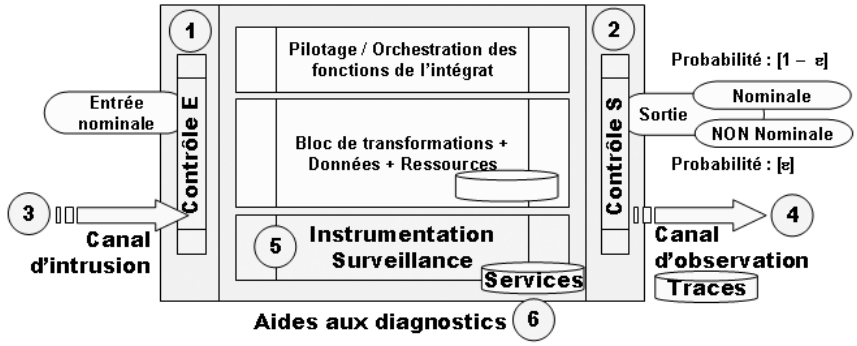


Figure 13.11 - Intégrat testable

Par rapport au schéma général, la notion de testabilité introduit six éléments architecturaux nouveaux qui, d'un point de vue théorie de l'information, sont des redondances, en particulier :

- Deux organes de contrôles pour vérifier/valider tout ce qui entre et tout ce qui sort, i.e. les contrats négociés entre l'intégrat et les appelants/appelés qui doivent être enregistrés dans le référentiel.
- un canal d'intrusion à partir duquel, via l'organe de contrôle, on va pouvoir activer les instruments d'observation.
- un canal d'observation à partir duquel on va pouvoir extraire les observations effectuées et les archiver dans un fichier journal.
- un organe d'instrumentation et de surveillance, permettant de vérifier et valider les états mémoires (contrôle de la cohérence syntaxique et sémantique) consécutivement aux transformations effectuées.
- Des services pour faciliter l'aide aux diagnostics.

Dans un monde parfait sans erreur, il n'y aurait nul besoin de ces éléments additionnels. Mais si ces informations n'existent pas, il faudra les reconstruire, au gré des acteurs en charge des tests, et dans le plus grand désordre qui est le destin des fins de projets, généralement au détriment de la qualité. Mieux vaut donc les installer à demeure dans le produit fabriqué, conformément à ce que font toutes les autres branches de l'ingénierie (i.e. le « *design for test* »).

Le rôle de l'architecte est de définir rigoureusement le pattern correspondant de façon à permettre aux intégrateurs de concentrer leurs efforts sur le diagnostic et non pas sur la désintrication des états résultant d'une programmation sans rigueur où les différents aspects s'enchevêtrent.

Pour bien comprendre la nécessité de définir des intégrats testables il est commode de revenir à la notion de « panneau de commande » telle que les programmeurs système l'ont connu et pratiqué avec les premiers ordinateurs. Le panneau de

commande permet de surveiller tout ce qui se passe dans la machine, grâce à différents modes de surveillance qui font partie des fonctionnalités de base :

- Mode **pas à pas**, qui permet de reprendre la main après chaque instruction exécutée. Ce mode de contrôle avait donné naissance à une instruction dont disposaient certaines machines, comme les UNIVAC de la série 1100 (aujourd'hui UNISYS), appelée « `EXEC, adresse` » l'adresse pointant un mot mémoire contenant une instruction machine. C'était bien commode pour réaliser des programmes interpréteurs. Le mode pas à pas présuppose l'indivisibilité des instructions (propriété **A** et **I** de ACID), d'où, à nouveau, l'importance de la notion de transaction, faute de quoi l'état mémoire est incompréhensible.
- Mode **arrêt sur adresse**, soit adresse d'instructions, soit adresse de données ; ce qui permettait aux programmeurs de demander un arrêt prévu à l'avance, soit sur une instruction particulière, soit sur une donnée particulière. C'était indispensable pour la mise au point des programmes avec des données partagées car le programmeur pouvait ainsi savoir quels programmes accédaient à la donnée en produisant un événement (appelé « *trap* » dans le jargon) associé à la donnée. NB : Cf. la notion de « trigger » disponible dans certains SGBD, qui est la même chose, mais gérée par le moniteur d'accès aux données du SGBD.
- Mode **arrêt sur un événement**, selon la nature des événements gérés par la machine, en particulier ceux concernant les ports d'entrées et de sorties de la machine, l'accès aux ressources, ce qui permettait de disposer d'une photo instantanée (dans le jargon, on disait « *snapshot* ») de la dynamique événementielle et d'analyser l'état des files d'attente associées aux événements.
- Mode **diagnostic** permettant d'exécuter, à la demande (par exemple à chaque appel de procédure, à chaque commutation de processus, etc.), tel ou tel programme d'aide aux diagnostics (OLTD) prévu à l'avance, pour vérifier la cohérence de l'état courant de la machine.

Un debugger symbolique, tel qu'on en trouve dans la plupart des environnements de programmation, ne fait que simuler partiellement ce qui était possible sur le tableau de commande. D'une certaine façon, il définit un panneau de commande virtuel, mis à la disposition du programmeur, pour tel ou tel langage de programmation. Les interfaces correspondantes ne font pas partie du langage, et de ce fait ne sont pas standardisées.

Un intégrat testable dispose de six éléments additionnels indiqués sur la figure 13.11. La bibliothèque de services d'aides aux diagnostics matérialise l'effort de tests associé à l'intégrat (cf. la figure 12.4) pour garantir le contrat de service de l'intégrat.

Par rapport au modèle d'intégrat proposé au chapitre 8, on voit que le mode pas à pas correspond à la surveillance du monitoring de l'intégrat (cf. figure 8.12, rendu possible par une application stricte du principe de modularité (cf. figures 8.11 et 9.12).

Du point de vue du moniteur l'intégrat joue le rôle d'« instruction » généralisée, et l'on comprend bien la nécessité de définir des intégrats avec un point d'entrée et un point de sortie, ce que garantit le modèle de programmation transactionnelle. On peut ainsi clairement séparer la logique d'enchaînement des intégrats, de la logique d'enchaînement des constituants de l'intégrat, ce qui va faciliter la recherche dans les différentes traces en cas de défaillance de l'intégrat. L'application des propriétés d'atomicité et d'isolation, le **A** et le **I** de ACID, garantit par ailleurs qu'il n'y aura pas enchevêtrement : a) des états mémoires communs à plusieurs intégrats, et b) des états mémoires internes à l'intégrat ITG_X qui ne concernent que lui seul. Toute violation des règles de modularité revient à entrelacer des flots qui n'ont rien à voir entre eux et à enchevêtrer des états qui ne devraient pas l'être, donc à augmenter la quantité d'information que le testeur aura à gérer, donc à augmenter le temps de recherche de la « bonne » information.

Le mode arrêt sur données ne peut être rendu possible que par un monitoring des données conformément au modèle CRUD présenté au chapitre 10, ce qui sera le cas pour toutes les données gérées via un SGBD et/ou un SGF (données persistantes). Le moniteur d'accès aux données constitue un point d'observation privilégié pour filtrer les requêtes d'accès et garantir la sécurité (droit d'accès, intégrité).

Pour les données non persistantes que le programme crée statiquement ou dynamiquement, on voit qu'il y a intérêt à disposer d'un service qui assure de façon centralisée la gestion de ce type de données. L'état de la mémoire des données non persistantes est l'état de la mémoire géré par ce service. Si l'architecte laisse les programmeurs gérer eux-mêmes la mémoire, il y a tout lieu de penser que les programmes seront saupoudrés d'ordres d'allocation dont la prise en compte dépend des enchaînements et du contexte d'exécution. Par construction un tel état est non reproductible d'une exécution à l'autre, ouvrant ainsi la possibilité de faire apparaître des défaillances elles-mêmes non reproductibles. Ce style de programmation est à proscrire ; c'est à l'architecte d'édicter la règle de programmation correspondante et à s'assurer (via le système qualité mis en place) qu'elle est effectivement appliquée par les programmeurs.

Du point de vue de l'intégrateur la situation est la suivante (figure 13.12) :

Dans le cas 1, l'état des données non persistantes est inobservables, du point de vue de l'intégrateur car celui-ci n'a pas (ne doit pas avoir) accès à la structure interne de l'intégrat qui pour lui est une boîte noire. L'état des allocations dépend de l'intégrat et de l'environnement ; il est, par construction, non reproductible d'une exécution à l'autre.

Dans le cas 2, le service d'allocation a lui-même une structure d'intégrat testable, avec un point d'entrée et un point de sortie. L'intégrateur peut demander à « voir » l'état interne de la mémoire allouée ; il suffit pour cela que le service soit conçu conformément au pattern CRUD. On remarquera que les tests de performance, les tests de robustesse qui peuvent être fabriqués concernent le service d'allocation lui-même, mais que si ce service n'existe pas, les tests non plus n'existeront pas ! Le cas 1 est un bon exemple de programme non testable que l'architecte devra proscrire.

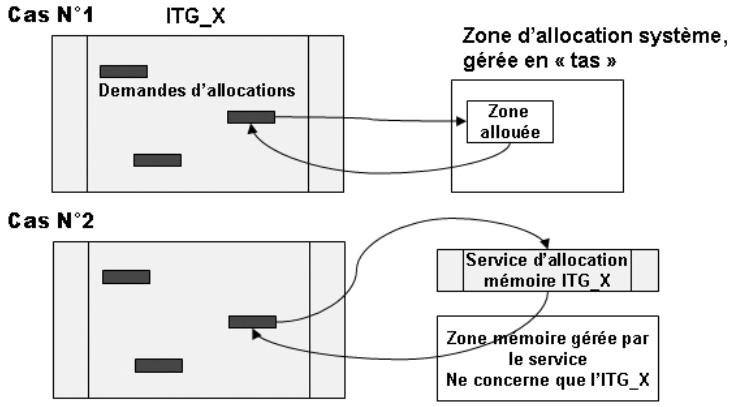


Figure 13.12 - Ordre et/ou désordre de la mémoire

On notera que dans le cas 2, on peut pousser la testabilité aussi loin que nécessaire. Supposons, par exemple, que pour être valide, du point de vue de ITG_X, l'allocation doit avoir une durée < *x* milli-secondes. Il est tout à fait possible, par scrutation des timers du processus d'allocation de mesurer cette durée et de la restituer à l'appelant de façon systématique (cf. figure 7.2). C'est quasiment impossible à faire dans le cas 1, et de toute façon non prévue dans les ordres d'allocation mémoire intégrés aux langages de programmation.

Concernant les contrôles E et S, ceux-ci vont résulter des descriptions de contraintes associées aux interfaces. S'il s'agit de messages (cf. les figures 10.19 et 10.20), on a toute latitude pour doser exactement ce qu'il y a lieu de faire compte tenu de l'exigence de disponibilité applicable à l'intégrat. La violation des contraintes provoquera l'émission de messages d'auto-diagnostic qui seront édités via le canal d'observation dans le fichier traces. Le format de ces messages devra être standardisé au plus haut niveau possible du système, par exemple sous la forme d'une DTD XML que chaque intégrat devra appliquer, quel que soit son rang.

Dans une approche méthodologique comme le TDD (*Test Driven Development*) préconisée par l'X-programming et certaines méthodes agiles, la notion d'intégrat testable permet de bien répartir les rôles entre le programmeur et le testeur qui forme une paire indissociable (en théorie, du moins). Celui qui a le rôle du programmeur programme le code nominal de l'intégrat. Celui qui a le rôle du testeur programme les six organes additionnels du pattern intégrat testable introduit dans la figure 13.11 et la bibliothèque de services qui contient les programmes de test.

Dans cette approche de la testabilité on constate que tester et programmer, c'est la même chose, quoiqu'avec des points de vue différents. L'indicateur de complexité/complication ICC introduit au chapitre 12 prend tout son sens, car tout est programmation y compris l'activité de test. Optimiser l'ICC est un « jeu », au sens de la théorie des jeux dont la stratégie (règles du jeu) doit être définie par l'architecte.

13.3 RECONSTRUIRE L'HISTOIRE D'UNE DÉFAILLANCE

Lorsqu'une défaillance se manifeste dans un intégrat ITG_X, cela signifie qu'il existe une incohérence entre le code et les données, et/ou que l'environnement a cessé d'être nominal. Le problème est de remonter à la cause de la défaillance et d'identifier le ou les intégrats qui n'ont pas rempli leur contrat.

Il y a une triple analyse à effectuer (cf. figure 13.1) et un triple questionnement :

- Quels sont les prédécesseurs de l'ITG_X, dans le graphe des appelants/appelés, y compris les composantes non connexes ?
- Quels sont les ITG qui ont eu accès aux données Dx et les ont modifiées, soit les C, U et D de CRUD ?
- Quels sont les ITG qui ont modifié l'environnement et consommé des ressources partagées ?

13.3.1 Analyse des prédécesseurs

Cette analyse dépend de la façon dont l'exécution est monitorée et de la granularité de l'observation de l'exécution.

- Au niveau le plus fin ont peu disposer de la trace d'exécution des instructions sources, et des données modifiées (en général, une par instruction) ; ce qui peut conduire à une trace volumineuse rapidement inexploitable.
- Au niveau modules et/ou transactions regroupant un ensemble de traitements sémantiquement apparentés la taille de la trace diminue mais la taille du vecteur des données modifiées augmente. Et ainsi de suite, jusqu'au niveau des composants applicatifs qui peuvent regrouper quelques dizaines de milliers de lignes de code source. La précision de l'observation dépend du niveau de granularité du monitoring.

À chaque niveau de trace, il faut disposer de suffisamment d'information pour analyser la cohérence des données « photographiées » avec la trace (cf. les figures 8.12 et 8.13).

Cette analyse est facilitée par la mise à disposition dans le référentiel système du graphe des appelants/appelés, des matrices N2 et de tout ce qui concourt à tracer les chemins d'exécution, sachant que certains chemins ne peuvent être matérialisés que dynamiquement (programmation par événements).

13.3.2 Analyse des données

Il faut cette fois retrouver tous les ITG susceptibles d'avoir modifié la ou les données jugées incohérentes. Dans une optique CRUD, cela revient à s'intéresser aux ITG qui ont fait les opérations CUD sur les données (le R est neutre, de ce point de vue), ce qui ouvre la possibilité d'analyser de nouveau chemins et de nouvelles données.

Les outils de références croisées {fonctions/services×données} (gestion de configuration, dictionnaire de données) sont indispensables pour effectuer ces analyses.

Le combinatoire engendrée par l'enchevêtrement du code et des données devient rapidement énorme, ce qui fait que si le temps de latence est long, il n'y a aucune chance de remonter facilement à la cause de la défaillance.

D'une façon générale, cette combinatoire résulte des modes de couplages occasionnés par le style de programmation adopté. Là encore, c'est de la responsabilité de l'architecte de s'assurer que le style de programmation n'engendre pas de couplages cachés qui seront autant de situation à risques.

Pour réduire le temps de latence (propriété dite « *fail-fast* » dans la programmation transactionnelle), il est indispensable d'effectuer périodiquement des contrôles de cohérence sur les données en entrée et en sortie de l'intégrat. Ceci n'est faisable que si les intégrats respectent strictement le principe de modularité (cf. figure 8.11). Ces contrôles doivent être installés à demeure dans le code, mais ne s'exécuter que sur la demande expresse de l'intégrateur (mode d'exécution en diagnostic contrôlé).

13.3.3 Analyse de l'environnement

Il peut arriver que la défaillance ait comme cause une perturbation de l'environnement d'exécution occasionnée par le partage des ressources de la plate-forme d'exécution de l'ITG. Par exemple, un ITG qui n'a rien à voir fonctionnellement avec l'ITG X où se manifeste la défaillance, a saturé la mémoire, ce qui va perturber le comportement de l'ITG_X et le faire sortir de l'état nominal où son bon fonctionnement est garanti.

Dans ce cas, il n'y a pas d'autre alternative que de surveiller l'environnement avec des outillages ad hoc (Cf. le *capacity planning*, *system management*, entre autres), soit au niveau plate-forme, soit au niveau du composant applicatif.

Au niveau plate-forme, il peut être difficile de faire la relation entre une ressource particulière (par exemple un disque ou une ligne de télé-communication) et un composant applicatif particulier.

Au niveau composant applicatif, il peut être difficile d'accéder aux informations gérées par le système d'exploitation qui requiert des modes d'exécution privilégiée, sauf si le système d'exploitation dispose d'interfaces standard d'accès à ces informations. C'est tout l'enjeu de démarches comme l'*autonomic computing* dont IBM s'est fait le promoteur.

On voit que la reconstitution de l'histoire d'une défaillance requiert un grand nombre d'informations que l'on a tout intérêt à expliciter dans l'intégrat lui-même, faute de quoi cette information sera perdue d'une intégration à la suivante. Une saine économie de la gestion de la disponibilité exige que tout cela soit fait explicitement, sous contrôle de l'architecte, et programmé avec les mêmes standards qualité que le code fonctionnel pour l'usager. C'est la meilleure garantie de pérennité de

cet investissement qui pèse couramment 15 à 20% du coût de projets. D'un point de vue qualité, c'est un coût qualité, au bon sens du terme.

Si tout cela n'a pas été organisé correctement, il s'en suivra une grande disparité d'un intégrat à l'autre, selon l'humeur des programmeurs, avec au final, lors de l'intégration, des temps de diagnostic prohibitif dont la seule cause sera le laxisme des concepteurs/programmeurs livrés à eux-mêmes sans directive claire de l'architecte. C'est l'intégration qui « paiera » la note d'une situation dont elle n'est pas responsable, sauf à rendre l'architecte garant de la performance de l'équipe d'intégration. Notons que dans le BTP il existe une garantie décennale qui rend l'architecte responsable des malfaçons éventuelles, pour une durée de 10 ans. A quand une telle garantie dans les projets informatiques ?

14

Adaptabilité – évolutivité

14.1 INTRODUCTION

Les propriétés d'adaptabilité et d'évolutivité caractérisent la capacité d'un programme à prendre en compte de nouvelles exigences, de nouveaux besoins, à remplacer et/ou intégrer de nouveaux équipements, à corriger facilement les défauts, etc. En bref il s'agit de minimiser, en terme CQFD, la prise en compte de toutes les modifications petites ou grandes susceptibles d'apparaître durant la vie du programme. Ces propriétés sont essentielles pour garantir une bonne efficacité des équipes en charge de la maintenance, période souvent beaucoup plus longue que le développement proprement dit. B.Boehm a souvent dit que pour 1$ investit en développement, il faut en prévoir 2 pour le MCO, d'où l'importance économique de ces propriétés.

Du point de vue de l'architecte, cela revient à anticiper tout ce qui est susceptible de changer dans la vie du système qui intègre le logiciel, tant au niveau du monde métier M1, qu'au niveau du monde informatisé M2 et des programmes. Ce qui fait la caractéristique essentielle du software par rapport au hardware est précisément son côté « mou » et adaptable. L'adaptabilité est de ce fait un acte majeur de l'architecte pour tirer le meilleur parti de cette matière informationnelle si « facile » à modifier qu'est le logiciel, relativement aux éléments matériels du système.

Concrètement, une adaptation se traduit toujours par des opérations conjointes sur le texte source et les tests du programme à adapter : modification, suppression et ajout de code source et/ou de tests. L'adaptation peut être diffuse, un peu partout dans le programme et dans les tests, ou au contraire être précisément localisée. Le coût de l'adaptation en termes CQFD dépend de l'architecture du programme et du style de programmation. Idéalement ce coût ne devrait dépendre que de l'adaptation proprement dite mais on peut facilement comprendre que les relations de toute nature qui peuvent exister avec le reste du programme nécessitent un travail de véri-

fication et de validation qui lui est dépendant de la taille du programme à adapter, et potentiellement de l'environnement d'exploitation. À côté du code source et des tests, l'ensemble du référentiel devra également être mis à jour.

14.2 ADAPTABILITÉ DU POINT DE VUE DES MÉTIERS ET DE LA MAÎTRISE D'OUVRAGE

Du point de vue métier, le besoin d'adaptabilité du SI global vient toujours des objectifs stratégiques fixés par la direction générale de l'entreprise aux différentes organisations métiers. Les motivations pour adapter le SI sont multiples :

- À partir de l'existant du SI, créer de nouvelles chaînes de valeur, modifier et/ou supprimer des chaînes de valeurs existantes.
- Echanger des informations et/ou des savoir-faire, mettre en commun des moyens informationnels avec des partenaires (cf. BD géologiques des pétroliers) et/ou lors de fusions-acquisitions.
- Ouvrir tout ou partie du SI au grand public et/ou aux clients de l'entreprise via les portails Internet et le Web.
- Faciliter les échanges de données informatisées de toute nature avec les fournisseurs afin d'accélérer les cycles de conception-production de nouveaux biens et/ou services.
- Dématérialiser certaines fonctions de l'entreprise comme les bureaux d'études grâce aux technologies de simulation (cf. bureaux d'études virtuels chez les avionneurs).
- Tirer parti le plus rapidement possible des innovations technologiques aux bénéfices des métiers existants et/ou pour la création de nouveaux métiers (cf. le cas de la téléphonie mobile qui est exemplaire).

Derrière chacun de ces thèmes, il s'agit d'adapter le SI de l'entreprise aux besoins suscités par l'environnement, ce que nous avons résumé par le sigle PESTEL, et ceci avec un objectif général d'améliorer la performance globale de l'entreprise.

On peut faire rentrer toute cette problématique dans deux schémas généraux d'interopérabilité :

- Mise en cohérence de plusieurs systèmes développés indépendamment mais partageant une problématique commune (y compris un composant applicatif métier intégrant déjà un progiciel métier) en vue de rationaliser le SI.
- Intégration d'un nouveau système dans le SI de l'entreprise, soit une nouvelle version d'un système et /ou d'un composant applicatif existant, soit un progiciel métier.

Dans tous les cas de figure, il s'agit pour la maîtrise d'ouvrage de définir des projets cohérents avec l'ensemble du SI, en respectant les contraintes économiques de l'entreprise en termes CQFD, TCO et de risques.

Dans le **premier cas**, il s'agit de fédérer n systèmes et/ou composants applicatifs. La démarche d'obtention de projets cohérents pour réaliser cette adaptation est la suivante (figure 14.1) :

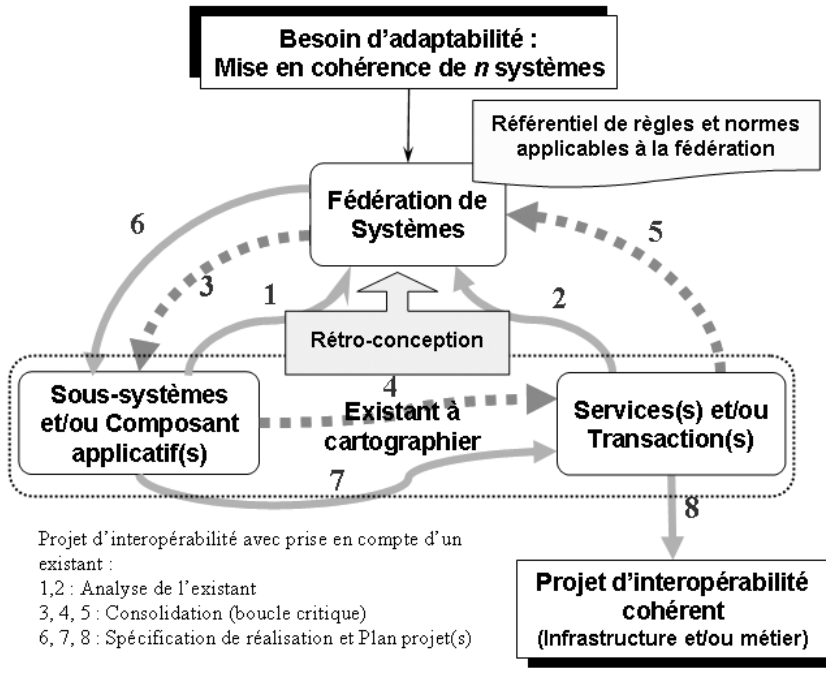


Figure 14.1 - Démarche de mise en cohérence d'une fédération de systèmes

Dans cette démarche il s'agit de faire l'inventaire de l'existant en termes de composants applicatifs, c'est-à-dire à cartographier cet existant, et de définir précisément la cible que l'on cherche à atteindre, i.e. cartographie cible. Les éléments factorisés vont constituer le référentiel de la fédération, i.e. du système de systèmes qui constitue le SI de l'entreprise. Les projets qui en résultent déterminent la trajectoire à suivre pour passer de l'existant à la cible, en garantissant la cohérence via le référentiel de la fédération.

On remarquera sur le schéma que le référentiel de la fédération s'obtient en deux étapes :

- Une première étape résulte d'une rétro conception (« *re-factoring* », dans le jargon) des composants applicatifs en vue d'identifier le modèle commun (cf. le chapitre 10).

- Une seconde étape, après la mise en cohérence du référentiel de la fédération consiste à revisiter les référentiels spécifiques de chacun des composants applicatifs, afin d'évaluer les adaptations à réaliser et de trouver le meilleur compromis CQFD/TCO entre le niveau fédération et le niveau applicatif. De cette seconde itération résultera le schéma d'adaptation définitif.

Dans le **second cas**, il existe une infrastructure avec un référentiel formalisé et l'on se propose d'intégrer un nouveau système au SI de l'entreprise. Pour être intégrable, le nouveau système doit s'adapter aux contraintes du référentiel existant. La situation, plus simple que le premier cas, est la suivante (figure 14.2) :

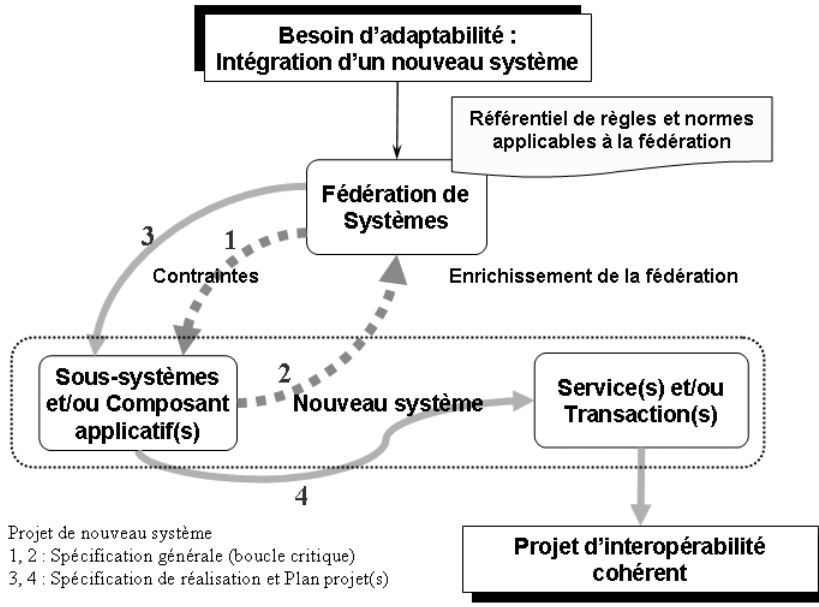


Figure 14.2 - Intégration d'un nouveau système à une fédération de systèmes

Le nouveau système hérite d'un certain nombre de capacités provenant de ce qui existe déjà (d'où économie d'échelle). En échange, il enrichit la fédération de capacités qui lui sont propres, dans un premier temps, et qui pourront être utilisées par d'autres systèmes de la fédération au fur et à mesure des évolutions.

L'adaptabilité, du point de vue métier, va se traduire par une évolution de l'urbanisation du SI de l'entreprise qu'il faudra gérer comme telle, en respectant autant que faire se peut la compatibilité avec ce qui existe déjà. Des analyses architecturales assez fouillées peuvent être faites à ce stade, indépendamment de toute solution informatique.

Par exemple, de nombreux systèmes d'information embarquent en leur sein des pans entiers de législation tant au niveau des données que des procédures de traite-

ments (cf. le code des impôts, le code du travail, la loi informatique et liberté, la régulation des télécommunications [l'ART] ou de l'énergie [la CRE], etc.). Les informations correspondantes doivent évoluer au rythme de la législation ; il convient de bien les distinguer des autres informations qui les utilisent mais qui sont sur des rythmes différents.

C'est un des aspects de la sémantique des données dont il faut se soucier dès l'analyse du besoin et des modèles fonctionnels associés (cf. chapitre 2).

De même, l'ergonomie du poste de travail est une contrainte métier forte qui par bien des aspects conditionne l'efficacité des acteurs métiers. On en a un exemple frappant avec les systèmes de contrôle-commande (cf. chapitre 10, section 10.4) qui permettent à des opérateurs de piloter des processus complexes comme le contrôle aérien, l'équilibre offre-demande dans le transport de l'énergie, le pilotage d'une centrale nucléaire, etc.

Tous ces systèmes utilisent aujourd'hui des GUI sophistiqués qui ont été définis par le métier au fil des ans et que la capacité des matériels informatiques permet d'informatiser. La présentation de l'information doit prendre en compte les aspects cognitifs, indépendamment de la logique de traitement qui, elle, va dépendre des capacités des plates-formes informatiques.

Il est fondamental de bien distinguer ces deux logiques pour les adapter, si besoin est, indépendamment l'une de l'autre. Les SI géographiques, ou les SI ayant une composante géographique forte comme les systèmes de gestion de crises, font un usage intensif de la symbologie définie par les cartographes depuis plusieurs siècles.

Il est fondamental de prendre ce type de contraintes au sérieux si l'on ne veut pas s'exposer à des risques de rejet du système par les usagers, ou à des coûts de conduite du changement, accompagnés de risques aggravés d'erreurs humaines. La même remarque vaut pour les systèmes de santé où les acteurs médecins, infirmières et soignants ont des habitudes et des réflexes qu'il ne faut pas rompre.

14.3 ADAPTABILITÉ DU POINT DE VUE DE L'ARCHITECTE

Une fois le besoin d'adaptabilité défini du point de vue de la maîtrise d'ouvrage, le travail de l'architecte va consister à traduire ce besoin en termes opérationnel du point de vue des équipes projets. Cette traduction peut conduire à réviser le besoin initial compte tenu des contraintes spécifiques à l'architecture et aux projets (complexité, maturité des acteurs projets, disponibilité des ressources, plates-formes d'exploitation, etc.).

Le besoin d'adaptabilité n'est que la partie visible, du point de vue des métiers, de l'iceberg architectural que l'on peut représenter comme suit (figure 14.3) :

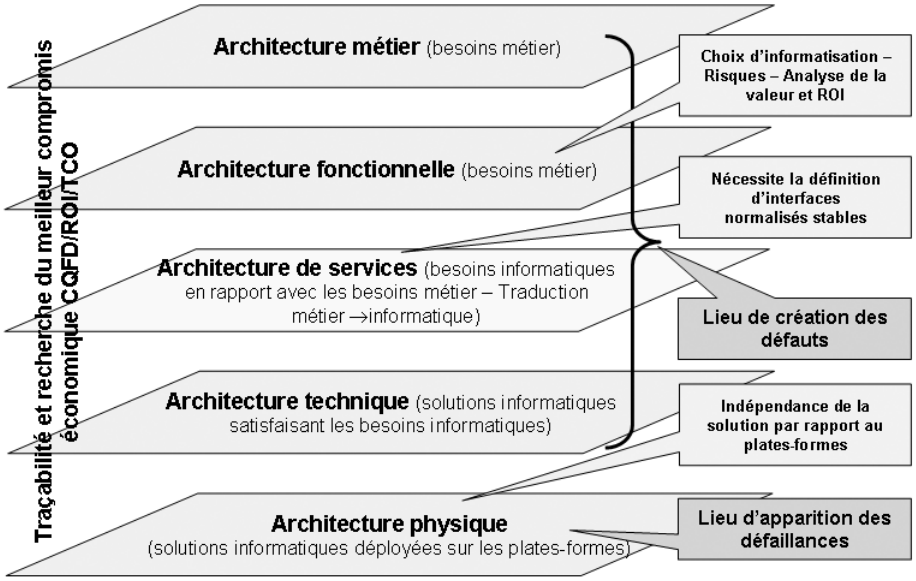


Figure 14.3 - Les différentes vues de l'architecture

Dans le chapitre 2 (cf. Tableau 2.3) nous avons utilisé la terminologie « phase » ou « changement d'état »[1] pour caractériser les transformations successives que subissent les modèles, jusqu'à obtention du code et des tests qui matérialisent les livrables finals du projet. L'adaptabilité de l'architecture revient à manipuler globalement cette structure à l'aide de règles dont les modalités d'emploi constituent le méta-modèle du système. Cette terminologie a été adoptée pour caractériser d'une part le modèle du système et/ou du/des composants applicatifs qui le constituent, et d'autre part le méta-modèle qui fixe les règles auxquelles sont astreints les différents textes du modèle, code source et tests inclus.

Cette distinction, classique en linguistique ou en logique mathématique, mérite d'être bien comprise pour ne pas ouvrir une nouvelle boîte de Pandore informatique, car les confondre ouvre la porte à des paradoxes logiques qui, en informatique, se traduiront par autant d'erreurs nouvelles.

Le rapport modèle/méta-modèle est de même nature que le rapport entre un texte français et la grammaire du français. Modifier la grammaire, créer des terminologies ambiguës ou redondantes, peut rendre le texte initial incohérent : il faut donc être prudent quand on modifie le méta-modèle.

Au chapitre 6, dans l'exemple des compilateurs, nous avons vu quelques exemples de méta-modèles qui permettent d'abstraire a) la grammaire du langage que l'on compile et b) la machine cible sur laquelle on génère le code. Compilateurs et tra-

1. Par analogie avec ce qu'en physique on appelle phase ou état solide, liquide, gazeux.

ducteurs de langages sont certainement les entités informatiques qui permettent le mieux de comprendre cette distinction fondamentale. Le méta-modèle, qui est un nom pédant pour parler du référentiel, peut lui-même être plus ou moins formalisé. S'il est rigoureusement formalisé, comme dans le cas des grammaires formelles, on peut utiliser le terme méta-modèle qui dénote alors quelque chose de précis. Dans la vie réelle, une grande partie du référentiel sera décrite de façon informelle, en français/anglais de tous les jours, ce qui ne veut pas dire sans rigueur, et une petite partie pourra utiliser des notations formelles, comme la BNF (largement utilisée et popularisée via le langage XML). Toutes les notations et/ou langages utilisés pour décrire les entités architecturales participent au méta-modèle (cf. chapitre 4).

Pour l'architecte, la situation peut être schématisée comme suit (figure 14.4) :

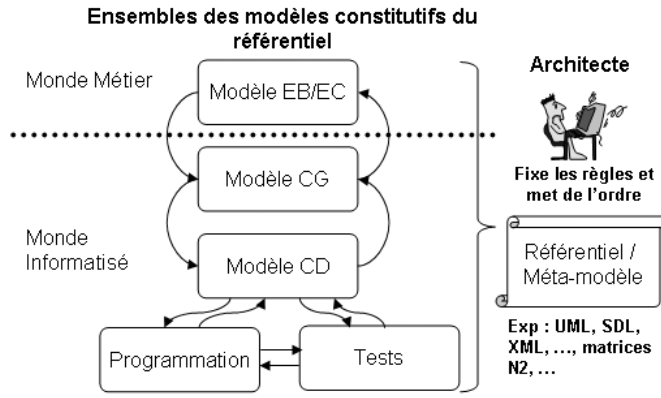


Figure 14.4 - Relation Modèles/Méta-modèle

Pour qu'une modification des modèles soit licite, elle doit se conformer aux règles du référentiel/méta-modèle. Concernant la propriété d'adaptabilité, l'architecte doit se poser deux questions :

- Question N° 1 : Qu'est ce qui bouge le plus dans la partie du métier que l'on souhaite informatisé ?
- Question N° 2 : Comment faire pour prendre en compte le plus rapidement possible l'adaptation dans la partie informatisée du SI ?

La première question exige une bonne compréhension du métier ; la seconde question exige une excellente maîtrise des technologies et des techniques informatiques, ce qui, in fine, permettra de traduire le besoin exprimé.

D'une façon générale, l'architecte est confronté à une double contrainte, d'une part les besoins d'adaptabilité suscités par les métiers, et d'autre part les besoins d'adaptabilité suscités par les évolutions des plates-formes (cf. le niveau architecture physique sur la figure 14.3). Son problème est de trouver un compromis acceptable entre les différentes contraintes CQFD/PESTEL/FURPSE/TCO.

14.3.1 Cas des adaptabilités métier

Illustrons cette problématique à l'aide de deux exemples tirés du métier.

Exemple N° 1 : un méta-modèle rudimentaire pour les tables de décisions logiques

De nombreuses applications informatiques font usage de table de décisions qui sont susceptibles de changer fréquemment, selon les aléas de l'environnement PESTEL. Une table de décisions manipule un ensemble de conditions qui permettent d'effectuer telle ou telle action, sur le modèle :

```
Si expression logique (C1, C2,...) alors action N° 1 ;
Si expression logique (...    ...) alors action N° 2 ;
Si ... etc.
```

On sait programmer ces tables dans n'importe quel langage de programmation depuis toujours, y compris COBOL qui dispose même d'une norme AFNOR NF Z 67-200, *Spécification des tables de décision employées en informatique*. La question est : faut-il les programmer ?

Si elles sont programmées, toute modification entraînera la livraison d'une nouvelle version de l'application, une intégration et un déploiement sur toutes les plates-formes ou l'application a été installée. En terme CQFD, cela peut coûter très cher.

Une autre technique consiste à transformer la table de décisions en un tableau de données qui seront transmises à un programme qui saura les interpréter.

Pour cela, il faut codifier la représentation de la table de décisions, donner à chaque condition un nom de condition et à chaque action un nom d'action de façon à ce que le programme interprète de la table puisse gérer la correspondance de ces noms avec les morceaux de code source qui leur correspondent. C'est ultra-classique pour quiconque a une connaissance élémentaire de la programmation et des design-patterns.

- Si l'on veut que la table de décisions soit compréhensible par les acteurs métiers, tous les noms doivent correspondre à des concepts métiers qui leur sont familiers.
- Si l'on veut que la table de décisions soit interprétée de façon simple, il faut remplacer tous les noms en clairs par des codes qui faciliteront le traitement informatique.

Dans le référentiel il faudra décrire cette correspondance car c'est elle qui assure la cohérence Monde 1/Monde 2, sur cet aspect particulier.

Ceci étant fait, on peut alors distribuer l'adaptation sous la forme d'un fichier de données dont seul le nom et le format auront fait l'objet d'une convention. C'est beaucoup plus simple et beaucoup moins coûteux que la livraison d'une nouvelle version de l'application.

Le coût architectural de cette facilité est un ensemble de règles qui devront être intégrées au référentiel et dont l'architecte devra s'assurer qu'elles sont effectivement respectées. Si la correspondance FORMAT MÉTIER → FORMAT INFORMATIQUE est complexe, un outil informatique pourra s'avérer nécessaire ; cet outil fera alors partie du référentiel. C'est une traduction de formats de données, comme on en a vu au chapitre 2, que l'on peut représenter comme suit (figure 14.5) :

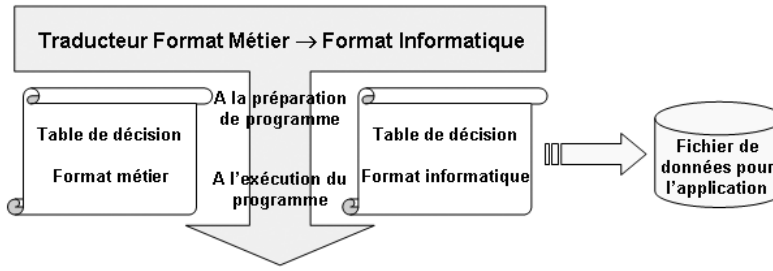


Figure 14.5 - Traduction d'une table de décision.

Sur cet exemple simple, on peut envisager deux modalités pour effectuer une telle adaptation.

- 1ère modalité : la MOE pilote l'opération. C'est elle qui fabrique le fichier et le distribue, en mettant à jour la configuration. Le métier est tributaire de la MOE.
- 2ème modalité : le métier, via la MOA, livre directement la nouvelle version de la table, comme si c'était un fichier de données dont elle a la responsabilité entière. Le traducteur de format doit être intégré à l'application (ce qui est la meilleure garantie de pérennité). La traduction se fera lors du lancement de l'application, ce qui permet une mise à jour optimale.

Dans ce deuxième cas, on a encore amélioré la capacité d'adaptation en faisant du maître d'ouvrage le pilote de l'opération. Le métier n'est plus tributaire de la MOE. Pour cela, on a complexifié l'application en lui intégrant une fonction de type outil. Le maître d'ouvrage devient un acteur de la sûreté de fonctionnement, ce qu'il n'était pas dans la modalité N° 1, avec toutes les responsabilités que cela implique (validation, vérification, test, procédures de reprises, messages d'erreurs, etc.).

Pour réaliser cette adaptation, l'architecte a inventé un mini-langage qui lui a facilité le dialogue avec les experts métiers. On peut imaginer des situations où il devra inventer différents langages, selon la nature des adaptations à réaliser et des habitudes langagières des acteurs métiers. Certains peuvent être complexes (cf. par exemple le langage idéographique des acteurs de la gestion de crise décrit dans le Mil-std 2525B, *Common warfighting symbology*, du DOD). Se pose alors le problème de la prolifération des langages pour l'adaptation et des outils de traduction associés qui peuvent complexifier l'application au-delà du raisonnable. Pour uniformiser les

notations, il faut un méta-langage universel permettant de représenter ces différents textes métiers de façon homogène. Ce langage existe depuis toujours, il s'agit de la BNF, mais on peut lui préférer la variante XML qui joue exactement le même rôle. Dans le jargon pédant et inutile de l'OMG, c'est ce que l'on appellerait un langage de méta-modélisation ou encore un méta-méta-modèle ; terminologie qui ferait certainement se retourner dans sa tombe le frère Guillaume d'Ockham. Pour l'architecte, l'agencement des différents niveaux de méta-modélisation peut se représenter comme suit (figure 14.6) :

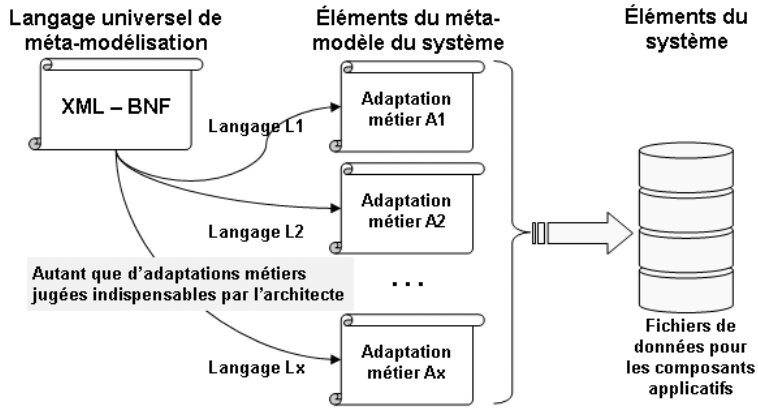


Figure 14.6 - les étapes de méta-modélisation

Devant permettre de décrire tous les langages métiers, la sémantique du tandem XML-BNF est réduite à la plus simple expression. Par contre, il y aura autant de DTD (i.e. de grammaires) que de langages métiers différents, chacun avec sa sémantique spécifique (i.e. un interprète spécifique, comme dans le cas des tables de décisions). La sémantique peut être formulée de façon impérative (i.e. une séquence d'instructions d'une certaine machine abstraite) ou de façon déclarative (i.e. une formule que l'on sait interpréter de façon non ambiguë : par exemple une formule mathématique, une sélection en SQL qui identifie un ensemble en extension, une expression de contraintes en OCL, une expression régulière pour décrire un texte simple, etc.).

Exemple N° 2 : un méta-modèle pour l'interopérabilité

Nous avons rencontré à différentes reprises le problème de l'architecture de l'interopérabilité. Dans la figure 14.1 nous devons spécifier les données communes qui s'échangent entre les systèmes de façon à construire une chaîne de traduction comme celle de la figure 10.21.

Ce qui suit n'est qu'une ébauche d'architecture restreinte aux données. Dans une architecture réelle, il faudrait prendre en compte les données, les services et les événements. Pour ce qui concerne les données, la rétro-conception, à partir de l'existant, fait apparaître les flux d'échanges (i.e. les messages échangés) et les données

constitutives de ces messages. En utilisant les langages de description de données (DDL) utilisés dans chacun des systèmes S1 et S2, on peut en déduire deux MCD spécifiques de chacun des systèmes mais sémantiquement équivalent. La démarche de reconstruction (i.e. *re-factoring*, en anglais) et illustré par le schéma 14.7.

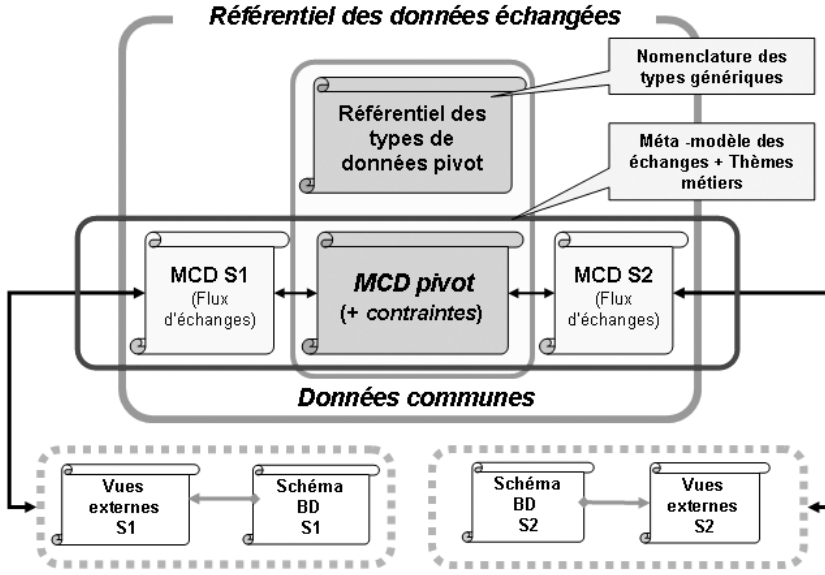


Figure 14.7 - Construction d'un référentiel d'interopérabilité

Ce qui est appelé « pivot » sur le schéma matérialise la sémantique des éléments communs aux différents systèmes de la fédération. Là encore, il est préférable de choisir un langage neutre comme XML pour représenter ce qui est commun à tous les systèmes. Le référentiel distingue les types de données génériques (i.e. le vocabulaire de base des entités données) et les entités proprement dites, avec leurs relations et leurs contraintes. C'est à partir de ce modèle pivot qu'il faut générer les éléments du système qui implémentent l'interopérabilité. Dans la figure 10.21, il s'agit des MCD d'échanges et des adaptateurs d'échanges.

La chaîne de traduction peut se représenter comme suit (figure 14.8) :

La structure de la chaîne de traduction est un choix d'architecture qui devra être expliqué comme tel dans le référentiel. Dans la figure, nous avons choisi de conserver un niveau MCD S1, indépendamment des schémas des bases de données des composants applicatifs du système S1. Le MCO S1 peut en effet servir à générer tout ou partie des schémas et/ou vues utilisés par les bases de données, mais également des séquences de codes source qui seront insérées dans des endroits ad hoc de l'applicatif

(cf. la cohérence des messages dans les figures 10.19 et 10.20). Ce niveau conceptuel concerne l'ensemble du système et non pas seulement les données.

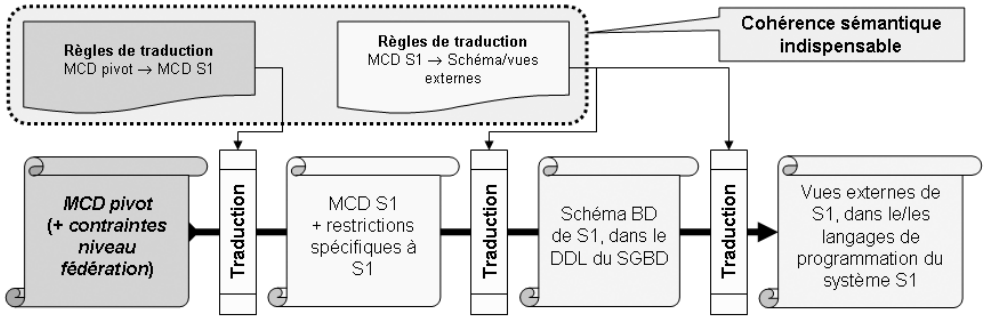


Figure 14.8 - Chaîne de traduction des modèles

14.3.2 Cas des adaptabilités aux plates-formes et aux socles techniques (architecture physique)

Il s'agit ici de s'adapter aux évolutions des plates-formes, en minimisant les coûts en termes CQFD. Là encore, tout bouge. La pérennité des interfaces des plates-formes n'est pas garantie au-delà de quelques années ; les progiciels évoluent, parfois de façons incompatibles ; des fournisseurs disparaissent, parfois sans reprise du parc par des tiers. Ce qui était relativement stable à la « belle époque » des mainframes est devenu une espèce de jungle où tous les coups sont permis. Le « *small is beautiful* » des premiers PC est devenu le cauchemar des exploitants et des DSI qui se retrouvent avec des plates-formes où les équipements se comptent par centaines ou par milliers ; disponibilité et sûreté de fonctionnement sont redevenues de vrais problèmes. Peut-on s'affranchir des perturbations correspondantes, où tout au moins les contrôler. Dans le chapitre 10, section 10.2, Architecture en couches, nous avons vu quelques-uns des dispositifs qui permettent de gérer ce type d'adaptation.

Pour illustrer le propos, prenons le cas d'un SGBD dont l'architecte a décidé de contrôler l'emploi dans l'ensemble des composants applicatifs qui l'utilisent. Il souhaite garder un pouvoir de négociation vis-à-vis de son fournisseur[2], et éventuellement changer de fournisseur. Pour cela, il va utiliser un modèle de type CRUD, comme celui de la figure 10.15.

On fait l'hypothèse que les composants applicatifs sont constitués de transactions qui peuvent s'enchaîner via un workflow qui pourrait faire lui-même l'objet d'une adaptation (cf. le cas des transactions longues, ébauchées au chapitre 9).

2. Voir le schéma classique de M. Porter, dans *Competitive Strategy, Techniques for Analyzing Industries and Competitors*, chapitre 1, Forces driving industry competition, Free Press, 1980.

L'architecture du système, avec son adaptateur, aura la structure suivante (figure 14.9) :

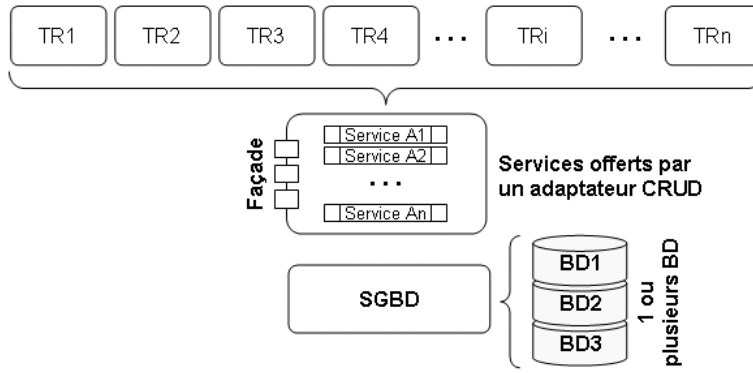


Figure 14.9 - Adaptateur CRUD pour un SGB

L'intérêt de l'adaptateur CRUD est évident :

- Les transactions TRx sont indépendantes de la structure des bases de données BD1, 2 et 3. Il suffit qu'elles connaissent les modalités d'appel du service CRUD.
- L'adaptateur peut être réalisé par une équipe qui dispose des savoir-faire et des connaissances nécessaires, d'où une productivité optimale des équipes, chacune restant dans son rôle, soit métier, soit technique.
- La combinatoire a été cassée ; l'adaptateur peut être testé et validé indépendamment des transactions, en conséquence l'indicateur de complexité ICC baisse.
- L'adaptateur est au point de passage obligé, ce qui est une propriété intéressante en matière de supervision et d'administration, ainsi qu'en terme de maturité pour la disponibilité et la sûreté de fonctionnement. Le prix à payer est la compétence de l'architecte, indispensable pour concevoir et superviser la réalisation de l'adaptateur.

Dans chacun des exemples cités (on pourrait en donner de nombreux autres) on remarquera que l'adaptabilité se traduit par le passage d'un mode d'expression impérative (une séquence d'ordres) à un mode d'expression déclarative (une structure de données, i.e. un texte structuré) qui donnera naissance à un fichier paramètre du composant applicatif ou du système. Ce passage du mode impératif (représentation en extension) au mode déclaratif (représentation en intention) est l'essence de la technologie MDA prônée par l'OMG. C'est effectivement la bonne démarche, mais elle est semée d'embûches[3], à commencer par la compétence des architectes. Pour

3. Cf. L'article de l'IBM Systems journal, cité au chapitre 12, ci-dessus.

utiliser cette technologie, il faut maîtriser la théorie des langages (grammaires, automates, langages formels) et les techniques de compilation. Si ce n'est pas le cas, mieux vaut ne pas s'y risquer, car un langage « mal ficelé », dépendant du contexte, se transforme vite en cauchemar combinatoire de cas particuliers ingérables et non testables.

15

Interfaces

15.1 INTRODUCTION

Le but de ce chapitre est d'expliquer en quoi la notion d'interface est fondamentale pour une organisation conjointe du travail du maître d'ouvrage et du/des maîtres d'œuvres des projets informatiques. C'est une première réponse à notre définition de l'architecture du point de vue projet : l'architecture est terminée quand toutes les interfaces sont définies. Les interfaces matérialisent les abstractions qui résultent du travail architectural ; elles sont le juge de paix, le tiers de confiance, qui matérialisent le contrat passé entre les acteurs. Une interface peut être un équipement comme une ligne de télécommunications avec ses protocoles, ou un ensemble de programmes purement logiciel comme dans une communication inter-processus vue au chapitre 7. Les interfaces sont des éléments de livraison de première importance qui permettent au maître d'ouvrage de suivre l'avancement du travail du maître d'œuvre durant la conception du système laquelle détermine les coûts globaux (TCO) du système. De ce fait, l'expression du besoin et les exigences en matière d'interface doivent faire l'objet d'un soin tout particulier de la part du maître d'ouvrage. Dans les standards de l'ingénierie système[1] on définit des documents à produire appelés IRS (*Interface Requirement Specification*), IDD (*Interface design document*) et ICD (*Interface control document*). De la spécification rigoureuse des interfaces va dépendre :

- Le découpage du système en sous-systèmes autonomes qui ne se connaissent que via les interfaces. Ce faisant, le développement du système pourra être organisé en tâches indépendantes. Cela facilitera l'identification de progiciels, et le remplacement éventuel de progiciels devenus périmés.

1. Cf. IEEE Std 1012, *Software verification and validation* ; IEEE Std 1220, *Application and management of the systems engineering process*.

- La simulation de certains éléments fonctionnels afin de faciliter l'intégration du système et l'automatisation des tests.
- La possibilité de modéliser des enchaînements fonctionnels afin d'étudier les performances du système, et plus généralement son comportement.
- La possibilité d'observer et d'enregistrer certains états internes du système ce qui facilitera la maintenabilité du système et améliorera la disponibilité.

Les interfaces sont des éléments stables de l'architecture sur lesquelles repose l'ensemble des composants du système et qui fondent sa solidité et sa robustesse. Elles doivent être décrites comme telle dans le référentiel du système. L'expression de besoin en matière d'interfaces est une condition nécessaire à la mise en œuvre des méthodologies d'acquisition de systèmes complexes où de nombreux acteurs sont appelés à intervenir.

Remarque : Représentation des interfaces dans les langages de modélisation.

Dans les langages évoqués au chapitre 4 (SADT/SART, IDEF, SDL, UML) les interfaces sont représentées par des traits ou des flèches, ou des variantes simples, ce qui ne rend pas compte de la richesse sémantique de la notion d'interface. Comme on va le voir, cette richesse ne peut pas se représenter visuellement de façon simple. Le symbolisme le plus élaboré est celui du langage SDL. Les matrices N2 (graphe de connexité) vues au chapitre 2 sont un aspect important des couplages induits par les interfaces.

15.2 RAPPEL SUR LA NOTION D'INTERFACE

La notion d'interface est consubstantielle à la programmation des premiers ordinateurs dans l'architecture de von Neumann. Le langage de programmation, fut-il l'assembleur des premières machines, affranchit le programmeur de la connaissance de l'intimité de la machine ; il assure une indépendance relative du mode d'expression de ce que le programmeur souhaite faire, par rapport au comment faire de l'univers matériel de la machine. Les premiers programmeurs, disons jusque dans les années 70, étaient particulièrement sensibles à cette vision des choses compte tenu des machines d'alors et des contraintes du matériel (peu de mémoire, puissance CPU faible, périphérie synchrone parfois exotique). Il y a dans la notion d'interface une dimension « langage » extrêmement forte que l'on redécouvre actuellement avec les approches MDA préconisées par l'OMG, les architectures orientées services (SOA) et le « *grid computing* »[2].

Une interface décrit les modalités d'interactions autorisées entre une entité architecturale particulière (intégrat ***appelé***) qui réalise une fonction et/ou fournit un service, et d'autres entités avec lesquelles elle est susceptible d'échanger de l'information (intégrats ***appelants***). L'aspects dynamique est fondamental.

2. Cf. IBM Systems journal, Vol 43, N°4, 2004, Grid computing.

Les entités architecturales partagent des éléments qui leur sont communs, à travers lesquels circule l'information : des zones mémoires, des moyens de communications qui constituent la frontière, ou le bord, de l'interface (cf. la notion de ports logiques de la machine). On dit que ces éléments communs sont « publics » ou « externes », ce qui dans les langages de programmation a motivé la création des concepts de règles de visibilité, de portée des déclarations associées aux aspects du langages ; notions d'ailleurs largement insuffisantes car il a fallu rapidement les compléter avec les dictionnaires de données et les gestionnaires de configuration.

Une interface est caractérisée par :

- Un ensemble de règles syntaxiques qui décrivent les modes d'appel et/ou d'interruption de l'interaction (signaux, exceptions, « trap » sur instruction et/ou donnée…), les formats des données échangées, les modalités de passage de l'information (paramètres d'appel, mémoire commune, messages, accusés de réception), les modalités de lecture des informations protégées par l'interface.
- Un ensemble de conventions sémantiques qui spécifient l'ordre dans lequel les opérations doivent être effectuées, les valeurs acceptables et la sémantique des données échangées ainsi que les modes de défaillance et les codes d'erreurs associés qui définissent le comportement.

Les modalités d'appel doivent spécifier la précision des résultats attendus, l'authentification des appelants (aspect sécurité), les contraintes de l'interaction (par exemple les time-out), les éléments nécessaires à une certification éventuelle par des tiers de confiance, les ressources nécessaires.

Dans le cas d'une interface Homme-Machine (cf. figure 1.4), les contraintes de type ergonomique (couleurs, bande passante, « *think-time* », etc.) font partie des conventions et des règles qui gouvernent l'accès et le comportement de l'interface.

La description d'une interface doit préciser :

- La portée et le besoin d'en connaître les différentes catégories d'usagers de l'interface : architectes, programmeurs, testeurs, administrateurs, mainteneurs. Certains de ces usagers voient l'interface comme une boîte noire, alors que d'autres auront à en manipuler la structure à différents niveaux de granularité.
- L'usage général de l'interface qui doit refléter une notion de contrat entre les différentes catégories d'usagers de l'interface, avant d'en effectuer l'analyse détaillée et la programmation. Ce point est particulièrement important si certains des usagers sont des sous-traitants car l'accord doit alors être de nature contractuelle et légale.
- La représentation et la traçabilité de l'interface, et ce en étroite relation avec l'environnement de conception et de programmation qui a été adopté pour le logiciel. L'interface doit pouvoir être explicitée dans les différents langages

autorisés, et répertoriée comme telle dans le gestionnaire de configuration du système.

On doit appliquer à l'interface les caractéristiques qualité FURPSE qui permettent la définition des exigences, comme à n'importe quel élément logiciel, telles que préconisées par la norme ISO 9126. Compte tenu du rôle critique que jouent les interfaces, l'usage d'une telle norme est particulièrement importante.

Une interface est tout à la fois une fonction de transcodage (i.e. un adaptateur), un protocole d'appel (modalités d'invocation de l'entité appelée), et une cinématique. Il est fondamental d'expliciter les caractéristiques comportementales attendues de l'interface, et plus particulièrement :

- la facilité de mise en œuvre de l'interface (vérification des contraintes à satisfaire, codes d'erreurs, aides en lignes, définition formelle ou non…),
- la fiabilité et la disponibilité (tolérance aux fautes, possibilité de récupération, fréquence des défaillances observée),
- la performance (temps de réponse, débit, consommation des ressources de l'intégrat appelé),
- la maintenabilité (facilité d'analyse en cas de défaillances, gestion des modifications, facilité de tests en particulier la non régression et la compatibilité ascendante, sensibilité à l'environnement (effets de bords, canaux cachés, ressources partagées) et aux conditions d'appels, innocuité et robustesse en cas d'appels et/ou de données erronées),
- la portabilité (adaptabilité éventuelle à différentes plates-formes d'exécution, facilités d'autotest et d'installation, aptitude à la réutilisation) et l'évolutivité selon le principe de la compatibilité ascendante.

Ces cinq caractéristiques constituent les caractéristiques non fonctionnelles de l'interface. Pour la fiabilité et la performance on définira le niveau d'instrumentation nécessaire à l'observation et aux mesures compte tenu des exigences de sûreté souhaitées.

Une interface peut fonctionnellement se schématiser comme suit (figure 15.1).

NB : On notera l'analogie profonde de ce schéma avec le modèle Traducteur-Transducteur du chapitre 10.

Les fonctions de conversion qui figurent dans le schéma assurent, si nécessaire, les changements de formats nécessités par les représentations de l'information dans les entités de part et d'autre de l'interface. La logique d'enchaînement pilote les interactions entre l'appelant et l'appelé.

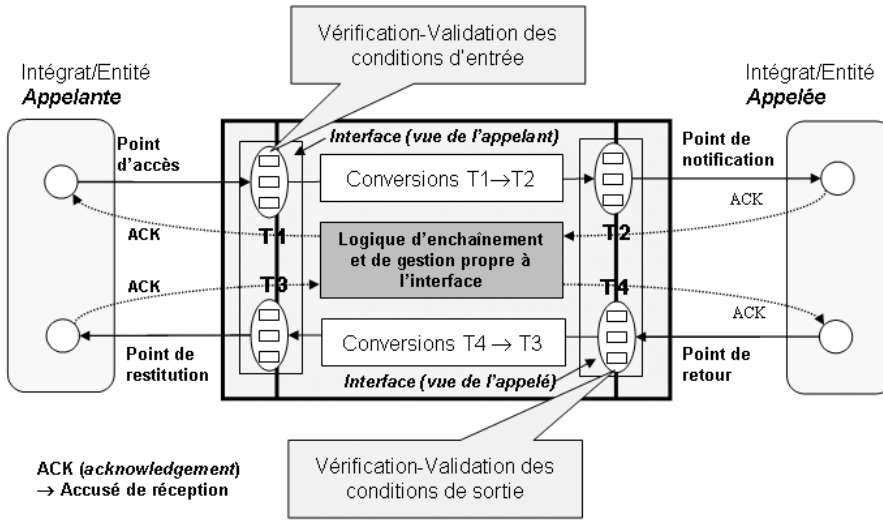


Figure 15.1 - Structure générale d'une interface

Dans une interface, il faut distinguer trois niveaux de description :

1. La description sémantique de l'interface (i.e. son modèle conceptuel). Cette description peut être en langage naturel et exploiter toutes les possibilités offertes par le multimédia, ou en notation semi-formelle voire formelle quand la rigueur l'exige. Elle doit être non ambiguë. Elle répond à la question du **quoi** : Qu'est ce que ça fait ? C'est la vue du concepteur/architecte du système.
2. La description programmatique de l'interface (i.e. son modèle logique dans tel ou tel langage de programmation). Cette description répond à la question : **comment** programmer ? C'est la vue du programmeur.
3. La description physique, sur une plate-forme particulière, qui matérialise le comment du point de vue de la machine que l'on a parfois tendance à oublier.

NB : les défaillances apparaissent toujours au niveau physique.

Dans le cas d'une interaction entre un acteur (pas nécessairement humain) et un équipement, on peut adapter le schéma précédent comme suit (figure 15.2).

Enfin, il faut être parfaitement conscient que les entités architecturales appelées (i.e. les services) importent leurs comportements dans les entités qui les appellent. Il faut alors s'assurer que le comportement résultant reste cohérent avec celui souhaité pour l'intégrat appelant. Si l'appelé sature une ressource partagée avec l'appelant, le comportement de l'appelant peut être modifié de façon aléatoire et de ce fait ne plus être nominal. La programmation de l'intégrat appelé (c'est un service) doit être rigoureuse du point de vue de la disponibilité et de la sûreté de fonctionnement. Une

programmation peu rigoureuse induira du non-déterminisme chez l'appelant ce qui sera fatal à la sûreté de fonctionnement globale.

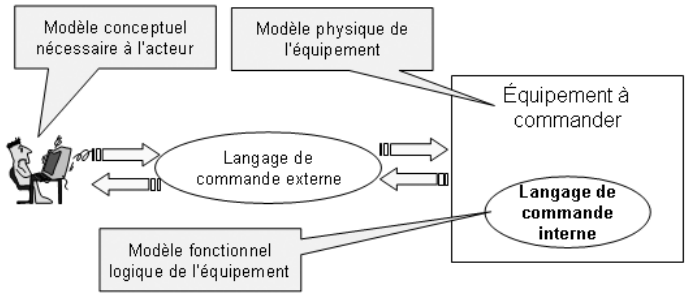


Figure 15.2 - Interface acteur – équipement

Comme le montre le schéma de principe ci-dessus, une interface est un organe fonctionnel actif qui réalise un couplage entre appelants et appelés, tout en garantissant une autonomie relative des appelants et appelés. En cas de modification de l'un et/ou de l'autre, c'est l'interface qui « encaisse » et joue le rôle d'amortisseur. Dans le jargon des informaticiens, l'interface est parfois appelée « joint de dilatation » par analogie avec la pièce qui désolidarise le tablier d'un pont des piles du pont, d'un point de vue mécanique.

La métaphore a ses limites car elle masque l'aspect dynamique de l'interface qui assure le transport du flux informationnel. En mécanique, c'est ce qui se passe pour le transfert du flux d'énergie du moteur vers les roues d'un véhicule, grâce à des organes comme les joints de Cardan et/ou les différentiels de vitesse. Dans une interface, il faut soigneusement distinguer la syntaxe, i.e. la représentation et/ou le codage de l'information, et la sémantique, i.e. le sens de l'information qui est un invariant. En se cantonnant aux sciences de l'information, la meilleure métaphore qui soit est celle du canal, au sens de la théorie de l'information.

Définir et réaliser une interface revient à définir et à réaliser un canal de communication. Selon la nature des exigences définissant les échanges (caractéristiques non fonctionnelles) il y aura plusieurs réalisations possibles pour les différents types de couplages.

La nature profonde d'une interface est toujours d'ordre sémantique. Elle matérialise le contrat appelant-appelé en donnant la liberté des modes de représentations (i.e. codage) de l'information, ce qui n'est pas rien, mais exige une parfaite rigueur pour ce qui concerne la sémantique. C'est la raison pour laquelle il est indispensable de définir les interfaces préalablement à toute réalisation. Si l'architecte ne procède pas ainsi, il est certain qu'il y aura des ambiguïtés et des « trous » sémantiques générateurs de défaillances graves car elles touchent le sens du système ; c'est le problème fondamental des projets d'interopérabilité.

Dans une chaîne de traitement, ce qu'il faut garantir c'est la sémantique du point de vue des UA métiers qui produisent, gèrent, transforment, consomment l'information dont elles ont besoin. Le schéma de communication est le suivant (figure 15.3) :

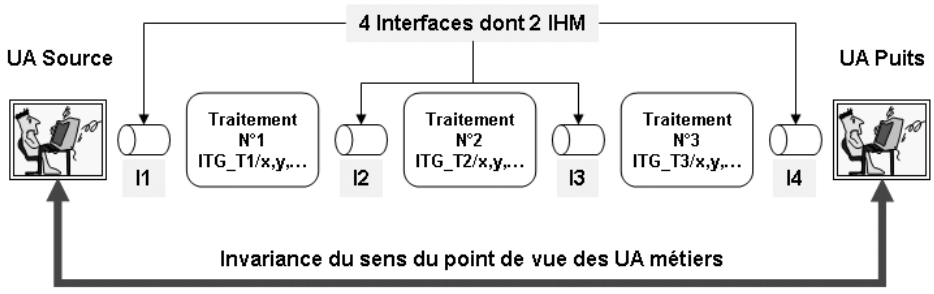


Figure 15.3 - Interfaces et invariance sémantique

Les interfaces doivent être définies a priori, et la « vérité » qu'elles matérialisent doit être toujours respectée. D'une certaine façon, elles constituent l'ossature logique du système, son axiomatique. D'où le rôle crucial des interfaces pour la sûreté de fonctionnement. Le langage dans lequel est définie l'interface doit privilégier la sémantique et la rigueur logique, indépendamment des modes de représentations que l'on trouvera du côté appelant/appelé. Le langage B, et la méthode de raffinement qui lui est associé, est un bon exemple de rigueur logique indispensable, sans équivalent, pour le moment, dans les évolutions d'UML (cf. le profil définit à l'OMG pour l'ingénierie système SYSML). On ne doit jamais définir une interface par essai-erreur a posteriori.

Du point de vue de la sûreté de fonctionnement et de l'intégration, les interfaces sont des points d'observation privilégiés des transformations et des comportements du système. L'observateur associé à l'interface est un être logique (un démon de Maxwell) qui voit passer l'information ce qui va lui permettre d'agir, soit en nominal, soit en surveillance, soit en cas de violation du contrat. Le schéma de principe est le suivant (figure 15.4) :

L'appelant dialogue avec l'appelé via le langage externe LE de l'interface qui matérialise le modèle conceptuel de l'interface. L'appelé réalise les requêtes de l'appelant via le langage interne LI de l'interface. L'interface traduit le programme exprimé , en LE en un programme opérationnel exprimé en LI.

LE peut être un langage déclaratif (type SQL dans les SGBD) alors que LI est par nature impératif (de type navigationnel) car c'est les instructions de LI qui actionnent les mécanismes de l'équipement sur lequel l'appelant s'exécute.

La logique fonctionnelle de l'interface se ramène ainsi à une logique de traduction, avec tout ce que cela comporte, qui est l'une des branches les plus matures de la technologie informatique (Théorie des langages et ingénierie des compilateurs, cf. distinction syntaxe concrète/syntaxe abstraite) dont nous avons donné un aperçu au chapitre 6.

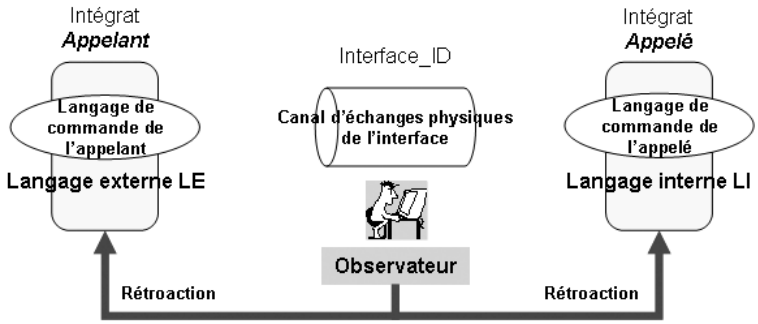


Figure 15.4 - Observateur de l'interface

D'un point de vue strictement logique, on retrouve ici une distinction fondamentale entre la notion de langage externe et de langage interne, effectuée par les logiciens du cercle de Vienne dans les années 1930s. Voir en particulier l'ouvrage fondateur de R. Carnap, *The logical structure of the world*, où tout ce fondement logique est parfaitement expliqué.

C'est la même distinction faite par Von Neumann, (lui-même logicien de premier plan, bien au fait de tous ces travaux) dans son dernier ouvrage, *The computer and the brain*, écrit quelques semaines avant sa mort. Le langage de l'acteur est différent du langage de l'actant, mais ils ont un noyau sémantique commun qui permet la traduction (dans le cas du cerveau humain, il s'agit d'une transduction, au sens propre).

Du point de vue de l'intégrateur, la structure de l'interface offre des points d'observation naturels et des possibilités de « bouchonnage » qui simplifient le travail d'intégration. Le travail de l'intégrateur est concentré sur la logique de l'interaction et non sur les structures internes appelant/appelé qu'il n'a pas à connaître. Les points principaux d'observation sont résumés par le schéma 15.5.

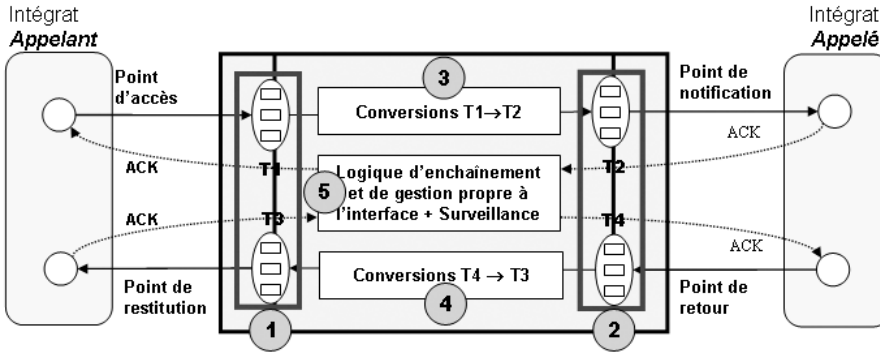


Figure 15.5 - Observation de l'interface

On retrouve le schéma général de la machine abstraite ébauchée au chapitre 1 :

- 1 et 2 permettent d'observer tout ce qui transite par l'interface.
- 3 et 4 permettent d'observer les conversions/traductions effectuées.
- 5 permet d'observer les changements d'états (automate de contrôle de l'interface) parmi lesquels on pourrait distinguer ce qui est nominal et ce qui est non nominal du point de vue de la sûreté.

En matière de bouchonnage et/ou de simulation, on peut connecter l'équipement de simulation soit sur le point N°1, soit sur le point N°2. Dans le cas N°1, l'interface joue le rôle de simulateur de l'appelant. Dans le cas N°2, l'interface existe indépendamment de l'appelant ; le simulateur utilise la logique de l'interface, ce qui a deux avantages : a) validation et test de l'interface, et b) simplification du simulateur qui n'a pas à connaître le détail des adaptations réalisées par l'interface.

La situation est résumée par le schéma 15.6 :

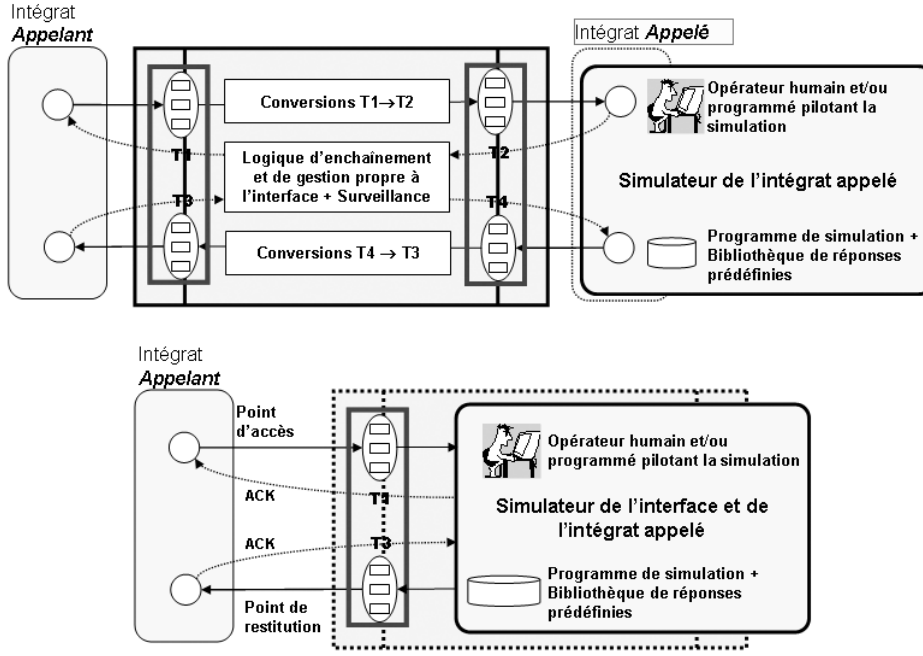


Figure 15.6 - Ancrages de la simulation

On notera, au passage, que tous les éléments doivent suivre le principe de modularité un point d'entrée – un point de sortie ; faute de quoi, l'instrumentation de l'interface est impossible du fait de l'enchevêtrement et de la complexité qui en résulte.

L'interface jouera également le rôle de filtre d'information en évitant de propager des informations qui n'auraient pas de sens (i.e. l'appelant et/ou l'appelé ne savent pas comment agir, car ils ne disposent pas de la programmation correspondante).

Il y a de bons exemples de filtrage avec les IHM ou les interface d'équipements complexes. Dans une IHM le serveur de transactions ne verra jamais les événements « souris » ou « clavier », il n'en verra que le résultat. Dans une interface réseau, l'appelant ne verra jamais les codes correcteurs d'erreurs, les « retry » ou les accusés de réception fins générés dans les couches physiques. Cette prise de recul qui est un niveau d'abstraction est un aspect fondamental de l'intégration pour lequel les interfaces sont l'élément critique.

C'est une situation très fréquente dans les systèmes distribués où, par suite d'un filtrage défaillant, ce qui concerne, au départ, un des éléments du système finit par affecter globalement le système.

C'est la raison pour laquelle il faut appliquer systématiquement une logique constructive, où tout est explicité, en matière d'ingénierie des interfaces. Le code correspondant aux interfaces doit être robuste (on dit aussi résilient ; on parle de propriété de résilience, i.e. *resiliency* en franglais informatique), car ce sont les interfaces qui constituent l'ossature de l'édifice. S'il y a un aspect du système qui doit être traité avec la plus grande rigueur, avec les techniques formelles ad hoc, c'est bien les interfaces car toute propagation d'information erronée, ou simplement inexploitable, dans un système d'une certaine taille, rendra le système non déterministe ; c'est la pire des situations pour un ingénieur.

Dans un schéma comme celui de la figure 12.6, le rôle de l'intégrat pivot comme élément de filtrage est fondamental pour assurer la sûreté de fonctionnement du point de vue métier.

15.3 CYCLE DE VIE ET MISE EN ŒUVRE

Comme toute entité informatique une interface est faite de donnée, d'algorithmes et de comportements (i.e. événements associés) dont la définition doit suivre le cycle de développement standard de toute entité. Ce qui distingue les interfaces des autres entités système est leur caractère a priori ; elles doivent être définies avant les intégrats fonctionnels. Idéalement, le taux de retouche d'une interface devrait être nul.

D'un point de vue concret, une interface prête à l'emploi est constituée de :

- texte programme à incorporer dans l'intégrat appelant, avant compilation, conforme aux règles du langage de l'appelant,
- texte binaire à intégrer, lors de l'édition de liens statiques et/ou dynamiques,
- fonctions de services disponibles (i.e. installées) sur la plate-forme d'exécution de l'appelant (bibliothèque de services).

C'est la chaîne de préparation des programmes qui assure l'intégration cohérente de ces différents textes (cf. les figures 2.1, 2.2 et 2.3) et des tests éventuels qui leur sont associés.

Dans le schéma de principe de la figure 15.1, il y a trois catégories d'entités distinctes :

1. la façade, côté appelant.
2. la façade, côté appelé.
3. le corps fonctionnel, qui est une structure intermédiaire regroupant les convertisseurs de formats et l'automate d'enchaînements.

Du point de vue de l'architecture physique, plusieurs cas de répartition sont à considérer, selon la nature des plates-formes et des équipements utilisés.

Cas 1 : appelants et appelés sont dans un même espace d'adresse, ou dans des espaces d'adresse différents, mais sur une même plate-forme, sous le contrôle d'**un seul** système d'exploitation (SE). La communication se fait par la mémoire commune.

Cas 2 : appelants et appelés sont sur deux plates-formes (et/ou équipements) distinctes. Le système peut être hétérogène, avec **deux** SE distincts. La communication se fait par un bus d'intégration (type EAI, ESB) via un réseau.

Cas 3 : Idem cas 2, mais tout ou partie du corps de l'interface est localisé sur un équipement intermédiaire (un « hub ») centralisé, par exemple pour satisfaire des exigences de sécurité.

Le niveau de complexité est croissant de Cas N°1 → Cas N°3.

Le cas 1 est un schéma de répartition centralisé dans lequel le SE assure la surveillance de bout en bout. La mémoire commune est gérée via les services du SE. La configuration est centralisée.

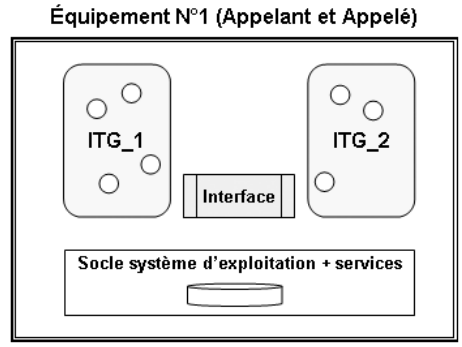


Figure 15.7 - Interface centralisée

Le cas 2 est un schéma de répartition distribué sur deux plates-formes. La surveillance est faite par l'organe de liaison car lui seul à la vision de bout en bout. Il peut avoir une gestion mémoire spécifique.

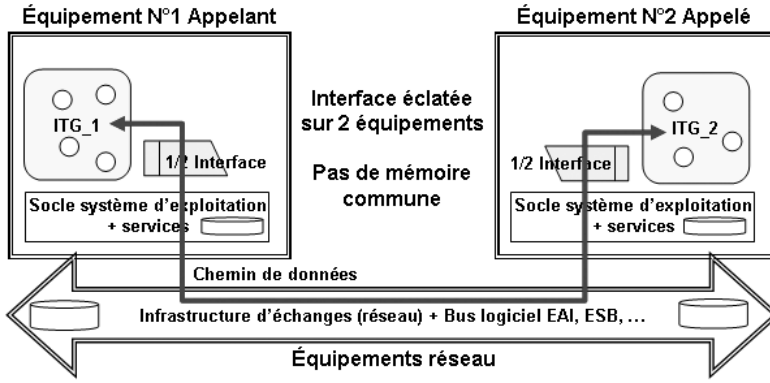


Figure 15.8 - Interface distribuée

Le bus de liaison assure la sûreté de fonctionnement. Seul l'exploitant de ce bus peut garantir l'intégrité des chaînes fonctionnelles (équivalent à une transaction longue). Chaque plate-forme peut avoir sa propre gestion de configuration (situation fréquente avec les systèmes « legacy » ce qui augmente le risque d'incohérence (cf. figure 12.8).

Le cas 3 est un compromis centralisé/distribué ce qui a des avantages et des inconvénients. L'ingénierie (en particulier le déploiement) est plus simple mais la centralisation sur l'équipement intermédiaire peut occasionner des problèmes de performance (engorgement) et de disponibilité (point de fragilité en cas de panne).

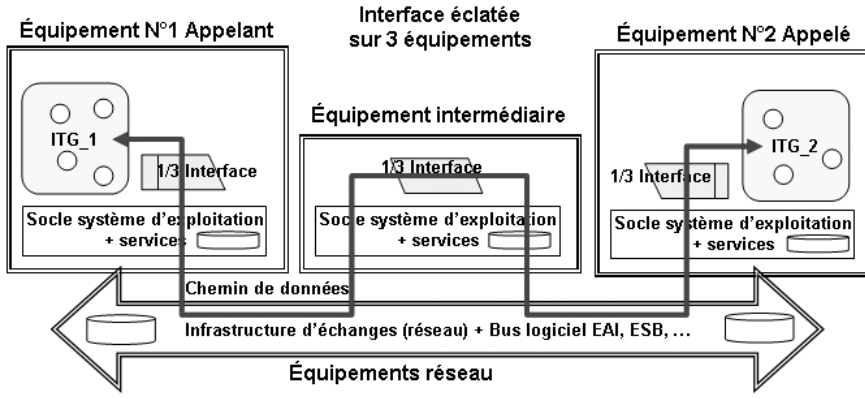


Figure 15.9 - Interface distribuée avec hub

La chaîne de liaison système est plus longue, d'où des problèmes de latence et de contrôle de charge. L'équipement intermédiaire qui héberge le corps de l'interface est un point de fragilité qu'il faut gérer comme tel (redondance, tolérance de pannes, reconfiguration automatique), selon les principes de l'autonomic computing.

L'interface, du point de vue système, doit être conçue comme un intégrat autonome, indépendant du contexte appelant/appelé. Elle peut être spécifiée de façon plus large et anticiper des évolutions prévisibles, en étant temporairement bridée, selon le profil des appelants.

Du point de vue de l'intégration (IVVT), le cycle d'ingénierie de l'interface présente les situations suivantes, selon l'état de réalisation. Chacun des éléments de la chaîne peut être soit simulé/partiel, soit réel/complet, ce qui fait 8 cas. Le tableau complet est comme suit (tableau 15.1) :

Tableau 15.1 - Interface et intégration (IVVT)

N°	Appelant	Interface	Appelé	Remarques
1	S	S	S	Faisabilité – Définition de l'interface
2	S	S	R	Faisabilité – Définition de l'interface
3	S	R	S	Développement – Intégration de l'interface
4	S	R	R	Développement – Intégration de l'interface
5	R	S	S	Faisabilité – Définition de l'interface
6	R	S	R	Faisabilité – Définition de l'interface
7	R	R	S	Développement – Intégration de l'interface
8	R	R	R	Développement – Intégration de l'interface

Toutes les cases marquées S dans la colonne Interface correspondent à la spécification /Conception de l'interface (documents IDD et ICD).

Toutes les cases marquées R dans la colonne Interface correspondent à différentes étapes d'intégration, jusqu'à la ligne 8 qui correspond au déploiement.

Modèles d'interfaces

Les modèles d'interfaces sont des adaptations des modèles généraux présentés dans la 3e partie.

Dans le cas N°1, l'invocation peut être séquentielle ou parallèle (cf. chapitre 9) ; l'échange d'information se fait par la mémoire via le modèle CRUD.

Le chemin de donnée du cas N°1 est le suivant (figure 15.10) :

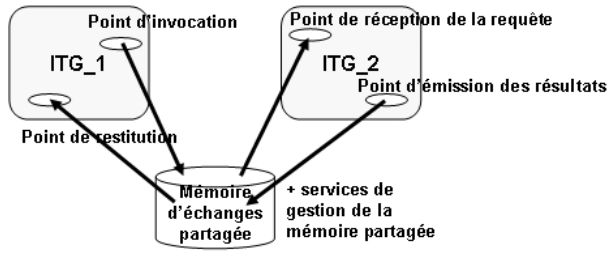


Figure 15.10 - Echange par la mémoire

L'interface gère la mémoire d'échange appelant/appelé soit via les services du SGF ou du SGBD en direct, soit via un service de gestion de mémoire ad hoc qui peut masquer le SGF/SGBD pour des raisons de portabilité.

En cas de défaillance, la restitution se fait explicitement au point non nominal de l'appelant.

Dans le cas N°2, l'échange se fait par le réseau, via les primitives `send receive` classiques. L'échange est asynchrone et met en œuvre des événements qui transitent via les files d'attente associées aux commandes `send receive`. Le chemin de données du cas N° 2 est le suivant (figure 15.11) :

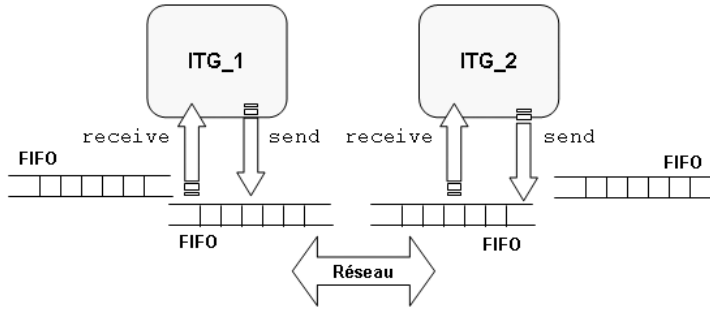


Figure 15.11 - Echange par le réseau

L'interface gère les accès réseaux via les méthodes d'accès réseau au niveau le plus fin, soit à l'aide de progiciels d'intégration de type MOM (*Message Oriented Middleware*) ou plus sophistiqué comme les EAI et/ou ESB. Dans tous les cas, l'architecte doit être conscient que l'exigence de stabilité des interfaces est primordiale et c'est ce qui doit commander le choix des technologies qui supportent les interfaces. Les technologies fondées sur des standards doivent être préférées à toute autre ; l'OPEN SOURCE offre une large gamme de services pouvant protéger efficacement les investissements des utilisateurs.

Le cas N°3 est une fusion Cas N°1 / Cas N°2.

Le cas d'une interface d'accès à un équipement a été ébauché au chapitre 10, section 10.2, Architecture en couche, en particulier figures 10.7 et 10.8.

La spécificité de ce type d'interface est que l'intégrat appelant peut dialoguer avec l'intégrat appelé via des séquences d'ordres pré-enregistrées ou construites dynamiquement à la demande, selon le contexte. Vue de l'appelant, l'intégrat appelé se présente comme une machine virtuelle dont les instructions sont les différents ordres émis par les appelés. Les ordres peuvent être exécutés un par un, ou communiqués à l'appelant par paquets réalisant une action complète du point de vue de l'appelant.

Le schéma de principe de ce type d'interface est le suivant (figure 15.12) :

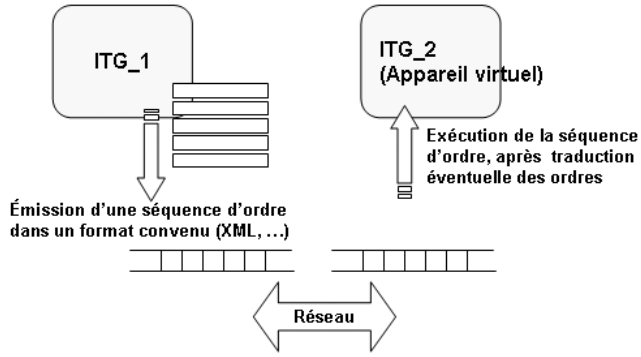


Figure 15.12 - Interface de machine virtuelle

On voit tout l'intérêt qu'il y a à standardiser ce type d'interface avec un langage comme XML, car une séquence d'ordres n'est qu'un document dont la seule exigence est d'être syntaxiquement et sémantiquement correcte. C'est l'approche choisie par les services Web avec le langage WSDL. En adoptant un mode de représentation textuelle des interfaces, on s'affranchit des représentations internes, ce qui est un progrès, mais également un risque stratégique car toute la technologie repose alors sur les capacités et les performances des traducteurs sous-jacents, ce qui peut rendre la partie la plus vitale du système dépendante de technologies que l'utilisateur ne contrôle pas. Dans ce cas, la seule protection pour l'utilisateur est la standardisation, non pas celle d'officines aux visées obscures, mais celle de vrais standards industriels dont l'OPEN SOURCE est un bon exemple.

NB : Toute cette approche nécessite des compétences en théorie des langages et en compilation, mais la technologie des compilateurs, après le collapsus de Bull, a disparu de notre pays depuis les années 90. Elle n'est quasiment plus enseignée, ni dans les écoles, ni dans les universités. Donc, prudence !

15.4 EVOLUTION ET COMPATIBILITÉ ASCENDANTE DES INTERFACES

Dans un système qui peut compter plusieurs milliers d'intégrats de rang 0 (cf. chapitre 2, section 2.2), il est indispensable de pouvoir modifier certains d'entre eux sans déstabiliser l'ensemble du système. Par les mécanismes de couplage vus au chapitre 9, il est très facile d'imaginer une situation chaotique où un changement de format (cf. le passage à l'an 2000) pourrait avoir des conséquences catastrophiques du point de vue CQFD/TCO du système.

Historiquement, les premiers architectes confrontés à cette problématique ont été ceux en charges des systèmes d'exploitation des machines dites de 3ème génération, avec des périphéries disques importantes. On peut évoquer deux grandes raisons.

1) Raisons d'ingénierie spécifique au système d'exploitation lui-même : très grande taille (4 à 5 millions de lignes sources), variété des équipements gérés, variété des configurations offertes, facilités de paramétrage offertes aux clients (i.e. la customisation), etc. C'était déjà de l'agilité.

2) Raisons économiques, du point de vue des utilisateurs métiers et exploitants, interdisant que des modifications du SE obligent à recompiler et re-qualifier les chaînes d'applications. Il fallait éviter les transferts de coûts fabricants/éditeurs vers les usagers. Ce qui était une règle absolue dans les années 70-80 s'est un peu relâchée, mais globalement elle est toujours respectée. C'est une exigence forte de tous les systèmes ayant un grand nombre d'usagers.

De nombreuses interfaces vitales pour la préservation des investissements des usagers ont été normalisées dans des standards internationaux, tant au niveau du système d'exploitation (norme POSIX) que des réseaux (norme de l'IUT) et des SGBD (norme SQL). Tous les langages de programmation sont complétés par des bibliothèques de fonctions qui sont soit intégrées directement dans le langage et normalisées comme tel, soit livrées par des éditeurs et constituant des standards de fait.

Pour livrer aux usagers des interfaces ayant l'indépendance souhaitée, celles-ci doivent être réalisées selon le principe dit de **compatibilité ascendante** (*upward compatibility*), depuis cette époque.

La propriété de compatibilité ascendante doit être construite par l'architecte. Les langages de programmation comme C++, Java offrent de nombreuses facilités pour implémenter correctement les interfaces, mais leur seul usage n'est évidemment pas une condition suffisante ; ni d'ailleurs nécessaire, car même en COBOL, on pouvait déjà travailler proprement. Les langages de programmation normalisés sont eux-mêmes d'excellents exemples de mise en œuvre du principe de compatibilité ascendante. Une évolution du langage qui crée une incompatibilité peu générer un coût significatif chez les utilisateurs, car ceux-ci devront modifier leurs programmes pour rester dans la norme. L'auteur se souvient de discussions acharnées au comité COBOL lors du passage à la norme 85 où certaines évolutions proposées ne respec-

taient pas le principe. À l'époque, le nombre de lignes écrites en COBOL se comptait en milliards ! D'où le coût induit : colossal.

Enoncé du principe de compatibilité ascendante

Les domaines de valeurs des paramètres de l'interface, et les contraintes à satisfaire déterminent un ensemble de cas possibles qui est l'extension de l'interface.

On dit qu'il y a compatibilité ascendante si toute évolution des domaines de valeurs et/ou des contraintes se traduit par une extension qui inclut strictement la précédente. Soit :

- Spécification I_id / dans la version V1→ Extension E_V1
- Spécification I_id / dans la version V2 → Extension E_V2 ⊃ E_V1

La forme géométrique de l'extension peut être compliquée, comme on peut le voir sur un exemple simple. Soit le calcul z = F (x,y) dans lequel x, y, z sont des entiers courts. La forme de l'extension dépend du type de calcul et de la façon dont est gérée la précision ; si c'est une multiplication, l'extension est une hyperbole, comme suit (figure 15.13) :

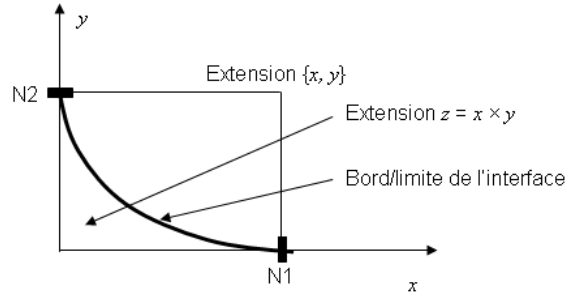


Figure 15.13 - Extension d'une interface

Avec *N* domaines de valeurs, l'extension est dans un espace à *N* dimensions. Notons, au passage, que la forme de l'extension donne un critère objectif de génération de cas de tests de l'interface. Tout ce qui est inclus dans l'extension est correct, tout ce qui est extérieur est un cas d'erreur.

La génération de cas de tests effectués sur le bord de l'extension et à son voisinage immédiat permet d'analyser la précision des contrôles effectués en entrée et en sortie de l'interface. Ce sont les tests de robustesse de l'interface. Ce type de test est fondamental pour l'intégration.

Dans le monde réel, il n'est pas toujours possible de respecter le principe. En cas d'erreur sur l'interface lui-même, ou d'évolution liée au développement de la technologie, des incompatibilités peuvent apparaître. Dans ce cas, il est indispensable de prévenir l'usager du problème potentiel. La situation normative du standard de l'interface se présente comme suit (figure 15.14) :

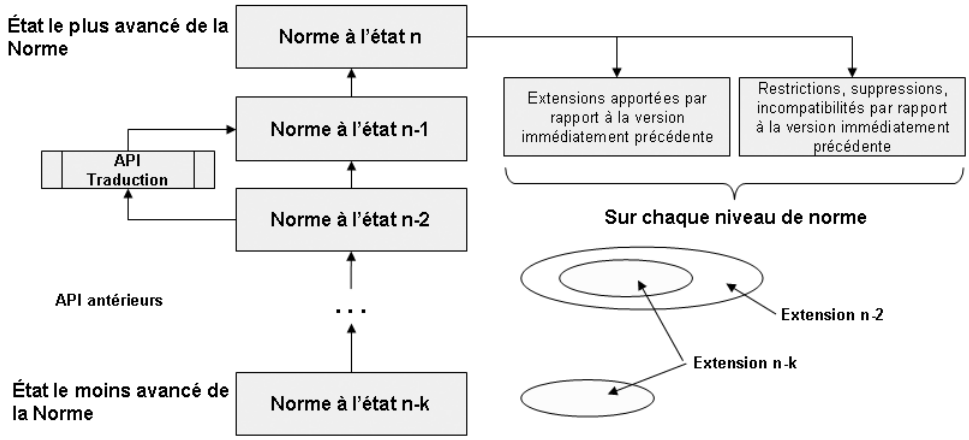


Figure 15.14 - Évolution de la norme d'une interface

À un instant donné de la vie du système, plusieurs états de la norme de l'interface I-id peuvent cohabiter. L'architecte a tout intérêt à minimiser ce nombre, pour des raisons économiques évidentes. Pour gérer une incompatibilité, il faut trois niveaux de norme : 1) l'état courant, 2) l'état où l'on annonce l'incompatibilité, 3) l'état où l'on résorbe l'incompatibilité.

La compatibilité ascendante garantit que l'on peut toujours traduire l'état n-1 dans l'état n à l'aide d'un traducteur, d'où la nécessité de « penser » l'interface comme un langage et les changements d'états successifs comme des traductions.

Pour la construction de l'interface de nombreuses implémentations sont possibles. Une solution simple consiste à définir une façade appelante unique dans laquelle il n'y a que trois paramètres : le nom de l'interface, son indice de version/révision conforme à la gestion de configuration, un vecteur d'état des données échangées, lui-même avec un indice de version/révision.

Le corps de l'interface doit être doté de traducteurs ad hoc, qui vont transformer le vecteur d'état dans l'état de la norme le plus avancé.

Sous la forme de diagramme de classe, le vecteur d'état peut être représenté comme suit (figure 15.15) :

D'un point de vue, physique, le vecteur d'état peut être transmis à l'appelant sous la forme d'un ou plusieurs messages (passage de l'information par valeur).

L'utilisation du méta-langage XML permet de gérer parfaitement les états normatifs (version/révision) des messages.

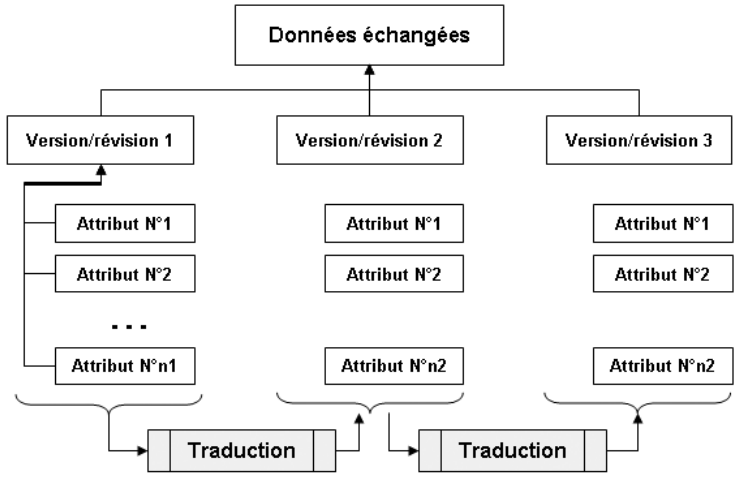


Figure 15.15 - Version/révision du vecteur d'état des données échangées

L'architecture fonctionnelle de l'interface, avec son jeu de traducteurs, est la suivante (figure 15.16) :

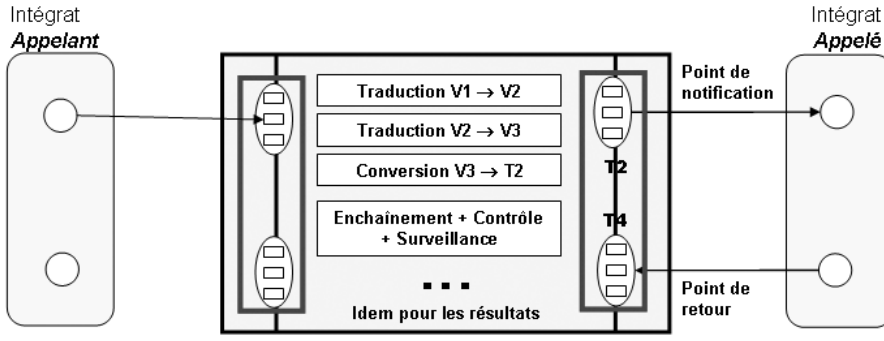


Figure 15.16 - Architecture fonctionnelle d'une interface assurant la compatibilité ascendante

Dans ce schéma, traducteurs et l'adaptateur sont la matérialisation de la sémantique gérée par l'interface. Les techniques d'analyse syntaxique et de compilation (cf. les figures 10.19 et 10.20, Cohérence des messages) permettent de contrôler parfaitement tout ce qui transite par l'interface, conformément à la spécification du contrat.

Avec cette méthode, faire évoluer une interface revient à rajouter un traducteur à l'interface. L'ajout peut se faire dynamiquement, grâce aux facilités d'édition de liens dynamiques (cf. la figure 2.3) que l'on trouve dans les environnements de programmation.

Des mécanismes comme le polymorphisme permettent des formulations très élégantes de l'implémentation, mais, comme nous l'avons déjà dit, il ne faut pas confondre la forme syntaxique et le fond sémantique.

Le module ECS Enchaînement - Contrôle - Surveillance qui gère le comportement de l'interface est à même de détecter toute anomalie liée à la sémantique comportementale et de le notifier à l'appelant via le mot d'état qui résume l'historique de l'exécution du service appelé.

Une interface construite de cette façon prend en compte les trois propriétés fondamentales de simplicité, disponibilité et évolutivité.

15.5 INTERFACES EXTERNES ET INTERNES D'UN INTÉGRAT AGRÉGÉ

Dans les parties I, chapitres 1 et 2, et III, chapitre 8, nous avons vu les principes des machines informationnelles et la structuration des intégrats. Le processus d'intégration défini, répétable et mesurable est basé sur une logique d'agrégation des interfaces des intégrats que l'architecte souhaite agréger pour former des intégrats de rang supérieur en respectant le principe de modularité. Nous proposons un exemple qui fait la synthèse de ces différentes notions.

L'intégrat agrégé se présente vis-à-vis de son environnement, sous la forme d'une machine abstraite composée de 7 intégrats élémentaires, comme suit (figure 15.17).

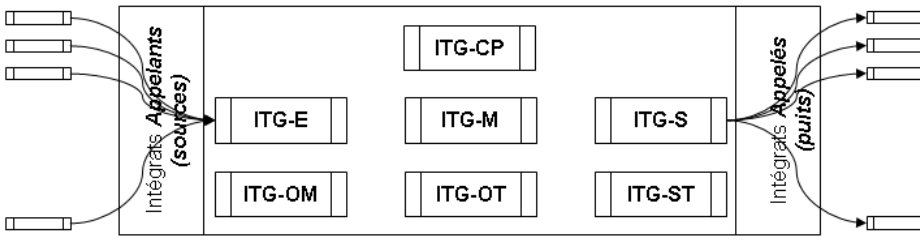


Figure 15.17 – Interfaces externes et internes d'un intégrat agrégé

La notion de sphère de contrôle de l'intégrat est matérialisée par ITG-E et ITG-S qui captent tout ce qui entre et tout ce qui sort de l'intégrat agrégé, y compris les intrusions pour l'instrumenter (on pourrait en décider autrement et faire apparaître les ports pour l'instrumentation).

La capacité de transformation (i.e. la sémantique) est matérialisée par ITG-OM (Objets Métiers) et ITG-OT (Objets Techniques). ITG-OT matérialise le jeu d'instruction avec lequel les objets métiers sont programmés. Si la plate-forme d'exécution évolue, seul ITG-OT est adapté.

ITG-M est le moniteur d'enchaînement et de contrôle (orchestration) de toutes les opérations effectuées par l'intégrat agrégé. C'est un point de contrôle unique, essentiel pour la sûreté de fonctionnement et la testabilité de l'intégrat agrégé. Il peut faire appel à un opérateur extérieur non représenté sur le schéma. Tous les événements, tous les messages transitent par ITG-M..

ITG-ST regroupe tout ce qui concerne la surveillance et les traces nécessaires au bon fonctionnement et au diagnostic (OLTD) en cas de panne.

ITG-CP regroupe tout ce qui concerne la construction et/ou le paramétrage de l'intégrat agrégé lors de l'initialisation/installation de l'intégrat. Il a connaissance des structures internes des intégrats constitutifs (méta-modèle).

Les interfaces externes, du point de vue de l'environnement, sont concentrés sur ITG-E et ITGS qui sont les ports logiques de l'intégrat agrégé. L'environnement ne verra jamais directement les autres intégrats, autrement que via ce qui est explicité dans ITG-E et -S. Le responsable de l'intégration dispose ainsi d'une information précise sur les interactions et couplages contractuellement autorisés.

Tous les autres interfaces sont des interfaces internes, privées à l'intégrat agrégé. On peut les représenter à l'aide d'une matrice N2 comme suit (figure 15.18).

ITG-E	×				×	
× (2)	ITG-M	×	×	×	×	×
	×	ITG-OM	× (1)		×	
	×	× (1)	ITG-OT		×	
	×		×	ITG-CP	×	
	×		×(3)		ITG-ST	
					×	ITG-S

Figure 15.18 – Matrice N2 des interfaces internes de l'intégrat agrégé

La colonne et la ligne ITG-M avec **×** signifie que le moniteur M peut appeler tout le monde et que tout le monde revient sur M (sauf quand on a décidé de sortir).

ITG-E ne peut appeler que M et/ou ST, et n'est lui-même appelé par personne.

ITG-OM peut appeler directement ITG-OT via **×**(1), sans repasser par M ; c'est une option, mais l'architecte pourrait décider de l'interdire et obliger à passer toujours par M.

ITG-ST peut être appelé par tout le monde, car tout le monde doit être surveillé, en particulier les ports E et S (selon des modalités du type de celles évoquées au chapitre 10, figures 10.19 et 10.20).

ITG-S peut revenir sur ST en cas de violation du contrat de sortie. On remarquera que pour sortir définitivement, il faut repasser par la case M, en application du principe de modularité.

La case ×(2) indique que le moniteur est en attente d'événements entrant via le port d'entrée E ; ceci dépend du mode de couplage, soit interactif, soit par lot (dans ce cas, la case serait vide).

La case ×(3) indique simplement que ST peut appeler des services techniques T qui lui seraient spécifiques.

Tous les échanges internes peuvent se faire en utilisant la représentation interne de l'intégrat, ou en utilisant sa représentation externe, par exemple des textes en XML, comme ce qui est recommandé dans les Web-services, avec des langages comme WSDL. Dans ce cas, il faut traduire dynamiquement les interfaces dans leur format interne, ce qui offre la souplesse maximum en évolutivité/adaptabilité, au prix d'une architecture plus complexe qui représente le « prix » de cette capacité fondamentale. Le traducteur correspondant devient un élément critique du méta-modèle géré par ITG-CP.

Conclusion

L'ingénieur et le bricoleur, texte de C.Lévi-Strauss, dans *La Pensée sauvage*, Plon.

[…]

Le bricoleur est apte à exécuter un grand nombre de tâches diversifiées ; mais, à la différence de l'ingénieur, il ne subordonne pas chacune d'elles à l'obtention de matières premières et d'outils, conçus et procurés à la mesure de son projet : son univers instrumental est clos, et la règle de son jeu est de toujours s'arranger avec les « moyens du bord », c'est-à-dire un ensemble à chaque instant fini d'outils et de matériaux, hétéroclites au surplus, parce que la composition de l'ensemble n'est pas en rapport avec le projet du moment, ni d'ailleurs avec aucun projet particulier, mais est le résultat contingent de toutes les occasions qui se sont présentées de renouveler ou d'enrichir le stock, ou de l'entretenir avec les résidus de constructions et de destructions antérieures. L'ensemble des moyens du bricoleur n'est donc pas définissable par un projet (ce qui supposerait d'ailleurs, comme chez l'ingénieur, l'existence d'autant d'ensembles instrumentaux que de genres de projets, au moins en théorie) ; il se définit seulement par son instrumentalité, autrement dit et pour employer le langage même du bricoleur, parce que les éléments sont recueillis ou conservés en vertu du principe que « ça peut toujours servir ».

[…]

Au terme de cette visite guidée de quelques-unes des concepts d'architecture les plus solidement établis, il nous faut insister, pour conclure, sur le trait de caractère qui illustre le mieux la méthode de travail de l'architecte : « Penser à cent ans ! », pour reprendre l'expression de Le Corbusier.

En matière de logiciel, le message est on ne peut plus clair : seules les abstractions logiques ont la capacité de transcender les contingences technologiques. Penser à cent ans ne peut pas vouloir dire autre chose que penser abstraction et interface, évolution et ouverture. L'abstraction est consubstantielle à la matière informationnelle ; elle est l'âme du système, pour parler comme Aristote, i.e. ce qui anime, donne la vie. Elle est vraie dans tous les mondes possibles, et intemporelle. Pour parler en informaticien, elle est « context free », ce qui est une qualité de pre-

mière importance. Toute entité logicielle qui n'a pas cette capacité est virtuellement morte née et condamnée au bricolage évolutif.

Parmi les notions que nous avons présentées, la notion de « machine », au sens que Von Neumann – et dans une moindre mesure Turing[1] – lui avait donné dès l'origine, a toutes les qualités requises pour durer cent ans. Elle est l'unité de construction élémentaire porteuse de sens, le module, pour reprendre un vieux terme des maîtres bâtisseurs des cathédrales, autour duquel va s'organiser le chantier de développement, aussi vaste et aussi long soit-il, et ceci à tous les niveaux d'abstraction du système. Cette propriété « fractale »[2] est si fondamentale que sa présence pourrait constituer le signe de reconnaissance du chef-d'œuvre architectural. La notion de machine est étroitement associée à celle de langage dont elle est le dual. Le problème de l'intégration est ramené à un problème de dialogue de machines à machines, donc à celui d'une traduction réversible des langages qui les caractérisent. Comme nous l'avons vu tout au long de ces pages, les techniques de traduction sont parfaitement maîtrisées, ce qui montre que nous pouvons bâtir une méthodologie d'intégration sur des bases saines.

Les machines d'aujourd'hui n'ont plus grand-chose à voir avec celle de von Neumann à l'IAS de Princeton[3], mais les principes logiques sont restés les mêmes. Parmi ceux-ci, la fiabilité et la sûreté de fonctionnement, dont on peut dire que ce fut l'obsession de Von Neumann, jusqu'à sa mort prématurée en 1955. Chaque fois que ces principes ont été ignorés ou oubliés, l'échec a été au rendez-vous. L'un des plus notoires, et des plus coûteux pour le contribuable européen, fut la tentative de réalisation de machines parallèles en architecture dite « Non Von », sous l'impulsion de la CEE dans le cadre des projets ESPRIT, qui était censée répondre au coup médiatique des ordinateurs japonais dit de 5ème génération, dans les années 80s.

Imaginer une architecture que l'on ne saura pas tester, est aussi vain que de vouloir démontrer la quadrature du cercle. C'était vrai pour le matériel et les systèmes d'exploitation hier, c'est encore plus vrai pour les architectures applicatives en environnement distribué aujourd'hui. Seule la simplicité peut contrecarrer la complexité inhérente à la réalité. C'est cette évidence qu'avaient simplement oubliée les promoteurs de ce projet déraisonnable. Mettre entre les mains des usagers une machine non déterministe, c'est comme livrer une centrale nucléaire sans ses enceintes de confinement. C'est pourtant ce qu'avaient réussi à vendre les lobbyistes bruxellois. Une erreur stratégique à quelques centaines de millions d'euros, avec en prime un discrédit général, et la déroute de l'informatique européenne[4]. Ceci explique peut-être, en partie, la méfiance des décideurs vis-à-vis des informaticiens ?

---

1. Un des mots attribués à Turing était : « *Keep the hardware as simple as possible* » ; cf. A.Hodges, *Alan Turing : the enigma*, Burnett Books, 1983.
2. Terme utilisé par Y.Caseau dans son livre, *Urbanisation et BPM*.
3. Cf. H.Goldstine, *The computer from Pascal to Von Neumann*, Princeton University Press, 1972.
4. Cf. L'article de J.Stern, ancien PdG de Bull, dans la revue Géopolitique N° 71, Septembre 2000 : L'Europe des occasions perdues.

Les plates-formes que nous utilisons aujourd'hui sont, toutes choses égales par ailleurs, ultra-fiables et ultra-performantes comparées au logiciel. Ceci ne doit rien, ni au hasard, ni à la technologie, fut-elle habillée des oripeaux de la « loi » de Moore. Ce miracle d'équilibre doit tout à l'intelligence des architectes[1] qui ont su faire ce qu'il fallait, qui se sont comportés en ingénieurs pragmatiques et non en joueurs de poker ou en stratèges de bistrot. Comme on l'a expliqué tout au long de ces chapitres, une machine est une construction que l'on sait tester, tant au niveau des parties que du tout. On sait l'accélérer de façon sûre avec des mécanismes comme les pipe-lines et/ou les caches, et, principe d'ingénierie fondamental, on sait toujours expliquer pourquoi cela marche grâce au « *design to test* ». L'existence des plates-formes, telle que nous les connaissons, démontre de façon évidente que l'on peut travailler au plan logique, indépendamment du plan physique, sans jamais confondre les deux. Il suffit pour cela d'avoir de bons langages et de bons compilateurs. La méthode d'obtention de la qualité est déterministe parce qu'on l'a définie comme cela ; elle est le dual de l'architecture. Ce sont ces pratiques et ces méthodes qu'il faut appliquer à tous nos systèmes informatisés.

L'architecte logiciel, faute d'un meilleur terme, est face à un triple défi : a) défi de la communication avec les parties prenantes, b) défi de l'imagination et de la simplification face à la complexité et aux débordements technologiques, face à la pression des vendeurs de solutions « prêt à porter », c) défi éthique face à l'abnégation dont il doit faire preuve vis à vis de réalisations dont il ne verra peut-être pas l'achèvement.

Le défi de la communication tient à la nature immatérielle des abstractions manipulées par l'architecte. Comment faire comprendre le juste prix de l'immatériel à des décideurs ancrés dans le concret ? Comment expliquer la différence entre un vaste bricolage, fut-il astucieux, et une construction d'ingénieur rigoureuse, où chaque pièce a sa fonction et sa juste place ? Ce qui ne posait pas trop de problème dans le monde d'ingénieurs[2] rompus aux sciences dures des premiers grands systèmes, est devenu un réel problème dans une communauté informatique où les ingénieurs sont désormais minoritaires. D'un côté une logique long terme, à base de modèles et d'expérimentation, de l'autre une logique court terme aux résultats financiers immédiats. Il n'est pas question ici de réactiver un conflit science dure/science molle, mais d'instaurer un dialogue constructif. Le scepticisme des décideurs, pour cause de promesses non tenues, est à mettre au passif des informaticiens qui ont parfois, pour rester diplomate, vendu du vent. À l'inverse, le décideur qui s'entoure d'« experts » complaisants ne peut s'en prendre qu'à sa naïveté, ou à sa mégalomanie dans les cas pathologiques.

Il nous semble que des notions comme l'architecture testable, revisitée sous l'appellation anglo-saxonne de *Test-Driven-Development*, appuyée sur des modèles de coût où l'architecture joue le rôle central, peut amener les parties prenantes sur un terrain neutre où tout le monde se comprend. L'ingénieur s'intéresse au coût des réalisations et à la compétence des équipes, ce qui n'est pas le cas de son

1. Cf. R.Colwell, *The Pentium chronicles*, Wiley, 2006.
2. Cf. R.Germinet, *L'ingénieur au chevet de la démocratie*, Odile Jacob, 2004.

collègue académique qui lui s'intéresse essentiellement à la faisabilité, démontrée par des modèles. Confondre les deux problématiques est la meilleure façon d'échouer et de décevoir. Par ailleurs, personne n'a le désir de nuire, et chacun souhaite sincèrement que cela « marche ». Il faut revenir à la vérité des coûts.

La communication par métaphore est utile mais trompeuse, analogie ne veut pas dire similitude. Thomas d'Aquin disait que la traduction est une trahison. Avec les métaphores, on pourrait parler de double trahison, celle du langage de l'émetteur, et celle du récepteur. Les informaticiens ont usé et abusé des métaphores, mais ils ne sont pas encore guéris comme les évangélistes de l'OMG, et plus récemment les thuriféraires de l'Alliance AGILE, nous en administrent régulièrement la démonstration. À notre avis, la seule bonne métaphore en matière d'information est celle du texte écrit que nous pratiquons tous depuis 3.000 ans. Chacun peut comprendre, quelle que soit sa culture, science dure ou molle, le prix d'un texte, quelle que soit l'unité de compte, le mot, la phrase, la ligne ou la page. Ne pas savoir expliquer le coût d'un texte correspondant à la réalisation des tests du système est une « *supporting evidence* » d'un défaut de maturité et d'une grande incertitude sur la réalisation : même un non expert peut le comprendre.

Le défi de l'imagination est celui de la prudence[1] dont l'architecte doit faire preuve vis-à-vis des mérites d'une technologie que des vendeurs peu scrupuleux peuvent entretenir chez des décideurs sceptiques[2]. À ce sujet, le rapport du Standish Group parle d'illettrisme technique, terme à propos duquel il ne faut pas se méprendre. Discourir, même de façon savante, sur la technologie et se gargariser de mots creux, ne donne aucune compétence architecturale. C'est la raison pour laquelle il y a tant de pseudos experts. Le problème de l'architecte n'est pas de discourir ou de philosopher sur les mérites comparés des technologies, mais de choisir, avec comme critère de décision le « penser à cent ans ». Pour cela, il doit comprendre en profondeur les forces et les faiblesses de la technologie en question. Malheureusement, cela ne s'apprend pas dans les livres, encore qu'un minimum de bon sens permette de comprendre bien des choses[3]. Empiler les technologies pour faire mode et complaire à un management qui sort de son domaine de compétence en s'improvisant architecte, c'est fabriquer une formidable complexité que l'on ne saura pas tester dans une enveloppe de coût raisonnable, d'où une chute inexorable de la qualité de service et du ROI informatique. La complexité est une maladie chronique du bricolage technologique, fatale à moyen-long terme, qui de surcroît entretient chez les décideurs le mythe du sauveur dont on connaît les ravages. La complexité n'est soluble que dans la simplicité résultant du travail architectural et de l'imagination de l'architecte qui est avant tout un « *problem solver* ».

Le défi éthique de l'architecte n'est que la conséquence politique du « penser à cent ans » qui a tous les attributs de la chose invendable. Le bien fondé d'une solu-

1. C'est une des vertus cardinales, dont Thomas d'Aquin a été le théoricien, après Platon et Aristote.
2. Cf. Ch.Morel, *Les décisions absurdes*, Gallimard, 2002.
3. Cf. l'ouvrage de M.Porter, *Competitive strategy*, Free Press, 1980.

tion architecturale ne se révélera que dans la durée, et jamais dans le temps court. En vérité, comme son lointain prédécesseur bâtisseur de cathédrales, l'architecte ne verra probablement pas le fruit de son effort dont d'autres s'attribueront peut-être le mérite à sa place. Sans faire de l'architecte une espèce de moine, il faut une force de caractère peu commune et un goût de la chose bien faite assez rare, pour la gloire et l'honneur de l'esprit humain, pour se détacher de son œuvre. Il y réussira d'autant mieux s'il est assez adroit pour gagner la confiance de son management, ce qui est le B.A.-BA du bon management, car sans la confiance[1] rien d'envergure n'est concevable. L'architecture, comme la stratégie, est un art d'exécution qui doit prendre en compte tous les paramètres techniques et environnementaux.

Pour les sceptiques qui ont quelques excuses à l'être, nous leur proposons de réfléchir au fait qu'en tout état de cause, il existera toujours une architecture, soit intentionnellement, soit par défaut. Le seul choix est donc, soit de la construire a priori, soit de la subir a posteriori. Dans le premier cas on est dans une démarche d'ingénieur qui tente de réduire le risque pour les usagers à l'imprévisible, alors que dans le second on reste au stade du bricolage et de la pensée magique, en croisant les doigts pour que cela marche.

L'architecture est un germe, un centre organisateur, que l'architecte devra instiller dans les équipes qui en développeront les potentialités. C'est sans doute là son rôle le plus difficile, le plus ingrat, mais également le plus gratifiant : faire partager une conviction dont celui qui la reçoit peut croire qu'il en est l'inventeur. C'est l'essence de la pédagogie et de la connaissance véritablement assimilée.

Après le temps des inventeurs dans les années 50-60, puis des développeurs dans les années 70-80, puis des éditeurs de progiciels à tout faire des années 90, voici venu le temps des architectes bâtisseurs au service des chaînes de valeurs de l'entreprise.

1. Cf. l'article : Les mathématiques de la confiance, dans D.Kahneman, A.Tverski, *Choices, values and frames*, Cambridge University Press, 2000.

Sigles et acronymes utilisés

ACID	Atomicité, Consistance, Isolation, Durabilité – Propriétés fondamentales d'un programme transactionnel
ACL	Article de Configuration Logiciel
ADT	*Abstract Data Type* – Type abstrait de données ; concept fondateur de la programmation objet.
AF	*Architecture Framework*
AP	Arbre Produit – PBS, *Product breakdown structure* ; voir OT
AQ	Assurance Qualité ; en anglais, QA
ASN.1	*Abstract Syntax Notation one* – Norme CCITT/UIT, Recommandations X.208 et X.209 – Normes ISO 8824 et 8825 ; compagnon du langage SDL
BNF	*Backus Normal* (ou *Naur*) *Form* ; J.Backus fut le chief designer du FORTRAN et l'un des promoteurs de la notation BNF utilisé pour la définition du langage ALGOL 60. P.Naur, l'un des principaux contributeurs. Cf. Norme ISO 14977, *Extended BNF*
BPEL	*Business Process Execution Language* ; voir BPML
BPM	*Business Process Model* – Modèle métier (ou modèle d'entreprise)
BPML	*Business Process Modeling language* ; voir BPEL
C4ISR	*Communication Control Command Computer Intelligence Surveillance Reconnaissance* ; antérieurement C4I, C3I (TerminologieUS DOD, OTAN)
CDV	Cycle de vie
CDVL	CDV Logiciel ; en anglais : SDLC, *Software Development Life Cycle*
CDVS	CDV Système

CG/CD	Conception générale / détaillée
COCOMO	*COnstructive COst MOdel* – Modèle d'estimation CQFD élaboré par B.Boehm ; cf. *Software engineering economics* et *Software cost estimation with* COCOMO II
COTS	*Component off the shelf* – Produit/progiciel sur étagère
CPU	*Central Processor Unit* – UC, unité centrale
CQFD	Coût Qualité Fonctionnalité Durée/délai
CRUDE	*Create, Retrieve, Update, Delete, Execute* – CRUD pour les données
DFD	*Data Flow Diagram* – voir SA, SADT, SART, IDEF
DG	Direction Générale
DSI	Direction des Systèmes d'Information
DSL	*Domain Specific Language*
DTD	*Document Type Definition* – C'est la grammaire d'un document ; fait partie du langage XML
EAI	*Enterprise Application Integration* – Progiciel d'intégration des applications de l'entreprise
EB/EC	Expression de besoin / Exigences comportementales
ECC	*Error Correcting Code* – Code correcteur d'erreur
ERA	*Entity Relationship Attribute* – Modèle entité relation attribut
ESB	*Enterprise Service Bus* – Bus d'intégration des services de l'entreprise ; voir EAI
ETL	Extract, Transform, Load
FCS	*First Customer Shipment* – Date de 1ère livraison d'un produit chez un client
FURPSE	Fonctionnalité, Usabilité (facilité d'emploi), Fiabilité (reliability), Performance, Maintenabilité (serviceability), Evolutivité – Classification des caractéristiques qualité selon norme ISO/CEI 9126
ICC	Indicateur de Complexité Complication
IDEF	*Integration Definition Language for Function Models* ; Voir IEEE Std 1320
IF	*Information Framework*
IHM	Interface Homme Machine
INCOSE	International Council on Systems Engineering
IOC	*Input Output Controler* – Contrôleur d'entrées-sorties, pour piloter les organes périphériques d'un ordinateur

IPC	*Inter Process Communication* – Communication entre les processus, via les mécanismes du système d'exploitation comme FORK et JOIN sous UNIX.
IS	Ingénierie Système ; en anglais *System Engineering*
ITG	Intégrat (en anglais, Building-block, ou Module)
ITIL	Information Technology Integrated Library
IVVT	Intégration et VVT
JVM	*Java Virtual Machine* ; voir MA et MV
LHN	Langage de Haut Niveau (3ème génération), comme FORTRAN, COBOL, PL1, Ada, C, C++, Java, C#, etc.
LM	Langage Machine (1ère génération), i.e. les assembleurs et macro-assembleurs
LRU	*Least Recently Used* – Se dit d'une entité (en mémoire cache) la moins utilisée, donc potentiellement à effacer du cach.
LS / KLS	Ligne Source – Unité de comptage pour la taille des programmes ; en K(kilo)LS dans COCOMO.
MA	Machine abstraite ; voir MAE
MAE	Machine abstraite à état – ASM, Abstract state machine
MC	Mémoire Centrale
MCD	Modèle Conceptuel de Données (dans la méthode MERISE)
MCO	Maintien en Condition Opérationnelle ; activité de maintenance au sens large
MCT	Modèle Conceptuel de Traitement (dans la méthode MERISE)
MDA	*Model Driven Architecture* ; modèle défini par l'OMG
MIB	*Management Information Base* (configuration d'un équipement)
MOA	Maître d'Ouvrage ; en anglais, *the process owner* (cf. I.Jacobson). Se dit également de l'organisation en charge de la maîtrise d'ouvrage (en anglais, Project Office)
MOE	Maître d'œuvre ; organisation en charge de la réalisation, généralement au sein de la DSI
MOM	*Message Oriented Middleware* – Progiciel de communication orienté messages
MSC	*Message Sequence Chart* – Défini par le CCITT/UIT, Recommandation Z.120
MTBF	*Mean time between failure* – Temps moyen entre pannes
MTTR	*Mean time to repair* – Temps moyen de réparation

MV	Machine virtuelle ; vor MA, MAE
MVC	*Model View Controler*
NCS	*Network Centric System* – Système réseau-centré ; voir SDS
NDL	*Network data language* – Langage de navigation normalisé ISO pour les SGBD/DBMS basés sur le modèle réseau
OLAP	*On Line Analytical Processing* ; cf. OLTP
OLTD	*On Line Test and Diagnostic* – Tests et diagnostics en ligne, i.e. intégré dans le programme
OLTP	*On Line Transaction Processing* – Traitement conversationnel en ligne, i.e. le transactionnel basique.
OMG	*Object Management Group* ; voir le site www.omg.org
OS	*Operating system* – Système d'exploitation
OT	Organigramme des tâches – WBS, *Work breakdown structure* ; voir AP
PCB	*Process Control Blok* – Bloc (ou structure) de contrôle d'un processus de la machine
PESTEL	Politique, économique, Social, Technologique, Environnement, Légal – Classification des facteurs de l'écosystème projet (terminologie INCOSE)
PF	Point de Fonction – Unité de mesure de la taille fonctionnelle d'une application
PIM	*Plate-form Independent Model* ; Terminologie OMG pour le MDA.
PSM	*Plate-form Specific Model* ; Terminologie OMG pour le MDA.
PTE	Présentation - Traduction - Édition
RAID	*Redundant Array of Inexpensive Disk* – Unité de stockage intégrant plusieurs disques afin d'améliorer la disponibilité
ROI	*Return Of Investment* – Retour sur investissement
RPC	*Remote Procedure Call* – Protocole d'appel de procédure distante ; mécanisme de base dans les systèmes distribués
SADT	Structured Analysis (SA) and Design Technology
SART	*SA Real Time* – Méthodologie développée par Ward et Mellor
SDL	*Specification and Definition Language* – Définit par le CCITT/UIT pour spécifier les protocoles ; Recommandation Z.100
SDS	Système de systèmes – Un système dont les éléments sont eux-mêmes des systèmes, interconnectés via tout type de réseaux.
SGBD	Système de Gestion de Bases de Données – en anglais : DBMS, *Data Base Management System*

SGML	*Standardized General Markup Language* ; ancêtre de HTML et XML
SI	Système d'information ; au sens large, incluant tous les acteurs
SLA	*Service Level Agreement* – Contrat de service ; voir QOS, Qualité de Service
SOA	Service Oriented Architecture
SPEC	*Standard Performance Evaluation Corporation* ; voir www.specbench.org
SQL	*Structured query language* – Langage de requêtes normalisé ISO basé sur l'algèbre relationnelle
SYSML	*System Modeling Language* – Profil UML définit par l'OMG adapté à l'ingénierie système
TCO	*Total Cost of Ownership* – Coût de possession totale / intégrale des coûts sur toute la durée du cycle de vie (coût complet)
TMA	Tierce Maintenance Applicative ; i.e. la maintenance de l'application est confiée à une société tierce
TP	Transaction Program, ou TPR (TP Routine)
TPC	*Transaction Processing Performance Council* ; www.tpc.org
TPM	*Transaction Programme Moniteur* – Moniteur transactionnel
TTCN	*Tabular and Test Combined Notation* – Langage de test définit par le CCITT/ UIT ; voir SDL et ASN.1
UA	Unité Active – Entité du modèle métier (*business process model*) qui produit un effet sur son environnement ; UA atomique et UA moléculaire/agrégée.
UC	Voir CPU
UIP	Utile, Important (ou Indispensable), Primordiale ; pour la graduation des exigences.
UML	*Unified Modeling Language* – Définit par l'OMG
UP	*Unified Process* – Processus unifié, compagnon d'UML
VVT	Validation Vérification et Test
WBS	Voir OT
WSDL	*Web Service Description Language*
XML	Extended Markup language
5W	What, Where, Who, When, Why ; sans oublier How.

Glossaire commenté

Le glossaire est un élément fondamental du référentiel et de la pertinence de la communication entre les acteurs. Par défaut, nous nous référons aux glossaires IEEE et aux glossaires ISO/CEI. Pour les termes généraux, nous utilisons le dictionnaire de A. Lalande, *Vocabulaire technique et critique de la philosophie*, PUF. Pour les termes venant des sciences du vivant, nous utilisons P. Tort, *Dictionnaire du darwinisme et de l'évolution*, PUF.

Abstraction	Propriétés communes à un ensemble d'individus (cf. terminologie de la théorie des ensembles ; dans Bourbaki, notion de relation collectivisante). Classe abstraite dans la terminologie de l'approche objet. À partir d'un certain niveau, l'abstraction ne véhicule plus aucune information intéressante pour l'architecte. En anglais : abstraction, abstract entity (e.g. abstract data type).
Adaptabilité	Capacité d'un système / d'un composant applicatif à s'adapter à un nouveau contexte d'exécution (changement et/ou modification de la plate-forme, etc.). Nécessite un référentiel à jour. Se mesure par un effort d'adaptation, en y incluant la conduite du changement. En anglais : adaptability.
Application	Une application des années 70-80, sur mainframe (1 ou plusieurs exécutable sur une même machine, n'a plus rien à voir avec une application dans une architecture en clients/serveurs. L'application devient un système composé de plusieurs composants applicatifs, sur différents serveurs, qui coopèrent via les API (*application programming interface*) et interagissent via un réseau et/ou des middlewares. En anglais : application.
Architecture	C'est à la fois une activité (la conception, le « design ») et un résultat : le résultat de la phase de conception du système, i.e. l'arbre produit dans la terminologie de la gestion de projet. Cf. le §2.4. En anglais : architecture.
Automate	Mécanisme à états/transitions défini par les règles permettant de passer d'un état initial à un état final, en fonction des événements survenus. Une des notions les plus fondamentales de l'informatique, avec celle de langage. En anglais : automaton.

Caractéristique	Ce qui caractérise, donne une qualité à quelque chose. En qualité, on distingue les caractéristiques fonctionnelles (le quoi) et non fonctionnelles (le comment) ; cf. FURPSE, PESTEL. En anglais : functional and non functional characteristic.
Classification	Critère de classification des éléments d'un ensemble ; classification arborescente. Ordre total (e.g. ordre alphabétique, ordre croissant en mémoire, etc.) et/ou ordre partiel (e.g. les idéogrammes chinois, réseau sémantique). Une classification est toujours arbitraire et dépend du point de vue adopté. En biologie : taxinomie, systématique En anglais : classification.
Complexe	Complexité (i.e. nature de ce qui est complexe) ; réseau de relations entre les éléments (réels ou abstraits) du système. Cf. la notion de réseau sémantique, de dépendances fonctionnelles. C'est une propriété intrinsèque. Voir le chapitre 12. En anglais : complex, complexity.
Complication	Propriété conjoncturelle, liée au système de représentation et à la taille du texte (i.e. le langage, plus ou moins bien adapté au problème à résoudre). Voir le chapitre 12. En anglais : complication.
Composant	Composant logiciel (selon Sziperski), par analogie avec la notion de composant matériel. Le composant réalise une fonction bien définie et a un comportement. Voir §2.1. En anglais : component, building block, chunk.
Couche	Architecture en couches ; découpage d'une transformation en une série d'étapes intermédiaires, i.e. modularité de la transformation. Une couche regroupe des fonctions apparentées. Voir §10.2. En anglais : layered architecture.
Couplage	Ce qui rend des systèmes, des composants applicatifs interdépendants. On distingue les couplages étroits et les couplages faibles. L'interopérabilité est une forme de couplage. Voir §9.3. En anglais : tightly coupled, loosely coupled.
Défaillance	Se manifeste lors de l'exécution d'un défaut. La défaillance peut engendrer une perte partielle ou totale d'un service rendu par le système (en anglais, *fault* peut avoir le sens de panne, comme dans *fault-tolerant*). En anglais : failure.
Défaut	Défaut dans un texte (i.e. une faute d'orthographe et/ou de grammaire, un faux sens, un contre-sens, etc.). Résulte d'une erreur humaine, ou d'une erreur de compilation lors de la traduction du texte en langage informatique. Le défaut est localisé physiquement dans un texte précis, ou diffus dans différents textes. Le taux ou la densité de défauts est une mesure de la qualité d'un texte (courbe de maturité). En anglais : defect, fault.
Déterminisme	Ce qui est prévisible et mécaniquement reproductible ; propriété fondamentale dans la déontologie de l'ingénieur qui doit toujours savoir expliquer les pannes. En anglais : determinism.

Discipline…	Discipline de programmation : concerne la rigueur des comportements et des actions de l'acteur programmeur (ou programmeur/testeur dans la *pair-programming*). Voir la notion de *Personal software process* introduite par W. Humphrey. Cf. le « do it right first time » des experts qualité. En anglais : discipline of programming, discipline of software engineering.
Disponibilité	Capacité d'un système, d'un composant applicatif à satisfaire son contrat de service. Voir le chapitre 13. En anglais : availability, service level agreement.
Dualité	Forme duale, relation de dualité, représentation duale (cf. la notion de dualité en mathématiques). Un automate peut être représenté sous la forme d'un programme impératif ou sous la forme d'une matrice : ce sont des représentations duales d'une même abstraction. Idem pour les langages impératifs versus les langages déclaratifs (i.e. non procédural). En anglais : duality.
Evolution	Capacité d'un système, d'un composant applicatif, à satisfaire de nouvelles exigences. Capacité de croissance (augmentation du périmètre fonctionnel) d'un système ; cf. loi cybernétique de la variété indispensable (R. Ashby). Voir le chapitre 14. En anglais : evolution, requisite variety.
Exigence	Ce que le système doit impérativement faire. Se distingue du besoin qui est une simple demande de l'utilisateur/usager du système. Toute exigence à sa source dans les besoins ressentis ou dans la technologie, mais tous les besoins ne donnent pas naissance à des exigences. On classe les exigences en PIU. En anglais : requirement.
Extension	En théorie des ensembles, énumération de tous les éléments (individus) de l'ensemble (cf. le diagramme sagittal). Une table de logarithme est une représentation en extension. Voir intention. En anglais : extension.
Forme	Il s'agit de la forme logique. Cf. forme logique d'une théorie en mathématique selon les axiomes de la théorie. En informatique, forme logique induite par la structure des interfaces (cf. exemple des compilateurs, avec les langages intermédiaires). En anglais : logical form, logical architecture, logical design.
Grammaire	Ce qui définit les constructions syntaxiques autorisées, dans un langage de programmation. La notation BNF, XML permettent de définir des grammaires. En anglais : grammar.
Hiérarchie	Décomposition hiérarchique d'un système, selon différents niveaux de grain ; arbre produit dans la terminologie de la gestion de projet. Éléments de la hiérarchie et relations entre ces éléments. C'est un problème de partitions (cf. la notion de partition en théorie des ensembles). Voir classification. En anglais : hierarchy.
Indépendance	Indépendance des données et des programmes. Principe fondateur de la gestion de données (data management), et des SGBD où l'on distingue le DDL du DML (i.e. SQL, NDL, ou langage de programmation dans les BDO). Voir §2.3. En anglais : data independance.

Information	Se distingue du bruit. Une information véhicule du sens et a de la valeur. L'information a une syntaxe, une sémantique et une pragmatique. Voir C. Shannon et L. Brillouin, dans la bibliographie. En anglais : information.
Intégrat	Élément du processus d'intégration ; c'est à la fois une entrée et une sortie du processus (intégrats agrégés). En anglais : module, building block.
Intégration	Processus de construction méthodique du système à partir de ses constituants élémentaires (i.e. intégrats de rang 0, ou building blocks). En anglais : integration.
Intention	Critère de définition ou d'appartenance à un ensemble, i.e. une fonction prédicative ; en logique, c'est la fonction propositionnelle qui définit une classe. Bourbaki utilise le terme de relation collectivisante. Un t-uple relationnel, un record COBOL, définissent un ensemble (appelé table en relationnel). En anglais : intention.
Interaction	Entre deux, ou plusieurs, éléments. Échange d'information (données, services, événements) organisée en fonction d'un but. Cf. la notion de couplage et d'interopérabilité. En anglais : interaction.
Interface	Ce qui connecte deux ou plusieurs modules/intégrats et organise la circulation de l'information. Une interface est fondée sur un contrat que les parties prenantes s'engagent à respecter. Voir le chapitre 15. En anglais : interface.
Interopérabilité	Échange, partage d'information (données, services, événements) entre deux ou plusieurs systèmes (acteurs inclus). En anglais : interoperability.
Langage	Concept fondateur de l'informatique. La notion de langage (i.e. ce qui permet d'agir) s'oppose à celle de simple notation (i.e. une sténographie). Un langage est caractérisé par une syntaxe, une sémantique, une pragmatique partagées par tous les locuteurs. Un langage informatique est toujours un compromis entre les usagers humains du langage et la machine sous-jacente (i.e. ce qui agit), qui doit minimiser le temps d'apprentissage et la probabilité d'occurrence d'erreurs humaines. En anglais : language.
Machine	Concept fondateur de l'informatique. machine physique, ordinateur, machine logique (e.g. machine de Turing), machine virtuelle (MV), machine abstraite, machine abstraite à état (MAE). Une machine et son langage sont en relation duale. En anglais : virtual machine, computing instrument (terminologie Von Neumann, i.e. computer), abstract state machine.
Méta…	Méta-langage, méta-donnée. Langage formel (nécessairement) permettant de décrire rigoureusement un autre langage (par exemple, la grammaire d'un langage de programmation exprimée en BNF). En anglais : metalanguage, metadata, metamodel.
Modèle…	modèle conceptuel (porte la sémantique), modèle logique (une représentation dans un certain langage), modèle physique (dans une machine particulière). En anglais : model, conceptual model, logical model, physical model.

Modularité	Mise en application du principe de construction modulaire. Tous les intégrats sont des modules (au sens de D. Parnas). En anglais : modularity.
Module	Unité de construction élémentaire satisfaisant à certaines règles de construction (Cf. D. Parnas). Voir intégrat. En anglais : module, building block.
Moniteur	Concept fondateur de l'informatique. Ce qui pilote, ce qui orchestre. Cf. moniteur temps réel (optimisé sur le temps vrai du monde réel M1) ; moniteur transactionnel (optimisé sur les demandes de ressources formulées par les usagers du système). Voir §10.4. En anglais : monitor, transaction monitor, real-time monitor, driver.
Nomenclature	Énumération des éléments constitutifs d'un système ; c'est un ensemble de types. Peut être organisée en hiérarchie. Configuration et articles de configuration (ACL), en gestion de configuration. En anglais : bill of material, configuration, configuration management.
Organe	Organes (logique et/ou physique) de la machine ; mécanismes qui définissent les modes d'actions de la machine. Voir §1.3.2 et §11.3. En anglais : organ.
Performance	Quantité de ressources nécessaire à l'exécution d'un programme : temps CPU, mémoire centrale, mémoire disque, entrées-sorties, etc. Cf. les informations d'accounting d'un processus. Le nombre de pas de calcul nécessaire à un algorithme est la complexité algorithmique. En anglais : performance.
Pivot	Langage pivot, modèle de données pivot, architecture pivot. Ce qui permet la correspondance N→M en évitant une combinatoire N×M, soit N→Pivot→M. En anglais : hub, neutral format.
Processus	Concept fondateur de l'informatique. Processus séquentiels versus processus coopérants. Un processus séquentiel est une suite d'opérations bien définies qui s'effectuent sur les données gérées par le processus. Plusieurs processus séquentiels peuvent coopérer et synchroniser leurs opérations. Un processus est un procédé, i.e. une façon de faire, qui se déroule dans le temps. Voir le chapitre 7. En anglais : process.
Réaction	Programme réactif (composante réactive du programme, i.e. réaction ou réponse à des événements). On distingue les événements observables et les événements contrôlables (i.e. ceux pour lesquels il y a une réponse appropriée). En anglais : reactive program.
Référentiel	Ensemble des règles à respecter pour maintenir la cohérence logique et physique d'un système, ou d'un système de système. Le niveau de formalisation est un choix de l'architecte, i.e. informel, en langage naturel + schémas, semi-formel (syntaxes rigoureuses, à la BNF ou XML), formel (sémantique rigoureuse, via domaines sémantiques et contraintes). Cf. §2.2.1 et §4.5. En anglais : framework.

Sémantique	Concept fondateur des sciences de l'information. Selon Wittgenstein, « ne demandez pas ce que ça veut dire, demandez à quoi ça sert ». On distingue la sémantique dénotationnelle (qui fait référence à quelque chose d'incontestable, comme les mathématiques) et la sémantique opérationnelle (qui fait référence à une machine abstraite sur laquelle on s'est mis d'accord par convention, i.e. la machine de Turing, ou la machine MIX de D. Knuth dans son *Art of computer programming* ; ou une machine conventionnelle de référence comme la JVM). En anglais : semantic.
Sémaphore	Organe de synchronisation des processus (i.e. un feu rouge). Voir le chapitre 7. En anglais : semaphore.
Service	Se présente comme une instruction généralisée (un block indivisible, FAIT ou NON FAIT), i.e. `sin` ou `cos` dans les bibliothèques de services mathématiques, ou `sort`, `merge`, … en gestion. Voir §9.2. En anglais : service, primitive.
Simplicité	Principe de simplicité (« rasoir d'Ockham ») ou de parcimonie. L'épistémologie (E. Mach, P. Duhem, etc.) utilise des termes comme « économie de pensée » qui est une forme du principe de moindre action de la mécanique. Voir complexité, chapitre 12 . En anglais : simplicity.
Sphère	Concept fondateur de l'informatique transactionnelle (cf. J. Gray). Sphère de contrôle ; ensembles des équipements qui font l'objet d'une surveillance, et pour lesquels un contrôle des événements est programmé (cf. déterminisme). En anglais : sphere of control
Sûreté	Sûreté de fonctionnement ; ce qui permet à l'usager d'avoir confiance dans le système (intégrité, sécurité, accès équitable aux ressources, etc.). Voir le chapitre 13. En anglais : safety, dependability.
Surveillance	Système et/ou organe de surveillance, qui supervise le bon fonctionnement du système et/ou de/des machines (cf. la notion de processeur de service) ; boucle autonome de l' « autonomic computing ». En biologie, c'est le système immunitaire. Du point de vue de la théorie de l'information, c'est une redondance. Voir §3.3. En anglais : autonomic computing, self-healing system.
Syntaxe	Ce qui concerne la structure de l'information, indépendamment du sens. En anglais : syntax, syntactic.
Taille	Longueur d'un programme en nombre d'instructions de tel ou tel langage de programmation ; généralement exprimée en millier, i.e. KLS. La taille fonctionnelle exprimée en point de fonction (PF) est indépendante du langage de programmation. En anglais : size, functional size.
Test	Ce qui établi que le système (ou le composant) est conforme à ses exigences. Le test peut être une expérience (i.e. un cas d'emploi dûment validé par les acteurs), ou une preuve (i.e. une démonstration, de type mathématique). Voir le chapitre 12. En anglais : test.

Testabilité	Aptitude à être testé ; architecture testable, i.e. ce qui optimise l'effort de validation, vérification, test (IVVT). Voir §8.3 et chapitre 13. En anglais : testability, design to test.
Transaction	Concept fondateur de l'informatique. Transaction métier, transaction « technique » ; transaction longue. Transformation irréversible de la réalité. Notion de compensation (dans le domaine du métier bancaire). Voir §9.1. En anglais : transaction, transaction programme routine.
Transformation	Définition commentée : programme transformationnel (composante transformationnelle du programme), i.e. qui effectue une transformation des données en entrée pour obtenir les résultats. En mathématique, une fonction (cf. en informatique la notion de fonction partielle). Voir le chapitre 6. En anglais : transformational program, function, partial function.
Unité…	Unité active (UA). Terme préférable à celui d'agent économique. L'UA est ce qui transforme, grâce à ses capacités et ses ressources. C'est l'élément de construction des chaînes de valeurs de l'entreprise. On distingue les UA atomiques et les UA agrégées. Dans le vocabulaire projet on distingue les processus, les tâches, les activités qui sont différentes formes d'UA. Voir §8.1. En anglais : activity, stream of activities.
Urbanisme	Au sens littéral, organisation rationnelle de la ville (en latin, urbs = ville) selon sa fonction sociale : défense (i.e. le modèle du camp romain, encore apparent dans de nombreuses villes européennes ; ou le château fort du moyen âge (bourg = burg, en allemand, = château) et ses faubourgs, à l'extérieur ; ou la métropole moderne, ouverte sur l'extérieur, qui combine travail, loisirs, culture, etc. Par analogie, organisation rationnelle du SI global de l'entreprise En anglais : information framework architecture, enterprise architecture.
Usure…	Phénomène d'usure (dégradation entropique) d'un logiciel en exécution, lié à sa durée de fonctionnement qui va se traduire par des pertes aléatoires de ressources qui affectent son comportement. Cf. la notion de ligne de vie. Voir §3.2. En anglais : wear out, decay.
Vecteur…	Vecteur d'état qui résume l'historique de l'exécution d'un service ou le statut d'une opération. Voir §11.3. En anglais : state vector, status, program status word (PSW).
Vérité	Garantie de conformité aux exigences à l'aide de tests et/ou de preuves. L'ordinateur est binaire, soit VRAI/FAUX, alors que la réalité est modale. La « vérité » du point de vue métier est portée par des cas d'emplois validés par les usagers. En anglais : verity, truth.

Bibliographie

Ne figurent dans cette bibliographie que les ouvrages connus de l'auteur directement en rapport avec la problématique des architectures logicielles. Elle ne prétend pas à l'exhaustivité. Le texte contient d'autres références. Celles marquées * sont ma « *Top 20 list* ».

Abrial,J – *The B-Book Assigning programs to meanings*, Cambridge University Press, 1996.

*Aho,A Sethi,R Ullman,J – *Compilers, Principles, techniques, and tools*, Addison-Wesley, 1986.

Aho,A Ullman,J – *The theory of parsing, translation, and compiling*, Prentice-Hall, 1972.

Arndt,C – *Information measures Information and its description in science and engineering*, Springer, 2004.

Ashby,W – *An introduction to cybernetics*, Methuen Ltd, 1979.

Bach,M – *The design of the UNIX operating system*, Prentice Hall, 1986.

Bass,L Clements,P Kazman,R – *Evaluating software architecture*, Addison-Wesley, 2002.

Bass,L Clements,P Kazman,R – *Software architecture in practice*, Addison-Wesley, 2003.

Bayley,R – *Human performance engineering*, Prentice Hall, 1989.

*Beizer,B – *Software testing techniques*, Van Norstrand Rheinhold, 1990.

Bernstein,P Newcomer,E – *Principles of transaction processing*, Morgan Kaufmann, 1997.

Binder,R – *Testing object-oriented systems*, Addison-Wesley, 1999.

*Boehm,B – *Software engineering economics*, Prentice Hall, 1981.

Boehm,B et alii – *Software cost estimation with COCOMO II*, Prentice Hall, 2000.

Börger,E StärkR – *Abstract state machines*, Springer 2003.

Boydens,I – *Informatique, normes et temps*, Bruylant Bruxelles, 1999.

Brillouin,L – *La science et la théorie de l'information*, Réimpression J.Gabay, 1988.

*Brinch Hansen,P – *Operating system principles*, Prentice Hall, 1973.

Brinch Hansen,P – *The origin of concurrent programming, From semaphores to remote procedure calls*, Springer, 2002.

*Brooks,F – *The mythical man-month*, Addison-Wesley, 1975.

Broy,M Denert,E – *Software pioneers, Contributions to software engineering*, Springer, 2002.

Caseau,Y – *Urbanisation et BPM, Le point de vue d'un DSI*, Dunod, 2004.

Chauvet,J-M – *Services Web avec SOAP, WSDL, UDDI, ebXML …*, Eyrolles, 2002.

Chevance,R – *Serveurs multiprocesseurs, clusters et architectures parallèles*, Eyrolles, 2000.

Clements,P et al. – *Documenting software architectures*, Views and beyond, Addison Wesley, 2003.

Codd,E – *The relational model for database management : Version 2*, Addison-Wesley, 1990.

Colwell,R – *The Pentium chronicles*, Wiley-Interscience, 2006.

Custer,H – *Inside WINDOWS NT*, Microsoft Press, 1993.

Cusumano, M Selby,R – *Microsoft secrets*, Free Press, 1995.

Cypser,R – *Communications for cooperating systems OSI, SNA and TCP/IP*, Addison-Wesley, 1991.

Delattre,P – *Système, structure, fonction, évolution Essai d'analyse épistémologique*, Maloine, 1985.

*Deming,P Dennis,J Qualitz,J – *Machines, languages and computation*, Prentice Hall, 1978.

Dijkstra,E – *Cooperating sequential processes*, 1965 (cf. Brinch Hansen).

DOD Architecture Framework (DODAF), Vol I, *Definitions and guidelines*, Vol II *Products descriptions*, Final draft, 2003.

Erl,T – *Service-Oriented architecture, concepts, technology, and design*, Prentice Hall, 2005.

Forrester,J – *Principles of systems*, Cambridge, Pegasus, ISBN : 1-883823-41-2.

Fowler,M – *UML Distilled*, Addison-Wesley, 2004.

Gamma,E Helm,R Johnson,R Vlissides,J – *Design Patterns*, Addison-Wesley, 1995.

*Gray,J Reuter,A – *Transaction processing: concepts and techniques*, Morgan Kaufmann, 1993.

Gries,D – *Compiler construction for digital computer*, Wiley, 1971.

*Hennessy,J Patterson,D – *Computer architecture, A quantitative approach*, Morgan Kaufmann, 1990.

Hennessy,J Patterson,D – *Computer organization and design, The hardware/software interface*, Morgan Kaufmann, 1998.

*Hopcroft,J Ullman,J – *Formal languages and their relation to automata*, Addison Wesley, 1969.

Jacobson,I Booch,G Rumbaugh,J – *The Unified Software Development Process*, Addison Wesley, 2005.

*Jain,R – *The art of computer systems performance analysis*, Wiley, 1991.

Jorgensen,P – *Software testing, A craftman's approach*, CRC Press, 2002.

*Knuth,D – *Art of computer programming*, 3 Vol, Addison Wesley.

Ladrière,J – *Les limitations internes des formalismes, Etude sur la signification du théorème de Gödel*, Réimpression J.Gabay, 1994.

Lehman,M Belady,L – *Program evolution, processes of software change*, Academic Press, 1985.

Leveson,N – *Safeware, System safety and computers*, Addison-Wesley, 1995.

Maier,M Rechtin,E – *The art of systems architecting*, CRC Press, 2002.

Manna,Z Pnueli,A – *Temporal verification of reactive systems*, Springer-Verlag, 1995.

Manna,Z Pnueli,A – *The temporal logic of reactive and concurrent systems*, Springer-Verlag, 1992.

McNutt,B – *The fractal structure of data reference*, Kluver, 2000.

Meinadier, J-P – *Ingénierie et intégration des systèmes*, Hermès, 1998.

Meinadier, J-P – *Le métier d'intégration de systèmes*, Lavoisier, 2002

Messerschmitt,D Szyperski,C – *Software ecosystem*, MIT Press, 2003.

Meyer,B – *Conception et programmation objet*, Eyrolles, 1997.

Minsky,M – *Computation: Finite and infinite machines*, Prentice Hall, 1967.

Mitschele-Thiel,A – *System engineering with SDL*, Wiley, 2001.

Muchnick,S – *Advanced compiler design implementation*, Morgan Kaufmann, 1997.

*NASA *Systems engineering handbook*, 1995.

NATO Science committe, *Software engineering*, Garmisch, 1968 et Rome, 1969.

*Normes : IEEE Software engineering – Standards collection (Une quarantaine de titres).

Normes : UIT-CCITT, Libre bleu, Tome VIII.4, Recommandations X.200 à X.219, Interconnexion de systèmes ouverts (OSI) ; modèle et notation, définition du service.

Normes : UIT-T, Z.100, Langage de description et de spécification du CCITT (SDL) ; Z.120, Diagrammes de séquences de messages.

Ogunnaike,B Ray,W – *Process dynamics, modeling, and control*, Oxford University Press, 1994.

Orfali,R Harkey,D Edwards,J – *Client/server survival guide*, Wiley & Sons, 1999.

*Organick,E – *The MULTICS system: an examination of its structure*, MIT Press 1972.

Parnas,D – *On the criteria to be used in decomposing systems into modules*, Communications of the ACM, Vol. 15(12), 1972 ; réimpression dans Broy.

Perroux,F – *Unités actives et mathématiques nouvelles*, Dunod, 1975.

Printz,J – *Le génie logiciel*, PUF, Collection Que sais-je ?N°2956, Traduit en chinois.

Printz,J – *Productivité des programmeurs*, Hermès, 2001.

Printz,J – *Puissance et limites des systèmes informatisés*, Hermès, 1998.

Printz,J Mesdon,B – *Ecosystème des projets informatiques*, Lavoisier, 2006.

Printz,J, Deh,C Mesdon,B Trèves,N – *Coûts et durée des projets informatiques*, Hermès, 2001.

Pugh,E Johnson,L Palmer,J – *IBM's 360 and early 370 Systems*, MIT Press, 1991.

*Rechtin,E – *Systems architecting, Creating & building complex systems*, Prentice Hall, 1991.

Redmond,K Smith,T – *From Whirlwind to MITRE, The R&D story of the SAGE air defense computer*, MIT Press, 2000.

Revues spécialisées : ACM Computing surveys – Revue de synthèse de l'Association for Computing Machinery avec de nombreux articles d'architecture (Quatre N° par an).

Revues spécialisées : IBM Systems Journal – Revue technique de haute qualité reflétant une partie des travaux des équipes de R&D d'IBM – Nombreux articles et/ou N° spéciaux sur l'architecture (Quatre N° par an).

Rumbaugh,J Jacobson,I Booch,G – *The Unified Modeling Language reference Manual*, Addison Wesley, 2005.

Scott Geller,E – The psychology of safety handbook, CRC Press, 2001.

Servin,C – *Réseaux et télécoms*, Dunod, 2003.

Shannon,C Weaver,W – *The mathematical theory of communication*, University of Illinois, 1959.

Shaw,M Garlan,D – *Software architecture, Perspectives on an emerging discipline*, Prentice Hall, 1996.

*Simon,H – *The science of the artificial*, MIT Press.

*Soltis,F – *Inside the AS400*, News400 Books, 1997.

Szyperski,C – *Component software*, Addison-Wesley, 2002.

Tanenbaum,A – *Computer Networks*, Prentice Hall, 1989.

Tanenbaum,A – *Distributed operating systems*, Prentice Hall, 1995.

Tanenbaum,A – *Operating systems, Design and implementation*, Prentice Hall, 1987.

*Teorey,T – *Database modeling and design*, Morgan Kaufman, 1999.

Thom,R – *Stabilité structurelle et morphogenèse, Essai d'une théorie générale des modèles*, InerEditions, 1977.

Tsichritzis,D Lochovsky,F – *Data models*, Prentice Hall, 1982.

Turing,A Girard,J-Y – *La machine de Turing*, Le Seuil, Points Sciences, 1995.

Ullman,J – *Principles of database and knowledge-base systems*, 2 Vol., Computer Science Press, 1988.

*Von Neumann,J – *Collected works*, Vol V, Pergamon Press.

*Von Neumann,J – *Theory of self-reproducing automata*, University of Illinois Press.

Walliser,B – *Systèmes et modèles Introduction critique à l'analyse des systèmes*, Seuil, 1977.

Walraet,B – *Programming, the impossible challenge*, North-Holland, 1989.

Wiener,N – *Cybernetics or control and communication in the animal and the machine*, Hermann, 1948.

Willcock,C & al. – *An introduction to TTCN-3*, Wiley, 2005.

Winograd,S Cowan,J – *Reliable computation in the presence of noise*, MIT Press, 1963.

Winograd,T – *Language as a cognitive process, Syntax*, Addison-Wesley, 1982.

Wirth,N – *Algorithms + Data structures = Programs*, Prentice Hall, 1976.

Yeh,R – *Current trends in programming methodology*, Vol I, *Software specification* and design, Vol IV, *Data structuring*, Prentice Hall, 1977.

Index

S

T

U

V

W

49910 – (I) – (1,8) – OSB 100° – ARC – MER

Achevé d'imprimer sur les presses de
SNEL Grafics sa
Z.I. des Hauts-Sarts - Zone 3
Rue Fond des Fourches 21 – B-4041 Vottem (Herstal)
Tél +32(0)4 344 65 60 - Fax +32(0)4 289 99 61
octobre 2006 – 40286

Dépôt légal : novembre 2006
Imprimé en Belgique

INFOPRO

TYPE D'OUVRAGE

L'ESSENTIEL	SE FORMER	**RETOURS D'EXPÉRIENCE**

MANAGEMENT DES SYSTÈMES D'INFORMATION

APPLICATIONS MÉTIERS

ÉTUDES, DÉVELOPPEMENT, INTÉGRATION

EXPLOITATION ET ADMINISTRATION

RÉSEAUX & TÉLÉCOMS

Jacques Printz

Préface de Yves Caseau

ARCHITECTURE LOGICIELLE

Concevoir des applications simples, sûres et adaptables

Cet ouvrage s'adresse aux décideurs que sont les DSI et les maîtres d'ouvrage, ainsi qu'aux chefs de projets et aux architectes. Il intéressera également les étudiants et élèves ingénieurs au niveau du master informatique.

Quelle place l'architecture logicielle tient-elle dans les projets en termes de coût/qualité/délai ? Comment les contraintes de simplicité, de disponibilité et d'évolutivité doivent-elles être prises en compte dès la conception ?

Cet ouvrage propose des modèles d'architectures indépendants des technologies utilisées qui permettent d'atteindre les objectifs de qualité recherchés. Il replace les concepts architecturaux récents (Client/Serveur, SOA, MDA, WSDL, BPM, REST…) sur les bases solides que sont les traducteurs, la communication inter-processus, les transactions et les machines virtuelles.

La première partie présente la problématique de l'architecture logicielle dans sa globalité en insistant sur les correspondances entre le monde réel et les modèles. La seconde explique les concepts fondateurs que sont l'architecture des compilateurs, l'architecture des processus et leur synchronisation. La troisième définit les principes des architectures fonctionnelles logiques et de leur construction méthodique. La dernière explique comment parvenir à respecter les quatre propriétés fondamentales des architectures :

- la simplicité,
- la disponibilité et la sûreté de fonctionnement,
- l'adaptabilité et l'évolutivité,
- la construction des interfaces.

JACQUES PRINTZ

est professeur au Conservatoire national des arts et métiers, titulaire de la Chaire de génie logiciel. Ingénieur diplômé de l'École Centrale de Paris, il a effectué sa carrière professionnelle chez Bull, où il a contribué à la création du système d'exploitation GCOS7, puis dans des sociétés de services, puis comme consultant auprès de grandes entreprises et administrations (ministère de la défense, EDF, RTE, THALES…).
Au CNAM, il a créé le Centre de maîtrise des systèmes et du logiciel (CMSL), sur le thème des systèmes complexes.

9 782100 499106

ISBN 2 10 049910 6

www.dunod.com

DUNOD