Eléments pour une mesure naturelle de la complexité des systèmes et de leur architecture

Jacques Printz, Professeur au CNAM, Chaire de génie logiciel

"The great progress in every science came when, in the study of problems which were modest as compared with ultimate aims, methods were developed which could be extended further and further. … The sound procedure is to obtain first utmost precision and mastery in a limited field, and then to proceed to another, some that wider, and so on. … The experience of more advanced sciences indicates that impatience merely delays progress, including that of treatment of the «burning» questions. There is no reason to assume the existence of shortcuts."

John von Neumann, *Theory of Games and Economic Behaviour*

TABLE DES MATIERES

TABLE DES FIGURES

TABLE DES SIGLES

CQFD	Coût Qualité Fonctionnalité Durée/délai – Les 4 grandeurs des projets
FURPSE	Fonctionnalité, Usabilité (facilité d'emploi), Fiabilité (reliability), Performance, Maintenabilité (serviceability), Evolutivité – Classification des caractéristiques qualité selon norme ISO/CEI 9126
PESTEL	Contraintes externes sur les exigences FURPSE : Politique, Economique, Social, Technologique, Environnement, Légal – Classification des facteurs de l'écosystème projet (terminologie INCOSE).
ROI	Return Of Investment – Retour sur investissement
SDS	Système de systèmes – Un système dont les éléments sont eux-mêmes des systèmes, interconnectés via tout type de réseaux.
TCO	Total Cost of Ownership – Coût de possession totale / intégrale des coûts sur toute la durée du cycle de vie (coût complet).

GENERALITES SUR LA MESURE DE LA COMPLEXITE DES SYSTEMES ET DES ARCHITECTURES

Le problème de la mesure de la complexité d'un logiciel et de son architecture est un thème récurrent du génie logiciel, depuis sa mise sur les fonds baptismaux lors des deux conférences du NATO *Science Committee*, en 1968 et 1969 (Réf. [1]). Depuis cette date de nombreux articles et ouvrages ont traité du problème, selon différentes perspectives. Parmi les plus notables, citons les références [2], [3] et [4].

J'ai moi-même abordé le sujet dans un certain nombre de mes ouvrages, en particulier [5], [6], [7] et [8], auxquels je renvoie pour plus de détails. Le sujet est ardu et s'apparente à la définition du Bouddha qu'avait donné un moine Zen à un néophyte : « C'est un bâton merdeux » qu'on ne sait pas par quel bout prendre sans s'en mettre plein les mains !

Pour clarifier le débat il nous faut trouver une « mesure » suffisamment simple pour que tous les acteurs concernés la comprennent, mais également pertinente pour traduire correctement l'intuition des acteurs privilégiés, chefs de projet et surtout architectes, afin que ceux-ci puissent l'utiliser pour prendre des décisions utiles au projet qu'ils pilotent. Sans prétention, ni préjugé, nous allons explorer quelques pistes.

- Pour un chef de projet, complexe va vouloir dire coûteux, en termes CQFD/FURPSE.
- Pour un maître d'ouvrage ou une direction informatique, complexe voudra dire non seulement coûteux à développer, mais surtout coûteux à déployer, à exploiter, à maintenir qui sont des coûts récurrents : c'est à la fois un coût de possession totale (TCO) et un retour sur investissement (ROI).
- Pour un architecte concepteur, le volume des spécifications à produire et la méthodologie de développement, le nombre de « contrats » d'interfaces pour les programmeurs, i.e. le nombre de « pièces », ou modules (au sens de D.Parnas, réf.[24]), à réaliser et leurs relations, seront des facteurs déterminants de la complexité.
- Pour un architecte intégrateur, ce sera plutôt, outre le nombre de pièces, le volume de tests à produire, le plan d'intégration, i.e. l'ordonnancement du travail d'intégration, les environnements de tests qui seront perçus comme significatifs.

Dans un article à paraître dans la revue Génie Logiciel, *Complexité des systèmes d'information : une famille de mesures de la complexité scalaire d'un schéma d'architecture*, Y.Caseau, D.Krob et S.Peyronnet ont proposé une mesure basée sur la notion de graphes récursifs pondérés, avec des idées intéressantes comme l'*architecture observable* et l'*architecture organique* permettant de séparer la composante métier de la composante technique d'une architecture au sens large. A l'aide de cette mesure de *complexité scalaire d'ordre p* (la profondeur du graphe récursif), ils retrouvent les propriétés bien connues des architectures hiérarchiques.

Le problème est cependant de s'assurer que : 1) l'architecture est bien hiérarchique, ou que le graphe est bien récursif, ce qui n'est pas une donnée mais une construction (NB : pour avoir longtemps œuvré dans le domaine des langages et de la compilation, la tendance naturelle des programmeurs est plutôt l'inverse, même si on interdit

l'instruction `go to`), et 2) la mesure est « naturelle » pour les acteurs qui auront à s'en servir. C'est le but de cet article.

Analogie physique

Réfléchissons d'abord à ce qu'est une mesure en prenant comme point de départ ce que nous enseigne la métrologie des grandeurs physiques (Cf. réf. [28]). Constatons que cette notion fondamentale pour les sciences de l'ingénieur n'émerge que très progressivement du brouillard métaphysique qualitatif (cf. la « théorie » du phlogistique encore en vigueur à l'époque de Newton !) pour devenir ce qui est aujourd'hui enseigné en physique, au chapitre métrologie. L'histoire des sciences et des techniques nous rappelle à la prudence : trouver de « bonnes » mesures utiles est un processus lent qui nécessite une compréhension profonde des phénomènes observés. Or les sciences de l'information ne datent que du milieu du 20^{ième} siècle !

Prenons pour illustrer le propos l'exemple de la température que l'on n'a su définir correctement qu'au 19^{ième} siècle. Au départ, comme souvent en physique, c'est une sensation, comme le chaud et le froid que l'on ressent par contact, ce qui nous fait dire que le fer est plus « froid » que le bois. En fait, la sensation ne mesure qu'un gradient énergétique et une conductibilité thermique qui certes résulte d'une différence de température, celle de notre corps aux alentours de 37° et celle du milieu ambiant. Cette sensation n'est pas une bonne mesure de la température. Le phénomène observé et l'observateur sont trop étroitement imbriqués (intriqués, dirait-on en mécanique quantique !)

Plus tard, en observant mieux la nature, on remarque que le « chaud » fait se dilater les corps solides, liquides ou gazeux, et ce indépendamment de l'observateur. L'augmentation de température provoque une dilatation que l'on peut repérer avec une règle : c'est le thermomètre. On ne sait pas expliquer ce que signifie « 2 fois plus chaud » mais on découvre au passage l'existence du zéro absolu et de la température absolue, qui sont des notions fondamentales en thermodynamique.

Encore plus tard, vers la fin du 19^{ième} siècle et début du 20^{ième}, avec la théorie atomique, on arrive aux termes du voyage. La température est une mesure de l'agitation des atomes et/ou des molécules, avec de nouvelles notions comme l'entropie, l'enthalpie, etc. et des théories profondes comme la cinétique des gaz où s'illustrent J.Maxwell (avec son « démon »), L.Boltzman et W.Gibbs. La température mesure le degré de désordre.

Enfin, l'ingénieur C.Shannon, en réfléchissant sur les moyens d'optimiser la capacité de transmission des lignes téléphoniques naissantes, dans les années 30-40, définit une mesure de l'information transmise qui est la taille d'un signal, en nombre de bits, qui code l'information à transmettre. La structure du code, le nombre de symboles utilisés, la fréquence d'emploi des symboles deviennent des facteurs dimensionnant, et la formule de Shannon révèle une étonnante parenté formelle avec celle de Boltzman, ce qui permet à certains d'affirmer que l'information, au sens de Shannon, équivaut à une entropie, ou plutôt une néguentropie, à cause du signe ; mais là, nous quittons le monde des ingénieurs et de la physique pour celui des philosophes et de la métaphysique. Sur cette histoire, nous renvoyons à deux ouvrages fondateurs [9] et [10].

De l'agitation moléculaire, et ce quel que soit le corps, quelle que soit la masse ou le volume, émerge une seule grandeur macroscopique scalaire : la température. Cette grandeur donne une vue statistique moyenne, simplifiée mais non simpliste, de la réalité complexe sous-jacente fort utile en thermodynamique industrielle.

Le thermomètre, instrument de mesure de la température, appartient lui-même au monde macroscopique. Sa construction et ses conditions de mise en œuvre requièrent quelques précautions si l'on veut obtenir une mesure fidèle, comme cela est expliqué dans les manuels de physique, au chapitre métrologie.

On sait par ailleurs que si l'on descend au niveau microscopique, la mesure en tant que telle disparaît, ainsi que l'instrument pour la mesurer qui perturbe le phénomène, ce qui nous renvoie à la physique quantique et aux relations d'incertitudes d'Heisenberg.

Comme toute mesure, la température n'est pas donnée mais construite. C'est une grandeur qui dépend de l'échelle à laquelle on observe le phénomène thermique.

Aspect fractal de la mesure – Le problème du choix de l'étalon de mesure

On se souvient de la question d'apparence anodine mais cependant profonde de B.Mandelbrot (cf. réf. [11]) : « Quelle est la longueur de la cote de Bretagne » ? Selon que l'on est bactérie, mouche ou porte-avions, ou … la réponse est différente bien qu'il s'agisse d'une même réalité, du moins en apparence. De plus, quelle que soit l'échelle de l'observation, il semble que la forme dite « fractale » soit la même. On retrouve la même structure à différents niveaux d'abstraction.

Mandelbrot s'est beaucoup intéressé (R.Thom également) à ces étranges lois d'échelles, comme la loi de Zipf (Cf. réf. [20]), qui semblent dénoter quelque chose de profond, qui marche mais qu'on ne sait pas expliquer. Un des premiers livres relativement complet sur le génie logiciel, celui de M.Shooman (Cf. réf. [12]) en fait un élément central d'un chapitre consacré à « *Complexity, storage and processing-time analysis* », soit quand même 72 pages. Puis cela disparaît des ouvrages classiques comme ceux de B.Boehm, R.Pressman, I.Sommerville (Cf. réf. [4], [13], [14]).

Ceci est d'autant plus surprenant que la notion de niveau d'abstraction est centrale dans l'ingénierie du logiciel, tant au niveau de l'architecture des programmes que de l'organisation des projets (certains auteurs comme W.Humphrey ont parlé, à juste titre, de *Programming process architecture* ; cf. réf [21]) pour mettre en évidence la dualité des notions.

Dans un ordinateur, il est classique de considérer différents niveaux d'abstraction qui correspondent à des machines abstraites qui s'imbriquent les unes dans les autres, chacune avec sa mémoire, ses données, son jeu d'instructions et sa périphérie. Merci von Neumann, merci Turing !

A chacun des niveaux, un langage associé qui constitue le « code » de la machine correspondante. Avec ce code, on écrit des programmes qui peuvent se traduire les uns dans les autres (ou s'interpréter), et ce jusqu'au niveau hardware où les abstractions immatérielles sont traduites en phénomènes électroniques.

Au plus haut niveau de la hiérarchie d'interfaces, on a les langages de commande de la machine où le « grain » sémantique est l'application, le fichier, la base de données, etc. …, puis des pilotes de processus (composants applicatifs) avec des langages comme

BPML/BPEL et les moniteurs transactionnels. Ces niveaux concernent surtout les exploitants et les usagers, mais des machines comme l'AS400 d'IBM avec son OS/400 ont proposé un continuum d'interfaces depuis le niveau commande jusqu'au niveau programmation, comme cela avait été ébauché dans le système MULTICS du MIT avec le langage PL1.

Pour programmer les processus eux-mêmes il y a les langages de programmation, pour les programmeurs. Ceux-ci sont traduits par les compilateurs dans le langage machine, i.e. l'assembleur, qui définit l'interface hardware-software. Ce jeu d'instructions ultime détermine les capacités de la machine. Dans les années 80s, on a beaucoup discuté des mérites comparés des machines CISC et RISC, et de la simplicité des jeux d'instructions.

Mais la descente n'est pas finie pour autant car au sein même du microprocesseur, il y a d'autres niveaux de décomposition avant d'atteindre le niveau physique proprement dit (Cf. les ouvrages de références de Hennessy et Patterson en [15] et [16]).

Sur chacun des niveaux, on aura une mesure de complexité différente pour une même sémantique.

Choisir un étalon, c'est d'abord choisir un niveau d'abstraction. Mélanger sans discernement les niveaux d'abstraction peut conduire à des absurdités logiques et à des « mesures » dénuées de sens.

Une hiérarchie de langages – LM et LHN

Bien que cela puisse apparaître comme une évidence dépassée, intéressons nous quelques instants à l'interface hardware software de la machine. Cette interface définit la façon dont le programmeur système, par exemple un programmeur de compilateur ou de SGBD, « voit » la machine. Elle est la connaissance nécessaire et suffisante pour produire des programmes sémantiquement cohérents qui pourront être vérifiés et validés. Pour le programmeur système, c'est **LE** référentiel de la machine, en quelque sorte sa bible, qu'il doit connaître aussi parfaitement que possible.

NB : sur une des machines sur laquelle j'ai eu l'occasion de travailler pendant 15 ans, c'était un document d'environ 3000 pages.

Cette interface est-elle pour autant l'architecture avec un grand **A** ? Certainement pas car de nombreuses structures hardware vont rester cachées aux programmeurs systèmes, comme les caches et les pipe-lines d'instructions et de données, les mémoires associatives pour l'adressage, les codes correcteurs d'erreurs (9 à 11 bits par mot mémoire) qui protègent la mémoire des phénomènes aléatoires comme les particules α, le bruit thermique, les perturbations électromagnétiques, etc. ; cf. réf. [17]).

Par contre, il n'est pas faux de dire que cette interface dénote la « complexité d'usage » de la machine du point de vue du programmeur système. Pour le concepteur du microprocesseur, c'est la spécification du besoin et les exigences FURPSE qu'il devra satisfaire vis à vis de l'extérieur.

Pour les ingénieurs de maintenance de la machine, dans l'équipe d'exploitation, le point de vue peut être différent. En particulier, cet acteur fondamental pour la garantie du SLA peut avoir accès au « panneau de contrôle », interface autrefois bien visible et faisant partie de la mythologie de l'ordinateur, i.e. les diodes qui clignotent, aujourd'hui

interface virtuelle, elle-même commandée par un langage ad hoc, mais toujours interface fondamentale pour la surveillance et le diagnostic des pannes hardware (cf. les logiciels d'aide à l'exploitation, l'initiative « autonomic computing » d'IBM, etc.).

Sur cet exemple de la machine, on voit apparaître 3 facettes de la complexité, et conséquemment 3 mesures différentes, selon l'acteur concerné :

- La complexité d'usage **C1-LM1**, i.e. le U de FURPSE, qui concerne ceux qui se servent de l'interface de programmation.
- La complexité d'exploitation **C2**, i.e. le SE de FURPSE, qui concerne les ingénieurs d'exploitation et de maintenance.
- La complexité de conception de l'interface **C3**, i.e. le FRP de FURPSE, qui dans ce cas concerne les ingénieurs hardware.

Avec la série 360, IBM a démontré dans les années 60s qu'il était possible de stabiliser C1-LM1, tout en offrant une large variété d'implémentation de C3 selon les exigences RP. Il est donc légitime de parler de COMPLEXITES, au pluriel ! Comme le montre le schéma ci-dessous.

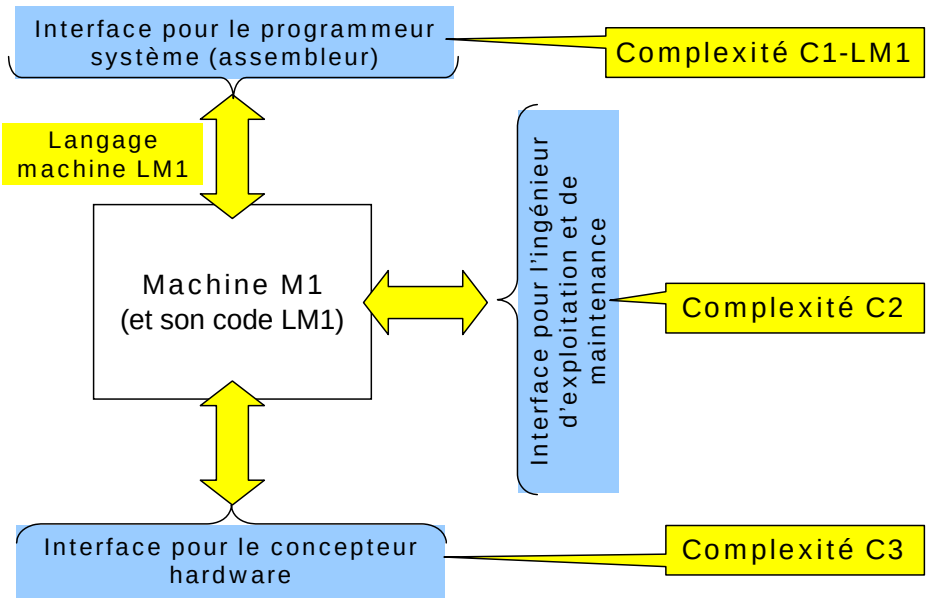


Figure 1 – Les complexités de l'architecture d'un ordinateur

Pour un chef de projet logiciel, seule C1 est intéressante.

Pour le directeur de l'exploitation seule C2 est intéressante.

Pour le DSI ou le directeur informatique, C1 et C2 sont intéressantes ; la somme des deux conditionne le TCO.

C3 intéresse la direction des achats dans la mesure où elle détermine le coût de la machine.

Très tôt dans la brève histoire de l'informatique l'interface C1-LM1 est apparue inadaptée aux besoins des programmeurs d'applications scientifiques et/ou de gestion qui n'avaient que faire de la gestion fine des ressources de la machine. Du point de vue

projet, les critères économiques CQFD deviennent prépondérants, c'est eux qu'il faut optimiser ; ceci était clairement établi dès le début des années 70s, B.Boehm (Cf. réf. [4]) ayant beaucoup œuvré dans ce sens. Ainsi sont nés FORTRAN et COBOL, sous la pression des usagers, et non celle des concepteurs de machines qui n'en voulaient pas. IBM s'est longtemps opposée à COBOL !

Du point de vue de la métrologie dans la perspective projet qui nous intéresse ici, les langages de haut niveau ont un double apport :

- Ils débarrassent le programmeur d'applications des tâches de gestion des ressources fines que doit prendre en charge son collègue programmeur système, et lui évitent ainsi de s'investir dans des problématiques certes intéressantes comme l'optimisation des ressources, mais qui ont toutes le désavantage de rendre l'application plus complexe, donc plus coûteuse.
- Ils permettent de créer des structures qui permettent aux programmeurs d'applications une description plus naturelle, dans un langage « à eux », des problèmes de conception et des solutions de programmation. Vecteurs et matrices pour FORTRAN, RECORD et fichiers pour COBOL, les deux complétés par des bibliothèques de fonctions préprogrammées intégrées à chacun des langages. COBOL ira même jusqu'à intégrer un langage de navigation pour les bases de données du modèle réseau défini par le CODASYL.

Il est à noter que ces deux langages, au départ, n'ont pas les idées claires sur se qu'on appellera plus tard les sous-programmes, et les procédures compilées séparément. COBOL invente même une construction franchement absurde pour un théoricien des langages, l'instruction **`PERFORM`**, mais qui à vrai dire ne choque pas vraiment les analystes et programmeurs COBOL. 20 ans plus tard, avec le langage C, on n'a pas vraiment fait mieux.

Ce faisant, on a augmenté ce qu'on a appelé le « niveau » du langage, ou sa « puissance », ce qui fait qu'à partir d'un même dossier de programmation, le programmeur produira un texte écrit 2 à 3 fois plus concis, en nombre d'instructions. Dans les années 70s, où les intégristes de l'assembleur étaient encore nombreux (et fiers de leurs astuces de programmation coûteuses) il fallait démontrer que les compilateurs étaient de meilleurs programmeurs que les « vrais » programmeurs. La performance, comme on disait, du code généré par le compilateur était un exercice d'explication obligatoire pour convaincre les sceptiques, d'où de nombreuses comparaisons et des statistiques abondantes, mais le débat n'a vraiment été tranché qu'au début des années 80s. On dispose désormais de tables qui expriment le coefficient de puissance du niveau de langage pour un grand nombre de langages (Cf. réf. [5], [6] et [18]), pour lesquelles Caper Jones a beaucoup œuvré).

Le langage de haut niveau donne une vue abstraite de la machine dans un langage plus naturel pour l'analyste et le programmeur d'application. Dans les années 70s, il y a eu un courant important sur ce qu'on a appelé les « machines langages », y compris avec des aides hardware (par exemple opérateurs microprogrammés pour le tri, le hachage, etc.). On parlait de « machines étendues » APL (un succédané du FORTRAN inventé par K.Iverson chez IBM), COBOL, PL1, etc. La figure 1 devient :

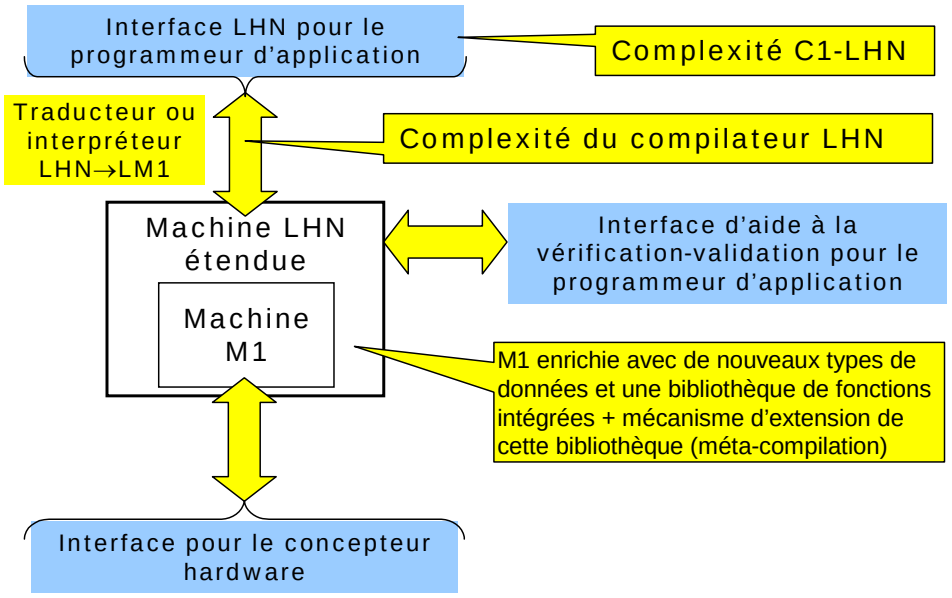


Figure 2 – Machine étendue par le langage de haut niveau

On notera au passage le dédoublement de l'interface C2, avec un appendice pour la mise au point des programmes qui se matérialisera avec les outils de *debuging* symbolique, plus commode que le panneau de contrôle, mais qui sont dépendants de chaque langage ; c'est une nouvelle complexité.

Remarque : il serait intéressant d'analyser sous cet angle les distinctions introduites par l'OMG dans l'approche MDA, i.e. le niveau PIM (*Plate-form Independent Model*) et PSM (*Plate-form Specific Model*) ; la plate-forme jouant ici le rôle de « machine ».

Quantité d'information et codage

En termes de code, au sens de la théorie de l'information, le code de la machine LHN est plus concis que celui de la machine LM. Il y a moins de types de données (instructions déclaratives) et beaucoup moins de types d'opérations (instructions impératives), 10 à 15 contre plusieurs centaines d'instructions assembleur. Pour un même besoin métier, la quantité d'information du texte LHN est beaucoup plus faible, car la fréquence d'emploi des opérations est plus grande. Si l'on prend le cas de l'opérateur + en LHN, il y aura couramment 20 à 30 instructions machine possibles selon le contexte, donc une fréquence d'emploi 20 à 30 fois moindre. On pourrait faire un calcul précis en utilisant les statistiques de fréquence d'emploi des instructions machine qui sont utilisées pour calibrer et optimiser les unités centrales des ordinateurs (Cf. le « *instruction* MIX », voir réf. [15] et [16]).

NB : Dans une architecture RISC, grâce au jeu d'instructions « plus carré » on fait baisser la quantité d'information des programmes. De fait, le générateur de code d'un compilateur LHN est significativement plus simple sur un RISC que sur un CISC. On pourrait le chiffrer en KLS (c'est l'architecte de compilateurs qui parle !). Cela montre la dualité entre complexité des interfaces et volume de code pour les compiler/interpréter, et le transfert de complexité du programme vers le compilateur qui factorise cette complexité au bénéfice de tous les programmeurs.

Pour un programmeur LHN, écrire `a=b` (3 symboles), `a=b+c` (5 symboles) ou `a=b+c×d` (7 symboles), c'est du pareil au même en termes de coût de fabrication (Cf. le chapitre 2.5, *Productivité du programmeur isolé parfait : le modèle PIP*, dans *Productivité des programmeurs* [6]). D'un point de vue psycho-cognitif, c'est un quantum d'information, une « vérité » statistique (merci la loi des grands nombres) qui émerge quand on analyse des programmes réels écrits par des programmeurs réels, mais qui disparaît avec les programmes « jouets » de quelques dizaines d'instructions. Il y a là une parenté étonnante avec la microphysique des composants électroniques où les phénomènes d'émergence génèrent des propriétés inattendues comme l'effet transistor (cf. l'ouvrage du prix Nobel R.Laughlin, *A different universe, reinventing physics from the bottom down*, Basic Books, 2005 ; une réflexion profonde sur ces phénomènes déroutants). C'est à ce niveau que se justifie le véritable fondement des modèles d'estimation.

Du point de vue de l'ingénierie des programmes, au sens large, ces phénomènes sont intéressants, car une loi d'échelle – ou une règle « de pouce », comme on veut, qui traduit cependant un vrai savoir faire – connue depuis très longtemps de ceux qui manipulent des programmes « en grand », stipule que la productivité moyenne d'une population de programmeurs est fonction du nombre de symboles manipulés, et non pas de la nature de ces symboles (Cf. la réf. archi-classique du nombre magique 7±2 [22] qu'il faut connaître, y compris pour le contester ! voir également le chapitre 7.5, *Stratégie de communication dans les organisations de développement*, de réf. [5]).

En conséquence, on programme aussi vite en assembleur qu'en LHN, mais comme on utilise 2 à 3 fois moins de symboles en LHN qu'en assembleur, pour un même programme, la production est améliorée d'autant. Ce constat est au cœur d'un modèle d'estimation comme COCOMO. Si c'est faux, le modèle est bon pour la poubelle ! C'est également ce qui fonde le succès de la programmation en LHN, d'où la frénésie d'invention de nouveaux langages, souvent au delà du raisonnable, mais ceci est un autre débat. Ada, dans les années 80s a constitué une espèce de cas limite, du fait de sa complexité intrinsèque (un « beau » et riche langage), qui lui a été fatale, les programmeurs lui ayant préférés C (un langage « sale », comme on dit « un sale gamin ») et sa descendance C++, … . Au passage, cela montre que la syntaxe n'a quasiment aucun intérêt, c'est juste un mal nécessaire, car notre cerveau s'adapte très bien à des syntaxes jugées absconses : écrire `a=b+c×d` (notation infixe) ou `abcd×+=` (polonaise inverse) c'est du pareil au même, juste une question d'habitude.

Pour améliorer la production de programmes, il n'y a qu'une seule règle : monter en abstraction, faire simple, diminuer le nombre et la variété des symboles utilisés par les programmeurs. C'est ce qui a été vérifié une nouvelle fois avec les langages associés aux SGBD, et le triomphe du SQL relationnel sur le DML de CODASYL.

Vers une approche de la mesure de la complexité textuelle

Avec une règle de comptage des instructions standardisée (Cf. réf. [19]), on peut mesurer la longueur d'un programme (communément appelée TAILLE du programme, en anglais SIZE) en nombre d'instructions. Quel est l'intérêt d'une telle mesure ? A QUI, et à QUOI peut-elle servir ?

Pour le chef de projet, c'est une information essentielle, pour établir le plan et l'organisation du projet, et la répartition du travail entre les programmeurs. C'est le domaine des modèles d'estimation, comme COCOMO ou les Points de Fonctions, qui sont fondés sur le constat d'une corrélation entre la taille du programme et l'effort à fournir pour le réaliser, i.e. programmation + tests + documentation. L'étalon de comptage est le millier d'instructions écrites par le programmeur, qui est un « mix » d'instructions, i.e. une grandeur statistique.

Pour l'architecte, c'est la structure du texte, au delà du détail, qui est l'information essentielle. C'est ce qui est écrit en amont (hier des pseudo-codes et PDL de toute nature, aujourd'hui une relative standardisation avec UML), mais que l'on retrouve, sous une forme ou sous une autre dans le texte même du programme. De fait, dans tous les langages de programmation, depuis ALGOL et PL1, il existe des moyens d'exprimer cette structuration. Que les programmeurs s'en servent intelligemment, c'est une autre histoire ! Certains styles de programmation conservent cette structure (programmation par machines abstraites, par systèmes états-transitions et automates, etc.) ce qui permet de garder lisible et visible les différents aspects de la sémantique (pilotage et contrôle, adressage des entités, transformations, entrées-sorties), mais le style ne fait pas partie du langage. L'absence de guide de programmation (le méta-texte qui dit au programmeur COMMENT programmer) est généralement un signe d'une carence architecturale grave, car revues de code et inspections deviennent impossibles, chacun a son style et interprète ce qu'il y a à faire comme ça l'arrange, en bref, c'est le chaos programmé.

Le chef de projet et l'architecte vont se retrouver autour de la notion de module (au sens de D.Parnas, cf. réf. [24]) ou de building-block, que nous avons traduit par intégrat (terminologie introduite par B.Walliser dans *Systèmes et modèles*, au Seuil). L'intégrat est l'unité de production, i.e. la pièce, au sens mécanique, qui permet d'articuler le travail de programmation et le travail d'intégration qui est l'assemblage progressif et méthodique de l'ensemble du système à partir de ses pièces élémentaires (voir chapitres 4.5.3 et 4.5.4 de réf.[7] ; également réf.[37] est intéressante).

En termes de complexité, la mesure se dédouble car il faut considérer :

- Le volume de programmation, i.e. la taille des pièces, dont la somme définit la taille du système.
- Le nombre de pièces élémentaires, les intégrats de rang 0, et les relations de ces pièces les unes avec les autres, dont l'agrégation progressive détermine la structure du plan d'intégration. La description statique de cet ensemble fait partie de la gestion de configuration

Avec cette approche, la grandeur complexité devient vectorielle (Cf. réf.[3] qui donne quantité d'informations utiles), soit :

Complexité → {Taille du système, Nombre de pièces, Nombre de relations entre pièces}

Remarque : le nombre de pièces dépend de la taille moyenne des pièces, et de ce que sont capables de produire les programmeurs, soit individuellement, soit par paire programmeur/testeur comme recommandé dans les méthodes « agiles » ou l'eXtreme Programming. Soit la figure ci-dessous :

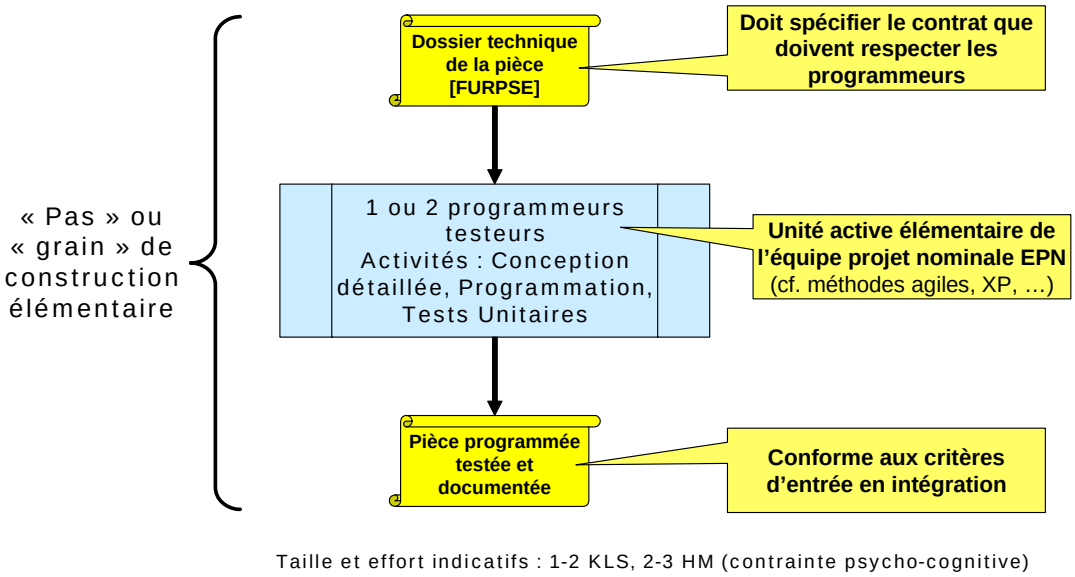


Figure 3 – Acte de programmation élémentaire

La capacité de compréhension moyenne des programmeurs détermine le nombre de pièces du puzzle (c'est un concept ergonomique).

Le « à quoi ça sert » est plus embarrassant, car un texte de programme réel embarque avec lui des éléments qui n'ont rien à voir avec la sémantique du problème et sa complexité. Ne pas avoir bien compris le problème, n'a jamais empêché un programmeur de programmer, il peut toujours faire du cas par cas et transcrire « bêtement » les cas d'emplois en programmes. B.Meyer a justement insisté sur le danger des « cas d'emploi » (cf. réf. [23], chapitres 5.2 et 22.4). Rappelons que le travail de conception-programmation consiste à passer d'une description du problème « en extension », via éventuellement des « cas d'emploi », à une description « en intention », qui est la solution algorithmique abstraite du problème. Dans une programmation au cas par cas, le texte garde un sens du point de vue projet, car il faut quand même le programmer, le tester et le documenter, mais il ne veut plus rien dire du point de vue de la complexité. Le texte a été parasité par l'incompétence du programmeur, et n'est que la « mesure » de cette incompétence dont le symptôme est un programme dont la taille est sans rapport avec la complexité du problème : des coefficients d'inflation de 10 à 15 ont été observés. Cette confusion est la source de beaucoup d'incompréhensions sur le bien fondé des modèles d'estimation. D'où l'importance du *benchmarking* qui seul permet de détecter ce type de pathologie.

Pour que le texte programmé présente une quelconque utilité quant à une « mesure » de la complexité, il faut qu'il soit exempt de ce type de défectuosité. Si le style de programmation est par machine abstraite (par exemple, sans exclusivité, cf. réf.[36]) ce sera le cas. Reste le problème du comportement du programme.

Dans un programme réel, le comportement du programme est tributaire des ressources nécessaires au programme, et de la façon dont ces ressources sont gérées, ainsi que des interactions avec l'environnement qui se manifestent par des survenues d'événements qu'il faut également gérer. Ce qui fait que le comportement attendu est incertain, au moment de la conception détaillée et de la programmation. Il n'est jamais évident de savoir a priori s'il faut ou non développer un cache pour économiser des entrées-sorties,

mais dans tous les cas cela pourra faire varier la taille du programme de façon importante. L'incertitude ne pourra être levée qu'en expérimentant le programme dans un contexte réel, ou relativement proche du réel (environnement simulé), soit en intégration, soit en exploitation.

En synthèse, on peut dire qu'un programme réel résultat d'une conception correcte amalgame, souvent de façon inextricable, trois types d'information (en mécanique quantique, on dirait intrication, ou *entanglement* en anglais ; on sait le rôle joué par l'observateur dans la notion de mesure, cf. réf.[25]):

- La complexité sémantique du problème, que l'on peut appeler **complexité intrinsèque**.
- La complexité **complication** engendrée par la notation utilisée et l'usage qui en est fait.
- La complexité **incertitude** engendrée par la nature et le volume des ressources disponibles, ainsi que les modalités de gestion résultant des choix d'architecture.

C'est ce que nous appellerons **complexité CCI** (complexité intrinsèque + complication + incertitudes), ou simplement complexité quand il n'y a pas ambiguïté. C'est une première approximation de la complexité intrinsèque, qui prend en compte la réalité concrète d'un projet particulier.

Le problème de l'étalon de mesure

On peut perfectionner la mesure en faisant des hypothèses simplificatrices, à la fois sur la machine qui sert d'étalon de mesure de la taille des programmes, et sur les caractéristiques psycho-cognitives des différents acteurs du projet.

En faisant l'hypothèse que les ressources sont infinies, la gestion des ressources disparaît, ainsi que le code qui va avec. C'est, toutes choses égales par ailleurs, ce qui s'est produit avec la généralisation de la mémoire virtuelle dans les années 70-80 par rapport à la génération d'avant où le programmeur devait gérer les *overlay* de la mémoire. Ce fut vécu comme une « libération » ! Mais cela a créé de très mauvaises habitudes.

Coté programmeur, on peut prendre comme étalon un programmeur « parfait », i.e. un robot programmeur, qui ne fait jamais d'erreur, qui a une connaissance parfaite du référentiel, lui-même parfait, et qui communique sans latence et sans conflit avec ses autres collègues également parfaits (Cf. notre concept PIP, **P**rogrammeur **I**solé **P**arfait, définit dans réf. [6], Chapitre 2.5). Cette hypothèse revient à fixer une vitesse limite de programmation (cinétique de la transformation) et permet d'éliminer le code des auto-tests, les assertions, la programmation défensive, etc. puisqu'il n'y a pas d'erreur.

Pour rester dans notre analogie thermodynamique, on définit ainsi une situation idéale qui correspondrait à la théorie des gaz parfaits, qui est un état limite qu'on ne peut pas dépasser (cf. le 2ème principe de la thermodynamique). Dans un moteur réel, les choses se passent différemment, mais la théorie fixe une limite indépassable au rendement du moteur. En chimie industrielle, la cinétique des réactions, la catalyse, la loi d'action de masse, … sont des aspects fondamentaux pour la construction effective des réacteurs chimiques qui déterminent la dynamique des processus (Cf. réf.[33]).

Sous réserve de ces hypothèses, la taille du programme idéal commence à tendre vers ce que l'on pourrait appeler mesure de la complexité intrinsèque.

Reste le problème de la machine. Quelles caractéristiques lui donner pour qu'elle garde une certaine ressemblance avec une machine réelle ?

La machine idéale la plus universelle est la machine de Turing. Son seul inconvénient est qu'elle est tellement éloignée des machines réelles qu'elle n'a aucun intérêt pour l'ingénieur, même s'il a une certaine largeur de vue, ce qui n'enlève rien à son intérêt théorique. La mesure de la complexité algorithmique de l'information (i.e. le plus court programme pour résoudre un problème), introduite par A.Kolmogorov et G.Chaitin utilise une machine de Turing comme étalon (Cf. réf.[26]).

Pour une machine abstraite plus réaliste, du point de vue de l'ingénieur informaticien, on peut envisager deux cas de figure :

1. Une machine MPI avec des performances infinies, i.e. infiniment rapide, sans contrainte d'aucune sorte, programmée par des PIP. C'est la machine d'un monde réel parfait.
2. Une machine MPF avec des performances finies connues, programmée par des programmeurs imparfaits qui fonctionnent avec un certain taux d'erreurs par acte de programmation. C'est la machine d'un monde réel contraint.

Les nuances ici introduites correspondent à des choix de modélisation effectués par les architectes à différents moments de la vie du projet, comme le montre la figure ci-dessous.

Dans le monde réel, il y a toujours des aléas, des contraintes, des incertitudes et des risques. Le problème pour l'architecte est de savoir comment les abstraire, donc de définir la machine de modélisation qui lui paraît le bon compromis entre les acteurs qui ont à connaître l'architecture. Dans la théorie de l'information, la notion d'erreur joue un rôle fondamental et le « bruit » est un élément de modélisation (Cf. réf.[9] et [17])

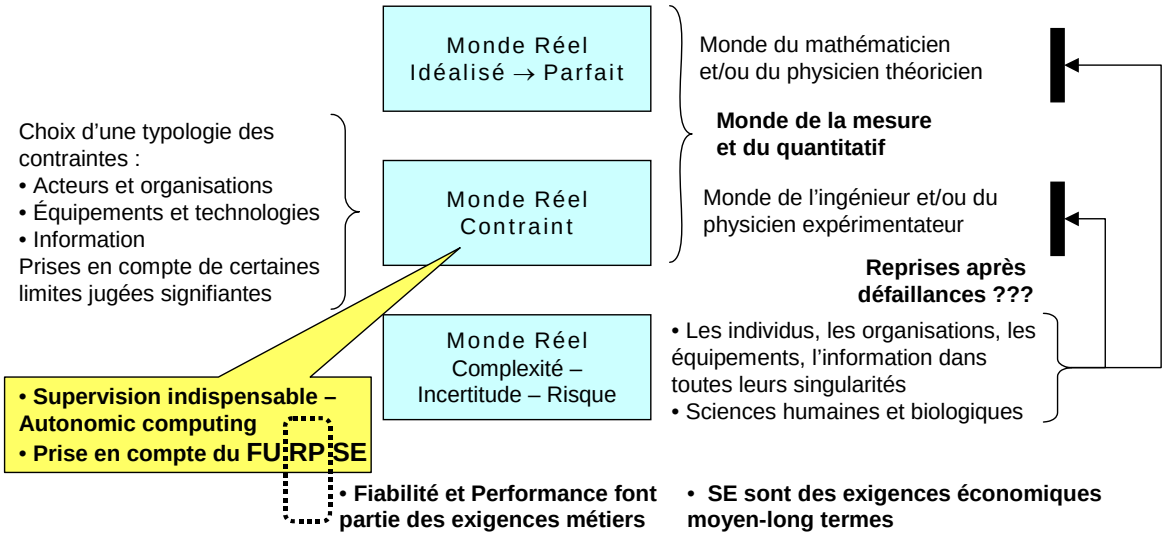


Figure 4 – Approche de modélisation par machine abstraite et simplification du monde réel

Dans une machine de Turing, la mémoire est un ruban infini indestructible. Dans un

ordinateur réel, c'est également un ruban dont la taille est fini, quoique grande, avec deux propriétés, ou contraintes, fondamentales : 1) la persistance (i.e. les disques, à accès lent), dont le contenu de la mémoire est conservé en cas de panne, et 2) la non-persistance, on dit aussi mémoire volatile (i.e. la mémoire centrale, plus petite, à accès rapide) dont le contenu est perdu en cas de panne.

La structure de la mémoire réelle a longtemps été purement séquentielle comme dans une machine de Turing (FORTRAN et COBOL, au début), avec un peu d'indexé (uniquement COBOL). Aujourd'hui on pourrait choisir une mémoire relationnelle (ce qui permet un adressage associatif), voire même une mémoire relationnelle XML_isée, avec des opérateurs CRUD ad hoc.

La machine dispose de deux catégories d'opérateurs : 1) les opérateurs de base, en nombre fini, qui font office d'instructions machine, et 2) les opérateurs dérivés (*built-in functions*), construits avec les précédents, qui définissent la machine étendue.

La machine dispose d'un mécanisme d'enchaînement des opérateurs, en fonction de l'état résultant de l'exécution d'une opération (i.e. un vecteur d'état, associé à chaque opérateur), et/ou d'événements résultant des interactions de la machine avec son environnement (i.e. via les ports de la machine qui détermine sa sphère de contrôle).

Dans un monde réel parfait, on peut faire l'hypothèse que la machine a des performances infinies, ce qui fait qu'il n'y a ni attente de fin d'exécution d'un opérateur, ni conflit de ressource, puisqu'une ressource acquise sera libérée instantanément. Un ensemble de machines coopérantes aura toutes les apparences d'une machine centralisée unique. L'organe de contrôle est réduit à sa plus simple expression. Les robots programmeurs, qui programment tous à vitesse constante, ne font jamais d'erreur, les contraintes (i.e. les contrats) sont toujours respectées et il n'est pas besoin de les vérifier. Il n'y a pas besoin d'assurance qualité.

Dans un monde réel contraint, la vitesse d'exécution des opérateurs est finie, on peut imaginer plusieurs quantum de coût, selon la nature de l'opérateur, mais elle n'est pas aléatoire comme dans les machines réelles. Ce qui fait qu'il y aura des attentes, donc des files d'attentes, et des priorités qu'il faudra gérer (arrêt des opérations non prioritaires, puis reprises). Il faut un vrai moniteur de contrôle, capable de gérer la distribution. Pour les programmeurs du monde réel contraint PMRC, on peut en imaginer plusieurs catégories selon le niveau d'expertise, ce qui se traduira par des vitesses de programmation et des taux d'erreurs différents selon les catégories. Le plan projet est quasi déterministe, mais une assurance qualité est indispensable.

NB : le fait que les temps d'exécution des opérateurs soient connus et finis permet, du moins en théorie, de calculer la durée de n'importe quel programme, et d'en déduire, toujours par calcul, l'ordonnancement optimum. Donc il n'y a pas d'incertitude de performance, et tout ce qui dépend des incertitudes disparaît ipso facto. Dans une machine réelle, il n'en est rien, même si la machine est surdimensionnée, ce qui reste un pis allé pour lever les incertitudes.

En synthèse, il est certain que le programme écrit pour la machine MPI sera plus court que celui écrit pour MPF, soit :

CCI (Programme P/PIP) sur machine MPI < CCI(Programme P/PMRC) sur machine MPF

On peut imaginer qu'en idéalisant suffisamment le monde réel, mais en lui gardant un minimum de propriétés des machines réelles (i.e. les ressources) et de caractéristiques

des programmeurs réels (i.e. cinétique, taux d'erreur), il existe une **limite** que l'ont pourrait appeler **Mesure CCI du programme**. Cette limite est très proche de la complexité intrinsèque, car de toute façon une notation est indispensable, et l'incertitude a été éliminée.

L'étalon de mesure de l'effort pourrait être la taille d'un programme ad hoc, jugé représentatif, programmé sur une machine abstraite permettant le mesurage (par exemple 20-25 pages de texte, soit 1-2 KLS, qu'un programmeur relativement expérimenté peut mémoriser). De tels étalons ont été définis pour mesurer les performances des systèmes réels (Cf. les normes SPEC, *Standard Performance Evaluation Corporation* [voir www.specbench.org] et TPC, *Transaction Processing Performance Council* [voir www.tpc.org]).

Considération sur les couplages et les graphes de couplages entre modules

On a vu ci-dessus l'importance du nombre de pièces, i.e. intégrats, et des relations entre les pièces.

Examinons, sans rentrer dans le détail, quelques situations intéressantes.

Enchaînement séquentiel d'intégrats

Le schéma de principe de ce type d'enchaînement est le suivant.

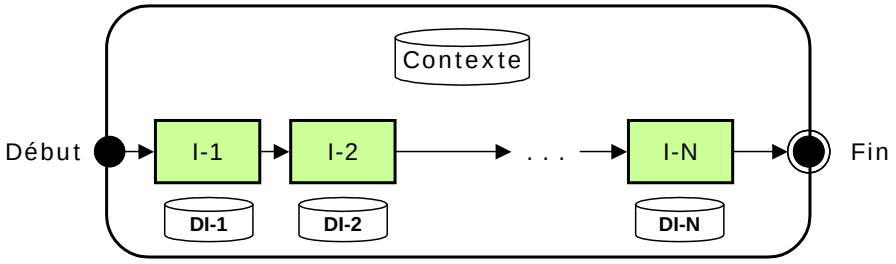


Figure 5 – Enchaînement séquentiel d'intégrats

C'est un enchaînement linéaire d'intégrats, le plus simple que l'on puisse imaginer. Chaque intégrat à une mémoire spécifique. De nombreux programmes suivent ce pattern. S'il n'y a pas de contexte entre les intégrats, et que la seule dépendance est l'enchaînement sur le suivant, le nombre de relations possibles est exactement $N-1$.

Dans ce cas, le travail d'intégration est proportionnel au nombre d'intégrats. C'est un coût forfaitaire par intégrat, selon la structure de l'interface. Il n'y a que deux programmeurs, ou deux équipes, impliqués dans le contrat d'interface.

Faisons maintenant l'hypothèse que tout en restant séquentiel, chaque intégrat hérite via la mémoire de contexte d'un état résultant de l'exécution des intégrats qui le précèdent. Ce qui veut dire que l'information entrante d'un intégrat provient de deux sources : l'interface et le contexte. Le nombre de relations possibles devient alors $N \times \frac{(N-1)}{2}$ car I-1 à $N-1$ successeurs, I-2 en à $N-2$, … jusqu'à I-N qui en a 0. L'effet du contexte sur le coût d'intégration est immédiat. La mémoire de contexte doit être négociée entre toutes les équipes et sa gestion doit être supervisée par l'architecte car c'est une source de complexité.

NB : dans les deux cas, le nombre cyclomatique est 1, ce qui montre que du point de vue de la complexité réelle CCI, ce nombre n'a pas de sens !

Prenons un exemple légèrement différent qui serait celui d'une exécution en mode transaction des intégrats, sans contexte. Le schéma est le suivant.

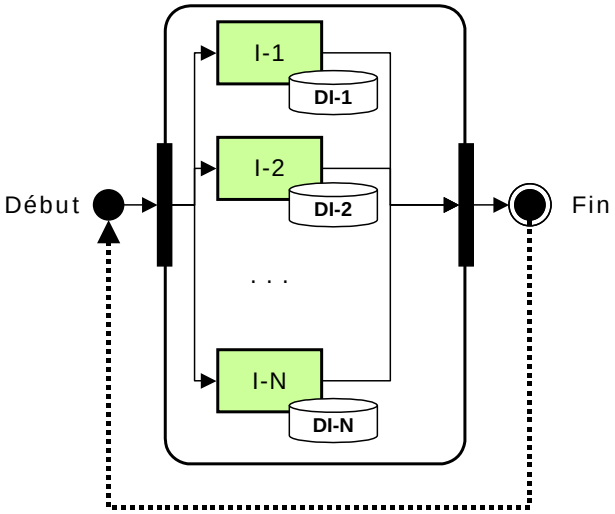


Figure 6 – Enchaînement sérialisé de transactions

Les données DI-1, DI-2, … , DI-N appartiennent à une base de données, mais du fait des propriétés d'atomicité et de sérialisation des transactions, le A et le I de ACID, tout se passe comme si ces données appartenaient en propre à la transaction pour la durée de la transaction. Comme par définition il n'y a pas de contexte, sauf erreur de programmation grossière de la part des programmeurs, on est dans le cas idéal d'une intégration à coût linéaire, ce qui justifie le modèle d'estimation des Points de Fonctions qui est un modèle linéaire.

NB : remarquons que dans ce cas, le nombre cyclomatique vaut N !

Influence du contexte mémoire sur la complexité – Architecture des données

L'existence de la mémoire contexte réalise un couplage des intégrats par les données partagées. Dans le cas de la figure 6, l'existence d'un contexte peut avoir un effet dévastateur sur la complexité.

La nature transactionnelle des traitements fait que l'ordre d'exécution des intégrats est quelconque. A un instant t, l'intégrat en cours d'exécution hérite d'un contexte qui résulte des exécutions précédentes, i.e. l'histoire du contexte. Si l'on considère les k dernières exécutions, on a un ordonnancement des histoires possibles OHP qui est un arrangement avec répétition de k intégrats parmi les N possibles, i.e. un ensemble :

{I-x1, I-x2, … , I-xk}, dont la cardinalité est : $OHP = k^N$.

Le nombre de relations de dépendances explose ! Si chacune des exécutions modifie quelques unes des entités et/ou attributs du contexte, on comprend tout de suite le danger (NB : c'est le problème des mémoires caches, qui photographient des états jugés répétitifs pour les réutiliser ultérieurement, d'où la difficulté des algorithmes de gestion des caches).

En cas d'anomalie du contexte, il sera très difficile, voire impossible, de reconstruire l'histoire. Pour apprécier correctement la complexité, i.e. le nombre de relations à gérer en intégration, le graphe de dépendance par rapport aux données est plus pertinent que le graphe des flux, d'où l'importance des dictionnaires de données et plus généralement de la gestion des configurations.

Moralité : moins il y a de contexte, mieux cela vaut, mais cela augmente le nombre et la taille des messages, d'où problème potentiel de performance – Si un contexte est prouvé nécessaire (décision impérative et explicite de l'architecte), il doit faire l'objet de règles d'emploi précises ainsi que d'une surveillance constante en assurance qualité.

Si les intégrats ne fonctionnent pas comme des transactions et sont dépendants des interruptions toujours possibles (i.e. la propriété d'atomicité n'est plus respectée), on ne peut pas considérer que les données D-I1, D-I2, … leur appartiennent en propre. Une interruption de I-x laisse les données D-Ix dans un état instable. Tout le travail de synchronisation effectué par le moniteur transactionnel est à la charge des programmeurs. Les négociations entre programmeurs pour gérer correctement le partage de l'information sont concevables au sein d'une équipe, mais elles deviennent difficiles si plusieurs équipes doivent se synchroniser, voir impossibles quand les équipes sont dans des organisations et des entreprises différentes (cas des SDS, systèmes de systèmes).

NB : le fait d'être interrompu, quelle qu'en soit la cause (événement prioritaire, time-out, erreur, etc.), revient à dérouter le flot du contrôle, ce qui fait que derrière le graphe de flux explicite résultant des instructions de contrôle du programme, se profile un autre graphe dont les arcs sont implicites car toutes les instructions deviennent des points de déroutement possibles. La perte de la propriété d'atomicité des intégrats a un effet catastrophique sur le graphe de contrôle (cf. réf.[8], chapitre 12).

En cas d'interruption, le nombre de nœuds du graphe de contrôle est le nombre des instructions, ou des séquences d'instructions, réellement indivisibles, soit potentiellement le nombre d'instructions machine, ou le nombre d'instructions impératives LHN, si celles-ci sont protégées par le *run-time* du langage (cf. les possibilités offertes par le langage Ada dans sa version 95).

Moralité : en termes de couplages, il faut non seulement s'occuper des dépendances fonctionnelles résultant des flux de contrôle (graphe des instructions impératives, graphe des appelants-appelés), mais également des couplages résultant des données partagées et des événements pouvant modifier le flux des traitements. Le coût de l'intégration évoluera en conséquence.

NB : c'est le cas des noyaux d'applications temps-réel, ou du noyau (superviseur) d'un système d'exploitation considérés comme des extremums de complexité. Dans de telles situations, en poussant à la limite, le nombre de pièces élémentaires à considérer du point de vue de l'intégration tend vers le nombre de séquences d'instructions indivisibles, soit potentiellement le nombre des instructions machine !

Le travail de l'architecte consiste à s'assurer que les couplages entre les intégrats ne fabriquent pas mécaniquement un nombre de relations qui rendrait le travail d'intégration incompatible avec les objectifs CQFD du projet.

Couplages hiérarchiques – Architectures en couches

Une bonne façon d'apprécier la complexité de l'intégration, et donc son coût, est de se poser la question, pour chaque intégrat, de son réseau de dépendances vis à vis des autres intégrats résultant des couplages entre intégrats.

On peut faire ainsi apparaître une distance de couplage. Les intégrats à une distance d=1 sont directement couplés à celui qui nous intéresse, ceux à une distance d=2 le sont indirectement, et ce ainsi de suite jusqu'à ce qu'il n'y ait plus de couplage. La complexité de l'intégration de cet intégrat particulier dépend du nombre d'intégrats qui sont effectivement couplés, soit directement, soit indirectement.

Les premiers architectes (Cf. réf. [1], [2] et [27]) ont compris que c'était là un enjeu crucial, et qu'il fallait particulièrement se méfier des données, et encore plus des pointeurs sur les données. D'où l'invention des architectures hiérarchiques en couches dont le plus bel exemple est celui des protocoles de communications.

Le schéma de principe d'une intégration construite selon cette méthode est le suivant :

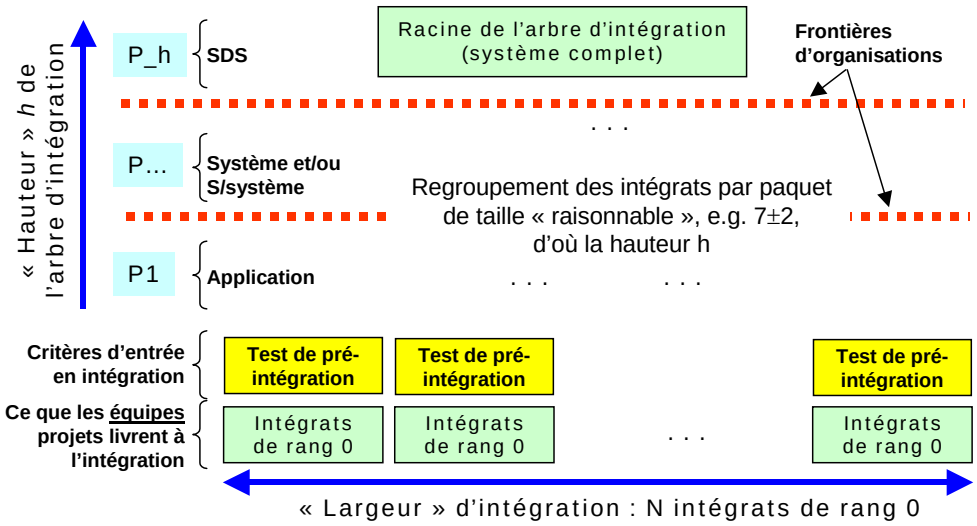


Figure 7 – Arbre d'intégration

Dans une architecture hiérarchique « VRAIE », i.e. dûment vérifiée et validée par l'architecte, l'intégrat de rang 0 hérite des propriétés qui sont celles des intégrats sur la ligne hiérarchique, et ce jusqu'à la racine de l'arborescence. La « longueur » de la ligne hiérarchique, i.e. la hauteur de l'arbre, est de l'ordre de $O(Log_{base}(N))$, ce qui détermine le nombre de couplage potentiel que nous avons appelé distance de couplage.

Les feuilles de l'arbre d'intégration sont les intégrats de rang 0 produits par les programmeurs (cf. figure 3) qui peuvent faire l'objet d'un assemblage par l'équipe projet (intégration équipe). Toute anomalie détectée dans la suite du processus nécessitera une intervention des programmeurs qui devront corriger le défaut découvert par les tests, et mettre à jour la configuration projet.

NB : le processus d'intégration est décrit en détail dans réf.[7], chapitre 4.5.

Le processus d'intégration proprement dit démarre au niveau application. Pour concrétiser, il suffit d'imaginer une application client-serveur avec 3 composants

applicatifs : le poste client avec les IHM, le serveur de transactions, et le serveur de données ; chacun d'eux étant livré par une équipe distincte.

Le critère d'entrée d'un intégrat dans le processus d'intégration est le passage sans anomalie des tests de pré-intégration.

La figure fait également apparaître des frontières d'organisations qui matérialisent des contrats d'interfaces passés entre les organisations, contrats devant être respectés impérativement pour éviter de déstabiliser l'ensemble de l'arbre, ou du sous-arbre, correspondant.

Au plus haut niveau (système de systèmes – SDS), on est au niveau du SI global de l'entreprise avec de l'interopérabilité entre les systèmes constitutifs (i.e. les différents métiers de l'entreprise) de ce SI global (cf. réf.[29]).

Toute la bonne marche du processus d'intégration est basée sur le fait que l'arbre d'intégration est effectivement un arbre. Pour que cette propriété soit vraie, il faut qu'il n'y ait aucun couplage latéral, à quelques niveaux que ce soit, qui shunte le principe hiérarchique. Toute information de couplage doit circuler explicitement sur les chemins déterminés par l'arborescence. D'où le schéma de principe :

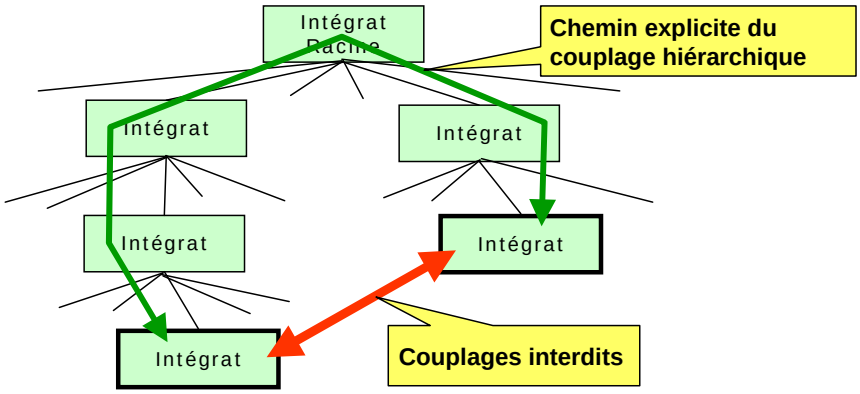


Figure 8 – Chemins de couplages hiérarchiques

Le schéma stipule que toute négociation entre responsables d'intégrats (i.e. les programmeurs) doit se faire par le canal hiérarchique déterminé par l'arbre d'intégration qui est un résultat direct du processus de conception qui crée effectivement l'architecture. Si cette règle est violée, il y aura des canaux cachés dans le système, et conséquemment des erreurs difficiles à diagnostiquer. La tentation de négociations bilatérales entre programmeurs est toujours forte, les motifs invoqués étant, au choix : l'optimisation des performances, le manque de temps, l'absence de documentation des interfaces, etc. ; c'est l'aspect « discipline » du métier de programmeur sur lequel de nombreux auteurs ont insisté, cf. réf. [30] et [34] parmi d'autres). Ce qui montre une fois de plus l'importance cruciale des interfaces, et des contrats associés. En particulier, les contrats d'interfaces aux frontières d'organisations doivent faire l'objet d'une spécification rigoureuse (enjeux économiques forts en termes CQFD et TCO).

Cela peut paraître « usine à gaz », mais c'est le prix à payer si l'on veut éviter la complexité générée par : « tout le monde négocie avec tout le monde » qui sera en $O(2^N)$, i.e. exponentielle, et condamnera le projet à l'échec.

Tests et complexité

Dans réf.[8], chapitres 12 et 13, j'ai défendu l'idée que le texte des tests a autant d'importance, sinon plus, que le texte du programme que ces tests sont sensés vérifier et valider. Avis également partagé par W.Humphrey, entre autres auteurs, dans réf.[30]. De fait, l'effort consacré aux tests dans les projets est généralement très supérieur à l'effort de programmation proprement dit. Ceci étant, les tests n'ont jamais fait l'objet de mesure textuelle, comme on a pu le faire avec les lignes de code.

De la formulation des exigences, il résulte deux types de texte : 1) le texte **PROGRAMME** pour l'expression fonctionnelle, et 2) le texte **TESTS** pour l'assurance qualité (chacun d'eux sont supposés correctement documentés).

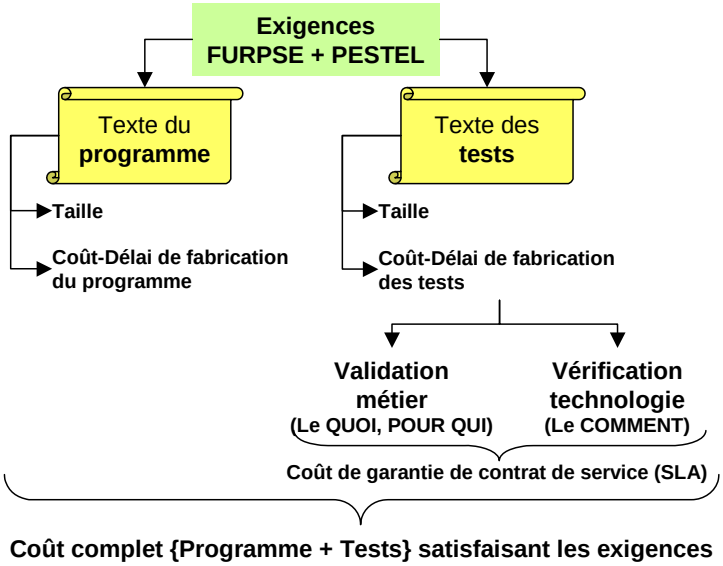


Figure 9 – Les deux textes fondamentaux : programme ET tests

Outre l'importance économique de l'effort de test dans les projet, les tests reflètent le caractère combinatoire et événementiel du comportement du programme. Ce qui fait qu'un programme perçu comme « petit » en taille, avec peu de données, mais à forte combinatoire (typiquement, un programme temps réel, comme un protocole) aura de « gros » tests, d'où l'effort pour les produire. Le test, dans ce cas, révèle mieux la complexité que le programme lui-même ! C'est une mesure « naturelle » du type de celle recherchée. Toutefois, additionner « bêtement » la taille du programme et la taille des tests n'a a priori aucun sens. Mais on peut additionner les coûts, comme le fait le modèle COCOMO.

Si l'on regarde ce qu'est l'activité de test, i.e. l'acte de test, du point de vue du testeur, il y a une forte parenté avec l'acte de programmation. Le bon testeur « joue » (au sens musical, pas au sens casino) de façon méthodique avec l'objet à tester (ce sont des expériences, en positif et en négatif) de façon à épuiser raisonnablement la combinatoire de l'intégrat, et ce jusqu'à obtenir une fiabilité compatible avec le contrat de service négocié avec les usagers du système.

Le résultat de ce « jeu » est un, ou plusieurs, scénarios de tests qui seront joués, et rejoués, tout au long du processus d'intégration et enrichis avec les défauts découverts en exploitation.

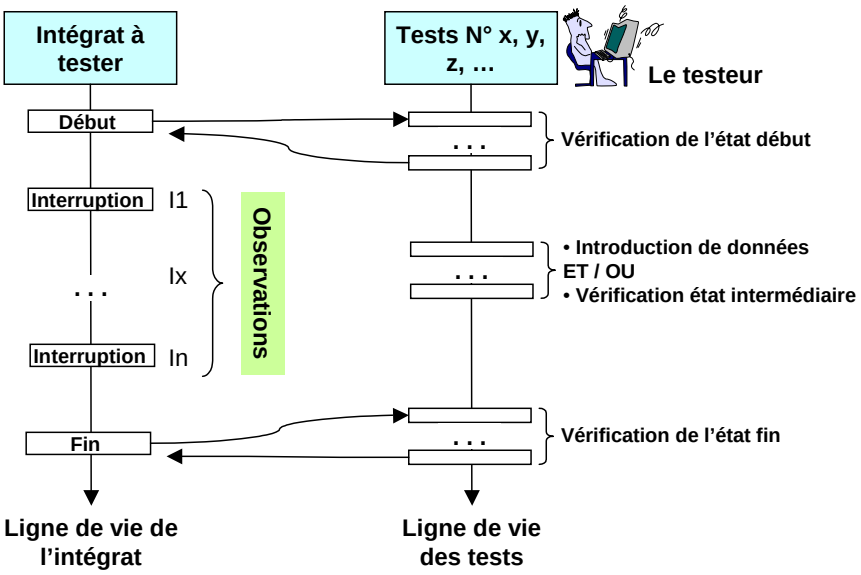


Figure 10 – Acte de test

Sur le schéma, le testeur fabriquera autant de tests différents que de lignes de vies jugées « intéressantes », i.e. des chemins, conformément aux exigences FURPSE de l'intégrat sous test. La fréquence d'emploi de tel ou tel chemin, ou fragment de chemin, est une information importante pour produire des tests pertinents. La productivité du testeur que l'on mesure en nombre de défauts découverts (tradition instaurée par la NASA, et les logiciels de la navette, cf. réf.[35]) dépendra évidemment de la qualité de la spécification de l'intégrat en termes FURPSE et du contexte que cet intégrat partage avec ceux auxquels il est couplé.

Je voudrai maintenant revenir sur l'argumentation, et les problèmes que soulève une mesure textuelle des tests.

Tout intégrat, tout programme, est une assertion considérée comme VRAIE d'une transformation T que l'intégrat effectue sur ses données d'entrée pour produire le résultat que constitue les données en sortie, à la fin de la transformation ; soit $D_E \xrightarrow{Transformation\ T} D_S$. Pour être effective, cette transformation va consommer des ressources qui doivent être disponibles pour que la transformation arrive à son terme normal (aspects cinétique et dynamique), mais de ce fait indisponibles pour d'autres programmes qui partagent ces ressources. Tous les praticiens de la programmation savent cela depuis toujours, et savent également que sur des programmes industriels réels, produits par des programmeurs réels, c'est loin d'être simple, et que cela constitue souvent, en programmation système et/ou interactive, la difficulté essentielle.

L'assertion a la forme logique d'un théorème, soit :

SI D_E valide (Contrat d'interface validé) **ET SI** Ressources **R** disponibles alors $D_E \xrightarrow{T} D_S$ **VRAIE**

L'explicitation des ressources (l'« énergie » informationnelle), notion centrale en programmation réelle, permet une analogie physico-chimique que J-Y.Girard a exploité, et formalisé, dans sa logique linéaire (cf. réf.[31]) que nous ne développerons pas ici.

Quant au test, qu'il soit expérimental (comme une expérience en physique) et/ou de l'ordre de la preuve (comme en mathématiques), c'est ce qui permet d'affirmer la vérité de la transformation et du résultat.

De même qu'un pont ne se réduit pas aux équations différentielles utilisées pour le modéliser, un programme réel ne se réduit pas aux abstractions utilisées pour le spécifier. En tout état de cause, les tests resteront nécessaires, comme dans tout travail d'ingénierie, pour valider le comportement du programme par rapport à la réalité.

Il y a donc une dualité évidente et profonde entre un programme et ses tests.

Ceci étant, la mesure textuelle des tests pose de vrais problèmes. Malgré les proclamations des évangélistes des méthodes « agiles » et de l'eXtreme Programming, avec le TDD (*Test Driven Development*), il n'est pas évident d'écrire les tests avant le programme, ni même d'entrelacer les deux activités, pour des raisons psycho-cognitives. Mais il faut quand même saluer cette proclamation iconoclaste en ingénierie logicielle, où l'on s'est « assis » sur les tests pendant des décennies ; tout le contraire de nos collègues en ingénierie hardware qui pratiquent le *Design To Test* depuis les origines.

Une deuxième difficulté vient de l'absence de langages spécialisés pour les tests, si l'on met à part les exceptions notables que sont des langages comme TTCN (langage de tests des télécommunications ; cf. réf.[32]) et ATLAS (Norme IEEE, pour le test des équipements électroniques). Dans l'histoire un peu ancienne du génie logiciel français, il y a eu une tentative intéressante avec le langage DEVISOR, d'inspiration Ada, développé par la société Dassault Electronique (aujourd'hui intégrée à THALES), mais tombé depuis en désuétude.

La bonne formule serait certainement d'associer à tout langage de programmation une extension avec des *built-in* fonctions orientées tests, et ce qu'il faut pour les exécuter dans le *Run-Time* du langage. Cela reviendrait à normaliser l'interface du *debugger* symbolique dans le langage de programmation hôte. Ce n'est malheureusement pas la tendance actuel des architectes de langage et des éditeurs qui préfèrent empiler les traits de langages les uns sur les autres, sans finalité industrielle claire, créant ainsi une complication inutile (une exception notable est le langage EIFFEL, de B.Meyer, qui malheureusement n'a eu aucun impact industriel). Dans cette optique, les extensions d'UML, dont le manuel de la version 2.0 fait déjà 700 pages, sont tout à fait inquiétantes. Les bons programmeurs, quant à eux, ont découvert depuis longtemps les vertus des auto-tests (là encore, c'est une notion classique en hardware) qui sont des tests intégrés aux programmes, écrits dans le langage des programmes, que l'on peut gérer sans difficulté en gestion de configuration, et que l'on peut compter avec les même règles que le texte programme (cf. réf.[19]). Les deux textes peuvent être additionnés. Pour les systèmes de grande taille, c'est évidemment la bonne formule, car c'est la garantie que l'instrumentation du système aura la même durée de vie que le système (là encore, c'est un classique de l'ingénierie !).

En désespoir de cause, l'effort de test est une mesure indirecte de la quantité de tests à fournir (le modèle d'estimation COCOMO est particulièrement explicite sur le sujet), mais, comme en physique quantique, tout est dans l'interprétation de la mesure et dans le bruit de fond introduit par l'observateur qui peut complètement fausser la mesure. Seul l'architecte est à même d'extraire quelque chose d'intelligible des données du projet.

Synthèse : la complexité dans les projets

En final, ce que le chef de projet, et son management, verra de la complexité se résume aux grandeurs projet CQFD, dont la partie QF est décomposable en exigences FURPSE qui engendrent la complexité CCI.

Le schéma ci-dessous montre les principales dépendances entre les notions qui viennent d'être discutées brièvement.

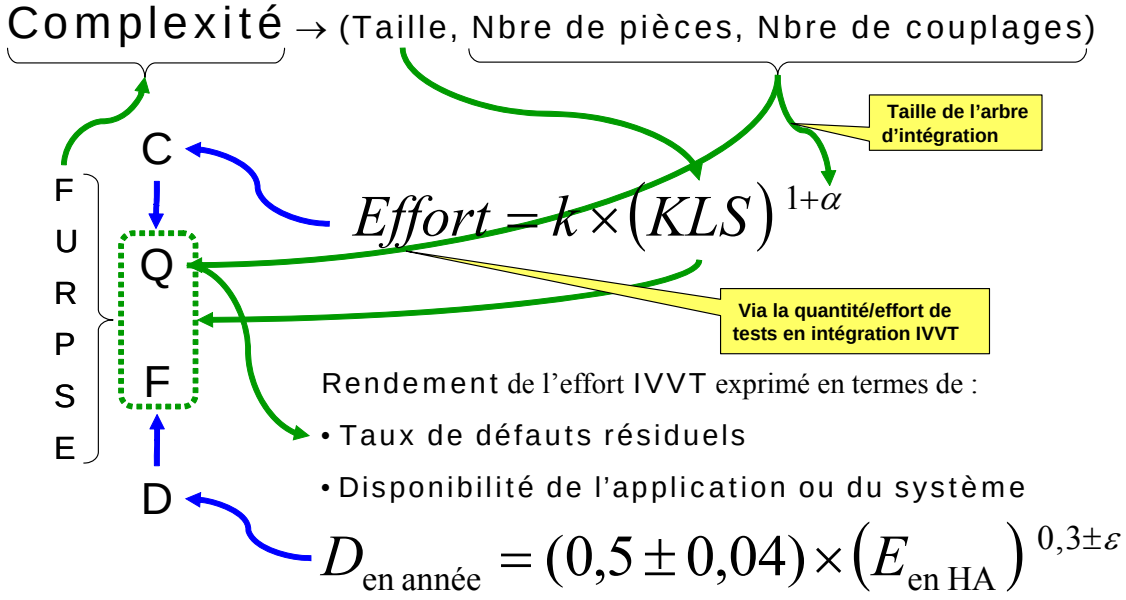


Figure 11 – Relations entre la complexité et les grandeurs projet

On remarquera, en particulier, la boucle **QF-FURPSE** → Complexité → {KLS, α} → **QF** qui détermine les grandeurs économiques projet **CD** qui agissent comme des contraintes. Si **CD** est incompatible avec le budget alloué il faudra revoir soit le budget, soit **QF**, soit les deux, et rééquilibrer **QF-FURPSE** jusqu'à obtention d'un équilibre acceptable par tous les acteurs de la décision. Sans modèle et sans mesure, on en est réduit au bricolage.

La notion la plus importante pour apprécier la complexité d'un système est matérialisée par l'arbre d'intégration, car c'est l'abstraction qui explique comment l'on passe de façon méthodique de l'ensemble des pièces élémentaires, i.e. les intégrats de rang 0, au système final, intégré dans le SI global de l'entreprise.

Pour rester dans le domaine ludique, la construction du puzzle d'intégration dépend de la découpe des pièces, i.e. du contexte et des couplages des intégrats (cf. réf.[37]). Plus cette découpe est arbitraire et sans règles explicites, au hasard des cas d'emploi et des négociations bilatérales entre les programmeurs, plus coûteuse et risquée sera l'intégration.

Spécifier à l'aide de machines abstraites (Cf. réf.[36], la 2ème partie : *Abstract machines*) aux propriétés rigoureusement vérifiées est le meilleur moyen de créer de l'ordre, de faire baisser la quantité d'information, et de simplifier le travail d'intégration.

La machine abstraite permet une séparation claire entre :

- La complexité d'usage (interface de programmation) qui in fine déterminera la taille du programme écrit sur cette interface. Cette taille est une mesure de la complexité CCI modulo la « puissance » expressive (niveau de langage) de cette machine abstraite.
- La complexité de la machine elle-même qui se matérialisera par la taille du traducteur ou de l'interpréteur correspondant, sur les niveaux d'abstraction inférieurs.

Le nombre α, comme nous l'avons établi par ailleurs (cf. Extrait N°1, ci-dessous) est un nombre pur qui ne dépend que de l'arbre. Il matérialise l'invariance d'échelle de l'imbrication des machines abstraites les unes dans les autres.

BIBLIOGRAPHIE :

[1]	NATO Science committee report, *Software engineering*,1968 et 1969
[2]	Brooks,F *The mythical man-month*, Addison Wesley, 1975
[3]	Lehmann,M Belady,L *Program evolution, Processes of software change*, Academic Press, 1985
[4]	Boehm,B *Software engineering economics*, PH, 1981 + COCOMO II, 2000
[5]	Printz,J *Puissance et limites des systèmes informatisés*, Hermès, 1997
[6]	Printz,J *Productivité des programmeurs*, Hermès, 2001
[7]	Printz,J *Ecosystème des projets informatiques – Agilité et discipline*, Hermès, 2006
[8]	Printz,J *Architecture logicielle – concevoir des applications simples, sûres et adaptables*, Dunod, 2007
[9]	Shannon, C Weaver,W *The mathematical theory of communication*, 1959
[10]	Brillouin,L *La science et la théorie de l'information*, 1959 ; réimpression J.Gabay
[11]	Mandelbrot,B *Les objets fractals*, Flammarion, 1975
[12]	Shooman,M *Software engineering – Design, reliability and management*, Mac Graw Hill, 1983
[13]	Pressman,R *Software engineering, a practitioner's approach*, Mac Graw Hill, 1987
[14]	Sommerville,I *Software engineering*, Addison Wesley, 2001 (6th edition)
[15]	Hennessy,J Patterson,D *Computer architecture – A quantitative approach*, MK, 1990
[16]	Hennessy,J Patterson,D *Computer organization and design – The hardware software interface*, MK, 1998
[17]	IBM J. Res. Develop., Vol.28, N°2, March 1984, Chen,C Hsiao,M *Error-correcting codes for semiconductor memory applications : a state-of-the-art review*
[18]	Capers Jones,T *Estimating software costs*, Mac Graw Hill, 1998; voir le site de sa société SPR www.spr.com pour obtenir la Programming Language Table
[19]	CMU/SEI-92-TR-20, Sept. 92, Park,R *Software size measurement : a framework for counting source statements* – disponible sur le site du SEI
[20]	Zipf,G *The principles of least effort – An introduction to human ecology*, Addison Wesley, 1949
[21]	Humphrey,W *Managing the software process*, Addison Wesley, 1989
[22]	Miller,R *The magical number seven plus or minus two*, Psychological review, Vol. 63, 1956
[23]	Meyer,B *Conception et programmation orientées objet*, Eyrolles, 1997

[24]	Parnas,D *On the criteria to be used in decomposing systems into modules*, Communications of the ACM, Vol.15, N°12, Dec.72
[25]	Omnès,R *Comprendre la mécanique quantique*, EDP Sciences, 2000
[26]	Chaitin,G *Information randomness & incompleteness*, World scientific, 1987 ; bon résumé dans J-P.Delahaye, *Information, complexité et hasard*, Hermès, 1994
[27]	Simon,H *The sciences of the artificial*, MIT Press, 1996 (3rd edition)
[28]	Perdijon,J *La mesure – Histoire, science et philosophie*, Dunod, 2004
[29]	Caseau,Y *Urbanisation et BPM*, Dunod 2005
[30]	Humphrey,W *A discipline for software engineering*, Addison Wesley, 1995
[31]	Girard, J-Y *Le point aveugle – Cours de logique*, 2 Vol., Hermann, 2007
[32]	Wilcock,C & al. *An introduction to TTCN-3*, Wiley, 2005
[33]	Ogunnaike,B Ray,H *Process dynamics, modeling and control*, Oxford UP, 1994
[34]	Dijkstra,E *A discipline of programming*, Prentice Hall, 1976
[35]	NASA *Software engineering laboratory* – Nombreuses publications sur les métriques
[36]	Abrial,J-R *The B-book*, Cambridge UP
[37]	Holland,J *Emergence, from chaos to order*, Basic Books, 1998

EXTRAIT N° 1, DE : *PRODUCTIVITE DES PROGRAMMEURS*, CHEZ HERMES.

Aspects de la complexité

Le défi de la complexité

Le plus grand défi auquel les organisations de développement, comme d'ailleurs n'importe quelle organisation, sont confrontées est celui de la complexité des systèmes que ces organisations sont sensés faire fonctionner. La complexité est l'un des principaux facteurs d'échecs de nombreux projets et la cause principale du vieillissement accéléré des applications dont le coût de maintenance devient prohibitif. Rappelons sans entrer dans les détails quelques uns des facteurs de complexité que l'on trouvera dans tout système informatisé :

Le facteur taille

Plus le nombre d'éléments d'un système est élevé, plus le système est complexe. Cette complexité dépend également de la variété des éléments, ce qu'en informatique on appelle le type de l'élément. Le facteur taille est donc double : nombre de types d'éléments et nombre d'occurrences des éléments. Une base de données qui ne contient que quelques types de table mais des milliards d'occurrences est tout de même très complexe.

Les éléments peuvent être regroupés pour former des éléments plus gros, selon les règles habituelles des nomenclatures. L'outil de base du facteur taille est la gestion de configuration du système.

Le facteur interactions

Plus le maillage entre les éléments est important, plus le nombre d'interactions possibles est élevé. Un graphe avec cycles et composantes connexes est plus complexe qu'un graphe connexe sans cycle, qui lui même est plus simple qu'une arborescence, qui elle même est plus simple qu'une chaîne séquentielle d'éléments.

Le besoin d'interactivité se traduit immédiatement par un maillage plus dense des éléments constitutifs du système. Ce maillage définit une structure d'ordre comme celle que l'on manipule en algèbre : ordre total pour la simple séquence, ordre partiel pour une arborescence, treillis, groupement avec ou sans pré-ordre où chacun est libre d'interagir avec tout le monde.

L'outil de base de l'interaction est constitué des différentes références croisées qui existent dans la description du système. Toutes sont en fait des méthodes de représentation des graphes correspondants aux interactions souhaitées. La difficulté avec l'informatique est qu'il faut faire des références croisées simultanément sur les données, les instructions et/ou les fonctions, sur les différents moniteurs de contrôles qui seront utilisés pour gérer les différents enchaînement. Ceci explique l'existence de nombreux canaux cachés qui sont autant de points faibles potentiels, à la disposition des réalisateurs de virus ou des intrus, évidemment préjudiciable à l'intégrité du système.

Le facteur couplage

Le couplage introduit la dimension du temps dans le système. Plus le degré de couplage est élevé, plus les informations se propagent rapidement dans le système ; meilleur est le temps de réponse. Dans un monde parfait, c'est la situation idéale. Dans un monde où il y a des erreurs, un couplage fort va accélérer la propagation des états incohérents jusqu'à la défaillance et/ou la panne avec arrêt du système. Plus le débit des canaux de communications est élevé, plus fort pourra être le couplage. Le besoin naturel d'interactivité, « tout, tout de suite, quand on veut », va se traduire par une forte augmentation du facteur de couplage.

Il est remarquable que les performances extraordinaires des différents équipements constituant les plates-formes informatiques (ordinateurs beaucoup plus puissants en nombre quasi illimité, réseaux haut débit, points d'entrées sorties en grand nombre, etc.) poussent toutes dans le sens d'un formidable accroissement de la complexité sous-jacente. L'usager des systèmes n'en a d'ailleurs pas conscience, mais il en constate facilement les effets néfastes : documentations incohérentes, augmentation du taux de pannes, pannes incompréhensibles, déni de service, virus et intrusion, etc.

Le facteur taille de l'équipe

Un projet complexe implique généralement une équipe nombreuse car il y a beaucoup à faire, mais l'inverse n'est pas forcément vraie car beaucoup de projets sont rendus artificiellement compliqués du fait de mauvais choix architecturaux et/ou méthodologiques. Le management des très grands projets reste une gageure et les risques sont considérables. La sociologie des grandes équipes[1] fait que sur une population nombreuse, il y a toujours un pourcentage non négligeable de mauvais coucheurs, de caractériels, d'incompétents par rapport aux tâches à effectuer, de paresseux … mais bien sûr on ne les connaît que lorsque les dégâts sont faits.

Le statut de l'erreur dans un système complexe

Dans les années 60s-70s, les constructeurs d'ordinateurs, avec IBM à leur tête, ont accrédité l'idée que les utilisateurs de ces systèmes étaient partie prenante dans le processus de vérification, validation et test. Peut-être n'y avait-il pas d'alternative, et de toute façon les utilisateurs ont accepté le risque. Pour pouvoir avoir accès à ces merveilles technologiques, il fallait aider à leur mise au point. Avec les nouveaux entrants dans le marché, essentiellement les fabricants de progiciels, Microsoft en tête, les choses ne se sont pas vraiment arrangées et tout a continué comme avant. En dépit d'énormes profits, et pas seulement Microsoft, la qualité telle que perçue par l'usager ne s'est pas vraiment améliorée, sans d'ailleurs que cela ne déclenche de révolte, car individuellement on peut faire beaucoup plus de choses qu'on en faisant auparavant. En fait, de façon subreptice, nous nous sommes habitués à l'inacceptable.

Dans un système complexe, ce type de comportement va devenir complètement inacceptable. Le propre d'un système complexe est de coupler un maximum d'éléments pour offrir le maximum de services dans un temps minimum. Ce qui fait que l'interaction d'un usager avec l'un des points d'entrée du système va créer dynamiquement une chaîne fonctionnelle, ou chaîne de liaison, de très grande taille entre le stimulus initial et la réponse finale du système jugée satisfaisante par l'usager. Le schéma de cette interaction est le suivant :

[1] Cf. W.Humphrey, *Managing technical people*, Addison Wesley, 1997.

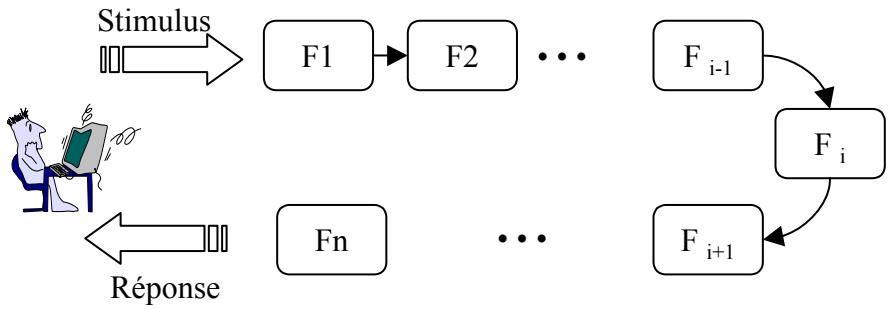


Figure 21 – Chaîne de liaisons dans un système complexe

Si chacune des fonctions F_i a une fiabilité f_i, la fiabilité de la chaîne sera :

$$f = f_1 \times f_2 \times f_3 \times \ldots \times f_i \times \ldots f_n$$

Si l'on couple étroitement des systèmes qui individuellement auraient une fiabilité acceptable, il est néanmoins certain que la fiabilité de la chaîne pourra être, quant à elle, parfaitement inacceptable, sans que l'on puisse d'ailleurs blâmer tel ou tel système particulier de la chaîne.

NB : il est bon de garder en mémoire quelques chiffres comme :

$0{,}9^{10} = 0.35$, $0{,}99^{100} = 0.37$, $0{,}999999^{10^6} = 0.37$ (cela tend vers $\frac{1}{e} = 0.368\ldots$)

mais également que $0{,}99^{200} = 0.13$, $0{,}99^{300} = 0.05$, $0{,}99^{500} = 0.00\ldots$.

Autrement dit, il suffit d'être nombreux sur le système, ou que les chaînes soient longues pour qu'une défaillance se produise inéluctablement. La même remarque vaut d'ailleurs pour les acteurs d'une organisation de développement.

Dans une organisation de développement, chacun s'efforce de travailler le mieux qu'il peut, compte tenu des règles communément admises ; personne n'a le désir de nuire. Il faut une intelligence particulièrement perverse pour imaginer que Microsoft s'est donnée comme objectif d'inonder la planète de logiciels réputés peu fiables[2]. Les ingénieurs de Microsoft ne sont probablement pas plus mauvais que les autres, l'inverse est même probable, mais ils ont simplement le plus grand nombre de clients, ce qu'il est difficile de leur reprocher.

Dans une organisation de développement où chacun travaille au voisinage de ses limites personnelles, voir au delà quand la pression des délais à respecter monte, et que la nature du système à développer nécessite un couplage très étroit entre les acteurs alors l'organisation risque de n'être plus fiable. Remarquons qu'un excès de couplage peut avoir diverses causes, soit une mauvaise architecture du produit (pas de couches indépendantes, pas d'interfaces, beaucoup de canaux cachés et de dépendances non documentées, etc.), soit un mauvais processus de développement du type niveau 1 du CMM. C'est faire preuve d'une grande naïveté que de blâmer les acteurs individuels qui n'y sont rigoureusement pour rien.

[2] Cf. les pamphlets comme *Le hold-up planétaire* ou *Comment Microsoft a fait main basse sur votre ordinateur*.

Dans un schéma de développement nécessitant la mise en œuvre de plusieurs projets en parallèle, comme celui de la figure xxx, *Manager le multiprojet* ci-dessous, le rôle de la mise en cohérence est précisément de fixer les règles qui vont permettre aux acteurs des système S1, S2, … de travailler indépendamment les uns des autres. La mise en œuvre de la méthode est donc un facteur de *dé*-couplage et de réduction de la complexité. Si cette mise en cohérence est insuffisante compte tenu du besoin d'interopérabilité exigé, par exemple les interfaces et les modèles ne sont pas suffisamment formalisés, alors le risque que les acteurs des systèmes S1, S2, … découvrent de graves problèmes en intégration est élevé.

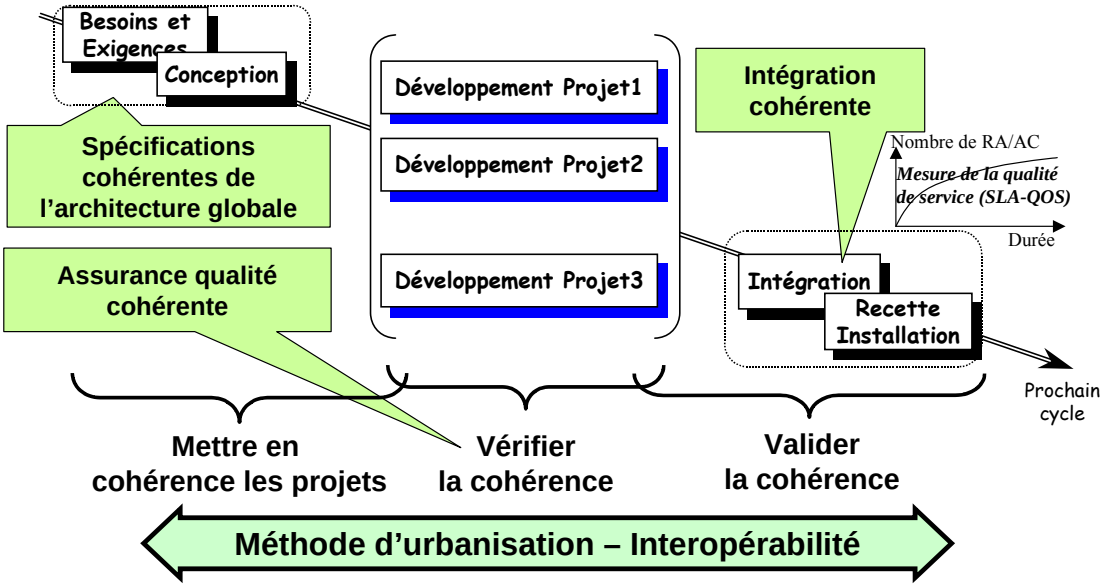


Figure 22 – Mise en cohérence des projets, le bon usage des référentiels

Il y a d'ailleurs un parallélisme absolu entre la façon de travailler en milieu complexe et les mécanismes architecturaux qui vont permettre l'interopérabilité des différents systèmes en interaction. Pour qu'un ensemble de systèmes S1, S2, … Sn puissent continuer à croître et à se développer à leur rythme propre, tout en se fédérant, il faut deux conditions :

a) D'une part qu'ils aient leurs règles propres, ce qui évitent les couplages implicites et les canaux cachés au sein même des systèmes,

b) D'autre part qu'ils aient des règles communes sur lesquelles seront fondées les échanges inter-systèmes

Comme pour les mécanismes du système qualité, il faut que tout ce qui entre et sort, fasse l'objet d'une vérification explicite par rapport à ces règles. Il faut de plus que les règles communes soient compatibles avec les règles spécifiques à chacun des systèmes, ce qui veut dire que l'on sait les traduire les unes dans les autres. Il est clair que si l'on a fait remonter trop de règles au niveau commun, ou si ces règles privilégient l'un des systèmes, on peut arriver à des situations de blocages. Il faut donc appliquer à la lettre le principe de subsidiarité.

En milieu complexe, il est impératif de valider cette compatibilité au niveau des acteurs, ce qui peut se traduire par un suivi particulier de tout ce qui touchent à la cohérence d'ensemble (y compris les acteurs eux même). Le suivi de la cohérence tout

au long du cycle de développement est donc une nécessité, avec les techniques ad hoc de l'assurance qualité. Si ce n'est pas le cas, chacun des systèmes se transforme en « trou noir » d'où plus rien ne sort ; l'incertitude est alors à son maximum.

En reprenant la notion de sphère de contrôle décrite dans l'avant propos du chapitre, et en l'appliquant à la partie du système proprement informatique, on est conduit à architecturer le système lui-même en sphères de contrôle de taille raisonnable, tant en ce qui concerne le produit à réaliser que l'équipe. Du point de vue de l'ingénierie, un projet informatique qui nécessite un pic d'effectif à 100-150 personnes est probablement une limite à ne pas dépasser. A partir de 4 à 500 personnes, les chefs de projet que l'industrie informatique française est capable de former se comptent sur les doigts de la main. Mieux vaut ne pas s'y risquer !

En disant « architecturer le système », il ne faut pas, comme les médecins de Molière qui expliquaient l'effet de l'opium par une mystérieuse vertu dormitive, croire qu'en ayant nommer le problème on l'a de ce fait résolu ! La question de fond à laquelle tous les MOA et les MOE sont confrontés est : comment s'y prendre ? que faut-il faire ? Pour beaucoup, architecturer veut dire faire des paquets que l'on baptisera objets si l'on veut faire moderne. C'est une vue relativement statique de l'architecture qui privilégie l'aspect nomenclature. Mais ce qui crée vraiment la complexité, comme on vient de le voir, ce sont les interactions et les couplages qui sont des phénomènes dynamiques. Si les regroupement effectués ne se traduisent pas par une baisse importante du niveau des interactions, on n'a en fait rien architecturé du tout !

Pour bien comprendre la nuance, prenons le cas d'une architecture client/serveur 3-tiers classique, comme ci-dessous :

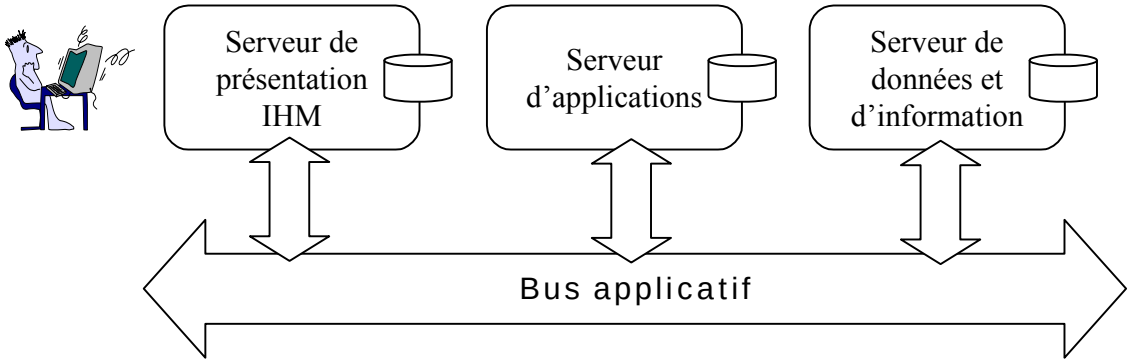


Figure 23 – Canaux et dépendances cachés dans une architecture client/serveur

Si l'on n'est pas extrêmement vigilant, et malgré le fait que le projet ait été organisé en trois sous-projets avec trois équipes distinctes, les trois serveurs peuvent devenir totalement dépendants les uns des autres si les données qu'ils échangent ne sont pas organisées en conséquence. La première décision architecturale à prendre est de découpler les modèles de données de chacun des serveurs, tant au niveau logique qu'au niveau physique, de façon à assurer leur autonomie. La seconde décision est de mettre en place une structure d'échanges autonome, avec son propre modèle de données qui donne la syntaxe des messages échangés. La troisième décision est de faire une analyse sémantique des informations du système susceptibles d'être échangées entre les différents serveurs, de façon à s'assurer que chacun des modèles à les capacités de

codage nécessaires et suffisantes à l'échange. Cela signifie que chacun des serveurs est libre de représenter l'information à sa convenance, pour autant que l'on sache traduire les différentes syntaxes les unes dans les autres. L'analyse sémantique garantie que ces traductions conservent le sens de l'information.

En l'absence d'une telle analyse, il y aura automatiquement des conventions implicites qui auront été faites entre les différentes équipes sans que cela soit documenté ailleurs que dans la programmation des différents serveurs, qui peuvent être dans trois langages différents : JAVA ou Smalltalk pour la présentation, C++ et une base de données objet pour le serveur d'applications, C et SQL pour le serveur de données. Les dépendances fonctionnelles et les canaux cachés peuvent faire, par exemple, que toute la structure de données est dépendante de la représentation adoptée dans les tables SQL du serveur de données, à moins que ce ne soit celle du bus d'échange fut-ce à travers une représentation des interfaces en IDL[3] ! Les difficultés apparaissent lors des évolutions des différents serveurs et du renouvellement naturel des équipes, lorsque les nouveaux chefs de projet se retrouvent avec du personnel qui n'a pas la connaissance des conventions implicites initiales. Il est alors trop tard pour restructurer le système dans un sens qui lui aurait permis d'évoluer. Des erreurs de cette nature ne se révèlent qu'à l'usage, lorsqu'on veut faire évoluer les serveurs indépendamment les uns des autres. Pour ne pas tout refaire, les responsables des différents serveurs traitent alors leurs problèmes au cas par cas, ce qui a pour effet de compliquer artificiellement la programmation pour pallier un manque de généralité de l'architecture ; les tests sont plus difficiles à développer, la fiabilité diminue, le coût de réparation des défauts augmentent, etc. Ceci montre que l'on ne peut juger la solidité d'une architecture que sur le long terme, et que de plus cela n'a rien a voir avec les technologies utilisées.

La philosophie sous-jacente à cette approche est celle de l'interopérabilité des différents serveurs. Architecturer et urbaniser un système, c'est rendre les éléments de ce système interopérables, c'est à dire les rendre dépendants d'une structure d'échanges et rien que de cette structure, par une construction explicite, laquelle ne doit donc véhiculer aucune dépendance à caractère sémantique qui viendrait de l'un quelconque des serveurs. La structure d'échange est une pure syntaxe qui doit être parfaitement neutre vis à vis des systèmes dont elle va assurer le couplage. De telle syntaxe existe depuis longtemps dans le monde des protocoles de communication (syntaxe ASN.1 du CCITT/UIT pour les messageries X200), et plus récemment dans celui des échanges de documents formatés avec EDIFACT, et HTML/XML pour les documents Internet. Ce qui dicte le choix de telle ou telle syntaxe est une pure commodité, mais la règle absolue est qu'il faut que cette syntaxe soit parfaitement explicite, et non pas le résultat d'un codage conjoncturel qui privilégiera tel ou tel serveur. La cohérence sémantique vient du fait que ces différentes syntaxes sont traduisibles les unes dans les autres, ce qu'il est très facile de valider et de vérifier, pour autant qu'elles soient explicites.

Cet ensemble de règles va permettre à chacun des éléments d'évoluer à son rythme, voir d'anticiper les évolutions d'autres serveurs, sans mettre pour autant son intégrité en péril.

En conséquence, nous pouvons dire que maîtriser la complexité, c'est :

- Se donner les moyens architecturaux et méthodologiques d'assumer ce qu'on a décidé de réaliser. Architecture et méthodologie sont deux faces d'un même

[3] C'est le langage de définition des interfaces de la norme CORBA proposé par l'OMG.

problème, mais la méthodologie doit être la servante de l'architecture, et jamais l'inverse. Il ne faut pas hésiter à réduire les fonctionnalités avant qu'il ne soit trop tard et trop coûteux pour procéder à des retraits qui seuls garantiraient la tenue des délais et de la qualité. L'architecture est le moyen de garantir la qualité.

- Se méfier tout particulièrement des caractéristiques non fonctionnelles, surtout lorsqu'elles n'ont pas été analysées en détail, tant au niveau du besoin de l'organisation cible, que de leur expression en langage informatique. Quoiqu'il se passe, les programmeurs feront toujours des hypothèses qui donneront au système des propriétés implicites qui ne seront probablement pas celles que souhaitent les usagers.
- Se méfier des modes, et ne jamais céder à la tentation de faire comme tout le monde, sans savoir exactement pourquoi on le fait. L'imitation n'est jamais un substitut à l'intelligence. Chaque situation est spécifique. Le cas des architectures distribuées est exemplaire : bien utilisées elles peuvent donner une grande souplesse d'évolution et d'adaptation ; utilisées de façon naïves, elles peuvent rapidement tourner au cauchemar (administration, fiabilité, maintenabilité, etc.).

Tout ceci est du strict ressort du management qui doit faire preuve de professionnalisme et de lucidité. La facilité n'existe pas et il n'y a jamais de solution miracle, ce qui, dans le langage de P.Strassmann se dit : « *There is no zero cost for something of value* ». Un bon manager n'embarquera jamais ses équipes dans des aventures irréalisables, de même qu'un bon général est celui qui n'exposent pas ses troupes en vain. Il faut toujours qu'il y ait un enjeu et un défi pour motiver les équipes, mais cet enjeu doit être atteignable : c'est là l'essence de la stratégie.

Expression analytique et forme des équations de l'effort en fonction de la taille et de la durée en fonction de l'effort

Rappel des équations du modèle COCOMO

Pour le modèle de base, on a :

Typologie des programmes		
Simple de type conversion (Organic)	***Algorithmique*** (Semi-detached)	***Réactif*** (Embedded)
$Effort_{HM} = 2.4 \times KLS^{1.05}$ $D_{mois} = 3.5 \times KLS^{0.40}$	$Effort_{HM} = 3.0 \times KLS^{1.12}$ $D_{mois} = 3.8 \times KLS^{0.39}$	$Effort_{HM} = 3.6 \times KLS^{1.20}$ $D_{mois} = 3.8 \times KLS^{0.38}$
$D_{année} = 0.54 \times (E_{HA})^{0..38}$	$D_{année} = 0.50 \times (E_{HA})^{0..35}$	$D_{année} = 0.46 \times (E_{HA})^{0..32}$
L'approximation $D_{année} = 0.5 \times \sqrt[3]{Effort_{HA}}$ est légitime		

Equation de l'effort

Dans le modèle *PIP*, le programmeur est parfaitement isolé du monde extérieur. Dans la réalité des projets, il n'en est évidemment rien. Nous allons montrer par des raisonnements de type heuristique comment cet environnement influe sur l'équation d'effort, et finalement détermine sa forme mathématique.

Prenons comme étalon d'effort, l'effort qui a été nécessaire à un *PIP* pour produire un module *M* de *x* KLS ; nous noterons cet effort : $Eff_x(1)$. Pour produire un programme constitué de 2 modules de même longueur (ou taille) *x*, deux cas sont à considérer :

1er cas : les 2 modules sont totalement indépendants. Ils n'ont aucune relation entre eux et ne partage aucune donnée ; l'information de la somme des modules est exactement la somme de l'information de chacun des modules. Nous pouvons alors écrire : $Eff_x(2) = 2 \times Eff_x(1)$.

Plus généralement, si nous avons à faire à 2 modules indépendants de longueurs *x* et *y*, nous pouvons écrire $Eff_{x+y}(1) = Eff_x(1) + Eff_y(1)$

ainsi que $Eff_{p\times x+q\times y}(1) = p \times Eff_x(1) + q \times Eff_y(1)$ pour une combinaison linéaire de module de types *x* et *y*.

Du point de vue de l'effort, indépendance des modules signifie additivité de l'effort. Dans ce cas, l'équation d'effort reflète la proportionnalité résultant de l'additivité des efforts ; le monde parfait est linéaire et conservatif, soit pour *n* modules :

$$Eff_x(n) = n \times Eff_x(1)$$

NB : Dans la théorie de la mesure, il y a un des axiomes qui correspondrait dans notre notation à : $Eff_{x-y}(1) = Eff_x(1) - Eff_y(1)$. Une telle expression signifie que le retrait

d'une fonctionnalité y correspondrait à un retrait équivalent en termes d'effort. On sait qu'il n'en est rien ; il faut écrire, pour traduire notre intuition :

$$Eff_{x-y}(1) = Eff_x(1) - Eff_y(1) + CR\{M_x, M_y\}$$

dans laquelle $CR\{M_x, M_y\}$ représente le coût du retrait de M_y dans M_x. Bien qu'il s'agissent d'un retrait, ce retrait à un coût en termes d'effort, car ***toute manipulation d'information a un coût*** comme indiqué ci-dessous.

2ème cas : Les 2 modules ne sont plus indépendants. Ils partagent de l'information ; ils ont des données communes et ils peuvent interagir en s'échangeant des messages, quel que soit le canal d'échange, sur un mode `SEND` et `RECEIVE`, en séquence ou en parallèle, en synchrone ou en asynchrone. Ceci revient à dire que certaines entités de M_x sont vues de M_y et réciproquement. Dans ce cas, on est sûr que l'effort pour produire un ensemble intégré des deux modules, sera exprimé par une équation de la forme :

$$Eff_{x+y}(1) = Eff_x(1) + Eff_y(1) + CI\{M_x, M_y\} \qquad \text{Equation } \textbf{\textit{Eq1}}$$

dans laquelle le terme $CI\{M_x, M_y\}$ représente le coût correspondant aux interactions des modules et des PIP qui doivent se mettre d'accord sur un certain nombre de protocoles et/ou de conventions de communications, quelle que soit la nature de ces interactions ; le monde réel est dissipatif, l'information a toujours un coût. Par exemple, la simple mise en place d'une convention de nom, comme la « notation hongroise » en vigueur chez Microsoft[4], aura un coût d'apprentissage et d'emploi non négligeable ; à fortiori pour le découpage en modules qui est le résultat de l'architecture, et pour le processus d'intégration.

Les termes $CR\{M_x, M_y\}$ et $CI\{M_x, M_y\}$ traduisent la nature entropique de l'information.

- Tout retrait ou ajout de fonctionnalité sur un existant M_x à un coût.

Forme générale de l'équation d'effort

D'une façon générale, si n modules interfèrent, l'effort correspondant à CI devra considérer la partition des modules 2 à 2, 3 à 3,… n à n, c'est dire potentiellement l'ensemble des parties restreint à ce qui est visible d'un module à l'autre.

NB : ce qui est vu d'un module peut l'être de façon directe, par exemple au moyen d'allocation mémoire identique, par des recopies à l'identique, ou de façon indirecte à travers des pointeurs et/ou tout type de canaux cachés.

Le bonne partition du code et des données, et plus encore de tout ce qui traite du contrôle (enchaînements, exceptions, erreurs, etc.) est un aspect fondamental de l'architecture dont on voit ici l'importance en termes de coût.

Pour la simplicité du raisonnement, nous allons considérer des modules de même taille de façon à privilégier une équation de la forme :

$$Eff_x(n) = n \times Eff_x(1) \times CUI(n)$$

[4] Cf. M.Cusumano, R.Selby, *Microsoft secrets*, Free Press ; Chapitre 5.

dans laquelle *CUI* est une fonctionnelle à déterminer, mais dont nous connaissons a priori quelques propriétés.

Appelons coût unitaire, ou coût ramené à l'unité, des interactions la quantité :

$$CUI(n) = \frac{Eff_x(n)}{n \times Eff_x(1)}$$

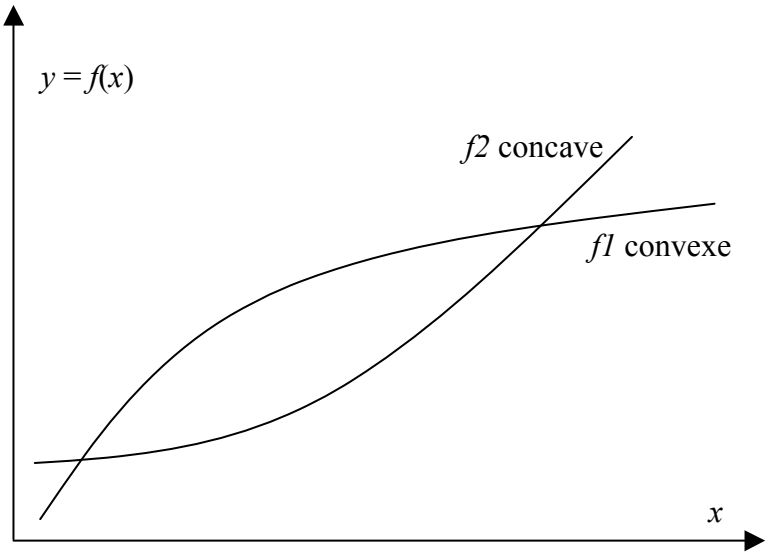
Cette quantité mesure le surcroît d'effort moyen occasionné sur l'un quelconque des modules du fait de son appartenance à la communauté des *n* modules formant l'application complète $Appli[\mathrm{A}] = \{M_1, M_2, M_3, \ldots, M_n\}$.

Compte tenu de l'équation *Eq1*, nous pouvons écrire :

$$CUI(n) = 1 + \frac{\sum \{Coûts\ des\ Interactions\}}{n \times Eff_x(1)}$$

La fonction *CUI*(*n*) est une fonction croissante de *n*, ce qui est intuitivement évident, mais, ce qui l'est moins, est également une fonction convexe, sous certaines hypothèses architecturales que nous allons préciser.

Rappelons que ce qui caractérise la convexité, ou la concavité, est le signe de la dérivée seconde de la fonction. Si $f(x)$ est convexe, alors $f''(x) < 0$, ce qui détermine la forme générale des courbes :



Dans ce schéma, *f1* est convexe ; le signe de la dérivée seconde signifie qu'il y a décélération régulière de la croissance ; *f2* est concave, il y a accélération de la croissance.

Parmi les fonctions convexes les plus simples qui pourraient convenir à la description du phénomène d'accroissement des coûts, notons, sur l'intervalle $[1, +\infty[$:

1. $y = \log(x)$ ou $y = \sqrt[n]{x}$, très simples à calculer, mais qui croissent à l'infini, ou encore :

2. $y = 1 + \alpha^2 \times \dfrac{x-1}{\alpha \times x + (1-\alpha)}$ dont l'asymptote est $y = 1 + \alpha$; et la dérivée seconde

$$y'' = \frac{-2\alpha^3}{(\alpha \times x + (1-\alpha))^3} < 0$$

Le choix d'une telle fonction pourrait se justifier en faisant l'hypothèse que le coût d'interaction décroit comme une progression géométrique de raison $r < 1$, soit une dissipation correspondant à :

$$C + C \times r + C \times r^2 + \ldots + C \times r^{n-1} = C\frac{1-r^n}{1-r} \cong C\frac{1}{1-r} \quad \text{pour } n \text{ grand,}$$

selon une formule bien connue.

Analyse du coût des interactions

Dans le meilleur des cas, le coût des interactions 2 à 2 est un terme de la forme :

$\dfrac{n(n-1)}{2} \times K_2$ selon les formules bien connues de l'analyse combinatoire. D'où une première approximation de la fonction

$$CUI(n) = 1 + \frac{n(n-1)}{2} \times \frac{K_2}{n \times Eff_x(1)} = 1 + k_2 \times (n-1)$$

dans laquelle on a normalisé K_2 en le ramenant à l'effort unitaire par module, soit k_2.

Remarquons qu'en exprimant le nombre de modules à partir de la longueur du code source comptant X KLS, on aurait : $n = Partie\ entière\left[\dfrac{X}{x}\right] + 1$, d'où la forme finale :

$$CUI(n) = 1 + a_1 \times (X)$$

Si l'on considère les termes correspondants aux interactions 3 à 3, 4 à 4, … on va voir apparaître des termes en $X^2, X^3, \ldots$ affectés de coefficients $a_2, a_3, \ldots$ dont on ne connaît ni la valeur ni le signe, mais dont on peut imaginer le type d'influence qu'ils doivent avoir sur *CUI*.

Dans une architecture comme celle schématisée ci-dessous, une augmentation de taille du système, donc une augmentation probable mais pas certaine du nombre de modules, ne va pas se traduire par un accroissement proportionnel de la fonction *CUI*, car tous les modules ne sont pas affectés par ce qui a été nouvellement introduit. La fonction *CUI*, bien que nécessairement croissante, n'est pas linéaire.

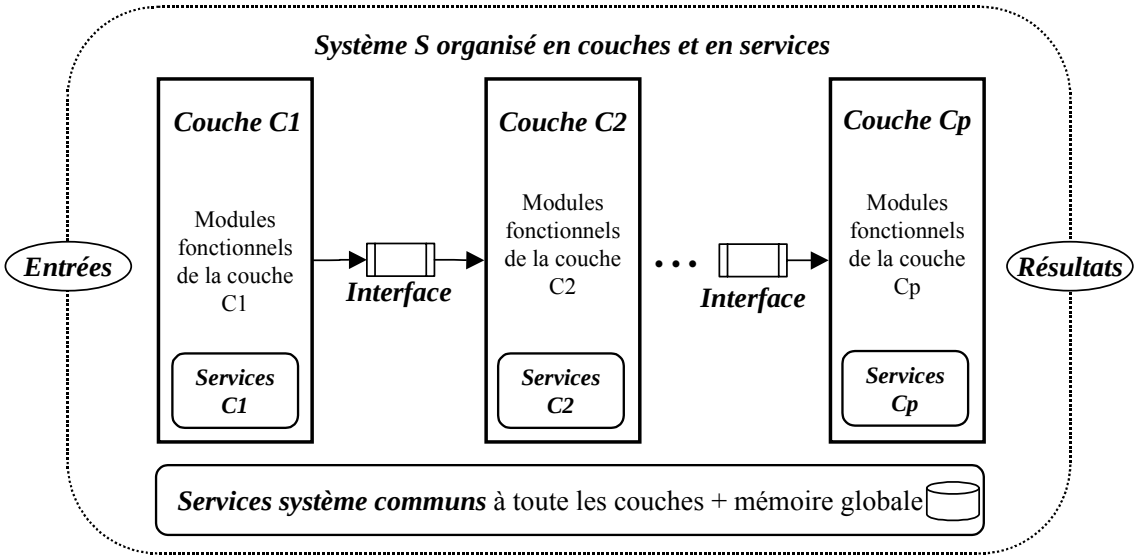


Figure 24 – Schéma d'une architecture canonique en couches

Si les modules fonctionnels sont organisés en hiérarchie, conformément aux règles de la programmation structurée correctement pratiquées, et a fortiori de la programmation objet, les interactions se trouvent restreintes aux lignes hiérarchiques. Or on sait que dans un arbre comportant n nœuds, la profondeur moyenne de l'arbre est de l'ordre de $O(\log(n))$. Dans une architecture canonique à p couches, il y a une hiérarchie de modules par couche ; la couche i n'étant en interaction avec les couches $i-1$ et $i+1$ qu'à travers les interfaces de couches[5]. On sait également que la fonction *Log* est convexe, et a un développement en série comme suit :

$$Log(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} + \cdots + (-1)^n \frac{x^{n+1}}{n+1} + \cdots$$

ce qui suggère une forme améliorée, beaucoup plus conforme à l'intuition de l'architecte pour ce qui concerne l'accroissement du coût correspondant à notre fonction *CUI*. L'inconvénient d'une fonction *Log*, est qu'elle croît indéfiniment, ce qui n'est pas conforme à l'observation des coûts d'un système correctement structuré. On est donc conduit naturellement à rechercher des formes de fonctions d'effort du type :

- $Eff_x(n) = n \times Eff_x(1) \times \left(\sqrt[b]{n}\right)$; *la plus simple*,

NB : on sait que que la fonction puissance est décomposable comme suit

$$(1+x)^b = 1 + b \times x + b(b-1)\frac{x^2}{2!} + b(b-1)(b-2)\frac{x^3}{3!} + \cdots + b(b-1)\cdots(b-n+1)\frac{x^n}{n!} + \cdots$$

- **(2)** $Eff_x(n) = n \times Eff_x(1) \times \left(\log(n)\right)$ que l'on peut développer avec la série *Log* précédente,

[NB : ces deux fonctions ont l'inconvénient de croître à l'infini, donc il vaut mieux être un expert pour les manipuler correctement ; avec ces équations, il n'y aura jamais

[5] Une excellente description de l'architecture en couches se trouve dans la norme UIT/CCITT X.200-X.219 (*Blue book*), Tome VIII-Fascicule VIII.4, Section 1 *Modèle et notation*.

d'économie d'échelle, ce qui est contraire aux phénomènes constatés dans les grands systèmes correctement architecturés. (1) et (2) ne sont valides que sur une certaine plage $[n_1; n_2]$ bien spécifiée]

ou encore, en utilisant la fonction avec asymptote :

- **(3)** $Eff_x(n) = n \times Eff_x(1) \times \left(1 + \alpha^2 \times \frac{n-1}{\alpha \times n + (1-\alpha)}\right)$

[NB : avec cette équation, l'économie d'échelle apparaît quand on est au voisinage de l'asymptote, car le coût devient quasi linéaire]

Notons qu'en utilisant les développements en série, on peut choisir, par plage de valeurs, des polynômes d'approximation très voisins ; donc, en application du principe *KISS*, on choisira une équation de la forme (1) qui est celle de COCOMO 81.

Le choix des coefficients est un problème d'analyse de données et de statistique qui n'est pas trivial, mais qui est fortement simplifié quand on connaît la forme des équations.

- Le résultat fondamental de l'analyse qualitative de la forme des équations est que toute déficience concernant le contrôle, tant au niveau des interactions entre les acteurs PIP et les architectes, qu'au niveau des interactions des modules et/ou des services consécutives à une mauvaise architecture, se traduira par une augmentation de l'exposant et la disparition de l'asymptote, ce qui aura un effet catastrophique sur les coûts, et ce d'autant plus que cette déficience ne pourra être révélée que tardivement dans la phase d'intégration.

Un traitement plus approfondi, et mathématiquement plus rigoureux, de la forme de l'effort correspondant aux interactions nécessiterait l'emploi de la théorie de l'information (l'information correspondante aux interactions à un coût en termes d'entropie ; les interactions sont assimilées à du désordre), ce qui pourrait constituer un beau sujet de thèse et reprendre sur une meilleure base les travaux de M.Halstead déjà cités !

Equation de la durée

L'équation qui donne la durée d'un projet pour un effort donné peut paraître étrange, voire magique, alors qu'elle ne fait que traduire des relations de bon sens comme nous allons le montrer maintenant.

Chacun sait que dans la dynamique projet, l'effectif ne peut pas passer brutalement de l'effectif initial, que nous appelons *germe*, à l'effectif maximum considéré comme optimal, que nous appelons *pic*.

Il y a plusieurs raisons à cela :

1. Il est inutile d'augmenter l'effectif du projet, si les nouveaux arrivants ne savent pas ce qu'ils doivent faire ; l'effectif va augmenter en fonction de la capacité du germe à préparer rapidement et de façon fiable le travail pour les nouveaux arrivants.
   NB : Ce que préconise des experts qualité comme Crosby avec son leitmotiv « *Do it right first time* » est une condition *sine qua non* du germe.
2. Les ressources humaines disponibles font qu'il est plus ou moins difficile de recruter le personnel surtout si le projet nécessite des profils de compétences rares.

Le pic d'effectif correspond à la phase de programmation /tests unitaires. Dès que cette phase sera terminée, l'effectif décroîtra progressivement jusqu'à l'effectif indispensable à la maintenance.

En première approximation, on peut considérer que l'effectif a un profil correspondant au cas N°1.

Cas N°1

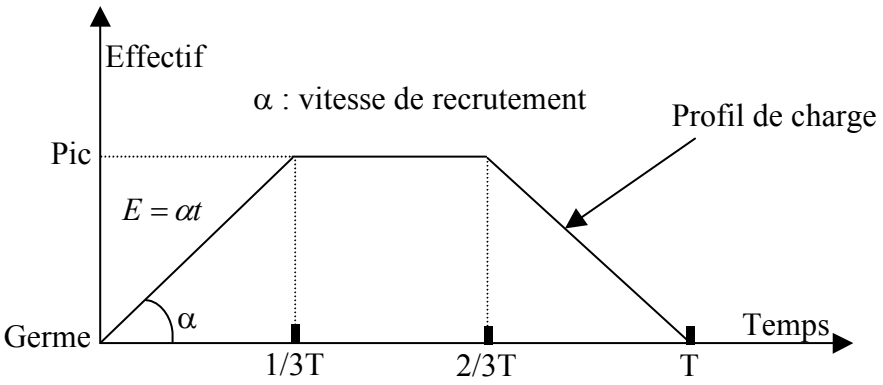


Figure a12 : Vitesse de recrutement linéaire

Pour la simplicité du calcul nous avons considéré trois périodes dans la vie du projet :

- La montée en charge, qui se fait avec une vitesse constante α ; au bout d'un temps *t*, l'effectif est donc $E = \alpha t$.
- Le plateau, où l'effectif est stable, même s'il y a des renouvellements de personnes.
- L'intégration qui, dès que les tests d'intégration sont satisfaisants, va permettre de restituer les programmeurs au pool de ressources disponibles pour les autres projets.

L'effort libéré par un tel profil est donc égal à la surface sous la courbe du profil de charge.

Dans le cas N°1, nous considérons que les trois périodes ont la même durée, ce qui permet d'écrire :

$$Effort = \frac{1}{2}\frac{\alpha T}{3}\times\frac{T}{3} + \frac{\alpha T}{3}\times\frac{T}{3} + \frac{1}{2}\frac{\alpha T}{3}\times\frac{T}{3} = \frac{2\alpha}{9}T^2$$

Selon cette hypothèse on voit que :

$$T = 1.5\sqrt{\frac{2}{\alpha}}\times\sqrt{Effort}$$

ce qui est une première approximation de la durée probable.

Perfectionnons le modèle et considérons le cas N°2 où la vitesse de montée en charge s'accélère avec le temps.

Cas N°2

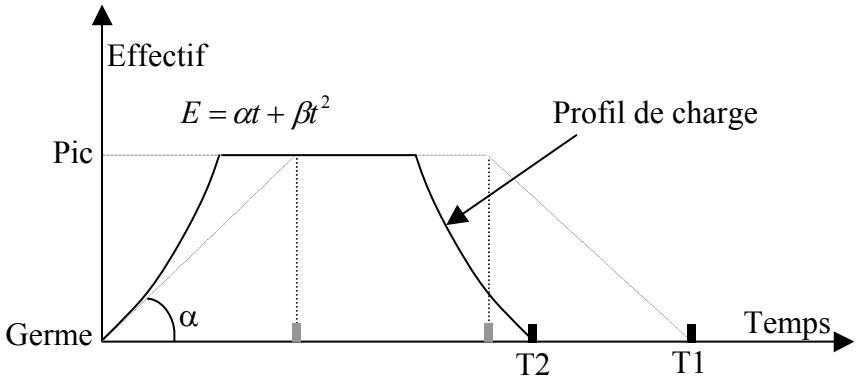


Figure a13 : Vitesse de recrutement uniformément accélérée

Notons que dans une dynamique de croissance proportionnelle à l'effectif à l'instant t, l'évolution de la population serait exponentielle. Pour simplifier le calcul, nous ne considérerons que le terme en t^2 de l'exponentielle, d'où l'équation de croissance : $E = \alpha t + \beta t^2$

Toujours pour rester simple, nous supposerons que les trois périodes de temps restent identiques du point de vue de l'effort et que la décroissance est symétrique de la croissance (ce qui est tout à fait raisonnable).

Sous cette hypothèse, la somme des efforts devient :

$$Effort = 2\int_0^{\frac{T}{3}} (\alpha t + \beta t^2)\, dt + \frac{\alpha}{9} T^2$$

Après intégration, on obtient :

$$Effort = 2\left[\frac{\alpha}{2} t^2 + \frac{\beta}{3} t^3\right]_O^{\frac{T}{3}} + \frac{\alpha}{9} T^2 = \frac{2\alpha}{9} T^2 + \frac{2\beta}{81} T^3$$

Si l'on compare les deux équations, on voit que la seconde contient un terme de la forme kT^3, en plus du terme identique en T^2 ; ce qui fait que l'écart de durée entre les deux profils diminue comme la $\sqrt[3]{Effort}$, terme que l'on retrouve dans les équations T_{DEV} du modèle COCOMO ou d'autres modèles semblables ; il n'y a donc aucun mystère !

On peut également comprendre que dans un projet complexe où il y a beaucoup d'interactions, et donc beaucoup de tests à faire, le profil de la décroissance soit beaucoup plus lent, pour un même volume de programmation, d'où l'allure des équations du modèle COCOMO qui sont en fait intermédiaires entre une $\sqrt[2]{Effort}$ et une $\sqrt[3]{Effort}$.

De ce calcul, on peut tirer plusieurs **remarques très importantes** :

- La **capacité d'embauche au démarrage** du projet est primordiale ; les coefficients $\{\alpha, \beta\}$ fixent à eux seuls l'allure générale de la courbe de charge. ***Si le germe est médiocre, l'ensemble sera [très] médiocre*** car l'effet néfaste correspondant est au mieux cumulatif, ce qui dégradera la productivité.
- Le **délestage**, ou relaxation des ressources, dans la période d'intégration, doit être organisé avec une très **grande précision**. En particulier s'il est prématuré (i.e. pas assez de tests unitaires) il faudra rappeler les programmeurs qui sont peut être déjà réaffectés sur d'autres projets pour leur faire corriger après coup ce qu'ils auraient du faire dans la phase P/TU[6] ; la perturbation sera alors très importante.

Perfectionnements possibles du modèle de montée en charge

Le calcul ci-dessus fait l'hypothèse que dans le cas N°2, un effet d'accélération se superpose à la montée en charge linéaire.

Une approximation plus fidèle de la réalité est donné par le cas N°3.

Cas N°3

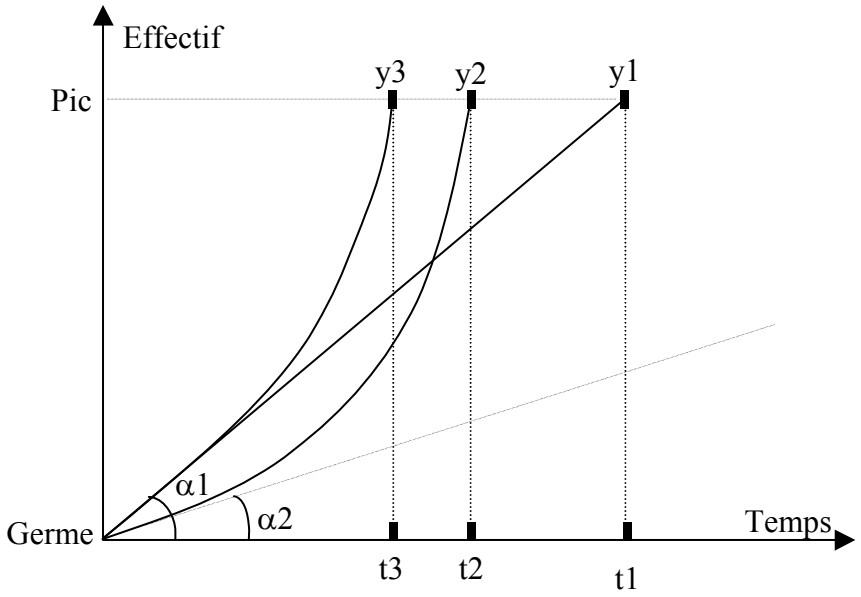


Figure a14 : Différents profils de montée en charge

La montée en charge est plus lente au départ (ce qui correspond au recrutement d'experts) puis s'accélère. La condition d'équivalence des surfaces s'écrit alors :

$$\int_0^{t3} y_3 dt = \int_0^{t2} y_2 dt = \int_0^{t1} y_1 dt$$

On voit que pour la courbe *y3* qui est complètement au dessus de *y1*, il faudrait peut-être compléter la surface sous la courbe par un petit rectangle additionnel sur une tranche de temps Δt, ce qui revient à utiliser notre sigmoïde en *S*.

6 D'où l'importance des critères quantifiés en termes de couverture recommandés au chapitre 1.2 ci-dessus.

La résolution des équations correspondantes (NB : elles sont du 4ème degré !) permettrait de trouver des jeux de coefficients $\{\alpha, \beta,\}$ qui garantissent l'égalité des surfaces (i.e. des efforts). Tous ces calculs ne changeraient d'ailleurs pas l'allure des courbes et l'on peut penser que les écarts rentreraient dans le bruit de fond du modèle d'estimation qui a bien d'autres paramètres d'incertitudes.

Une modélisation plus utile consisterait à déduire le profil de charge utile en fonction du profil théorique visible de la comptabilité analytique du projet (i.e. la grandeur C). Le profil utile caractérise le rendement du projet (i.e. les ressources effectivement productives).

Le profil de charge utile reflète, d'une certaine façon, la capacité de management, et la maturité du chef de projet et de l'équipe. D'où :

Cas N°4

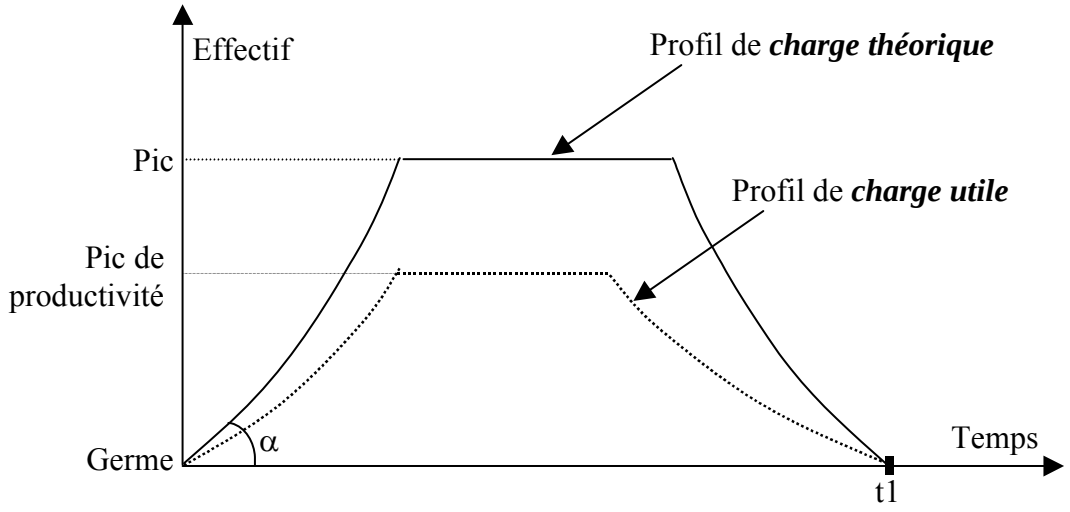


Figure a15 : Profil de productivité

Un chef de projet au stade 4 ou 5 sera très vigilant sur la « réunionite » et contrôlera très soigneusement les critères de fin de tâche (i.e. le paramètre S du modèle VEST) de façon à éviter les retours arrière qu'il sait être très coûteux en productivité et qui risquent de rendre le profil utile complètement chaotique.

Il sera très attentif à la compétence des personnes qui vont constituer le germe de son projet car il sait, et le modèle le confirme, que c'est d'eux que dépendra la dynamique de la croissance et au bout du compte la productivité de son équipe.

L'ensemble de ces courbes explique complètement le phénomène connu sous le nom de loi de Brooks :

- L'ajout d'effectif à un projet en retard ne fait que le retarder encore plus.

Nos équations montrent, en effet, que dans le meilleur des cas l'ajout d'une personne ne peut réaliser qu'un gain de temps de l'ordre de la $\sqrt[3]{\Delta ajout}$, ce qui n'est pas beaucoup ! Mais ceci ne tient pas compte des perturbations engendrées par le nouvel arrivant qui vont dégrader le profil de la charge utile (surcoût de communication, formation indispensable sinon il commettra de nombreuses erreurs, etc.) ; il suffit de faire perdre

le temps de quelques personnes clés très productives pour annuler complètement le gain !

L'une des difficultés principales du chef de projet est que son management ne voit que le profil de charge théorique, alors que lui ne voit que le profil de charge utile. Si le management est peu informé des spécificités du développement de logiciels l'incompréhension est inévitable ; et ce d'autant plus que la productivité visible de chaque programmeur ne reflète aucunement leur contribution réelle au succès global de l'équipe. D'où une autre caractéristique du chef de projet expert :

- Le chef de projet doit être, en plus de beaucoup d'autres qualités, un bon pédagogue ; il doit utiliser tous les aléas rencontrés pour expliquer et former son environnement organisationnel et humain.

EXTRAIT N°2, DE : *ECOSYSTEME DES PROJETS INFORMATIQUES*, CHEZ HERMES

Architecture, Complexité, Intégration, programmation

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 définit 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, 1ère 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 de l'OS 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*, 1ère 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 profondeur car il ébauche une analyse quantitative entre les trois aspects qui nous préoccupent ici (données, transformations, événements).

Depuis les années 90s, 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, D.Garlan, *Software Architecture*, Prentice Hall 1996, on trouve une définition (page 3) « *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. … 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.

Le cadre d'architecture développé par le ministère de la défense américain[7] qui est un peu la référence en matière d'architecture de système 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 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 d'information sérieux, ne renierait ce genre de recommandations.

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 complexes qui incombe à l'architecte.

Parmi les équipes, celles 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 inclut 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 !

On peut compléter les définitions précédentes de l'architecture, par une définition beaucoup plus opérationnelle et constructive, intégrant la problématique projet et donnant un critère de terminaison :

[7] Il s'agit d'un ensemble de travaux intitulés DODAF, pour DOD Architecture Framework.

Règle d'architecture dans une perspective projet : L'architecture d'un système est terminée quand, dans le projet de réalisation chacun sait ce qu'il doit faire (aspect descriptionnel et fonctionnel 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) compte tenu des contraintes économiques de coût, qualité et délai, i.e. CQFD et FURPSE.

Dans cette définition, « chacun » peut être un individu, une équipe nominale (≈ 7 personnes ± 2 est une bonne pratique), un ensemble d'équipes partageant une même finalité technique et/ou métier. 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 termes de risque, en particulier combinatoire.

Ce qui a été dit pour l'équipe d'intégration est parfaitement pris en compte dans cette définition. Cette équipe teste et assemble, 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 notre 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 la taille du système en points de fonctions ou en volume de code, la taille des tests, son coût, sa qualité, le délai de la réalisation de la 1ère version, sa durée de vie (combien de versions est-il raisonnable de prévoir), 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 compenser 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 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.

On sait que prendre en compte trop tardivement la problématique de l'erreur, ou ne pas se préoccuper assez tôt des performances conduit droit à l'échec ; mais il est très difficile de s'en préoccuper sans une première formulation du système, d'où les retours

arrière et, ipso facto, la non linéarité. Ceci vaut pour toutes les caractéristiques non fonctionnelles.

D'un point de vue théorique, c'est un problème de stabilité ou de point fixe. Les évolutions successives du système laissent le schéma architectural invariant, ce que l'on peut exprimer par une équation du type :

$$\text{Arch (Arch (x))} \rightarrow \text{Arch (x)}$$

Les étapes successives ne doivent pas altérer le schéma initial, mais le compléter ; faute de quoi, le processus sera divergent.

En analysant plus finement les conditions de stabilité, on est amené à confronter deux séries : a) une série d'incréments fonctionnels, à partir d'un germe F0, soit *F0*, $\Delta_1[F0 \rightarrow F1]$, $\Delta_2[F1 \rightarrow F2]$, … , etc., comme par exemple une suite de cas d'emploi, et b) une série d'incréments architecturaux construits à partir d'un germe Arch0, soit $\Delta_1[Arch0 \rightarrow Arch1]$, $\Delta_2[Arch1 \rightarrow Arch2]$, …, qui, si l'architecture est stabilisée, doit tendre rapidement vers 0 ; la signification de ce terme est qu'il n'y a plus d'effort architectural à fournir.

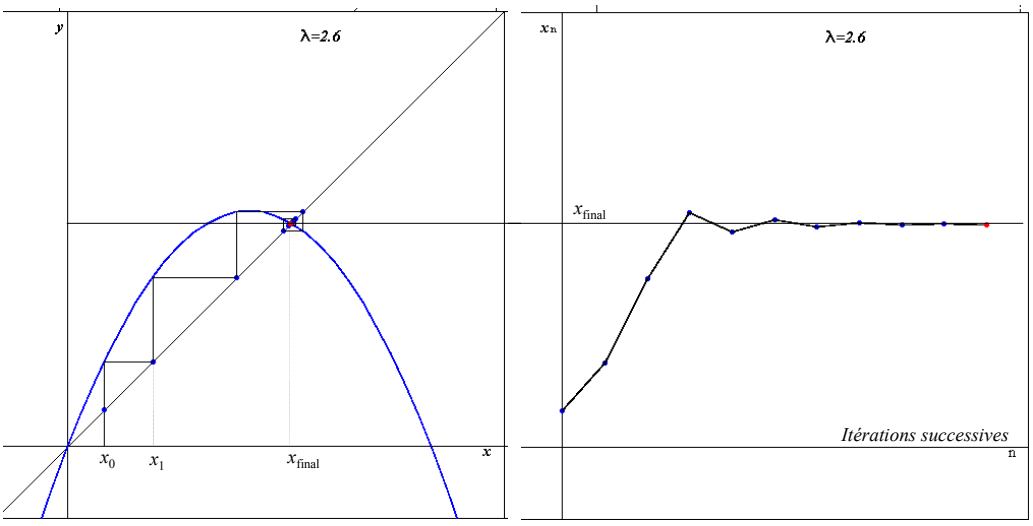
Pour les incréments architecturaux, on peut dire que l'incrément de l'étape $(\mathrm{n}+1) \rightarrow \Delta_{n+1}[Arch(\mathrm{n}+1) \rightarrow Arch(\mathrm{n})]$ est le « produit » de ce qui a été acquis à l'étape *n*, et de ce qui reste à découvrir en matière d'architecture par rapport aux évolutions envisagées. S'il n'y a plus rien à découvrir, l'incrément architectural est nul, l'architecture est alors stabilisée. A cet incrément correspond un travail qui résulte de ce qui a été déjà acquis, mais qu'il faut éventuellement retoucher, et de ce qui vient d'être découvert.

Intuitivement, la forme de ce terme est $Eff_{n+1} = Eff_n \otimes (Eff_{final} - Eff_n)$.

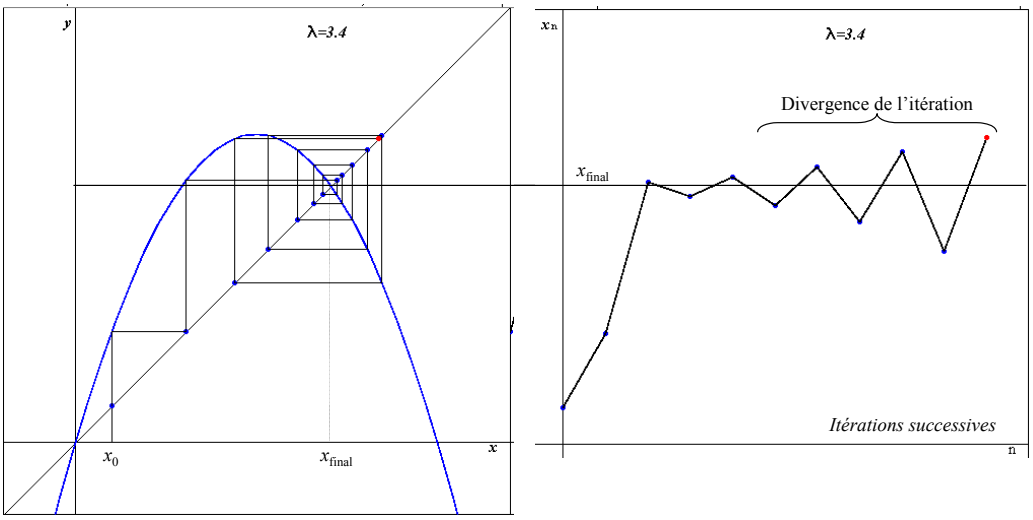
La question qui se pose alors est de savoir si il y a convergence automatique où non ? Comme il s'agit d'équations à différence finie, la réponse n'est pas triviale, mais on peut l'illustrer à l'aide de l'équation logistique que nous avons déjà rencontrée en gestion de projet, soit $X_{n+1} = \lambda X_n (1 - X_n)$ qui a la même forme que l'équation précédente. Sous sa forme différentielle, c'est l'équation de la courbe en S.

Ce qui est remarquable avec cette équation très simple est que nous sommes certain qu'il existe une solution non nulle $x = 1 - \frac{1}{\lambda}$ qui détermine la valeur du point fixe, mais que si nous recherchons la solution par une méthode itérative le résultat n'est pas évident comme le montre les graphiques ci-dessous.

La première courbe avec λ=2,6 montre un cas de convergence. En partant avec une valeur de *x* approchée, on converge vers la solution par itération successive.



La deuxième courbe avec λ=3,4 est beaucoup plus surprenante car elle montre un cas de divergence. Dans un premier temps, les itérations successives se rapprochent de la solution, puis se mettent à diverger.



L'analyse du pourquoi de la dynamique de l'itération n'a pas lieu d'être ici, mais on peut consulter le livre de R.Holmgren, *A first course in discrete dynamical systems*, Springer, 1996, et le remarquable article de R.May, *Simple mathematical models with very complicated dynamics*, Nature, Vol. 261, June 10, 1976. Les systèmes biologiques comme les écosystèmes offrent une très large gamme de dynamiques de ce type.

En reprenant une analogie architecturale, cela montre combien il faut se méfier de la non linéarité du processus de construction du système. Tout va dépendre du germe initial, et plus particulièrement de la capacité de l'architecte à identifier le cadre architectural qui convient à ce projet particulier, même si ce cadre n'est pas complètement développé dans l'incrément N°1. S'il anticipe correctement la solution finale, tout se passera bien. Si ce n'est pas le cas, il faut s'attendre à quasiment tout refaire à chaque itération. C'est ce que de nombreux auteurs en génie logiciel ont décrit de façon qualitative sans aller au fond des choses.

L'architecture est un processus d'optimisation multicritère, où l'on recherche des *Mini-Max* sur les critères CQFD et FURPSE, dont le résultat est une description textuelle et/ou graphique plus ou moins formalisée. C'est un jeu, au sens de la théorie des jeux, où l'ordre des évènements influe sur la solution adoptée in fine, et d'une certaine façon détermine la trajectoire du projet.

Si les données constituent le cœur du système, comme le proclame Brooks à juste titre, et induisent les traitements qui en constituent le cœur fonctionnel, ce sont les événements, résultant de la non linéarité du processus, qui vont en caractériser la représentation programmatique qui à l'instant t constituera la livraison. La typologie des événements et l'ordre dans lequel ils vont être pris en compte par l'équipe de conception, sont la matière première des décision architecturales qui, au final, constituera non pas LA solution, mais UNE solution, qui elle même produira son lot d'événements conduisant à une deuxième version, et ainsi de suite. L'architecture initiale et son histoire événementielle détermine la capacité évolutive et adaptative du système, et d'une certaine façon sa durée de vie. Comme le montre l'équation logistique, des divergences restent possibles, même quand cela semble converger.

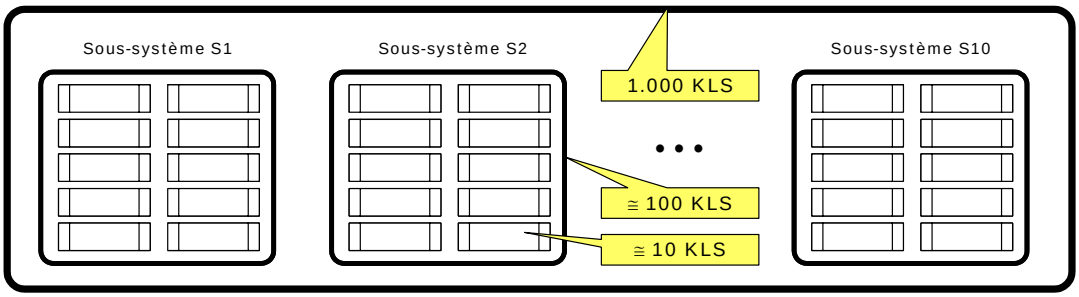
Architecture et complexité

Pour illustrer quantitativement la problématique considérons l'intégration d'un système formé de 1000 composants logiciels élémentaires.

Pour la simplicité des calculs on considère un système dont la taille est de 1.000.000 de lignes de code source (LS), soit, en millier de LS, 1.000 KLS (NB : cette taille correspond à un « gros » système d'information comme il en existe aujourd'hui des dizaines dans les entreprises).

Ce système, conformément à la terminologie de l'ingénierie système est formé de : 10 éléments comportant chacun 100 KLS (ce sont des ***sous-systèmes*** et/ou des ***applications*** constitutifs du système complet) ; chaque élément/sous-système est formé de 10 modules/packages dont la taille est de 10 KLS ; soit au total 100 modules/packages à intégrer pour constituer le système complet.

Le schéma ci-dessous donne la topographie d'un tel système.



Le nombre d'« intégrats[8] » dans une telle structure hiérarchique est donc égal à :

[8] Correspond à la notion de « building blocks » ; du point de vue de l'intégration, ce sont des boites noires qui ne sont connues de l'intégration que par leurs interfaces.

100 + 10 + 1 = 111 chaque intégrat regroupant 10 entités de rang inférieur (appelées sous-système/application, ou module, ou « composant » selon le niveau).

On fera l'hypothèse que les composants (on peut les assimiler à des objets) sont eux mêmes formés de « pièces/méthodes » logiciel de 1 KLS en moyenne (NB : c'est l'unité de comptage du modèle COCOMO) ; dans ce modèle l'équation d'effort pour un système de ce type est donnée par la formule : $Effort_{en\ hm} = 2{,}4 \times (Taille_{en\ kls})^{1{,}05}$).

- Le système complet est constitué de 1.000 « composants » élémentaires (ou de 10.000 « pièces » selon le niveau de granularité considéré.
- La décomposition hiérarchique est : Système → Sous-système/application → Modules/packages → Composants/Classes–Objets → Pièces/Méthodes

Les composants logiciels sont le résultat du travail de petites équipes de programmation dont l'effort de réalisation moyen est de $2{,}4(10)^{1{,}05} = 27hm$ soit 2,3 ha par composant, ce qui correspond à une productivité industrielle moyenne de l'ordre de 370 LS par programmeur et par mois (lignes de code documentés et testées, conformément aux normes en vigueur). Une pièce est le travail d'***un*** programmeur, mais un programmeur peut réaliser plusieurs pièces.

On remarquera que si l'intégration ne coûtait rien, le coût d'un tel système serait simplement de 100×2,3=230 ha. On peut vérifier la cohérence des coûts et l'impact de l'intégration en comparant l'effort de développement de 1.000 KLS à celui de 10 fois 100 KLS et de 100 fois 10 KLS, en utilisant la formule du modèle COCOMO basique ci-dessus, ce qui donne les résultats suivant :

Taille de l'intégrat en KLS	Nombre d'intégrats	Effort en ha	Différentiel conception intégration	Remarques
1.000 KLS	1	282 ha	–	
100 KLS	10	250 ha	32 ha	
10 KLS	100	230 ha	52 ha	Dont 20 ha de préintégration des 10 composants

Ce premier calcul illustre ce que va coûter le fractionnement d'un gros problème en 10 moyens problèmes ou en 100 petits problèmes et la reconstruction du puzzle pour obtenir le système de 1.000 KLS dans son état final. Pour cela, on a appliqué sans nuance le modèle COCOMO à des volumes de programmation, respectivement de 10 KLS, 100 KLS et 1.000 KLS.

Un calcul plus conforme à l'intuition que l'on peut avoir de la complexité d'un tel système devrait utiliser une équation d'effort rectifiée à l'aide des règles de paramétrage du modèle, en corrigeant le facteur de coût et le facteur d'échelle comme suit :

- La vitesse de programmation et la productivité moyenne vont très certainement décroître car il y a nécessairement plus de règles à respecter ; on va donc augmenter le coefficient 2,4 de respectivement 5%, 10% et 15%.

- Le nombre d'acteurs en interaction est beaucoup plus grand, les équipes sont plus nombreuses donc la maturité moyenne va baisser, il y aura plus d'erreurs, les consensus seront plus difficiles à atteindre, donc plus coûteux, ce qui rejaillit sur le coefficient 1.05 que l'on va passer à 1.08, soit environ 3% d'augmentation.

Le tableau ci-dessous résume la prise en compte de cette rectification.

Rectification (2,4 ; 1.08)		(2,52 ; 1.08)		(2,64 ; 1.08)		(2,76 ; 1.08)	
Effort	Intégration	Effort	Intégration	Effort	Intégration	Effort	Intégration
348 ha	–	365 ha	–	382 ha	–	400 ha	–
289 ha	59 ha	303 ha	62 ha	318 ha	64 ha	332 ha	68
240 ha	108 ha	252 ha	113 ha	265 ha	117 ha	276 ha	124 ha

Les calculs montrent l'impact prépondérant du facteur d'échelle et donc le sérieux avec lequel il faut prendre la complexité en considération. Compte tenu du nombre d'intégrats à considérer, l'influence du facteur d'échelle fait plus que doubler, à lui tout seul, le coût de l'intégration. La productivité n'est affectée que de façon linéaire.

NB : Pour tous les calculs d'effort avec le modèle COCOMO, on raisonne en année pleine, soit 1 année ≈220 jours ouvrés ; 1 jour ouvré = 8h brutes, avec 1hm de ≈20j (les congés, les 35 heures, … modifient substantiellement ces calculs ; on ne travaille pas plus vite pour autant) ; pour plus de détail, voir nos deux ouvrages *Productivité des programmeurs*, et *Coûts et durée des projets informatiques*.

Le responsable de l'intégration du système S a le choix entre deux stratégies,

1. Stratégie N°1 (parfois appelée big-bang) : il intègre les 1.000 pièces d'un seul coup car il ne connaît pas la nature des relations qui peuvent exister entre les différentes pièces qui lui sont livrées ; dans cette stratégie, le responsable d'intégration reçoit donc 1.000 pièces, en vrac.
2. Stratégie N°2 : le système est structuré en couches/serveurs indépendant[e]s, structurées en blocs de 10 entités selon le niveau, jusqu'à arriver au système complet ; dans cette stratégie, le responsable d'intégration reçoit toujours 1.000 pièces, mais il sait les assembler par paquets de 10, jusqu'à constitution de l'intégrat final. Il organise le processus d'intégration en fonction des informations qui lui ont été communiquées par l'architecte du système.

On remarquera qu'en termes de combinatoire, la stratégie S1 est de l'ordre 1.000 ! (i.e. factorielle 1.000, qui est un nombre immense), alors que S2 est de l'ordre $10 \times 100! + 10!$, également très grand, mais beaucoup plus petit en relatif.

Le niveau de test des pièces élémentaires (obtenu par la mise en œuvre d'une politique rigoureuse de tests unitaires) est tel que l'on peut estimer leur fiabilité f à 0,99%, ce qui signifie que sur 100 exécutions de la pièce on observera, en moyenne, une seule défaillance ou anomalie (NB : les données d'entrée d'une pièce peuvent changer d'une

exécution à la suivante, ainsi que l'environnement de la pièce, conformément au phénomène de dégénérescence qui se manifeste lors de l'exécution des pièces[9]).

Fiabilité des chaînes de liaisons

La formule classique donnant la fiabilité d'une chaîne de liaison de longueur *n* qui regroupent n intégrats I1, I2, … In, en prenant comme hypothèse simplificatrice que leur fiabilité est égale à f, est :

$$F(n) = f^n$$

soit pour n=10 → F=0,90 ; n=100 → F=0,37 ; n=200 → F=0,13 ; n=500 → F=0,007 et n=1.000 → F=0,00004

En valeur brute, la fiabilité tend donc inéluctablement vers 0.

Il faut toutefois faire très attention à l'interprétation de ces valeurs dans la réalité des modes d'exécution et d'emploi des systèmes informatiques ; c'est ce que nous allons maintenant esquisser.

La question de fond concerne l'interprétation du ratio des fiabilités relatives d'un agrégat de 10 éléments agrégés progressivement par paquet de dix (stratégie d'intégration N°2) sur un agrégat de 1.000 pièces (stratégie d'intégration N°1) ? Quel est le ratio d'effort de non régression dans l'une ou l'autre de ces stratégies, compte tenu du nombre moyen d'éléments sur lequel porte la non régression.

NB : On peut remarquer que dans la stratégie N°1, l'intégrat système est de 1.000 pièces ; et que dans la stratégie N°2, il y a 111 intégrats dont chacun comporte 10 pièces (élémentaires, ou déjà agrégées) ; dans ce dernier cas, la fiabilité du système complet est $F = \frac{f^{10}}{111}$, car la fiabilité — qui est équivalente à une probabilité — est additive dans ce cas, alors qu'elle est multiplicative dans l'autre.

Dans ce qui suit, le système S, au moment ou démarre l'intégration, contient un certain nombre d'erreurs résiduelles dont la distribution ne dépend pas de la stratégie d'intégration. Ce taux dépend de l'efficacité des tests unitaires faits sur les différentes pièces qui ont conduit à une fiabilité de 0,99 pour chaque pièce.

Il est intuitivement évident que la stratégie S1 est plus coûteuse que la stratégie S2, mais la vraie question est de savoir de combien ? qq. % ? ou 1 ou 2 ordres de grandeur, soit ×10, ou ×100 plus.

Ce qui suit ébauche la ligne de raisonnement, sur la base des hypothèses simplificatrices que nous nous sommes données, et détermine les ordres de grandeur.

NB : pour des raisonnements[10] plus fouillés, voir en particulier le chapitre 7.2 du livre *Puissance et limites des systèmes informatisés* et *Productivité des programmeurs*.

Le système le moins fiable occasionne en moyenne plus de défaillances ; on peut donc s'attendre à ce que le coût des interventions avec S1 soit plus élevé que avec S2.

Dans la stratégie S2, l'intégration se fait en trois étapes :

---

9 Cf. J.Printz, *Puissance et limites des systèmes informatisés*, déjà cité.

10 Voir également un raisonnement très intéressant dans H.A.Simon, *The sciences of the artificial*, MIT Press 1996, 3rd edition ; en particulier le chapitre 8, The architecture of complexity : hierarchic systems.

- 1ère étape : 10 « pièces » de fiabilité 0,99 sont intégrées pour constituer un module auquel est associé un jeu de test qui permettent d'obtenir une fiabilité de 0,99. Ce travail est à faire 100 fois.
- 2ème étape : on regroupe les modules par paquet de 10 pour former les sous-systèmes et les applications auxquels sont associés de nouveaux jeux de tests qui permettent d'obtenir une fiabilité de 0 ,99. Ce travail est à faire 10 fois.
- 3ème étape : intégration finale des 10 sous-systèmes pour former le système complet, avec un nouveau jeu de tests qui permet d'obtenir une fiabilité finale de 0,99.

Le rôle des tests d'intégration est donc de ramener la fiabilité des intégrats à celle de leurs constituants élémentaires, et ce jusqu'à l'obtention de la fiabilité globale exigée par la qualité de service auquel le système doit satisfaire.

Si les 111 intégrats issus de la stratégie S2 ont tous la même fiabilité (soit 0,99), on est en droit de dire que la fiabilité de S issue de la stratégie S2 est égale à $\frac{f^{10}}{111} \approx 8 \times 10^{-3}$ (les intégrations des différents intégrats sont indépendantes, donc les probabilités sont additives). Le quotient des fiabilités s'interprète comme le nombre moyen d'interventions à effectuer sur S compte tenu de la stratégie d'intégration S2 rapporté à S1, soit $N = \frac{\frac{f^{10}}{111}}{f^{1000}} \approx \frac{8 \times 10^{-3}}{4 \times 10^{-5}} \approx 200$.

Sous ces hypothèses, il y a 200 fois plus d'interventions avec la stratégie S1 que avec la stratégie S2, donc il y aura beaucoup plus de non régressions à effectuer avec S1 que avec S2.

Reste maintenant à estimer le coût moyen d'une intervention, dans l'une ou l'autre des stratégies d'intégration.

Avec S1, on a moins de scénarios, mais les scénarios sont plus longs à écrire et ils sont plus complexes à vérifier que avec S2 car les ensembles à considérer ont des cardinalités très différentes : 1000 versus 10 .

La taille de l'intégrat (i.e. le nombre de constituants élémentaires de l'intégrat) détermine la taille du/des scénarios de tests correspondant à cet intégrat. Lors d'un test, certaines pièces sont utilisées plus fréquemment que d'autres. Si l'intégrat contient ***n*** pièces, il faudra exécuter beaucoup plus que ***n*** pièces pour pouvoir les exécuter toute, au moins une fois ; ce qui complique à la fois l'écriture du scénario et sa vérification (donc cela augmente son coût). En effet, en l'absence d'information sur la structure de l'intégrat, il faut s'en remettre au hasard, et au phénomène de doublon que le hasard implique.

Avec la stratégie S1, on teste à l'aveugle en faisant complètement confiance au hasard ; on ne sait même pas quelles sont les pièces effectivement utilisées, et combien de fois elles l'ont été : on ignore tout de la distribution, et cette ignorance a évidemment un coût en termes de non qualité (ou d'effort additionnel, selon les points de vue).

Dans la stratégie N°2, le processus d'intégration est entièrement guidé par l'architecture (cf. le concept d'architecture testable). Si une défaillance apparaît on sait exactement dans quel intégrat parmi les 111 cette défaillance apparaît ; en conséquence, on en

cherche la cause dans le sous-ensemble correspondant qui est beaucoup plus petit (10 au lieu de 1000).

Le processus peut être schématisé comme suit : on intègre par paquet de 10 de façon à ce que la fiabilité de l'intégrat ainsi obtenu soit égale à 0.99, soit :

$$Intégration\{P1,P2,\ldots,P10\} \xrightarrow[Coût]{} Intégrat(I1)$$

Le coût d'intégration correspond à l'écriture et à la vérification des scénarios pour obtenir une fiabilité de 0,99 ; c'est un critère d'arrêt d'intégration.

En prenant les valeurs extrêmes de statistique généralement admises[11] (2 à 5 heures versus 50 heures), il n'est pas absurde d'estimer le différentiel de coût de correction des défaillances dans S1 par rapport à S2 de l'ordre d'un facteur 10.

En résumé : il y a beaucoup plus d'interventions de l'équipe d'intégration avec la stratégie S1 que avec la stratégie S2 (≈200), et ces interventions sont statistiquement beaucoup plus coûteuses (≈10).

La stratégie d'intégration est une conséquence du processus de conception. L'absence de ce livrable fondamental est un risque très élevé de non respect des caractéristiques CQFD et une quasi certitude d'un manque de maturité de l'équipe projet, qui aura d'autres conséquences.

Même si nos hypothèses sont volontairement simplificatrices pour la commodité des calculs, on voit qu'une architecture bâclée ou naïve (en général, personne n'a le désir de nuire, mais avec la complexité, le bon sens est souvent pris en défaut) aura des conséquences catastrophiques au moment de l'intégration, ce qui se traduira par deux constats, aux symptômes cependant bien visibles si l'on observe correctement le processus de développement au moyen du système qualité :

1. Allongement considérable de la durée de l'intégration (et en conséquence de son coût), car il est très difficile d'organiser le parallélisme des tâches d'intégration.
2. Défaut de qualité flagrant (performance, fiabilité, sûreté de fonctionnement, … entres autres) dès les premières installations, auquel il faudra rajouter, plus tard, de grandes difficultés d'évolution et maintenance.

Une économie sur l'effort de conception se paye toujours très chère en intégration et/ou en qualité.

On voit également l'importance de ce qui est fait en matière d'automatisation de la non régression, car la réduction du coût correspondant est d'un ordre de grandeur, au moins.

Architecture et programmation

Comme indiqué précédemment, l'un des rôles de l'architecte est de préparer le travail des programmeurs. Le référentiel d'architecture, quel que soit sa forme et son degré de formalisation, doit leur permettre de déterminer la nature des fonctions qui sont à programmer ainsi que la nature des contraintes à prendre en compte pour savoir comment les programmer, conformément aux exigences comportementales auxquelles doit satisfaire le système complet.

[11] Voir en particulier R.Grady, *Practical software metrics for project management and process improvement*, Hewlett-Packard professional book, Prentice Hall.

En suivant la terminologie de la norme ISO/CEI 9126, *Caractéristiques qualité des produits logiciels*, chaque sous-système/application, chaque module/paquetage, chaque composant/objet et chaque pièce/méthode sera déclinée en : Functionnality, Usability, Reliability, Performance (i.e. efficiency), Maintenability (i.e. serviceability), Portability (i.e. evolutivity) qu'il est commode de résumer par le sigle FURPSE ; la partie URPSE constitue les caractéristiques non-fonctionnelles. Ceci veut dire que dans le code source correspondant, nous allons trouver le code fonctionnel nominal qui assure la fonction auquel sera associé du code, en plus ou moins gros volume, pour assurer chacune des caractéristiques non-fonctionnelles. L'ensemble est généralement enchevétré, ce qui fait qu'il est très malaisé de dire si telle instruction fait partie de telle ou telle caractéristique. Une même instruction peut satisfaire simultanément plusieurs caractéristiques.

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.

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

1. ***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 fondamental du langage COBOL, à l'origine. Une telle programmation peut être validée par une simple comparaison aux textes définissant les règles métiers.
2. ***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 l'architecte, la question est donc de savoir, et de pouvoir, construire des blocs de code de caractéristiques homogènes minimisant la composante réactive, et surtout de donner des règles et des guide de programmation permettant aux programmeurs d'agir conformément à la logique de construction du système. Quiconque a pratiqué la programmation « en grand » sait que cela est un exercice non trivial qui nécessite une compréhension profonde de la sémantique du programme par rapport au contexte système dans lequel le programme opère.

La composante réactive est, in fine, formée des instructions de contrôle du flot :

> `IF … THEN … ELSE …` , `GOTO`, `CALL` *`fonction`*, `EXIT`, les boucles, la communication inter-processus, les pilotes d'événements, etc. Mais l'inverse n'est pas nécessairement vrai.

L'instruction IF peut servir à construire des assignations conditionnelles que l'on aimerait pouvoir écrire comme :

> *`a`* `= IF` *`cond`* `THEN` *`b`* `ELSE` *`c`*

pour dire que *a* vaut *b* ou *c* selon la condition. La même remarque vaut pour une table de décision.

Par ailleurs, l'automate de contrôle associé à la composante réactive se représente très bien par un tableau correspondant au graphe de l'automate, donc sous forme de données (par exemple une grammaire).

La simple considération des instructions est insuffisante, et de trop bas niveau, pour travailler sur la structure de contrôle, en particulier lors des tests. De plus, la structure du flot de contrôle peut elle-même être hiérarchisée, mais la hiérarchie correspondante n'est pas immédiatement évidente ; c'est une abstraction qui est à construire. Dans un langage, par exemple, on distingue depuis toujours le niveau lexical du niveau syntaxique, mais dans la réalité la frontière n'est pas toujours aussi nette : il y a des abréviations, des onomatopées, des locutions formant blocs, des métaphores, etc. Cela sera encore plus vrai dans le cas des workflows associés à un métier particulier.

Règle d'architecture de la composante réactive : La composante réactive est elle même une structure que l'on peut organiser en hiérarchie. 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) et l'optimisation de l'effort de VVT.

Si l'on prend comme exemple de programme un traducteur, comme l'approche MDA en présuppose de nombreux et de toute nature, à toutes les étapes de la transformation des modèles, on sait, pour de tels programmes, séparer la composante réactive de la composante transformationnelle (si tant est que l'architecte ait été formé correctement en techniques de compilation). La taille de la composante réactive est de l'ordre de 10 à

15%, voire moins en utilisant des automates de reconnaissance. La composante transformationnelle peut être très algorithmique (cas des compilateurs) où relativement simple (cas d'un générateur d'applications, où il s'agit généralement de transcription, sans optimisation, ni mémorisation de ce qui a déjà été traduit, i.e. pas de gestion de cache).

NB : Il est remarquable de noter que le pattern MVC dont nous avons fait le modèle de la machine informationnelle, est parfaitement conforme au pattern général des traducteurs.

La structure des coûts d'un tel programme est :

Coût total = Coût de la composante réactive + Coût de la transformation

Soit, pour 100 KLS : 57hm + 270hm = 327hm.

Si l'architecte (et les programmeurs) ne connaissent pas les techniques de compilation, ils peuvent cependant réaliser un traducteur dans lequel la composante réactive sera enchevêtré avec la composante transformationnelle, ce qui fait que tout le code doit être assimilé à du réactif ; la proportion s'inverse :

Soit, pour 100 KLS : 797hm + 27hm = 824hm.

L'amplification du coût est de 2,5 ; et la durée du développement sera en conséquence !

De plus, le programme sera très difficile à maintenir et à faire évoluer. Il sera également le reflet des idiosyncrasies de son concepteur, et de ce fait très difficile à transmettre à des tiers ayant une tournure d'esprit différente. Le système contient une bombe à retardement du point de vue du coût total d'acquisition (TCO).

Dans le cas de l'architecte expert en technique de compilation, on peut raisonnablement supposer que s'il manie parfaitement le concept d'automate, il a su également décomposer la partie transformationnelle en actions indépendantes (c'est une pure approche machine abstraite), ce qui fait que les 90 KLS de transformation peuvent être assimilée à 90 actions indépendantes de 1 KLS chaque (pour la simplicité du calcul), ce qui ramène le coût de la transformation à 90×2,4 = 216, soit un gain de 54hm. Enfin, *last but not least*, on peut également penser qu'une telle organisation du code facilite l'identification d'actions communes que l'on pourra factoriser, et donc l'émergence de nouvelles abstractions ; c'est ce que B.Boehm, dans le modèle COCOMO II, appelle le facteur d'échelle. Les 90 KLS initialement estimées ne sont peut-être que 70, d'où le coût final de la transformation : 168hm.

L'effet d'une bonne architecture de programmation est donc pour ce cas d'école, au final, un coût instantané de 225hm, à comparer au 824hm (soit un facteur 3,7), sans parler du coût récurrent de maintenance et d'évolution.

Le même raisonnement vaut pour la séparation du code transformationnel en transcriptions simples et cas particuliers, et en algorithmes. Un algorithme doit toujours être encapsulé de façon à pouvoir être remplacer, le moment venu, par un nouvel algorithme satisfaisant mieux les exigences non-fonctionnelles ; seule l'interface d'appel doit rester invariante.

Conclusion : le travail d'organisation du système en sous-systèmes et modules autonomes (i.e. leur sphère de contrôle est explicite) pour contrôler le coût d'intégration système et la fiabilité de l'assemblage ainsi obtenu, doit être accompagné d'une

réflexion approfondie sur la nature du code associé aux modules et composants correspondants.

Les différentes composantes réactives doivent être explicitées en fonction de la nature des interactions qu'elles régissent : services ou ressources de la plate-forme. Une ressource de mémoire persistante ou de mémoire non-persistante n'a pas les mêmes caractéristiques non-fonctionnelles : il faut donc les distinguer. Un service transactionnel centralisé (mettant en œuvre des transactions courtes, au sens base de données) ou distribué (mettant en œuvre des transactions longues, dans une logique métier) ne peut pas gérer les défaillances selon une logique uniforme ; les procédures de sauvegardes et de reprises seront nécessairement différentes. Un service métier défaillant peut être réparé en interrogeant l'opérateur humain ; ceci est impossible pour un service gérant des ressources matérielles. Les paramétrages seront également très différents, selon que l'on est du coté PIM, ou du coté PSM.

L'approche par machines abstraites permet non seulement d'organiser le systèmes en hiérarchie de machines coopérantes, mais également d'organiser le code en blocs de complexité homogène, évitant ainsi de transformer des pans entiers de code transformationnel en code réactif. Elle permet également de gérer au mieux la ressource programmeur en confiant la réalisation des composantes réactives et des composantes transformationnelles complexes aux plus expérimentés d'entre eux.

Pour reprendre la notion d'invariant architectural ébauché ci-dessus, l'organisation du système en hiérarchie de machines abstraites revient à décomposer l'invariant global en une hiérarchie d'invariants dont l'architecte peut contrôler le niveau d'autonomie. Modifier un invariant ne rejaillit pas nécessairement sur les autres. Si cette propriété est vraie (cela dépend bien sûr des capacités de l'architecte), alors l'instabilité est localisée sur une des machines abstraites, soit sur la composante réactive, soit sur telle ou telle composante de la transformation opérée par la machine, éventuellement sur les deux. L'architecture des microprocesseurs ou des systèmes d'exploitation offre de nombreux exemples d'architecture de ce type où l'on n'est pas obligé de tout refaire chaque fois que l'on modifie substantiellement les fonctionnalités. Il n'y a donc pas de fatalité à l'instabilité architecturale, pour autant que l'architecte ait mis en place les bonnes structures et se soit soucié de la façon dont celles-ci sont prises en compte dans la programmation du système. Même s'il ne programme pas lui-même, il doit s'assurer de la qualité du travail de programmation, par exemple en s'intéressant de très près à l'effort de test et à la qualité des tests qui en constituent un excellent révélateur. Indirectement, cela justifie pleinement l'importance donnée aux tests par les méthodes agiles et l'eXtreme Programming.