Livre Blanc CESAMES

⌘⌘⌘

Intégration et complexité des systèmes numériques

Table des matières

Figures

Bibliographie sommaire

[1] Y.Caseau, *Urbanisation, SOA et BPM*, Dunod, 2011, 4ème édition.
[2] B.Mesdon, *Les Points de Fonctions en Ingénierie Logicielle*, Hermès 2012.
[3] J.Printz, *Architecture logicielle*, Dunod, 2012, 3ème édition.
[4] J.Printz, N.Trèves, B.Mesdon, *Estimation des projets de l'entreprise numérique*, Hermès, 2012 [Parution fin 2012].
[5] J.Printz, B.Mesdon, *Ecosystème des projets informatiques*, Hermès 2006.

Remerciements

CESAMES remercie très sincèrement les participants et les conférenciers du Groupe de Travail *Intégration & Complexité*, et leurs sociétés d'appartenance, pour la qualité de leurs interventions, pour leurs remarques toujours pertinentes, pour leurs encouragements, sans lesquels la rédaction de ce livre blanc n'aurait tout simplement pas été possible.

Synthèse du document

Intégrer l'information dans l'entreprise numérique, c'est **mettre en cohérence trois ensembles de trajectoires évolutives**, bien distinctes mais étroitement corrélées :

- Les **processus métiers** et/ou **industriels** qui créent la valeur de l'entreprise, compte tenu des missions que l'entreprise s'est assignée et des nouveaux usages rendus possible par l'innovation technologique. De fait, une **co-évolution** des métiers, de la technologie et de l'ingénierie.
- Les **programmes**, i.e. le support informatisé, numérisé, des règles et des comportements de tout ou partie de ces processus qui représentent aujourd'hui des dizaines de millions de lignes de code et des milliards d'enregistrements dans les bases de données.
- La **technologie numérique**, i.e. les plates-formes et infrastructures TIC, aujourd'hui extraordinairement diversifiée, ubiquitaire, mobile, microscopique mais puissante, ... tant au niveau des capacités de traitement que du stockage de l'information.

Chacune de ces trajectoires a sa dynamique évolutive et sa complexité propre. Chacune a ses modalités organisationnelles mais toutes ont en commun la contrainte de se transformer et de s'adapter en permanence. Le **PROJET** est le moteur de cette co-évolution vitale pour l'entreprise.

Ce qui fait la complexité du numérique est la nécessaire **cohérence** de ces trois trajectoires prises individuellement, mais surtout de rendre leurs combinaisons deux à deux, et trois à trois également cohérentes, ce qui est beaucoup plus difficile compte tenu de la taille des patrimoines applicatifs des entreprises. C'est la recherche de l'équilibre et du bon compromis entre ces trois trajectoires, de l'organisation du patrimoine applicatif, qui donnera un avantage compétitif à l'entreprise qui saura comment organiser toutes ces interactions.

➔ La clé de la découverte de ce nouvel équilibre est l'architecture système, de tous les systèmes, industriels et/ou tertiaires

Les architectes des systèmes numériques sont confrontés à deux grands types de complexité :

- Une **complexité statique** appréhendée de longue date par les mécanismes de hiérarchisation qui sont au cœur de la technologie informatique depuis son origine. Hiérarchiser, classer, rechercher, créer/effacer, restructurer, ... sont les opérations au centre des technologies de management de l'information.
- Une **complexité dynamique** liée aux évolutions des environnements organisationnels et humains qui demandent toujours plus d'interactivité, aux évolutions du système lui-même qui se précise et s'adapte durant la réalisation, et aux évolutions de la technologie qui aujourd'hui permettent un asynchronisme massif des traitements, complexité qui se matérialise par des flux de transactions qui se chiffrent en dizaines de milliers par seconde. Les *Big Data* et le *Cloud Computing* sont l'image marketing de cette évolution irréversible qui exige fiabilité et performance, sûreté de fonctionnement, au niveau global système.

C'est l'organisation de cette complexité dynamique, pour des systèmes toujours plus sûrs, qui est aujourd'hui l'enjeu fondamental, le clivage décisif entre les entreprises qui développeront la capacité à passer l'obstacle de cette complexité en se formant sérieusement, et celles qui trébucheront. Le mot d'ordre proclamé par tous les acteurs du changement est **AGILITE**, mais la difficulté de l'agilité est dans le **COMMENT** la mettre en œuvre concrètement.

Rendre les trajectoires cohérentes et sûres, améliorer le contrat de service, c'est réaliser des couplages négociés entre tous les acteurs de ces trajectoires par les **INTERFACES** contractualisées. Les interfaces sont les éléments stables du système qui permettent l'échange et la transformation de l'information au niveau de finesse requis par l'interactivité. Plus d'interactivité, plus d'agilité = Découpage plus fin des fonctions = Plus d'éléments en interaction = INTEGRATION PLUS DIFFICILE, donc plus coûteuse.

Le but de ce Livre Blanc est de bien **poser le problème de l'intégration** qui est le socle fondamental sur lequel l'entreprise numérique pourra développer à la bonne vitesse les potentialités que l'innovation technologique lui met à portée de main.

➔ L'architecture de l'information est la condition nécessaire de l'agilité et de l'intégration réussie

Figure 1 : Bâtir toujours plus haut

Avertissement et terminologie

Nous utilisons le terme « numérique » pour caractériser l'état présent et futur des systèmes informatisés qui gouvernent le fonctionnement de la société et de ses grands équipements (énergie, télécommunications, transport, défense et sécurité, ...), tant au niveau des entreprises et des administrations, que des particuliers, où l'ordinateur est devenu un artefact quasi microscopique, « ça ne se voit plus », mais omniprésent, dont on ne peut plus se passer. Le meilleur indice de cette « numérisation » massive est la quantité de logiciel que chacun de nous utilise quotidiennement comme par exemple utiliser son IPad, rechercher un itinéraire avec Google Maps ou son GPS, ou simplement allumer la lumière, ou encore donner un bon coup de frein, ce qui active le dispositif ABS de sa voiture. Rappelons tout de même que le terme « informatique » forgé par la fusion de information et automatique dans les années 60, garde tout son sens, car il s'agit bien d'automatiser le traitement de l'information. Sans ces automatismes, il n'y aurait ni *Cloud Computing*, ni *Autonomic Computing*, ni *Big Data*, ni téléphone mobile, … pour utiliser des termes à la mode, car les flux associés dépassent désormais de très loin les capacités humaines.

Au sortir du dernier conflit mondial, cet indice logiciel était strictement égal à zéro ! Aujourd'hui, derrière nos gestes les plus élémentaires se cachent des dizaines de millions de lignes de programmes, fruit du travail et de l'intelligence de millions de programmeurs, logiciels qu'il faut concevoir, développer, valider, entretenir, et retirer proprement du service lorsqu'ils deviennent obsolètes, car on ne les trouve pas à l'état naturel comme les matières premières. Les ordinateurs et les équipements informatisés nomades qui permettent l'interaction en temps réel de millions d'usagers vont mettre en exécution ces logiciels pour rendre à l'usager le service attendu, avec le niveau de qualité requis par ces usagers qui ignorent tout de la complexité sous-jacente. La puissance de traitement et de stockage de l'information est fournie par des infrastructures regroupant des centaines de milliers de serveurs qui rendent accessibles à tous le « *Software as a Service* », [SaaS], un peu comme l'électricité ou le téléphone, par un simple branchement à cette nouvelle source d'énergie. Simplicité des usages, complexité des infrastructures et des plates-formes, complexité de la programmation des équipements et des interfaces, tel est le dilemme de l'intégration pour l'entreprise numérique.

NB : Dans le corps du texte nous seront amenés à parler de taille de programmes, exprimée depuis les années 70-80 en nombre d'instructions source réellement écrites par les programmeurs en COBOL, C, C++, Java, Python, ... [longue liste], à l'exclusion de celles fabriquées par des outils, conformément aux règles de comptage standardisées en vigueur ; *NB : 1.000 lignes de scripts peuvent activer un progiciel de grande taille, mais seules les 1.000 lignes ont été écrites*. Pour rendre concrète cette notion de ligne source [LS, ou KLS pour 1.000 LS] il est commode de les représenter au format imprimeur, i.e. en nombre de pages (en moyenne, dans l'édition courante, 50 lignes de texte par page), soit :

- Pour 1.000 LS, i.e. un texte de 20 pages, qui nécessite un certain effort de programmation[1]. Une erreur dans un tel texte, c'est une erreur sur 1.000 LS, déjà considéré comme une bonne performance : c'est une unité de compréhension qui respecte les règles ergonomiques humaines, en particulier la capacité de mémorisation du programmeur, indispensable à la qualité de son travail.
- Pour 10.000 LS, c'est un livre moyen de 200 pages.
- Pour 100.000 LS, c'est 2.000 pages, soit cinq « gros » livres de 400 pages, une taille d'ouvrage considérée comme une limite ergonomique, à la fois pour l'auteur et pour le lecteur ; difficile à mémoriser, y compris pour l'auteur qui doit avoir « en tête » tout le contexte. C'est typiquement le fruit du travail d'une équipe projet agile.

Il est bon d'avoir ces chiffres à l'esprit pour comprendre la nature du travail des programmeurs qui par certain coté est analogue à un travail d'écrivain et de traducteur, voire de mathématicien, mais surtout celui d'un créateur et architecte de modèles.

➔ Pour aller plus loin, voir les réf. bibliographiques [1], [3] et [4].

[1] Deux à trois mois calendaires est une statistique moyenne observée sur des milliers de programmeurs. Le code est documenté, validé et maintenable, conformément au contrat de service demandé par les usagers.

Le puzzle de l'intégration

Le vrai sens de l'intégration

La plupart des entreprises disposent de cartographies applicatives qui décrivent en grandes masses le contenu de leur patrimoine logiciel. Quelques unes d'entre elles disposent également de chiffrages de ce patrimoine, soit en termes de Ligne de code Source [LS], soit en termes de Points de Fonctions [PF, une mesure fonctionnelle associée aux données], unités utilisées pour estimer les projets informatiques depuis les années 80s (voir les réf. bibliographiques [2] et [4]). Dans ce LB nous utiliserons de préférence les lignes de code source, unité de comptage plus concrète, et mieux comprise par quiconque a touché à un ordinateur et vu un bout de programme ou un texte HTML/XML dans ses messages Internet.
Ces cartographies sont basées, soit sur l'architecture fonctionnelle des systèmes informatisés, soit sur les processus métiers et/ou les équipements qui les hébergent, soit un mélange des deux. Elles sont représentées à l'aide de diagrammes que l'on trouve dans les outillages qui supportent les méthodologies d'ingénierie, comme les langages UML, SysML, BPML, ... pour les plus courants. Le résultat de ce qu'on appelle communément URBANISATION est une cartographie du système qui a été urbanisé.

Il est important pour l'entreprise de connaître la taille du patrimoine logiciel qui lui permet de fonctionner et de se développer, exprimé en lignes de code source [LS]. Et plus encore de bien connaître celles et ceux qui entretiennent ce stock dont la valeur n'est pas vraiment comptabilisée : les programmeurs et les architectes. Aux Etats-Unis, programmeur est un métier noble, correctement valorisé.
Cependant, peu d'entreprises ont une idée précise de la taille et de la valeur de leur patrimoine applicatif.

- Pour une grande banque, dont l'informatisation démarre en même temps que l'informatique, disons dès les années 60, le patrimoine programmatique peut monter jusqu'à 300 millions de lignes sources, majoritairement encore en COBOL.
- Pour des ministères régaliens comme le MinEFi ou le MinDef, informatisés de longue date, on est dans les 150/200 millions de lignes source.
- Pour des entreprises de création récentes comme celles qui gèrent : a) les diverses réservations de billets d'avions, de trains, d'hôtels, etc. dont on a besoin dans la vie courante, ou b) les clients des opérateurs de téléphonie mobile, ou c) ou le transport de l'énergie, … on est dans les 100/150 millions de LS.

Il est commode, pour bien appréhender ce qu'une telle quantité d'information signifie à l'échelle humaine, de se la représenter aux normes de l'édition qui nous sont familières depuis quelques siècles : le livre. Un livre de 400 pages, à raison de 50 lignes par page, c'est l'équivalent d'un programme de 20.000 lignes sources.

➘ Un patrimoine de 100 millions de lignes source c'est l'équivalent d'une bibliothèque de 5.000 volumes.

Mais pour être complet, il faut ajouter à cette bibliothèque, le mode d'emploi, i.e. les référentiels système, les aides en lignes [souvent intégrées au texte même du programme], et surtout les scénarios de tests qui permettent de déclarer la bibliothèque « bonne pour le service » aux usagers. Ce qui reviendrait à doubler sa taille !

➘ Un patrimoine de 100 millions de lignes source, c'est un coût de développement incompressible d'environ 25.000 ha, mais beaucoup plus en coût réel si on introduit les facteurs de réussite bien connus (moins de 30% des projets), soit trois fois plus.

NB : Les analyses statistiques faites sur des milliers d'entreprises donnent comme productivité moyenne des programmeurs le chiffre de 4.000 lignes, à ±10%, de programmes

réellement écrites (et non fabriquées par des outils) par programmeur et par année calendaire « normale », i.e. 220 jours ouvrés de 8h[2]. Trois à cinq fois plus pour des programmeurs chargés de la maintenance selon le taux moyen de modifications à effectuer.

Les programmes sont la composante « intelligente » qui permettent de manipuler les données de l'entreprise grâce aux opérations basiques : CREER une donnée, RECHERCHER/LIRE une donnée, TRANSFORMER/MODIFIER une donnée pour la mettre à jour, ou encore EFFACER une donnée qui n'a plus d'intérêt [résumé par l'acronyme CRUD : CREATE, RETREIVE, UPDATE, DELETE]. Les programmes permettent également l'échange d'information via les messages et événements EMIS/REÇUS, par d'autres programmes et/ou l'opérateur du système [opérations SEND / RECEIVE]. Les données de l'entreprise, encore plus que les programmes, sont la vraie richesse de l'entreprise. Une immobilisation au sens comptable du terme, plus précieuse qu'un stock.

On peut également se représenter les fichiers et les bases de données avec les normes de l'édition. On parlera alors de lignes de données [LD]. Un grand fichier comme celui des contribuables, des clients d'EDF, des assurés sociaux, c'est au bas mot 25/30 millions de références clients. Si l'on compte une page d'information par référence, soit environ 2.500/3.000 caractères, on atteint facilement les 100 milliards de caractères. Si on édite ces fichiers sous forme de tableaux de type EXCEL avec un taux de remplissage de 50% par ligne compte tenue des blancs et des règles visuelles d'alignement, nous arrivons [en prenant 50 caractères par ligne] à 2 milliards de lignes de données soit 40 millions de pages, ou encore une bibliothèque de 100.000 livres.

➘ Un grand fichier de 25 millions de références, c'est l'équivalent d'une bibliothèque de 100.000 volumes.

Des fichiers de ce type, il y en a quelques centaines dans un pays comme le nôtre. Une grande administration comme la CNAM-TS gère plus de 1 milliard de bordereaux de remboursements par an, pris en charge par la sécurité sociale et/ou les mutuelles pour ceux des français qui en bénéficient. A supposer qu'il n'y ait que 250 caractères par bordereau, soit 10 fois moins que précédemment, on aboutirait cette fois à une bibliothèque de 400.000 volumes, en une seule année !
➔ Si on s'intéresse à la qualité de ces données et aux erreurs éventuelles qu'elles contiennent suite à des manipulations erronées, on est alors pris de vertige, et il faut bien s'accrocher à la rampe … nous ne ferons pas le calcul ici. Mais c'est une raison supplémentaire pour connaître ce patrimoine en détail.

Pour revenir à la cartographie, on peut donc séparer la partie PROGRAMMES, i.e. les procédures codées dans tel ou tel langage, et la partie DONNEES, i.e. l'information transformée et exploitée par les programmes. Du point de vue du comptage, l'entreprise numérique doit comptabiliser séparément la partie PROCEDURALE du patrimoine, i.e. les programmes proprement dits déduits des règles de transformation et des contraintes à respecter, et la partie DONNEES, i.e. les fichiers et bases de données, conformément à la logique de comptage des modèles d'estimation aujourd'hui largement utilisés en management de projet (Voir réf. [4]).

[2] Pour des comparaisons internationales, il faut raisonner en *heures ouvrées* réelles.

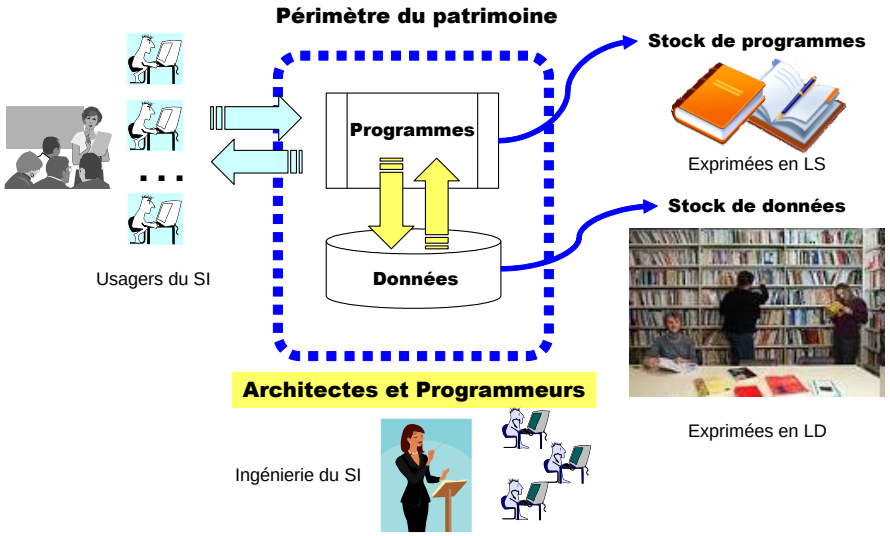

Figure 2 : Données et Programmes

C'est une première cartographie, certes pas très utile, mais qui montre bien la nature des liens que cette cartographie va entretenir avec son environnement.
Là où les choses deviennent vraiment intéressante est lorsqu'il va s'agir d'organiser cette « masse » par rapport aux organisations qui l'utilisent et qui la gèrent. Soit :

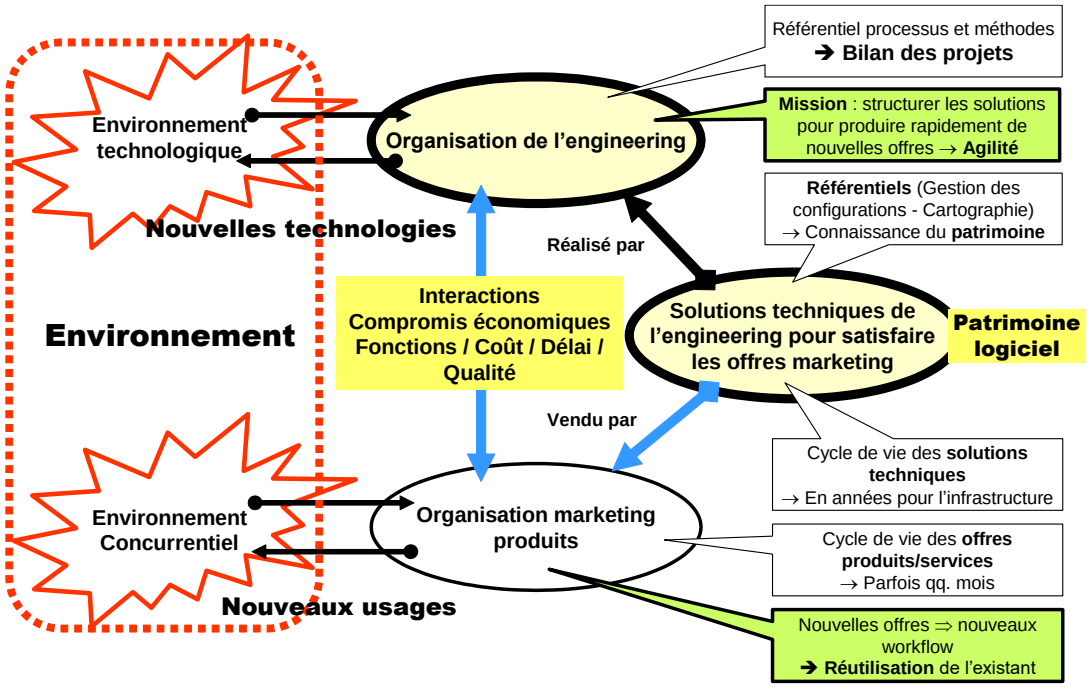

Figure 3 : Interaction des organisations – Ecosystème

Selon que l'on prend le point de vue de l'engineering, du marketing (i.e. des usagers), ou des technologies utilisées, on obtient des cartographies très différentes. Pour ce qui concerne l'intégration, c'est la relation Engineering/Patrimoine logiciel qu'il faut privilégier, mais en n'oubliant jamais qu'elle est au service des usagers. C'est l'engineering qui intègre les solutions, mais ce sont les clients usagers qui découvrent les erreurs résiduelles qui ont échappées à la vigilance de l'engineering.

Du point de vue de l'engineering, le patrimoine logiciel se présente comme un gigantesque puzzle, dont les « pièces » élémentaires sont les blocs de code source et des blocs de données associés à ces blocs de code.

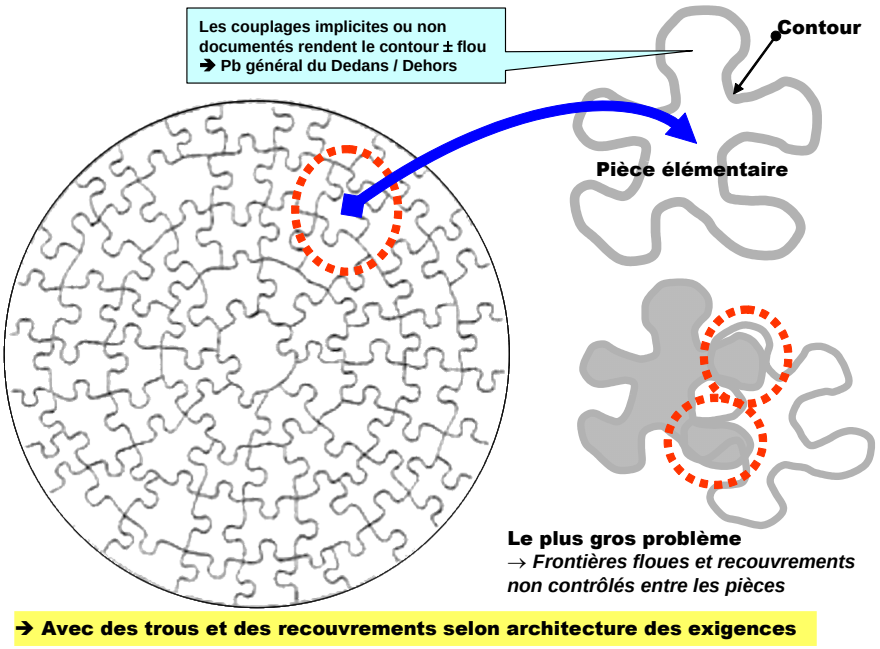

Figure 4 : Le puzzle de l'intégration

A ce stade on peut comprendre que tout le problème de l'intégration, et donc de l'architecture, va être de contrôler le nombre et la nature des relations que les pièces entretiennent entre elles. Mais avant, il nous faut regarder du coté de ceux qui sont en charge de l'ingénierie de ces pièces.

Unité de sens/compréhension – Performance des programmeurs

Une ligne de code source, i.e. une instruction, n'a généralement aucun sens pour le programmeur. Ce qui compte est l'agrégation des lignes de code en blocs sémantiques, l'équivalent de phrases, qui traduisent une action effectuée dans la réalité des métiers et/ou des équipements gérés par le programme. Des études d'ergonomie ont montré que ce qu'un programmeur correctement formé peut raisonnablement manipuler en terme de quantité d'information dépend de la capacité de mémorisation et d'organisation qu'il a su développer. On en connaît les limites statistiques :

- Notre champ visuel nous permet d'appréhender sans trop de difficulté deux pages de textes, i.e. un livre ouvert, soit une centaine de lignes, avec des conventions typographiques dont nous sommes familiers depuis des siècles.
- Pour des textes plus longs, il faut une structuration qui prend en compte le nombre de concepts qu'un cerveau correctement éduqué/instruit [le « savoir être » et le « savoir faire », i.e. la compétence] est capable de mémoriser et de manipuler pour agir de façon pertinente. Ce nombre, parfois qualifié de « magique », vaut 7±2 ; il est connu de tous les ergonomes[3]. Appliqué à nos deux pages, avec une table de matière commentée nous arrivons à une unité de sens programmeur de l'ordre de 20 pages de texte, soit environ 1.000 lignes source/instructions [quel que soit le langage].

[3] G.A.Miller, *The magical number seven plus or minus two: some limits on our capacity for processing information*, Psychological review, Vol. 63, N°2, March 1956.

➔ C'est la raison pour laquelle l'unité de compte du modèle d'estimation COCOMO est précisément 1.000 LS, ce qui représente un effort de développement approximatif moyen de 2,5 hm, soit 10 semaines calendaires [grandeur statistique].

➘ Un bloc de code d'environ 1.000 LS est ce qu'un programmeur correctement formé est en moyenne capable d'appréhender de façon sûre et durable : c'est une **unité de compréhension programmeur** [UCP], un quantum d'information qui définit le « **grain** » **sémantique** ultime de son travail de modélisation.
→ Tout dépassement de ce seuil doit être géré comme un risque.

NB : En programmation objet, avec des langages comme Java, C#, Python, ... c'est une classe de 8-10 méthodes, chacune comptant en moyenne 100-125 LS, soit deux pages de texte, i.e. une entité qui a du sens pour le programmeur.

➔ Quelques données ergonomiques qui structurent l'activité des programmeurs :

- Une lecture attentive de 20 pages de programme, c'est 2-3 heures de concentration intense, sans distraction ni interruption.
- On peut estimer qu'une reprise de ce code par le programmeur qui l'a conçu et développé, après 1 an, c'est environ 10% de l'effort initial, soit 5 jours de travail. Un bon programmeur gardera en mémoire la structure générale du programme, mais oubliera les détails qu'il reconstruira cependant sans difficulté.
- Le même travail de reprise, par un tiers, sans connaissance préalable, c'est 20-30%, soit 2 à 3 semaines d'appropriation, si la documentation est bien faite, pour effectuer des modifications.
- Une productivité de 20.000 LS par an est une performance atteinte par moins de 10% des programmeurs [c'est 5 fois la moyenne statistique de 4.000 LS à ±10%]. Si le programmeur qui annonce de tels taux n'est pas classé comme excellent, il est probable que plus de la moitié du travail a été oublié ou omis : programmation peu rigoureuse sans respect des règles de bonnes pratiques, absence de documentation, tests incomplets, etc. ... d'où la notion d'**age** « **moyen** » en **Compétence/Expérience** de l'organisation de l'engineering [sous-traitants inclus] qui est une moyenne pondérée, i.e. un barycentre, de l'expérience de chacun.

➔ On comprend pourquoi un turnover excessif dans une organisation d'ingénierie peut vite devenir un handicap insurmontable. Un turnover de 20-30% rend impossible l'atteinte des niveaux 4-5 sur l'échelle CMMI. Les méthodes agiles insistent sur la régularité de l'effort. Les meilleurs programmeurs sont des marathoniens, pas des sprinters !!! Google accorde une journée par semaine de travail libre à ses programmeurs pour les fidéliser.

Nombre de pièces à intégrer – Combinatoires

Le nombre de pièces du puzzle dépend de la taille des pièces élémentaires que sont capables de fabriquer les programmeurs dans une organisation de l'ingénierie (Voir réf. [5]).
A titre d'exemple dans une équipe projet agile, i.e. 8-10 personnes maximum, qui fonctionne sur des durées de 12-18 mois, on raisonne en binômes programmeurs testeurs.

- Un binôme va pouvoir gérer/produire, en moyenne, des blocs de code d'environ 20 KLS, soit une vingtaine de pièces « usinées » en 12-18 mois.
- Une équipe de 4-5 binômes va pouvoir gérer/produire, en moyenne, des blocs applicatifs d'environ 100 KLS, soit une centaine de pièces.

Pour faire le dénombrement du nombre de pièces d'un patrimoine, nous avons donc trois unités « naturelles » de comptages, faisant sens pour l'organisation de l'ingénierie :

- L'unité de compréhension programmeur **UCP** qui se réfère au programmeur individuel qui in fine écrira le texte du programme de la pièce, et le corrigera en cas de modification.
- L'unité de compréhension du binôme programmeur testeur **UCB**, environ vingt fois plus grosse que l'UCP. Dans les méthodes agiles il y a deux rôles importants interchangeables : celui qui programme et celui qui valide le programme, i.e. celui qui produit la preuve de bon fonctionnement : les tests.
- L'unité de compréhension équipe projet agile **UCE**, environ cent fois plus grosse que l'UCP, à charge de l'équipe de s'organiser, ou de s'auto-organiser comme le recommande certains auteurs, selon les compétences de chacun de ses membres, sous la responsabilité d'un programmeur senior.

En prenant des patrimoines : PETIT, 1 MLS, MOYEN, 10 MLS, et GROS, 100 MLS, on a un nombre de pièces basiques qui sont les nœuds du graphe correspondant du système informatisé de l'entreprise ; soit :

	Nombre de pièces à intégrer		
Unité de comptage	1 MLS	10 MLS	100 MLS
UCP	**1.000**	**10.000**	**100.000**
UCB	**50**	**500**	**5.000**
UCE	**10**	**100**	**1.000**

NB : Dans certains systèmes temps réel critiques, l'unité de compréhension est parfois l'instruction du langage de programmation, comme par exemple les instructions qui permettent la synchronisation des traitements et des événements.

Le paramètre le plus important, du point de vue de l'intégration, sont les relations que les pièces entretiennent les unes avec les autres. Un relation existe entre la pièce P1 et la pièce P2 si un couplage quelconque existe entre P1 et P2. Le couplage peut être fonctionnel, par exemple P1 et P2 ont des données communes, ou organique, P1 et P2 sont hébergées dans un même serveur, ce qui fait que si P1 bloque ou sature le serveur, P2 sera ipso facto bloquée également.

L'étude des matrices de couplages, appelées matrice N^2, est un aspect fondamental de l'ingénierie système (Voir réf. [3]), car le taux de remplissage de la matrice va donner une première indication sur la quantité de travail à effectuer sur le système pour valider l'intégration. Dans le cas le plus favorable, avec N pièces, on aura à valider $N-1$ relations de couplage ; les pièces constituent une simple chaîne, i.e. la diagonale de la matrice. Dans le cas le plus défavorable, quand la matrice est saturée, chaque pièce est couplée à toutes les autres, directement ou indirectement, on aura alors à valider $N(N-1)$ relations, avec l'hypothèse restrictive que lorsqu'une pièce est sollicitée plusieurs fois, elle réagit toujours de la même manière. Une pièce qui, compte tenue de sa programmation, ne se comporterait pas de la même façon à chacune des sollicitations, comptera comme autant de pièces différentes compte tenu des états possibles de la pièce en question, par exemple : état NOMINAL, tout marche bien, état NON NOMINAL, la pièce ne rend plus le service et le signale, une état intermédiaire avec MODALITE où une partie du service a été fait. Dans ce cas, il faut s'intéresser aux chemins possibles.

Une pièce qui occasionne des « fuites de mémoire » sur le serveur qui l'héberge va progressivement dégrader son contrat de service, premier problème, mais surtout occasionner des dysfonctionnements aléatoires dans les autres pièces qui partagent le même serveur. Un tel comportement est catastrophique dans le cas du *Cloud Computing*, où la pièce doit pouvoir être réallouée dynamiquement en fonction du niveau de saturation de l'infrastructure.

Dans un univers simple, comme celui des instructions machines, on peut raisonner VRAI FAUX, mais dès que la taille des pièces devient significative, dès le niveau individuel UCP avec ses 1.000 lignes/instructions source, la logique devient modale, en fonction du contexte.
La logique d'intégration, du point de vue de l'architecte, consiste donc à se poser systématiquement trois questions :

1. Quel est le bon nombre de pièces à considérer, de quelle taille ?
2. Quels sont les couplages que la pièce entretient avec les autres pièces, i.e. quelles sont les interfaces externes de la pièce [le contrat d'interface] ? Et ce, pour toutes les pièces.
3. Quel est le degré d'indépendance [isolation, confinement] de la pièce compte tenu des sollicitations ? Par défaut : être totalement indépendant, mais c'est une propriété qui doit être vérifiée à la construction [architecture] et validée, par exemple via les tests de performance et de fatigue, les tests de charge, etc.

Ebauche d'une typologie des pièces à intégrer

La nature des pièces à intégrer a une grande importance du point de vue de la combinatoire. Dans la figure 2, nous avons distingué la typologie la plus fondamentale, la typologie DONNEES / PROGRAMMES qui est le fondement du modèle de Von Neumann, la base de tous nos ordinateurs, ce qui correspond à la séparation Programmes / Bases de données dans tous les systèmes informatisés.
Du point de vue de la combinatoire c'est insuffisant, et il faut aller un cran plus loin dans la décomposition hiérarchique pour bien en mesurer l'importance. Les critères de classification indiquent l'importance que l'architecte attribue aux différentes entités qui constituent la classification. Dans la figure 5, on a fait apparaître les méta données et les méta programmes qui constituent une partie du référentiel système. Les informations correspondantes sont particulièrement importantes car elles doivent être partagées par tous les acteurs de l'ingénierie et de ceux des métiers qui sont en interactions avec l'engineering [voir figure 3]. On a également fait apparaître les messages et événements qui sont des interfaces fondamentales devant être contractualisées. On a aussi fait apparaître la distinction entre les programmes qui dépendent d'un équipement et/ou d'une plate-forme d'exécution et ceux qui dépendent des processus métiers car leur cycle de vie est totalement différent : on ne change pas les équipements tous les jours, alors qu'un nouveau besoin métier peut exiger une réponse de l'engineering dans un délai de quelques semaines. Cette dernière distinction est cruciale pour l'agilité du système.

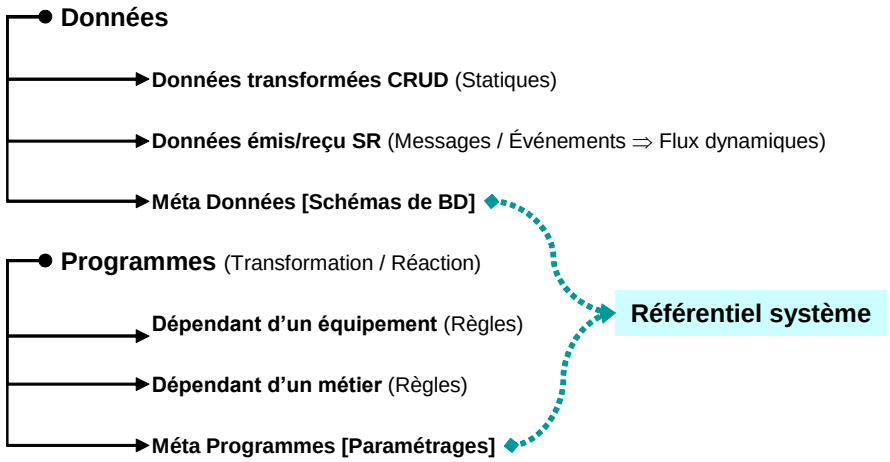

Figure 5 : Typologie

NB : sur l'arborescence, nous n'avons pas fait apparaître une typologie actuellement en vogue, celle des « services », pour des raisons de simplification. Un service, dans la mesure

où il est partagé est astreint à des règles strictes d'évolution/adaptation de même nature que celles des données partagées entre plusieurs programmes nécessitant un service d'accès.
➔ Dans tous les cas, c'est à l'architecte de définir ce qui est important pour le cycle de vie du système et de le faire figurer comme tel dans l'arborescence des entités constitutives du système. Ces entités sont des *types*[4] et doivent être rigoureusement documentées dans le référentiel du système.

Le problème de l'intégration

Toute adaptation/transformation significative du système va se traduire par la création de nouvelles pièces, la modification de pièces existantes en général en restant compatible [Principe de compatibilité ascendante] pour ne pas occasionner de régression auprès des usagers qui n'en aurait pas le besoin, la suppression de pièces obsolètes. Cet inventaire définit le périmètre de l'intégration. [Voir figures 4 et 6]. Les pièces de cette nomenclature seront appelés *Modules* ou *Building Blocks*, parfois traduit en *Intégrat.*
Les pièces dépendantes d'un équipement ou d'une plate-forme sont la partie émergée d'un « iceberg » d'interfaces système et/ou progiciel dont il convient de bien mesurer la hauteur [Voir figure 7].
C'est l'un des deux problèmes majeurs de l'intégration, le second étant la validation des propriétés transverses globales : interopérabilité, performance, disponibilité, sûreté de fonctionnement, facilité d'emploi, etc. pour ne parler que des plus importantes.
➔ Cette validation a été au cœur des problèmes débattus dans le groupe de travail I & C.

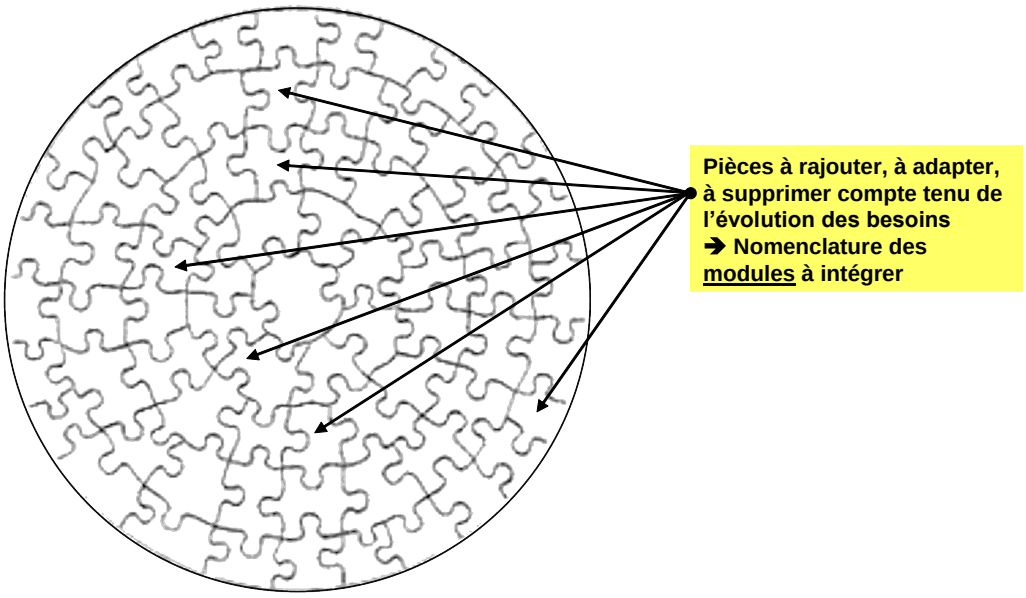

Figure 6 : Modifier le puzzle

Cet inventaire ayant été fait, on peut dresser la liste des modules externes qui sont en interactions directes [couplages], avec l'un quelconque des modules appartenant à l'inventaire.
➔ La gestion de configuration système est indispensable à ce stade.

[4] Au sens *Type abstrait de données*, [*Abstract data type* dans le jargon anglo-saxon] base de la programmation objet.

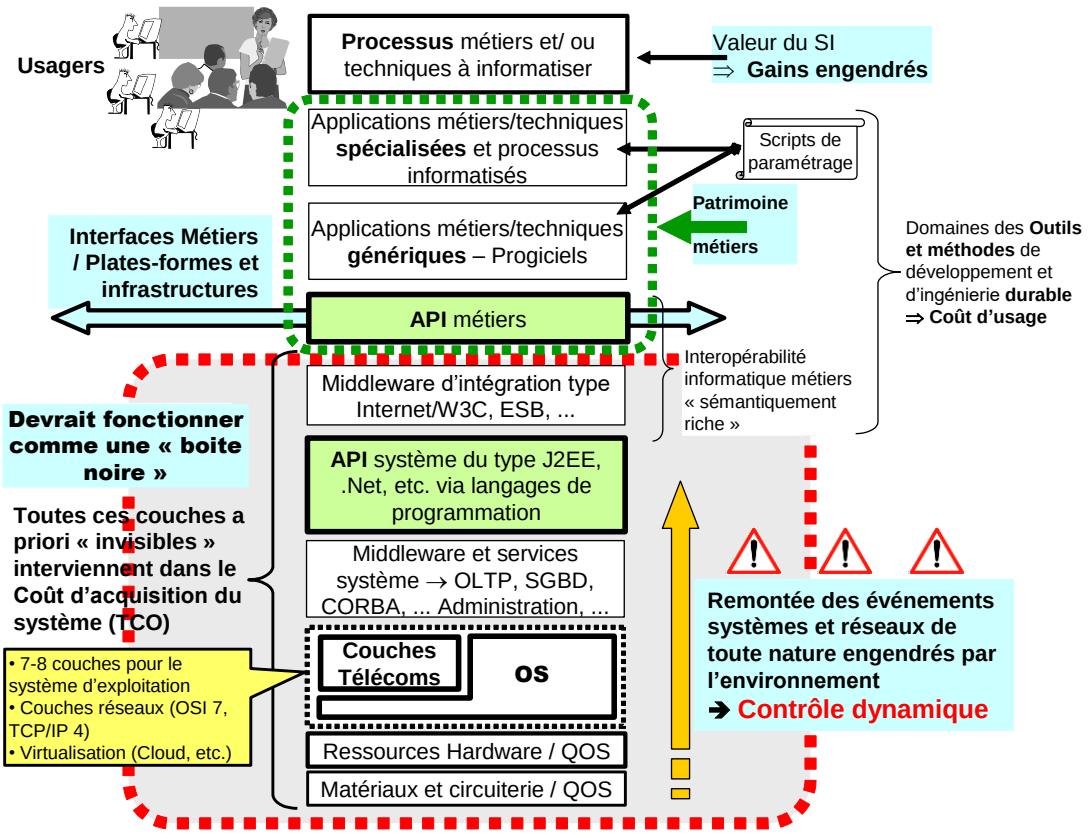

Figure 7 : Pile des interfaces systèmes

Le problème fondamental de la pile des interfaces systèmes est la remontée des événements en provenance des plates-formes et/ou des infrastructures, progiciels compris, qui doivent être récupérés par les API métier, car ils doivent être traduits en termes métiers pour pouvoir être exploités de façon cohérente. Ce contrôle ne peut être que dynamique (Cf. le concept d' *Autonomic computing*, introduit par IBM, et repris par tous : HP, Microsoft, Oracle, ...).

L'organisation de l'intégration

Dans la pratique, le processus d'intégration consiste à rassembler les nouveaux modules et les modules ayant fait l'objet d'adaptations dans un environnement spécifique, i.e. la plate-forme d'intégration, qui simule raisonnablement l'environnement d'exploitation réel. L'intégration ne se fait jamais directement dans l'environnement d'exploitation pour des raisons de sûreté de fonctionnement et de contrat de service évidentes.

La séquence qui conduit de la plate-forme d'intégration à la plate-forme de production, au minimum trois étapes, est représentée par le schéma de la figure 8.

En amont de la plate-forme d'intégration il y a la ou les plates-formes de développement qui ne sont pas représentées sur la figure, mais dont le responsable de l'intégration doit s'assurer d'une compatibilité raisonnable, en particulier pour ce qui concerne la pile des interfaces systèmes rappelée par la figure 7. C'est ce qui est parfois appelé à juste raison Usine Logicielle [la *Software Factory*] dont le rôle est de développer toutes les pièces logicielles constitutives des modules à intégrer de façon standardisée.

En entrée du processus d'intégration, le responsable dispose de la liste des pièces/modules à intégrer et de la place de ces modules dans l'architecture du système, tant au niveau fonctionnel qu'au niveau organique. Il dispose également de la traçabilité de ces modules avec les exigences qui ont nécessité ces adaptations, conformément au Triangle d'Or de l'ingénierie système préconisé par CESAMES.

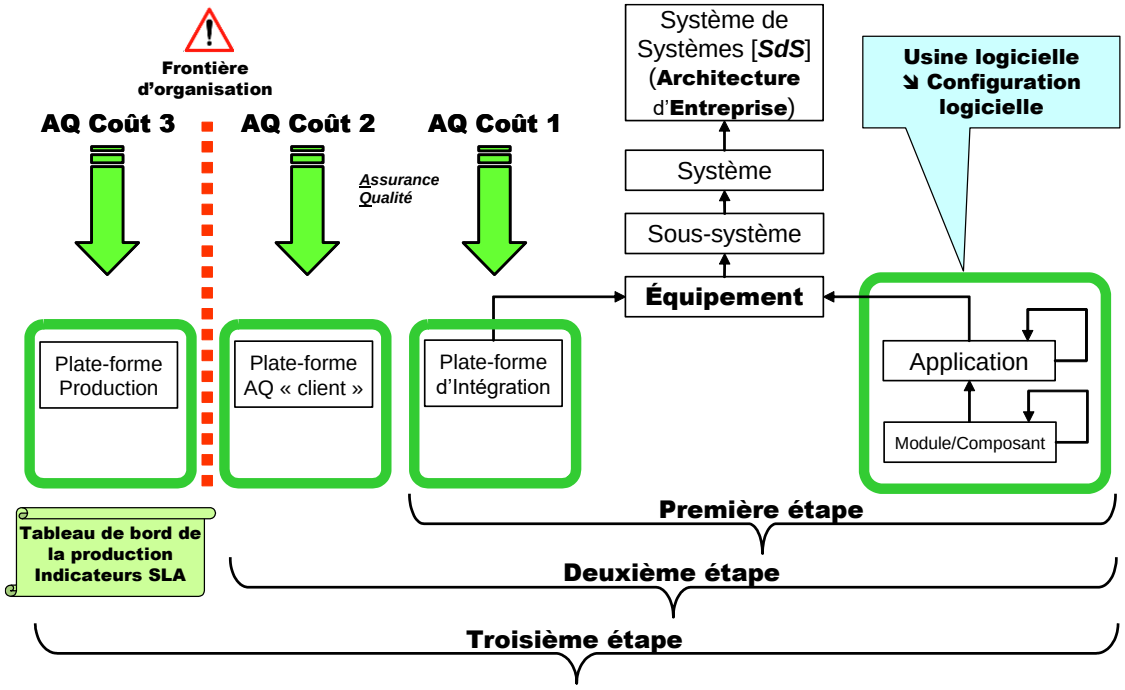

Figure 8 : Etapes du processus d'intégration

L'ordre dans lequel les modules vont pouvoir être intégrés dépend de la qualité du travail architectural. En effet, comme il a été dit ci-dessus, si c'est le laisser-faire qui prédomine, il faut s'attendre à ce que tous les modules dépendent chacun les uns des autres, ce qui va se traduire par la saturation de la matrice de couplage. Au contraire, si l'architecte a mis en œuvre les bonnes pratiques architecturales : architecture modulaire hiérarchique, architecture en couches, architecture des interfaces, les relations entre modules vont pouvoir être traduites dans la structure centrale qui pilote l'intégration : l'arbre d'intégration (Cf. réf. [4]). Cet arbre d'intégration est l'un des résultats fondamentaux du processus de conception de l'architecture. Pour l'intégration, la situation se présente comme suit [figure 9] :

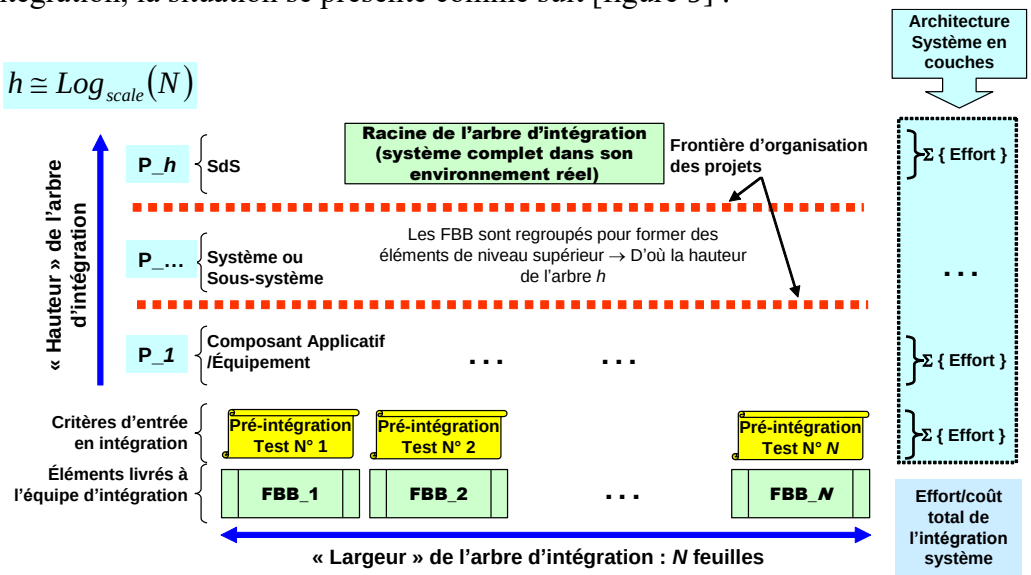

Figure 9 : L'arbre d'intégration

Chacun des modules entrant en intégration se présente en entrée du processus avec un ensemble de tests dit de pré-intégration requis par le responsable d'intégration qui valident le niveau de qualité du module et son aptitude à supporter les tests d'intégration proprement dit. Le but des tests d'intégration est de découvrir les erreurs liées aux couplages des modules ;

ces tests ne sont pas un complément des tests fonctionnels du module, lesquels font partie des critères d'entrée en intégration, ils matérialisent la validation des aspects transverses.
Les *N* blocs qui constituent les modules feuilles de l'arbre d'intégration sont regroupés par petits paquets, généralement aux alentours de 7±2, pour que leurs interactions restent compréhensibles en cas de dysfonctionnement ; la combinatoire de ces modules pouvant être décrit par des *workflows* ad hoc (i.e. une grammaire des enchaînements autorisés). Le taille moyenne des paquets définit l'échelle de l'arbre d'intégration, et finalement sa hauteur qui varie comme le *Log* du nombre de modules, comme cela est indiqué sur la figure 9.

➔ L'arbre d'intégration, déduit de l'architecture, traduit la mécanique de regroupement et d'enchaînements des modules jusqu'à la racine de l'arbre qui est le système complet.

La somme des tests à effectuer sur chacun des nœuds de l'arbre, en partant des tests de pré-intégration est une bonne mesure de la complexité de l'intégration, et la somme des coûts, à droite sur la figure 9, traduit le coût de l'organisation de cette complexité.
L'un des principaux résultats du groupe de travail I&C qui a fait l'unanimité de ses membres, a été en effet de valider que toute la remontée du cycle de développement dans sa formulation en V doit être anticipé dès la phase de conception du système. Après, il est trop tard, ou en tout cas beaucoup plus coûteux, pour faire les réajustements indispensables. La qualité du travail architectural doit donner suffisamment d'information pour mettre en œuvre des modèles d'estimation comme ceux que nous avons présenté dans nos ouvrages récents (Réf. [2] et [4]). Du point de vue de la complexité et des coûts, on peut représenter la situation comme suit [figure 10] :

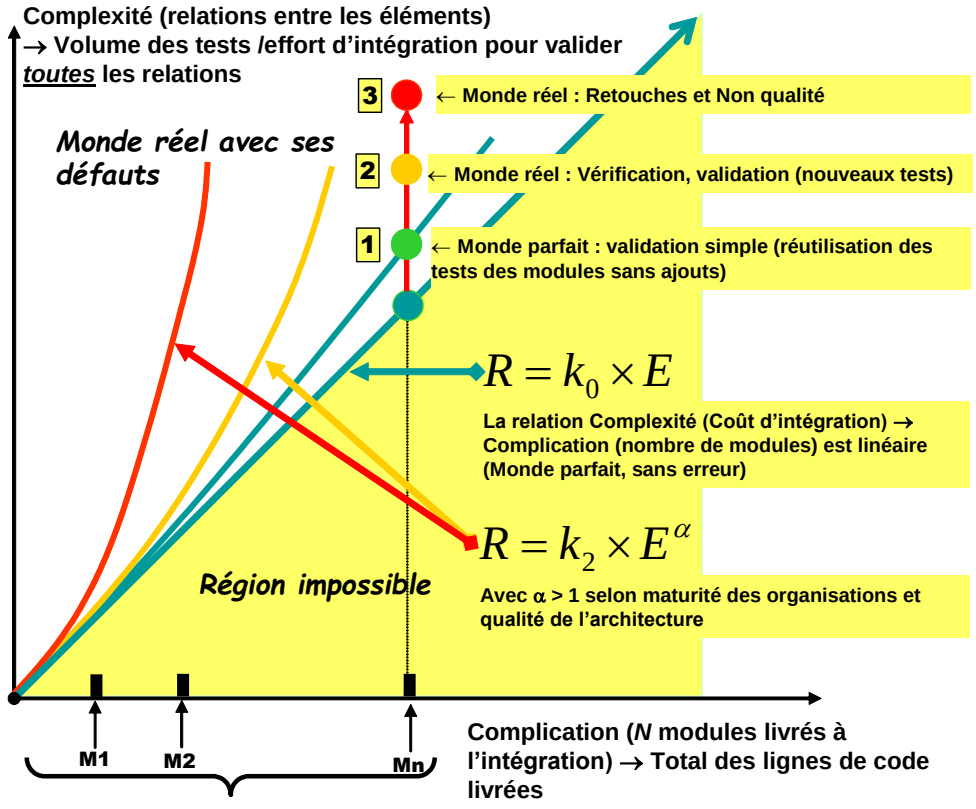

Figure 10 : Complexité complication de l'arbre d'intégration

Dans un monde parfait, sans erreur ni défaut, la complexité et le coût suivent la trajectoire 1 quasi linéaire. C'est ce que traduit la première relation $R = k_0 \times E$ dans laquelle E correspond à l'effort (volume de tests) à fournir pour valider tous les contrat d'interface des modules à intégrer.

Dans le monde réel raisonnablement organisé pour purger les erreurs, la complexité va croître comme la taille de l'arbre d'intégration, d'où l'apparition d'un loi de puissance (Voir réf. [4]). L'arbre d'intégration matérialise le coût d'organisation de la complexité, en fait la construction d'une arborescence directement issue de l'architecture, ce qui permet d'anticiper la nature des coûts. On pourrait ici introduire des distinctions faites par le modèle d'estimation COCOMO II, dans sa version 2000, qui fait intervenir la maturité de l'organisation.
Une maturité excellente, niveaux **4** ou **5**, dans l'échelle CMMI se traduira par un facteur d'échelle α légèrement supérieur à un, voire inférieur si réutilisation. Les tests sont réutilisés au maximum ce qui fait que le coût additionnel croît, mais cette croissance est faible.
Une maturité moyenne, niveaux **2** ou **3**, se traduira par un coefficient α élevé, typiquement 1,2 à 1,5. Le travail a été moins bien anticipé, on découvre plus de problèmes au fil de l'eau, donc des reprises de travail que l'on croyait acquis sont nécessaires, mais la bonne maîtrise des domaines clés du niveau **2** limite la casse.
L'absence de maturité se traduit toujours par des coût d'intégration élevés et un taux de retouches important après livraison à la production [SLA médiocre, MCO coûteux]. Dans le cas pire, encore fréquent comme l'indiquent les statistiques du Standish Group, on peut découvrir dans la période d'intégration des problèmes graves liés à la nature transverse de certaines exigences comme les performances ou la sûreté de fonctionnement ; la valeur du coefficient α va tendre vers deux. Les projets d'interopérabilité sont souvent dans cette catégorie par optimisme [manque de lucidité], manque d'anticipation et absence de modèles sémantiques qui caractérisent la nature de l'information qui s'échange entre les systèmes.
Ce qui caractérise le mieux l'absence ou le manque de maturité est la qualité du travail effectué pour expliciter rigoureusement l'architecture des interfaces. La raison d'être des interfaces est de découpler les modules ou les systèmes qui sont de part et d'autre de l'interface, comme les API métiers de la figure 7. Si l'interface ne joue pas ce rôle, alors il y aura automatiquement des couplages non maîtrisés. Les projets correspondants ne seront pas indépendants, ou beaucoup moins qu'ils pourraient l'être ; c'est une situation typique des projets d'interopérabilité mal conçus. Dans ce cas la trajectoire projet nominale de type **1**, s'écartera de plus en plus pour aller vers des trajectoires de type **2** ou **3**.

Pour conclure : Mesurer la complexité par les tests d'intégration

L'objectif N°1 du groupe de travail était de valider la pertinence d'une mesure de la complexité fondé sur les tests d'intégration. On voit que c'est effectivement le cas, dans la mesure où les tests sont directement issus de la conception.
Le mot complexité[5] est doté d'une riche polysémie qui peut rendre son emploi douteux si l'on ne prend pas quelques précautions. Dans le monde des systèmes informatisés on peut la définir en faisant une distinction de bon sens entre le nombre d'éléments constitutifs du système, et le nombre de relations, statiques et dynamiques, que ces éléments entretiennent entre eux et avec l'extérieur.
On dira qu'un système est **complexe** s'il y a de nombreuses relations entre les éléments constitutifs du système et avec l'environnement dans lequel il s'intègre. Décompter les relations/couplages est un exercice plus difficile que de compter les éléments car il y a des relations bien visibles, explicites, et des relations cachées, implicites, qui ne se révéleront qu'à l'usage : c'est le phénomène dit d' « émergence », bien connu des théories de la complexité [systèmes dynamiques]. La panne d'un système est un phénomène émergent redouté qui résulte d'une incohérence dans l'état global du système, lequel état n'a pas été visité lors de la validation, vérification et tests, i.e. lors de l'intégration. On en donnera un exemple simple ci-après avec la mutualisation et l'interopérabilité.

[5] Voir dans CSDM2011 la communication de Nancy Leveson du MIT, *Complexity and safety*.

On dira qu'un système est **compliqué** s'il comporte beaucoup d'éléments. D'où l'importance des analyses simplement volumétriques : on compte les instructions des programmes, on compte les défauts constatés [RA/Rapports d'anomalies] et les actions correctrices [Mises à jour ponctuelles, patches, ...] qui en résultent, on compte les équipements informatiques, on compte les usagers, on compte les programmeurs, on compte les messages d'erreurs et les aides en ligne, etc. ... etc. Mais on oublie souvent de dénombrer les états du système qui est cependant le facteur prépondérant de la complication, les états mémoire.
Pour bien appréhender la complexité, il faut donc soigneusement distinguer son aspect complication, i.e. un simple dénombrement [par exemple le nombre de règles et/ou de contraintes, le nombre de lignes source, le nombre de modules, etc.], et son aspect relations/couplages. Nous réservons l'usage du mot complexité à l'aspect dénombrement des relations/couplages qui prend en compte la dimension dynamique des interactions, lesquels seront matérialisés par des tests d'intégration, à un moment ou à un autre.
Par exemple, rendre deux systèmes interopérables c'est objectivement augmenter la complication, par le simple fait que les lignes de code et/ou les données [en y incluant les événements] des deux systèmes s'additionnent. Mais c'est aussi et surtout augmenter la complexité, car les deux systèmes vont échanger des informations et interagir de diverses façons. Il va donc y avoir de nouveaux couplages qui par définition n'existaient pas avant l'opération de fusion. Des éléments de chacun des systèmes vont ainsi se trouver « couplés ». Ainsi se crée une nouvelle relation qui stipule que toute modification de l'un des éléments couplés ne peut se faire sans l'accord de l'autre. Si l'un des systèmes a besoin d'une fonction qui existe déjà chez l'autre, il est inutile de la re-développer, il suffit de pouvoir l'utiliser en l'état à l'aide d'un protocole ad hoc, là où elle est, par exemple avec un simple workflow. C'est un nouveau type de couplage, i.e. une relation de partage.
Cette nouvelle situation est schématisée par la figure 11.
Le schéma met en évidence un aspect totalement contre intuitif relatif au nombre d'états résultant de ce couplage. Il faut en effet considérer l'ensemble des parties des états de chacun des systèmes, d'où la combinatoire « explosive » [si rien n'est fait, fonction du nombre de pièces] en 2^{n1+n2}, c'est-à-dire 2 à 2, 3 à 3, ... etc., soit l'ensemble des parties.

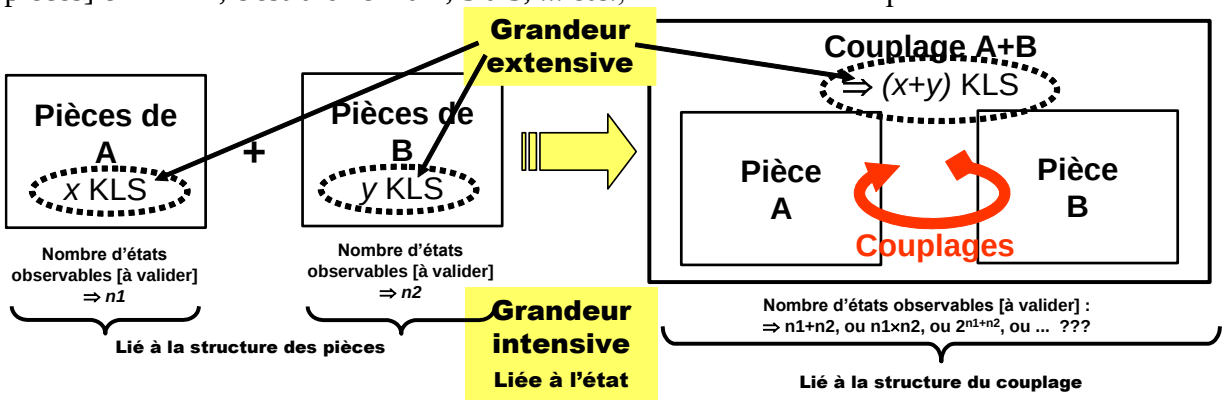

Figure 11 : Influence du couplage sur le nombre d'états à valider

NB : le nombre de lignes de code est une grandeur additive : elles s'additionnent. Le coût moyen de ces lignes de code est une grandeur intensive car le coût dépend de l'environnement, comme nous le précise le modèle COCOMO. 1.000 LS seules ne sont pas équivalentes à 1.000 LS parmi 500.000 LS car le facteur d'échelle α prend en compte ce contexte. En se sens, le coût/effort « abstrait » selon COCOMO fonctionne comme une « température » ou une « pression » (voir réf. [4]).

➔ Des lignes de code isolées sont plus « froide » que des lignes de code intégrés, i.e. moins coûteuses !

Dans une opération de mutualisation d'éléments d'un [ou plusieurs] systèmes, on baisse objectivement la complication car le volume de code et/ou de données mutualisés se trouve factorisé dans une bibliothèque et/ou une base de données. Mais on augmente corrélativement la complexité, car là où il n'y avait pas de relation, il en existe maintenant de nouvelles (couplages par réutilisation, par partage de ressources, etc.), ne serait-ce que pour utiliser/référencer/modifier l'élément mutualisé.
Pour que la mutualisation soit rentable il est impératif que le gain engendré par la baisse de la complication (moins de code et/ou de données), soit largement supérieur au coût de la mutualisation. Un élément mutualisé sera par définition plus sollicité après l'opération de mutualisation qu'avant. Or l'occurrence des pannes est une fonction monotone croissante de l'usage. Conséquence : son niveau de maturité doit être très supérieur à ce qu'il était avant mutualisation. Moralité : de nouveaux scénarios de tests seront nécessaires pour « durcir » l'élément, car s'il tombe en panne, ce sont tous les éléments appelants qui à leur tour ne rendront pas le service attendu. Si rien n'est fait il y aura dégradation inévitable du contrat de service, conformément à la loi générale d'augmentation du désordre [entropie du système]. Le code mutualisé doit être programmé en priorité à tout autre et intégré le plus vite possible de façon à l'exposer le plus fréquemment possible via la mécanique d'intégration continue.
Beaucoup de DSI ont été abusés par des décisions hâtives, sans une vraie réflexion, souvent poussés « au crime » par les éditeurs et/ou leurs consultants qui vivent « sur la bête », vendant de la simplification sans s'intéresser à la complexité, quand celle-ci n'est pas purement et simplement ignorée : le cas des EAI, technologies au demeurant intéressantes, mais extrêmement dangereuses une fois mises entre des mains inexpérimentées. Le non-dit étant avant tout d'éviter de parler de ce qui fâche, et si cela ne marche pas, on rajoutera une couche d' *"Add On"* sensés régler le problème.
On peut illustrer ces différents phénomènes à l'aide du diagramme figure 12.

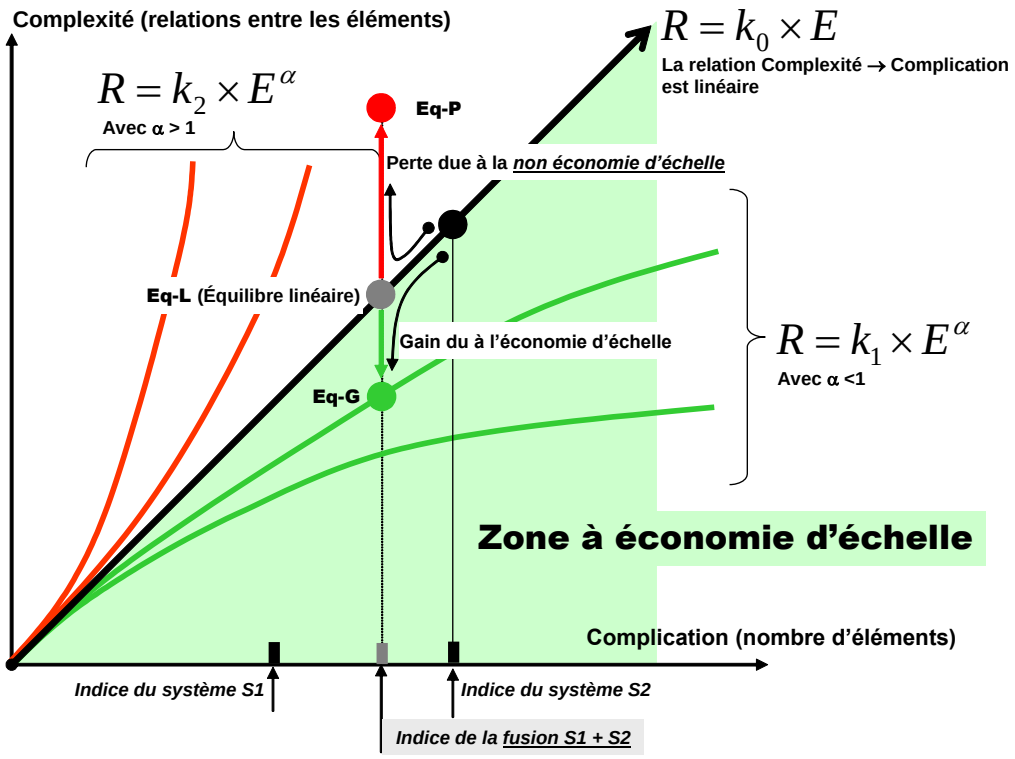

Figure 12 : Economie d'échelle et complexité

Ce qu'il faut absolument garantir dans une opération d'interopérabilité est que le coût après fusion des systèmes S1/S2, reste linéaire, c'est-à-dire que la baisse de l'indice de complication s'est accompagnée corrélativement d'une baisse proportionnelle du nombre des relations, une

fois les systèmes interopérés / fusionnés ; ceci est matérialisé dans la figure par le point d'équilibre linéaire Eq-L.
Une opération réussie se caractérisera par une économie d'échelle mesurable, d'où l'obtention d'un gain. Ce gain s'obtient grâce à un investissement en architecture qui permet de faire chuter le nombre de relations et d'atteindre le point Eq-G, grâce à l'architecture des interfaces entre S1 et S2. L'interopérabilité de plusieurs systèmes est d'abord un problème d'architecture qui doit être posé comme tel, indépendamment des technologies utilisées, conformément aux objectifs de cette mise en interopérabilité.
Une opération ratée, résultant de l'impasse sur l'architecture des interfaces, va se traduire certes par une baisse de la complication dans un premier temps, par exemple moins de lignes de code à gérer, partie visible de l'opération, mais accompagnée d'une augmentation corrélative du nombre de relations qui va se traduire par une augmentation du coût d'intégration et/ou du coût de non qualité. Du fait de l'augmentation du nombre des relations, il y a une augmentation du travail de vérification/validation indispensable au maintien du contrat de service (SLA). En conséquence, on se retrouve au point Eq-P. Si rien n'est fait ce sont les usagers qui paieront la note via le SLA médiocre et le MCO coûteux.

La définition intuitive de la complexité : PLUS COMPLEXE = PLUS DE TESTS = PLUS COUTEUX, peut donc être considérée comme validée. Mais à une condition : expliciter rigoureusement l'architecture du système, tant au niveau fonctionnel qu'au niveau organique, et surtout les interfaces.
➔ La structure du coût de cette complexité peut être anticipé et estimé comme un résultat de l'architecture, en particulier par la connaissance des matrices de couplages qui est un critère de terminaison du processus de conception du système.

Recommandations et bonnes pratiques

Il est contre-productif, voire futile, de continuer aujourd'hui à séparer artificiellement l'ingénierie des systèmes d'information et l'ingénierie des équipements massivement informatisés, anciennement baptisés systèmes industriels. La frontière entre les deux est devenue si poreuse que la distinction n'est désormais plus pertinente. Les langages de programmations généraux se sont banalisés, il y a du Java partout, dans nos téléphones mobiles, dans les systèmes de défense et de sécurité, dans tous les IHM des systèmes d'information récents et leur portails Internet, etc. ... etc., en attendant d'autres langages, comme peut-être Python, dans la communauté Internet. L'ingénierie des systèmes informatisés, quelle que soit leur nature, est **UNE** : c'est celle de l'entreprise numérique qui doit savoir gérer toute l'information qui en constitue l'intelligence même, et en exploiter la valeur.

➔ Il ne faut plus penser ni plateforme, ni infrastructure, il faut penser flux d' INFORMATION.

La machinerie numérique, plates-formes et infrastructures, qui supporte l'ensemble des processus est aujourd'hui complètement banalisée, virtualisée, il n'y a plus de machines propres aux systèmes industriels, et depuis longtemps, sauf dans quelques niches très spécifiques. Une grande variété d'organes d'interactions, capteurs et effecteurs, entre les hommes et les machines, entre les machines elles-mêmes dont certaines sont des robots capables d'initiatives, ont vu le jour, s'adaptant de mieux en mieux aux besoins de performance des acteurs, et mobiles de surcroît.
Les méthodes d'ingénierie sont fondamentalement les mêmes, ce qui se traduit par des familles d'outils de plus en plus génériques, avec des langages de modélisation à large spectre comme BPML, UML, SysML, etc. ..., voir des DSL, permettant une description rigoureuse des systèmes et des services, au bon niveau d'abstraction.

➔ Les langages de modélisation sont des instruments indispensables à une bonne compréhension des comportements et de la dynamique des systèmes, mais la notion fondamentale est bien celle du **savoir penser** MODELISATION.

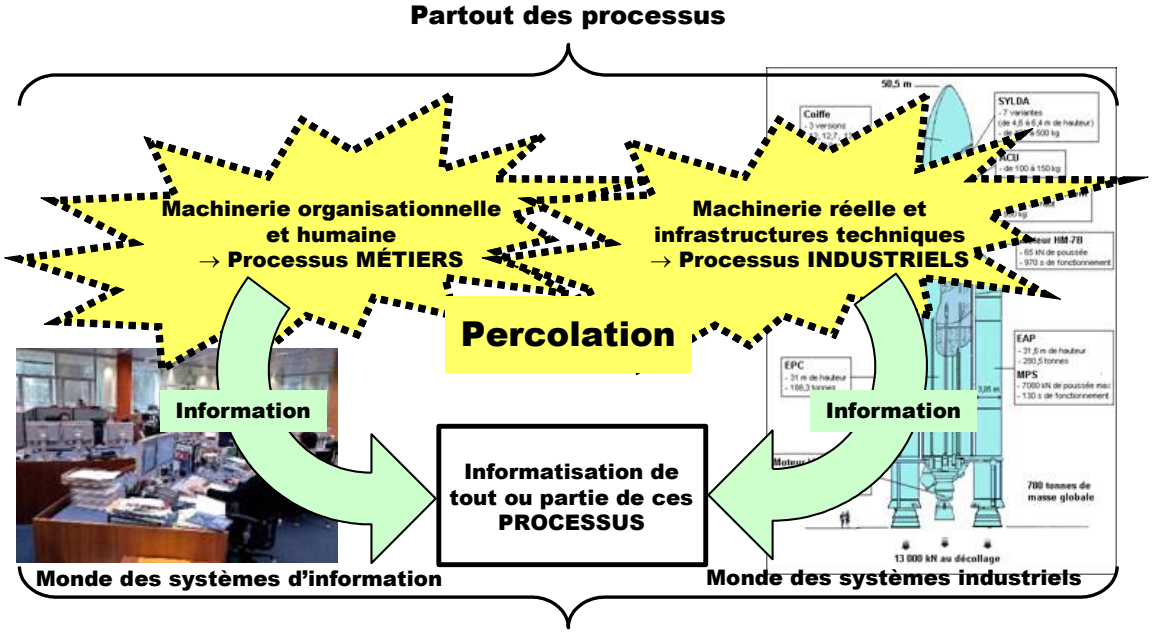

Figure 13 : Unification des ingénieries

Tant pour l'organisation des processus métiers que pour l'agencement des processus industriels l'architecture est le point de passage obligé d'une ingénierie proactive, prédictive et durable, qui sait prévoir ce qui va se passer et en mesurer l'avancement.

Il faut cesser de penser que l'architecture est un centre de coûts. Les témoignages recueillis par le groupe de travail montrent à l'évidence que tout investissement effectué en amont sera largement compensé par des gains en aval d'un facteur 10 à 50 pour l'intégration, beaucoup plus après la mise en production.
➔ Une incohérence d'interface non détectée peut conduire à des retours en développement / maintenance de centaines de pièces qui dépendent directement ou indirectement de cette interface.

Le résultat visible du processus de conception du système est une famille de MODELES COHERENTS qui vont permettre aux architectes de répondre aux questions de toute nature que pose la réalisation, de s'assurer que les pièces du puzzle forment un tout qui satisfait aux exigences globales et que ces pièces peuvent être assemblées de façon systématique :

- Découpe fonctionnelle et organique permettant aux équipes projets de travailler en parallèle : c'est le fondement de l'agilité.
- Simultanément à la découpe, ingénierie des tests permettant de valider fonctionnellement et organiquement le découpage, pour chacune des pièces prises individuellement : c'est le concept d'architecture testable, i.e. *Test Driven Development*.
- Contractualisation des interfaces à tous les niveaux et ingénierie de ces interfaces : tout doit être explicité pour démarrer en parallèle les scénarios de validation système, les pilotes et les « bouchons » pour le monitoring des tests.

- Arbre d'intégration qui est le protocole d'ordonnancement des pièces à assembler et des tests d'intégration à appliquer à chacune des étapes d'assemblage : c'est le concept d'usine système logiciel et d'intégration continue, analogue à une chaîne de montage, indispensable à la mise en œuvre de l'agilité.
- Ingénierie des exigences, en particulier toutes celles qui intéressent la globalité du système comme les performances, la sûreté de fonctionnement, la sécurité, l'interopérabilité, l'évolutivité de façon à disposer le plus tôt possible de scénarios de validation globale, de bout en bout : c'est le préalable de la démarche « centrée architecture » qui garantit que le système réalisé satisfera les exigences client.

Chacun doit se convaincre qu'il est désormais plus simple, moins coûteux, plus sûr de raisonner sur des modèles partagés entre les acteurs que sur le système lui-même. L'arbre d'intégration qui en est le résultat est l'instrument concret, pragmatique qui permet de synchroniser et de contrôler les trajectoires des différents projets qui contribuent à la réalisation du système conforme aux besoins de ses usagers réels.

➔ Les bonnes pratiques de modélisation et leur mise en œuvre par chacun des acteurs concernés est le sésame qui permet à l'entreprise numérique d'exploiter son bien le plus précieux : l'INFORMATION.

Annexe : les objectifs du groupe de travail

→ *Fixés en mars 2011 :*

Depuis déjà quelques années **les tests sont devenus une préoccupation centrale des organisations de développement** de logiciel et/ou de systèmes informatisés qui intègrent des volumes importants de logiciel. Le métier de testeur commence à être reconnu comme tel, avec des formations ad hoc, par les organismes professionnels (CIGREF, SYNTEC, ...).

Quel que soit le secteur industriel ou tertiaire vers lequel on se tourne, **l'accroissement de la taille des logiciels** est flagrant. Tous les responsables de développement et/ou d'intégration avec lesquels nous sommes en contacts sont conscients que cet accroissement de taille s'accompagne de difficultés de plus en plus grande pour valider les logiciels dans leur contexte système et d'exploitation réelle.

L'effort de test pour garantir le contrat de service requis, quelle que soit la nature des activités correspondantes dans le cycle de vie, devient la grandeur prépondérante des projets informatiques, et le facteur de risque le plus élevé. Dans un projet d'intégration, c'est de loin le coût le plus important, la programmation se réduisant souvent à l'écriture de scripts de paramétrage.

Dans la perspective du *Test Driven Development* et des architectures orientées test promues par les communautés de l' « eXtreme programming » et des méthodes agiles, **le test devient la pierre angulaire des projets** informatiques, et l'architecture testable le sésame de la simplification. Il est donc primordial de définir une **mesure objective de la taille des tests**, indépendantes des méthodes d'obtention des tests[6].

Nous proposons, dans le cadre de ce groupe de travail, de nous focaliser sur **les tests d'intégration**. Ces tests apparaissent comme un excellent **indicateur de la complexité réelle**, c'est-à-dire de toutes les relations, de tous les couplages, des dépendances directes ou indirectes, que l'élément à intégrer entretient avec son environnement. Ce travail passe par une **standardisation de la description des tests**, assimilé à un texte constituant une « preuve » de bon fonctionnement de l'élément correspondant, dans son contexte d'emploi réel. Cette standardisation est l'analogue des lignes de code et/ou des points de fonctions dans les modèles d'estimation.

[6] Voir J.Printz, CSDM2010, *A natural measure for system software complexity*.