# Des langages pour améliorer le développement et la maintenance des logiciels à base de composants

Régis Fleurquin

MÉMOIRE EN VUE DE L'OBTENTION DU DIPLÔME
D'HABILITATION À DIRIGER DES RECHERCHES EN INFORMATIQUE

*présenté devant*

L'UNIVERSITÉ DE BRETAGNE SUD
(SOUS LE SCEAU DE L'UNIVERSITÉ EUROPÉENNE DE BRETAGNE)

*Par*

RÉGIS FLEURQUIN

# Des langages pour améliorer le développement et la maintenance des logiciels à base de composants

*Présenté le 05/07/2010 devant le jury composé de*

M. Stéphane DUCASSE (rapporteur)
M$^{me}$ Marianne HUCHARD (rapporteur)
M. Houari SAHRAOUI (rapporteur)
M. Jean-Marc JEZEQUEL (examinateur)
M. Pierre-François MARTEAU (président et examinateur)
M. Flavio OQUENDO (examinateur)

## RÉSUMÉ

La définition d'éléments pouvant contribuer à améliorer le développement et la maintenance des logiciels est l'objectif des travaux de recherche menés en *Génie Logiciel*. Mon domaine de recherche porte depuis 2003 sur la définition et l'usage de langages « supports » (c'est-à-dire complétant les langages de développement) ; ceci dans le but de faciliter et d'améliorer le développement et la maintenance des applications logicielles conçues à l'aide de composants. Mes travaux se trouvent au carrefour de plusieurs disciplines du Génie Logiciel : le développement orienté composant, les architectures des logiciels, la maintenance et l'évolution, la qualité et l'ingénierie dirigée par les modèles.

Ce document constitue une synthèse de mes activités de recherche menées au cours des sept dernières années au sein de l'équipe SE du laboratoire VALORIA et de l'équipe-projet INRIA TRISKELL. Il commence par donner une vision assez personnelle des courants qui animent le Génie Logiciel et dresse un aperçu de l'état actuel de cette discipline. Puis il introduit les quelques notions dont la maîtrise est nécessaire pour évoluer dans les domaines dans lesquels se sont déroulées mes activités. Il retrace ensuite mon parcours thématique au cœur de ces disciplines et détaille mes travaux sur le contrôle de l'évolution des architectures, la sélection de composants et la documentation et l'exécution des bonnes pratiques de modélisation.

# REMERCIEMENTS

Je tiens tout d'abord à adresser mes remerciements à madame *Marianne Huchard* et messieurs *Stéphane Ducasse* et *Houari Sahraoui* qui ont pris le temps de revisiter mes activités de recherche et de commenter ce mémoire.

Même si le discours de ce mémoire est parfois (lorsque je le mérite) à la première personne du singulier, je n'oublie pas que les travaux présentés ici ne sont pas le fruit de mes seuls efforts mais également ceux consentis par les 6 doctorants dont j'ai assuré (parfois assure encore) l'encadrement : *Chouki*, *Bart*, *Réda*, *Soraya*, *Kahina* et *Vincent*. Je ne manquerai pas de rappeler leurs contributions aux moments opportuns.

La réussite de la préparation de cette habilitation doit beaucoup aux encouragements et à l'amitié que m'ont témoigné des années durant, mes collègues de l'IUT de Vannes (en particulier le très regretté *Paul Le Floch*, mais également *Jean-Luc Eveno* et *Jean-François Kamp*) et du laboratoire Valoria (tout particulièrement son directeur *Pierre-François Marteau* et une ex-co-déléguée INRIA, *Sylvie Gibet*).

Ce projet n'aurait sans doute jamais pu aboutir aussi rapidement sans ces deux années de délégation au sein de l'équipe INRIA-*Triskell* ; un grand merci donc à *Jean-Marc Jézéquel* pour avoir rendu possible dans ma carrière cet épisode très enrichissant et avoir accepté de siéger dans mon jury d'HDR. Merci également à *Olivier Barais*, *Noël Plouzeau*, *Benoît Baudry*, *Didier Vojtisek* et l'ensemble des non permanents de l'équipe *Triskell* pour leur chaleureux accueil.

Toutes ces personnes, par un moyen ou un autre, ont permis la germination de l'idée, l'envie d'entreprendre, l'épanouissement puis l'aboutissement de ce « projet » d'Habilitation.

Ma plus vive reconnaissance à l'égard de *Salah Sadou* pour son soutien indéfectible durant toutes ces années. Les travaux présentés sont aussi la résultante de nos enrichissantes discussions.

Mais ces remerciements ne pouvaient se conclure sans une pensée pour ces anonymes à la science qui me sont le plus chers, dont aucun papier ne verra jamais la signature à mes côtés, mais qui pourtant chaque jour font que la vie est là, avance, que l'énergie abonde en particulier pour mener mes activités de recherche : merci à toi *Claire*, merci à toi *Juliette*, merci à toi *Vincent*.

Table des matières

# Chapitre 1 Avant propos

## 1.1 Sur l'objectif du mémoire

Un mémoire d'Habilitation à Diriger des Recherches doit relever un double défi : synthétiser avec force recul les résultats scientifiques obtenus et mettre en lumière le cheminement scientifique dont ces résultats constituent la partie visible. Il faut à la fois convaincre de la profondeur, de l'originalité de l'apport scientifique des différents travaux entrepris et manifester une pensée scientifique ayant su conjuguer avec les aléas liés à notre métier.

La tâche n'est pas simple. Le risque est grand de se perdre dans les méandres des détails techniques. Il faut donc préserver l'essentiel tout en excluant les éléments secondaires qui peuvent être obtenus, si besoin, en parcourant les articles publiés et plus encore les mémoires des doctorants ayant soutenus. Exercice délicat que d'abstraire son discours tout en restant rigoureux et à même de présenter des raisonnements et des conclusions acceptables par les experts des domaines concernés.

Il faut se garder également de vouloir expliquer et confiner chaque travail dans un univers clos mais au contraire arriver à faire émerger les liens de toutes natures unissant l'ensemble des domaines abordés. Il faut montrer que les résultats sont le fait d'une réflexion profitant de débordements thématiques, de mélanges, de rencontres.

Le mémoire est également l'occasion de montrer une certaine maîtrise de sa discipline. Il faut être capable de dresser un état des lieux structuré et structurant, de pointer les courants, les enjeux. A travers la vision dégagée, il faut également prendre du recul sur ses propres travaux, leur positionnement et impact, leurs faiblesses et richesses.

## 1.2 Le plan du mémoire

J'ai essayé de répondre à tous ces impératifs au cours de ma rédaction. Le plan du mémoire en est le reflet.

Le chapitre 2 donne une vision assez personnelle des courants qui animent le Génie Logiciel. Il dresse un aperçu de l'état actuel de cette discipline.

Le chapitre 3 complète cette vision. Il introduit, plus en détail, les quelques notions dont la maîtrise est nécessaire pour évoluer sereinement dans les domaines dans lesquels se sont déroulées mes activités : les architectures et les composants, la maintenance et l'évolution du logiciel, les langages et les modèles.

Les chapitres 4, 5 et 6 constituent le cœur du mémoire. Leur lecture devrait permettre de comprendre mon cheminement, ses points de décision et d'inflexion, l'éthique à laquelle je me suis astreint, le contexte qui a facilité ou parfois contraint ce cheminement, et, bien entendu « l'essence » des résultats obtenus. Sur ce dernier point, l'objectif est plus de décrire la façon dont j'ai travaillé que de détailler les résultats obtenus. Pour ces détails, on pourra se reporter aux articles dont le texte égrène les références. Cette partie du mémoire dessine la carte de mes activités et pose les liens de dépendances. J'insisterai tout particulièrement sur un aspect commun à tous mes travaux : la définition et l'usage de langages dédiés. Aspect dans lequel, je fus tout particulièrement investi.

Le chapitre 4 positionne mes travaux et explique quelle fut ma démarche de recherche.

Les chapitres 5 et 6 présentent autant chronologiquement que thématiquement, ce que fut ma démarche scientifique. Ces deux dimensions sont entremêlées. La première dimension met en valeur les liens de dépendances, détaille les décisions qui furent prises à la lumière de leur contexte. La seconde confère de la structure au discours en facilitant le regroupement d'éléments proches au sein d'une section ou d'une sous-section.

Le chapitre 5 se concentre sur mes travaux les plus anciens et maintenant terminés, menés dans le domaine des architectures logicielles et des composants.

Le chapitre 6 expose pour sa part, mes travaux actuels relevant principalement de l'ingénierie dirigée par les modèles : la documentation des bonnes pratiques de modélisation et la composition de modèles. Il présente également des idées persistantes conséquences de réflexions personnelles dont il faudra assurer les contours. Autant de thèmes, sur lesquels j'espère, à l'avenir, poursuivre mes travaux.

Chaque section des chapitres 5 et 6 s'attache à la présentation d'un thème particulier. Chacun de ces thèmes représente une activité suffisamment riche, cohérente et autonome pour mériter une écriture propre. Un thème est généralement le fruit d'une, voire deux thèses de doctorat. La description de chaque thème suit, grosso modo, une progression à cinq temps, à savoir : la mise en évidence du problème traité, l'analyse des points forts et faibles des réponses existantes, la description de la solution proposée et son outillage, la présentation des expérimentations réalisées et enfin un bilan, a posteriori, sur le travail réalisé.

Le chapitre 7 conclut le mémoire.

## *1.3 Et avant 2003 ?*

Il n'échappera pas au lecteur que ce document ne fait état que de travaux entrepris depuis 2003. En effet, bien que recruté en 1997 en qualité de maître de conférences à l'Université de Bretagne Sud, les six premières années de ma carrière (1997 à 2003) ne me placèrent guère dans des conditions favorisant l'épanouissement d'une activité de recherche : de nombreux enseignements à monter, une charge d'enseignement très importante liée à un déficit en poste et une direction des études de trois ans.

La très faible densité des publications sur cette période, évidente à la lecture de mon curriculum vitae, est une conséquence visible de ces mauvaises conditions. Quelques travaux furent réalisés mais, de mon point de vue, de par leur éloignement avec les thèmes développés dans ce mémoire, je ne juge pas utile d'en faire état dans ce document. J'ai donc fait le choix de ne relater que les activités entreprises depuis 2003. Sept années durant lesquelles mes activités de recherche purent reprendre un rythme normal.

# Chapitre 2 Le Génie Logiciel

Ce chapitre donne une vision assez personnelle de la discipline Génie Logiciel, de la matière qui la constitue et des lois qui l'animent. La première section rappelle l'importance du Génie Logiciel. La section suivante introduit les quatre types d'outil qui aident les développeurs dans leur quotidien. La troisième section met en lumière les trois principes qui, depuis les premiers balbutiements de la discipline, ont permis aux chercheurs et aux industriels d'améliorer sans faillir les moyens de développement. Les trois dernières sections, enfin, tentent une esquisse des tendances et enjeux actuels de la discipline dans le domaine respectivement des procédés, des langages et des outils.

## *2.1 Le besoin d'un génie pour le logiciel*

La technologie du logiciel est le moyen le plus efficace pour injecter de l'intelligence et des capacités fonctionnelles dans un système. En effet, la mise en place d'une fonctionnalité particulière peut ne réclamer que quelques lignes de codes alors qu'à l'inverse sa prise en charge intégrale par le matériel nécessiterait le développement long et coûteux d'un circuit intégré dédié. Il est également bien plus facile d'amender les possibilités offertes par un dispositif au travers de la mise à jour du logiciel qu'il embarque que de reprendre une quelconque architecture mécanique et/ou électronique. Plus encore, la gestion des versions d'un logiciel ouvre la voie à la mise sur le marché de familles de produits aux fonctionnalités distinctes mais s'appuyant sur la même architecture matérielle. Le logiciel est désormais le médium véhiculant la part principale de la valeur ajoutée des dispositifs électroniques ou des processus automatisés. C'est à travers lui que l'on cherche à se distinguer de la concurrence, à capter des parts de marché tout en diminuant les coûts.

On a donc assisté à une augmentation continue et rapide de l'usage de cette technologie dans tous les secteurs et dans nombre d'objets de notre vie courante. Les logiciels prennent à leur charge de plus en plus de fonctionnalités avec pour conséquence une croissance de leur taille, au minimum d'ordre polynômial. Plus encore, ils s'exécutent souvent sur plusieurs plates-formes avec un double objectif : d'une part mettre à disposition l'information et les moyens de calcul à l'endroit où ils sont réclamés (ordinateurs portables, assistants personnels, téléphones intelligents, station de travail, etc.) en profitant des technologies du nomadisme et d'autre part optimiser l'usage des moyens informatiques à disposition sur le plan de la sécurité, des performances, de la fiabilité et des coûts. Les applications informatiques deviennent d'imposants « patchworks technologiques ». Ils sont désormais, pour nombre d'entre eux, compliqués voire dans certains cas complexes lorsqu'ils deviennent le lieu de propriétés émergentes.

Non content de se retrouver confrontées au développement de produits de plus en plus compliqués, les entreprises du secteur logiciel doivent faire face à des clients de plus en plus exigeants. Ces clients expriment en particulier de nouvelles attentes en termes de qualité. En effet, les applications informatiques sont désormais le support de la plupart des processus métiers. La moindre défaillance, panne ou immobilisation peut conduire à une dégradation importante du service rendu. La désorganisation résultante est à l'origine, parfois, de pertes financières, voire, pour certains systèmes critiques, d'atteintes à la sécurité des biens et des personnes. On exige également des applications la plus grande flexibilité pour pouvoir accompagner durablement l'évolution des usages, des processus métiers et ainsi rester compétitif. On réclame la mise à disposition rapide des nouvelles versions dans le but de corriger des erreurs résiduelles ou d'adapter le logiciel à un nouvel environnement. La *maintenance* des applications joue ici un rôle essentiel. Elle constitue d'ailleurs, et on l'oublie bien souvent, la majeure partie des activités des sociétés du secteur. Pour limiter les risques, les donneurs d'ordre se montrent en début de projet, tout autant sensible aux coûts et délais qu'aux garanties affichées : niveau de maturité CMMI-DEV (Team, 2006) ou possession d'un certificat

ISO 9001. En cours de projet, maîtrise d'ouvrage et futurs utilisateurs souhaitent être associés plus fortement au développement pour, au fil des projets, valider les choix et contrôler le respect des objectifs. En fin de projet, les clients se montrent toujours plus exigeants sur la complétude fonctionnelle et la convivialité des produits livrés, sur leur sécurité et leur fiabilité.

Le secteur du logiciel, fer de lance de la mouvance "nouvelle technologie", doit donc relever le double défi posé par une complexité croissante et par un renforcement des attentes des décideurs et des utilisateurs. Le secteur arpente depuis plusieurs années maintenant le chemin menant à la quête de la plus grande qualité, à coûts et délais maîtrisés. L'expérience passée nous a enseigné qu'il est risqué de parier sur une avancée technologique ou méthodologique décisive capable de nous aider à relever ces défis durablement. Le prémonitoire « no silver bullet » (Brooks, April 1987) lancé dès 1986 n'a jamais été mis à mal par aucune technologie, langage ou méthode depuis presque un quart de siècle. L'histoire nous enseigne qu'il faut au contraire espérer la patiente et lente constitution d'un ensemble de réponses en perpétuelle évolution : « There is no royal road, but there is a road ».

Contrairement à ce que l'on entend souvent, la discipline du logiciel n'est pas en échec. Les moyens actuels permettent de développer des systèmes critiques fiables, comportant plusieurs centaines de milliers de ligne de code. Chose impensable, il y a deux ou trois décennies. Dans le domaine du Web des applications utiles et non triviales peuvent être générées en quelques clics. Il y a donc d'indéniables succès. Mais ces succès sont le terreau de nouvelles aspirations. Chaque évolution des pratiques de développement permet la prise en charge d'applications plus compliquées, mais la marge de manœuvre obtenue un temps stimule de nouveaux excès qui se concluent à terme par l'obsolescence des moyens déployés et par la nécessité d'en substituer de nouveaux. Toute autre discipline confrontée comme la nôtre, à des limites sans cesse repoussées, connaîtrait les mêmes difficultés, subirait les mêmes critiques, réclamerait sans jamais faiblir les mêmes progrès.

## 2.2 Les 4 éléments fondamentaux du Génie Logiciel

La définition d'éléments pouvant contribuer à améliorer le développement et la maintenance des logiciels est l'objectif des travaux de recherche menés en *Génie Logiciel*. C'est le domaine dans lequel prennent place toutes mes activités de recherche.

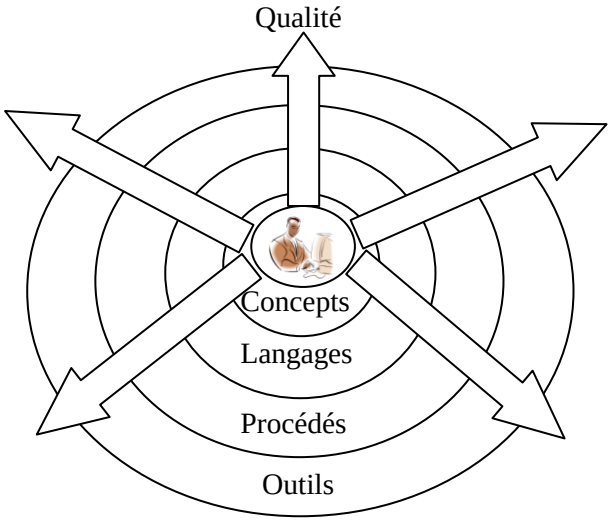


**Figure 2-1 Les 4 éléments fondamentaux du Génie Logiciel**

Le patrimoine à constituer pour cette discipline doit être capable de stimuler la compétence là où elle faisait défaut et de la faire fructifier plus encore, là où elle s'exerçait déjà. Les

intervenants d'un projet logiciel doivent, pour ce faire, évoluer dans un « éther », savant mélange incluant invariablement quatre éléments (Figure 2-1) :

- des *concepts* (module, classe, héritage, visibilité, contrat, etc.) ;
- portés par des *langages* de programmation ou de modélisation ;
- eux-mêmes encadrés par des *procédés* (cycles de vie, méthodes, procédures qualités, métriques, règles de modélisation et de programmation, etc.) ;
- le tout pris en charge par des *outils* (éditeurs, outils de test, outils de gestion des versions, etc.).

Les langages mettent en avant des concepts répondant à une double exigence. D'une part, ils doivent offrir un pouvoir d'expression suffisant pour représenter l'ensemble des systèmes ciblés par le langage. D'autre part, ils fixent l'attention de l'architecte ou du développeur sur certains points essentiels (le masquage d'information, le couplage, la cohésion, la précision, etc.). Ils l'orientent, voire le contraignent, à respecter certaines règles universellement reconnues comme pertinentes dans le domaine ciblé. Ils restreignent la forme des modèles ou des programmes représentables en excluant des formes non souhaitables. Ce faisant, les concepts augmentent les chances de produire des logiciels présentant de bonnes propriétés.

Cependant, les langages sont conçus pour permettre la représentation d'architectures répondant à une classe variée de besoins. Or, certains motifs architecturaux peuvent être jugés utiles dans certains contextes et à proscrire dans d'autres. La qualité d'un modèle ou d'un programme n'a pas de valeur absolue mais s'estime à la lumière des exigences (parfois contradictoires) liées à chaque projet (coûts, délais, exigences qualités, etc.). La qualité s'apprécie à la lumière des compromis qui s'imposent. Les langages doivent donc laisser des marges de manœuvre, se montrer tolérants et ne pas trop restreindre le champ des possibles afin de laisser libre cours à la créativité des développeurs.

En conséquence, l'usage d'un langage seul, aussi élégant qu'il puisse être, n'apporte pas la certitude, pour un projet donné, d'obtenir un modèle ou programme répondant aux exigences. Les procédés viennent alors fournir un outil complémentaire essentiel. Ils vont permettre, en s'appuyant sur le contexte propre à chaque projet d'orienter les développeurs vers le sous-ensemble des modèles du langage pertinent pour le problème abordé. A chaque étape, ils aident les développeurs à limiter leur espace de choix (Figure 2-2). Les procédés opèrent virtuellement, in fine, une restriction langagière progressive adaptée à chaque projet.

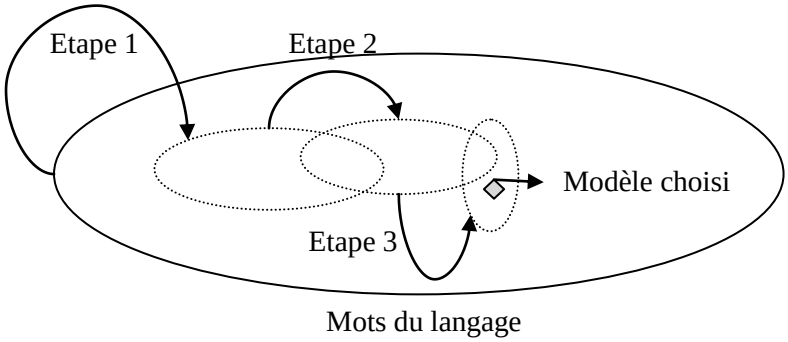


**Figure 2-2 un procédé vu comme une succession de restrictions langagières**

Les outils, de leur côté, en prenant à leur charge un maximum de tâches (celles calculables en temps et ressources acceptables) autorisent la gestion et la manipulation de données de grande taille repoussant d'autant les limites de la complication. Ils augmentent également la rapidité et la fiabilité des tâches qu'ils assurent.

L'écosystème ainsi constitué devra permettre de mener des développements bien plus complexes tout en produisant (rayonnant) plus de qualité, à compétence humaine égale, qu'un environnement ne laissant reposer sa réussite que sur les simples individualités.

La définition d'éléments d'un de ces quatre types est donc l'objectif de tous les travaux de recherche menés en Génie Logiciel. Mes activités de recherche aboutissent elles aussi, tout naturellement, à la proposition de concepts, langages, procédés et outils.

## 2.3 Les lois de la matière Génie Logiciel

Lorsque j'examine le chemin parcouru dans le domaine du Génie Logiciel, je reconnais, dans un grand nombre de propositions faites pour maintenir sous contrôle la complexité des développements, la permanence et l'influence de trois grands principes :

- le *découpage* de problèmes compliqués pour les réduire en sous-problèmes qui le sont moins ;
- la *réutilisation* aussi fréquente que possible des artéfacts et des pratiques du développement pour éviter de répéter les efforts et profiter de l'expérience acquise ;
- la *formalisation* qui est un préalable à toute forme d'automatisation des tâches « calculables » pour les fiabiliser et en augmenter la productivité.

Sans surprise, on retrouve ici des principes qui firent le succès d'autres secteurs industriels sujets à des contraintes de complexité analogues. Ces principes sont autant de « lois » fondamentales qui régissent la façon dont émergent et se combinent les avancées du domaine. A l'origine de toutes les techniques de développement modernes, les mêmes questions : « comment appréhender cet élément compliqué en le décomposant en sous-éléments cohérents suffisamment indépendants et simples pour être appréhendés individuellement tout en assurant l'existence d'un procédé de tissage permettant de retrouver l'élément de départ », « comment abstraire d'un élément ou d'une collection d'éléments, un nouvel élément réutilisable avec profit non dans une situation unique mais dans une multitude de situations », « quel niveau de formalisation mettre en place pour assister ou automatiser à l'aide d'un outil telle ou telle tâche ».

Ces trois principes ont façonné et façonneront encore dans l'avenir, le patrimoine de la discipline. Je vais dans les trois sous-sections qui suivent rappeler leur nature.

### 2.3.1 Le découpage

Le découpage est un allié fondamental dans la course à la gestion de la complexité. Il peut permettre de séparer les préoccupations sur le plan technologique aussi bien que méthodologique et donc contribuer à diminuer les ordres de complexité. Il intervient indifféremment selon trois axes orthogonaux (Figure 2-3) :

- l'axe temporel (une activité est découpée en sous-activités) ;
- l'axe spatial (un système est physiquement découpé en sous-systèmes) ;
- l'axe des préoccupations (un système est logiquement découpé en vues).

Le *découpage temporel* est à l'origine de toutes les démarches visant à répartir les efforts sur plusieurs niveaux d'abstraction. Le *découpage spatial* rend possible en particulier le développement séparé. Le *découpage selon les préoccupations* facilite la compréhension, la vérification d'un système et participe à la factorisation de propriétés transversales sur l'axe spatial. Il est désormais courant, dans les projets de développement modernes, de voir se côtoyer les trois types de découpage.

Il est important de bien comprendre que les deux derniers types de découpage sont radicalement différents. Avec le découpage spatial le système n'existe pas en tant que tel. Il émerge de la collaboration de ses parties. A l'inverse avec le découpage selon les préoccupations, le système existe. Souvent même, il est généré plus ou moins complètement en « composant » des vues.

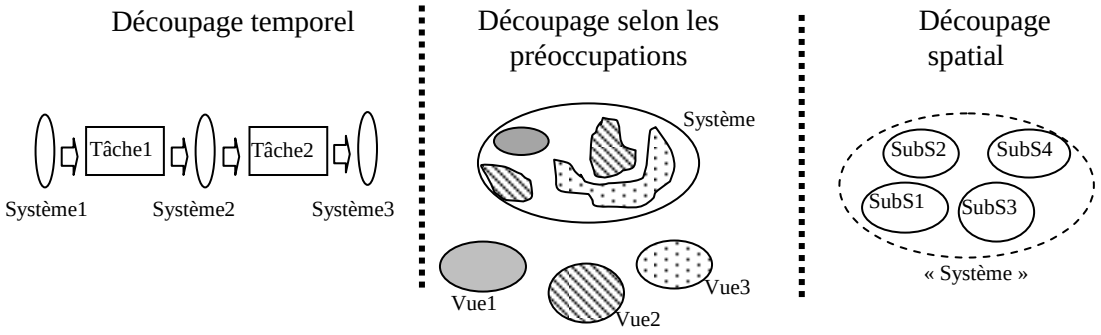


**Figure 2-3 Les trois types de découpage**

Tout découpage quel qu'il soit, pour être utile, ne doit pas substituer à la complexité du tout, une complexité des parties : éviter de transformer un système compliqué en une myriade de parties fortement interdépendantes encore plus difficiles à saisir, c'est-à-dire dont la complexité de la somme dépasserait celle du système initial. Un découpage doit donc impérativement rechercher trois propriétés.

- Un faible *couplage* : une indépendance maximum des parties, c'est-à-dire le moins de liens possibles entre eux. Le graphe connexe orienté manifestant les dépendances entre `N` parties doit avoir de préférence de `N-1` arcs (cas idéal d'une liste) que `N(N-1)` arcs (le pire des cas, avec un graphe complet). De plus, chaque lien doit avoir une « bande passante », une épaisseur aussi faible que possible. Ainsi pour une classe dépendre uniquement de l'existence d'une autre classe manifeste un couplage moins fort que de dépendre de la présence dans cette autre classe de certaines méthodes.
- Une forte *cohésion* : en regroupant au sein de chaque partie les éléments qui entretiennent des liens forts, durables et sémantiquement cohérents. Ainsi une classe regroupant un grand nombre de fonctionnalités très différentes sur le plan sémantique a une cohésion potentiellement moins forte qu'une classe ne prenant qu'une seule fonctionnalité de granularité réduite.
- Des modalités de dépendance entre parties rigoureusement définies : une description aussi précise et complète que possible des liens, pour déterminer exactement la nature des interactions entre les parties. Il est fondamental de savoir quelles sont les hypothèses sur lesquelles une partie peut compter vis-à-vis d'une autre partie dont elle dépend. Par exemple, une classe doit savoir que si elle appelle telle méthode d'une autre classe elle obtiendra tel service.

Le couplage et la cohésion ne sont pas indépendants mais contra-variants. Dans la plupart des cas, un faible couplage est concomitant avec une forte cohésion.

Plusieurs échelles de couplage avec des mesures associées ont été proposées dans la littérature pour le découpage spatial. On en trouvera une synthèse dans le cas du paradigme objet dans (Briand, et al., 1999). Le même type d'échelle et de mesure existe également pour la cohésion spatiale (Briand, et al., 1998). La définition d'échelles et de mesures de couplage et de cohésion pour le découpage selon les préoccupations est un sujet plus récent. Il n'est pas traité frontalement mais indirectement au travers de travaux s'intéressant à la cohérence (Paige, et al., 2007) et à la composition (Jeanneret, et al., 2008) de multiples points de vue à intersection non vide. Ces travaux doivent en effet formaliser les dépendances existant entre des langages et/ou des modèles. Le découpage de type temporel profite pour identifier les formes de dépendance entre activités de toute la littérature sur la modélisation de procédé de développement, les algèbres et langage de processus, etc.

La recherche d'un couplage aussi faible que possible ne doit pas laisser croire qu'il faille espérer un couplage nul. Le graphe orienté des dépendances est nécessairement faiblement

connexe. La présence de plusieurs composantes faiblement connexes doit donc alerter. Elles manifestent une perte. Car ce sont les liens, mêmes étroits, qui font émerger depuis les parties, le système. Une partie isolée ne contribue en rien. Elle ne peut s'ajouter aux autres. Il est également à noter que la granularité des parties influe sur le niveau de couplage et de cohésion. Plus il y a de parties, plus on a potentiellement de chance de diminuer le couplage et d'augmenter la cohésion. Cependant, cette multiplication accroit la complexité. Le découpage est donc un compromis entre le nombre, la taille des parties et leur niveau de couplage et de cohésion.

Tout l'objectif pour les chercheurs est de mettre sur pied des techniques permettant l'obtention de découpages pertinents. Il est par exemple reconnu qu'un découpage spatial dans le paradigme objet peut chercher à obtenir les propriétés précédentes en usant de techniques comme les types abstraits de données, le masquage d'information (Parnas, 1972) et la spécification rigoureuse des interfaces par contrat (Meyer, 2000).

Pour les trois types de découpage se pose l'épineux problème de la « recomposition », c'est-à-dire obtenir la démonstration que des parties émerge bien le système attendu. Pour le découpage temporel, on tente d'assurer la recomposition à l'aide de techniques capable de mettre en vis-à-vis les étapes du développement comme la traçabilité des exigences, la documentation anticipée des tests dans les phases en amont, etc. Le découpage spatial suppose l'usage de techniques de tests unitaires, d'intégration et de validation. Le découpage selon les préoccupations nécessite des opérateurs de composition capables de produire un unique système, cohérent, intégrant les propriétés décrites dans chacune des vues. Cette capacité à recomposer, pour tous les types de découpage, est entièrement conditionnée par l'existence d'une connaissance précise des « points d'intersection » de toutes les parties interdépendantes ; points qui constituent les lieux à partir desquels vont être entrepris les « recollements ».

### 2.3.2 La réutilisation

L'usage dans les projets d'artéfacts déjà existants et ayant fait leur preuve évitent des développements inutiles et contribuent à l'augmentation de la qualité des produits (Mohagheghi, et al., 2008). Une politique de réutilisation s'appuie sur deux circuits de développement concurrents et interdépendants :

- un cycle de développement *par la réutilisation* qui produit les artéfacts propres à chaque projet, en s'appuyant autant que possible sur des artéfacts préexistants mis à disposition dans des bibliothèques, des marchés, des ouvrages, etc. ;
- un cycle de développement *pour la réutilisation* produisant des artéfacts réutilisables mis à disposition ensuite dans des bibliothèques. Les artéfacts produits doivent présenter un certain niveau de généricité et de qualité. Ils sont éventuellement conçus en partant d'artéfacts produits dans l'autre cycle.

Ainsi pour des artéfacts de type code, le premier cycle se distingue du second par l'usage d'outils ad hoc permettant la recherche de composants logiciels, de classes, de modules ou de routines dans des bibliothèques, leur adaptation ou paramétrage, leur intégration et test. Le second intègre des outils de modélisation et de programmation spécifiques facilitant l'analyse de domaines, la définition de codes génériques et leur documentation, ainsi que des outils d'analyse statique de code et de test unitaire.

Les deux principales aptitudes qui rendent possible la définition d'artéfacts réutilisables sont d'une part l'*abstraction*, l'aptitude à ne retenir d'une chose que l'essence utile pour un contexte donné et d'autre part la *généralisation*, la capacité à identifier dans diverses situations des éléments communs et des points de variation (Krueger, 1992). L'abstraction est utilisée pour documenter efficacement les artéfacts réutilisables. La généralisation permet d'augmenter la capacité de réutilisation en explicitant pour tout artéfact une *partie fixe* et une *partie variable*. Si on prend l'exemple d'artéfacts de type code, la partie variable peut concerner les données manipulées et/ou les traitements réalisés. Dans le premier cas, il faut permettre à un même

traitement de s'appliquer à une classe plus large de données. Classiquement on introduit des super-types (abstrait) et/ou des types génériques. Dans le second cas, on introduit une hiérarchie de traitements optionnels, de traitements alternatifs (en ou exclusif), de traitements complémentaires (en ou inclusif) et des contraintes transversales régissant la forme des arbres concrets considérés comme viables.

C'est la partie variable qui détermine l'étendue des usages que l'on peut obtenir depuis un même artéfact. Cette étendue est évidemment dépendante de la puissance du langage utilisé pour l'exprimer.

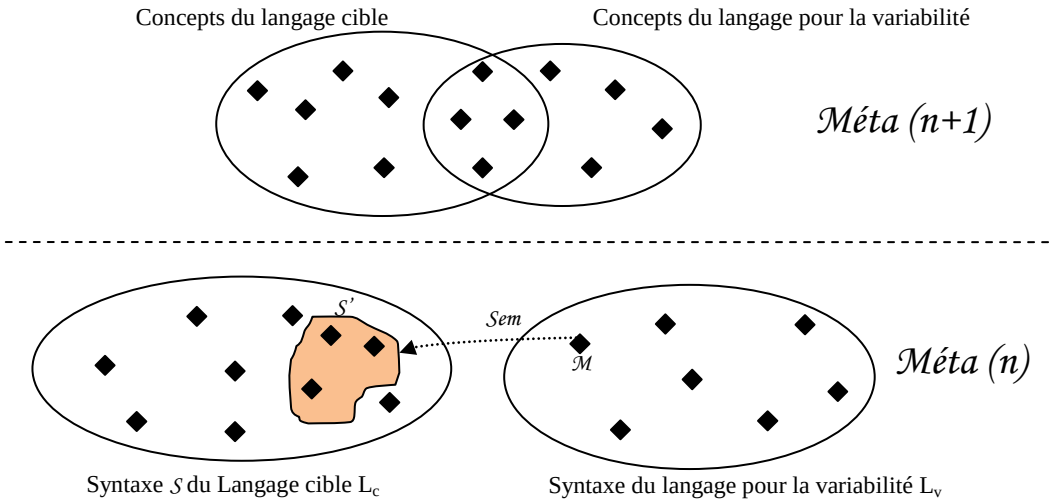


**Figure 2-4 Lien entre langage variable et langage fixe**

La partie variable ne décrit pas un système (comme pour le volet fixe) mais une collection de systèmes. Formellement parlant, un mot $\mathcal{M}$ (Figure 2-4) appartenant à la syntaxe du langage pour la variabilité $L_v$ définit un sous-ensemble $\mathcal{S}'$ de la syntaxe $\mathcal{S}$ du langage cible. Le langage $L_v$ est donc un langage capable de définir des restrictions langagières sur le langage cible. En d'autres termes, le domaine sémantique de $L_v$ est l'ensemble des parties ($\mathcal{P}(\mathcal{S})$) de la syntaxe $\mathcal{S}$ du langage cible $L_c$. $L_v$ doit en conséquence inclure, dans sa définition, une partie des concepts du langage cible : les concepts qu'il veut pouvoir contraindre. Les autres concepts de $L_v$ vont définir des opérateurs (alternative, ou inclusif, indéterminé, etc.) permettant d'agencer ces concepts pour désigner des collections de structure syntaxiques conformes à $L_c$. La puissance de $L_v$ est fonction à la fois du nombre de concepts de $L_c$ qui sont repris (quels sont les concepts pouvant varier) et du nombre et de la puissance des opérateurs d'agencement (les modes de variation offerts pour chaque concept).

Si les langages $L_c$ et $L_v$ appartiennent bien à la même couche de la pile de métamodélisation, on constate que $L_v$ opère à un niveau d'abstraction un cran plus élevé que $L_c$. L'écriture d'un mot de $L_v$ (comme $\mathcal{M}$) nécessite de classifier les traits unissant ou séparant un ensemble de système. Un mot de $L_c$, à l'inverse, raisonne sur les propriétés d'un seul système. Le niveau d'abstraction relatif par rapport à un modèle fixe est comparable à celui que l'on rencontre lorsque l'on définit par exemple des types de modèles (un type définit lui aussi une restriction langagière) ou des powertypes dans un diagramme de classes. En conséquence, la description d'un artéfact comprenant une part variable impose souvent aux développeurs de passer du niveau créateur de programmes au niveau créateur de (sous) langages. Si le langage cible embarque lui-même des constructions langagières permettant d'exprimer de la variabilité, il exige de maîtriser de concepts langagiers subtils : sous-typage, classe abstraite, polymorphisme, généricité, types de modèles, etc.

Un artéfact défini avec une partie variable ressemble à une formule de la logique du premier ordre non close. On ne peut l'utiliser tel quel. Il faut fixer ses variables libres : parmi toutes les

configurations possibles identifier celle que l'on souhaite retenir. Selon la nature de la partie variable, cette action réclame plus ou moins d'efforts : à l'exécution (le choix d'une valeur pour un paramètre lors de l'appel d'une méthode), à la compilation (le choix d'un type générique) ou plus en amont avec un processus conduisant à la génération de code. L'abstraction et la généralisation donnent de la puissance, mais une puissance chronophage, avide de compétence et d'intelligence.

La rentabilité d'une politique de réutilisation est résultante de l'harmonie avec laquelle les deux cycles décrits plus haut s'irriguent l'un l'autre. L'un des points clés est bien sûr la constitution d'un éventail d'artéfacts réutilisables répondant à deux contraintes qui se révèlent en pratique difficilement conciliables : la couverture de la plus large classe de besoin possible au coût d'adaptation le plus faible possible. Pour garantir un taux de réutilisation viable, deux options sont possibles.

La première option consiste à multiplier les artéfacts couvrant des besoins précis pour assurer cette couverture avec des efforts d'adaptation très réduit. Cependant, on se retrouve à soutenir les coûts de développement, de maintenance et d'archivage d'un nombre sans cesse grandissant d'artéfacts. Le second cycle doit fonctionner sans relâche à un rythme effréné pour abonder en artéfact les bibliothèques. Les utilisateurs doivent alors s'en remettre à des procédures de plus en plus lourdes pour parcourir des bibliothèques dont les couloirs deviennent progressivement obscurs, en quête de solutions qu'ils espèrent optimales.

La seconde option, à l'inverse, tente de limiter le nombre des artéfacts. Cela suppose, à niveau de couverture constant, de rendre encore plus génériques ces mêmes artéfacts. Or, plus un artéfact cherche à embrasser une large classe d'usages, plus il doit être abstrait et offrir des marges de manœuvre. Mais tout se paye. Plus un artéfact est abstrait, plus il faut faire d'efforts pour le rendre concret, le compléter, l'adapter au contexte d'un projet donné. Mais pire que tout, plus il est difficile de le développer, de le documenter et en corollaire pour les utilisateurs d'en cerner tout le potentiel et d'en user à bon escient.

Là encore, tout est affaire de compromis. L'obtention d'un taux de réutilisation permettant de justifier ce type d'approche au sein d'une entreprise suppose donc de déterminer de « bonnes » valeurs pour deux paramètres : le nombre et le niveau de généricité médian des artéfacts. Ces valeurs optimales, seront variables d'une entreprise à l'autre et dépendantes de plusieurs facteurs tels que ses domaines d'application ou le niveau de compétence de ses développeurs.

Il est frappant de constater à quel point la recherche incessante d'opportunités de réutilisation et l'impérieux compromis qui l'accompagne, façonnent l'ensemble des compartiments du Génie Logiciel et cela sans pour autant que nous ne le percevions comme tel : du module, à la relation d'héritage du paradigme objet, des patrons de conception, en passant par les styles architecturaux, les infrastructures, les bus logiciels, les cycles de vie, les lignes de produit jusqu'aux langages dédiés et aux méta-outils. Toutes ces avancées significatives sont la manifestation d'une volonté consciente ou inconsciente de promouvoir la réutilisation au niveau des langages, des procédés et des outils. Plusieurs ordres de grandeur d'un point de vue complexité ont pu être franchis en gardant en ligne de mire ce principe.

### 2.3.3 La formalisation

La formalisation correspond à l'injection de la rigueur mathématique dans les langages de modélisation et de programmation. Cet apport de rigueur, qui s'appuie en particulier sur la logique mathématique, va permettre d'obtenir des informations sur les artéfacts élaborés. Il est dès lors possible de « raisonner » sur des modèles ou du code, par exemple de démontrer qu'un programme ne « plantera » pas ou de manière plus générale qu'il vérifie une propriété particulière.

Cette rigueur peut n'être présente que sur une partie seulement d'un langage. Le langage diagramme de classes d'UML a ainsi certaines de ses constructions qui sont rigoureusement

définies comme par exemple la notion d'association entre classes qui est bâtie sur la notion mathématique de relation n-aire. Dans sa partie formelle, un langage devra se montrer irréprochable aussi bien au niveau de sa *syntaxe* (détaillant exactement tous les mots du langage) que de sa *sémantique* (la donnée d'une *fonction de projection* de chaque mot du langage sur un *domaine sémantique* de nature mathématique (Harel, et al., 2004)). Il est dès lors possible de procéder à des déductions sur, au plus, cette partie formelle du langage.

Si le champ d'action du raisonnement est fonction de la proportion du langage qui est formelle, la profondeur des raisonnements qui vont pouvoir être tenus est, pour sa part, liée à la richesse du domaine sémantique associé au langage. A titre d'exemple, les raisonnements réalisables sur des associations UML se limiteront à des preuves de l'existence de tel ou tel lien transitif entre instances de classes différentes ou à prouver qu'à tout instant le nombre des instances d'une classe sera compris dans un certain intervalle. La notion de relation n-aire, seule, ne peut guère conduire à des déductions plus poussées. A l'inverse, un langage usant d'un système de types construit sur un domaine sémantique usant de la logique linéaire par niveaux permet de démontrer qu'un algorithme bien typé est de complexité polynomiale… Malheureusement, plus le domaine sémantique d'un langage est riche, plus il est difficile pour les développeurs d'en comprendre les principes et donc d'en faire l'usage et encore moins d'en extraire manuellement des preuves.

Ce qui rend si attractive la formalisation, ce n'est pas tant la perspective pour quelques individus triés sur le volet de déduire des choses intéressantes à la seule force de leurs neurones, mais bien la possibilité lorsqu'on se retrouve face à des *techniques de preuve* tout ou en partie calculables, de laisser une machine le faire. Car l'outillage en prenant à sa charge une part de la difficulté, va rendre plus accessible l'usage du langage formel donc multiplier les contextes d'application. Plus l'outillage prend à sa charge de calcul, plus on a de chances de démocratiser les techniques de raisonnement. Le plus grand nombre peut en bénéficier. Si pour un même domaine sémantique plusieurs techniques de preuves sont possibles, on ne cherchera pas nécessairement à user de la plus performante mais de celle qui offre d'intéressantes possibilités d'automatisation.

Cependant, les techniques de preuve qui peuvent être déployées sur machine ont des limites. L'efficacité du *model checking* (Jhala, et al., 2009), qui consiste à vérifier des propriétés par une énumération exhaustive des états accessibles, dépend en général de la taille de l'espace des états accessibles et trouve donc ses limites dans les ressources de l'ordinateur pour manipuler l'ensemble des états accessibles. Il en est de même pour la preuve automatique de théorèmes, qui consiste à laisser l'ordinateur prouver les propriétés automatiquement, partant d'une description du système, d'un ensemble d'axiomes et d'un ensemble de règles d'inférences. Cette approche est connue pour être un problème non décidable dans le cas général en logique classique. Il existe bien des cas où le problème est décidable et où des algorithmes peuvent être appliqués. Cependant, même dans ces cas, le temps et les ressources nécessaires pour mener une preuve peuvent être inacceptables. Il est possible de pallier en partie ces limitations en proposant des outils interactifs qui permettent à l'utilisateur de limiter l'espace de travail ou de guider la preuve. Mais plus on réintroduit la variable humaine dans les calculs, plus on augmente en proportion le niveau de compétence requis pour user de ces techniques.

La formalisation est donc le moyen d'offrir une batterie d'outils théoriques permettant de tirer de l'information à valeur ajoutée des modèles et programmes. Elle ouvre des perspectives formidables sous réserve de viser in fine l'automatisation. On trouvera dans (Woodcock, et al., 2009) un état des lieux de l'usage des techniques formelles dans le monde industriel. L'enjeu des travaux de recherche est d'élever la profondeur des raisonnements et le nombre des modèles ou programmes analysables, et d'augmenter la part des activités pouvant être automatisées. Sans présumer des progrès à venir, il faut au présent, que chaque chercheur détermine en permanence pour chaque usage et type de projet, le niveau de formalisme devant et pouvant être mis en œuvre. Cela suppose de doser la richesse sémantique des concepts embarqués dans un langage, la nature du domaine sémantique associé et le type de technique de preuve que l'on utilise sur ce domaine.

L'exemple à suivre est sans doute à chercher dans le monde des langages de programmation (Pierce, 2002). Les *systèmes de type* dont sont dotés les langages actuels sont l'exemple même d'une mise en œuvre utile et utilisable de théories puissantes mais insondables pour le commun des mortels. Sur la base de quelques efforts (des déclarations de type à l'aide de quelques mots clés) qui ne nécessitent pas de connaissances théoriques particulières, les développeurs vont obtenir en retour la certitude qu'aucun appel ne restera sans cible adéquate, évitant du même coup l'occurrence d'un très grand nombre d'erreurs à l'exécution. Le compilateur masque la difficulté de la tâche et en silence empile consciencieusement des règles d'inférences compliquées. Tous les développeurs profitent, sans que l'on exige d'eux de compétences particulières, de la puissance de techniques formelles ardues.

Au fil des ans, les trois principes décrits dans la section précédente ont été déclinés, tour à tour, les uns profitant des autres, dans tous les compartiments du développement logiciel aussi bien au niveau des procédés, des langages que des outils. Les trois sections qui vont suivre dressent un état des lieux du Génie Logiciel sous l'éclairage de ces trois principes. Cet état des lieux ne prétend pas à l'exhaustivité au vu de la richesse et de la constante évolution de la discipline. Je débuterai par la description des procédés de développement actuels. Je décrirai ensuite l'état des lieux au niveau des langages puis des outils.

## 2.4 Les procédés

### 2.4.1 Les cycles de vie

Dans les années 60, des projets logiciels se sont essayés à découper l'activité de développement en étapes. L'objectif était de simplifier et de rendre visible les développements dans le but d'instaurer des techniques de gestion de projet (planification et suivi). Des points communs furent observés dans les découpages utilisés par certains projets couronnés de succès. De ces deux démarches découpage et abstraction ont émergé progressivement des *cycles de vie* comme celui en cascade de *Royce* ou en spirale de *Boehm*. Un cycle de vie décrit l'ensemble des activités et des rôles qui leur sont associés, pouvant être mis en œuvre, sous certaines conditions, par une entreprise dans le cadre du développement, de l'exploitation, de la maintenance et du retrait de certains types de logiciel.

Un cycle de vie introduit en général plusieurs types d'activités. Les *activités de production* (analyse des besoins, conception, codage, etc.) sont les plus souvent citées. L'une des activités de développement, dont l'importance est maintenant universellement reconnue, est celle de l'élaboration de l'*architecture* du futur logiciel. Une part importante de mon travail de recherche ces dernières années a concerné cette activité et les modèles qu'elle engendre. D'autres activités, toutes aussi importantes, mais plus éloignées de la production concernent : les *activités supports* (documentation, gestion de la configuration, assurance de la qualité, résolution de problème, etc.) et les *activités organisationnelles* (gestion de projet, infrastructure, amélioration de processus, formation, etc.). Les activités et les rôles apparaissant dans un cycle de vie dépendent de multiples facteurs tels que les domaines d'applications, la taille, la criticité des logiciels, la structure et la culture des entreprises cibles. D'un point de vue normatif la norme de référence est ISO/IEC 12207 (« Software Life Cycle Processes ») dont la dernière version date de 2008.

Les cycles de vie modernes préconisent des développements itératifs (la même séquence d'activité est réalisée plusieurs fois), incrémentaux (l'application est développée en plusieurs versions, chaque version étendant la version qui la précède), centrés sur les besoins du client (son implication doit être systématiquement recherchée) et sur l'architecture (la structure de l'application doit être modélisée et faire l'objet de la plus grande attention).

### 2.4.2 Procédés abstraits, concrets et réels

La notion de cycle de vie est l'illustration de l'enrichissement que procure l'entrelacement du principe de découpage et de réutilisation. Le découpage offre une réponse à la complexité. L'abstraction et la généralisation rendent utilisable le découpage non pas à un projet particulier mais à une classe plus ou moins vaste de projets. Un cycle de vie offre une trame, réutilisable à loisir, à partir de laquelle bâtir une stratégie de développement propre à chaque projet. Il décrit le « quoi faire » (les tâches à mener et les résultats attendus), laissant de côté une bonne part du « comment » (les méthodes, les techniques, les langages à utiliser), du « qui » (les ressources humaines, matérielles et logicielles affectées aux rôles) et du « quand » (les dates de début et de fin de chaque tâche et une bonne part de leurs relations de dépendance). Souvent, un cycle de vie comporte également des points de variation sémantique (un flou délibéré) et des possibilités de choix (des alternatives avec parfois des critères pour aider au choix).

Un cycle de vie n'est donc pas utilisable tel quel dans un projet. La définition d'un véritable *procédé concret de développement* pour un projet donné, sur la base d'un cycle de vie, suppose bien souvent un travail de fond : l'introduction de jalons temporels, l'affectation de ressources aux rôles et une adaptation au contexte par la résolution de tous les points de variation et de choix. Ce procédé de développement concret est matérialisé par un planning prévisionnel. Au démarrage du projet, ce procédé concret change de nature. Il va prendre vie et devenir un *procédé réel*. A l'inverse d'un cycle de vie de nature figé et générique, un procédé réel est une entité dynamique. Il évolue au gré des aléas. Les jalons, les affectations changent, de nouvelles activités apparaissent ou disparaissent, etc. Sa documentation prend à chaque instant la forme de la version mise à jour du planning complet du projet. Un procédé réel complet décrit, à un instant précis, ce qui a été, ce qui est et ce qui est planifié. Il n'est définitivement figé qu'une fois le développement terminé. Pour faire un parallèle avec le monde objet, le procédé abstrait serait une classe abstraite, le procédé concret serait une sous-classe concrète singleton et le procédé réel l'unique instance de la classe concrète.

Entre ces deux extrêmes (le cycle de vie qui est un procédé très abstrait et le procédé concret d'un projet amené ensuite à la vie sous la forme d'un procédé réel), il est possible de décliner toute une gamme de procédés plus ou moins abstraits, conçus pour répondre à des situations plus ou moins particulières (Figure 2-5). Partant d'un cycle de vie compatible avec tout type de projet, on va définir des procédés un peu moins abstraits adaptés à certaines classes de projets. Ces procédés peuvent eux-même servir de base à la définition de procédés encore plus spécifiques et ainsi de suite. Au final, une entreprise peut être amenée à maintenir un panel de procédés se distinguant les uns des autres par leur niveau d'abstraction (relatif, les uns par rapport aux autres) et par la classe des projets qu'ils supportent. Le niveau d'abstraction d'un procédé est lié à la place du procédé dans le graphe orienté ayant pour sommets les procédés et pour arcs les relations d'enrichissement. Cette relation d'enrichissement à l'instar de la relation de sous-typage est une relation d'ordre partielle. Il n'est donc pas toujours possible de déterminer le niveau d'abstraction relatif de deux procédés.

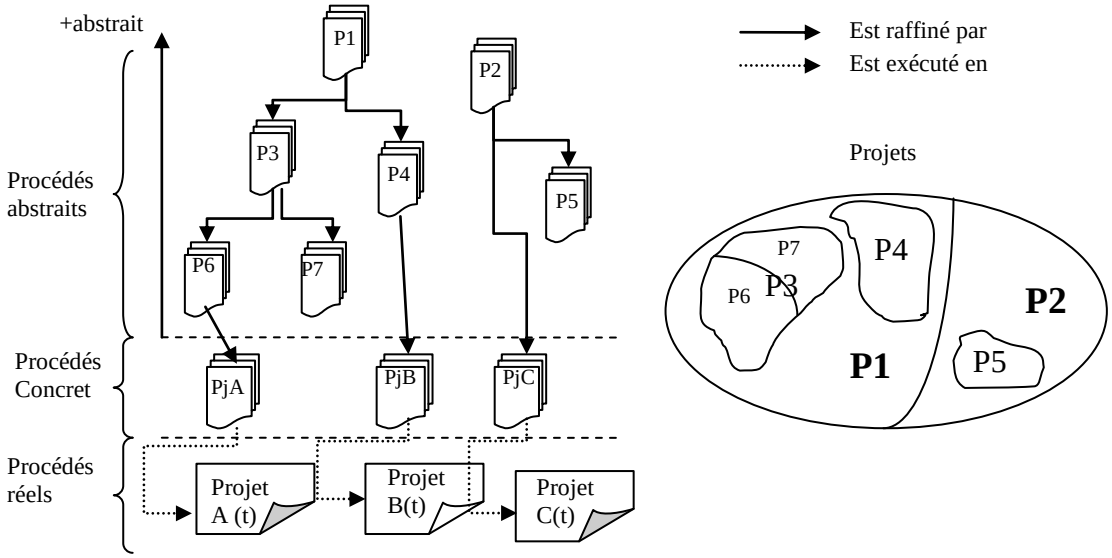


**Figure 2-5 Procédés abstraits et concrets**

### 2.4.3 Les bonnes pratiques

Les procédés se sont également enrichis au fil des ans d'une grande variété de *bonnes pratiques* (BPs), véritables patrons d'activités réutilisables et adaptables, dont l'efficacité a été constatée empiriquement. Les BPs sont des techniques, au sens le plus large, capitalisées par une entreprise ou par une communauté, dont l'efficacité a été constatée lors des projets de développement. Dans une entreprise, les BPs capitalisées proviennent de deux types de source : des sources externes à l'entreprise (en provenance d'autres entreprises par « Benchmarking », d'articles, de rapports, d'ouvrages faisant autorité, etc.) et des sources internes (par analyse de sa propre activité).

Une bonne pratique pointe une ou plusieurs activités d'un procédé. Elle vient greffer sur ces activités un (micro-) procédé interne, des métriques, des conseils, des critères, des styles de modélisation ou de programmation, etc. Certaines de ces bonnes pratiques sont, depuis plus d'une décennie, consciencieusement répertoriées dans des référentiels tels que CMMI-DEV et ISO 15504. Ces bonnes pratiques, lorsqu'elles sont documentées et capitalisées, viennent décorer les procédés décrits dans les documents qualité des entreprises. Une partie de mes travaux actuels porte sur la documentation et l'automatisation des bonnes pratiques associées aux activités de modélisation.

### 2.4.4 Documentation et exécution des procédés

Les cycles de vie et les processus de développement actuels qui tentent de respecter les référentiels en vigueur peuvent devenir tellement complexes qu'un langage dédié à leur description a vu le jour : le langage SPEM (« Software & Systems Process Engineering Meta-Model Specification », 2008) promu par l'OMG (Object Management Group), un consortium international faisant référence pour tout ce qui a trait au développement logiciel. Ce langage intègre l'ambition de promouvoir la réutilisation. Il permet de représenter des procédés à différents niveaux d'abstraction. Il permet également de définir, indépendamment de tout procédé, des éléments de procédé réutilisables : rôles, activités, etc.

La création de ce langage est un premier pas vers la définition de descriptions suffisamment précises (formelles) pour permettre leur manipulation et interprétation par des machines. L'objectif étant, dans l'avenir, de fournir comme support au développement des outils capables de mettre véritablement en action ces modèles de procédé, c'est-à-dire à même d'instancier et de contrôler le suivi des activités qu'ils décrivent.

Si le découpage a su s'imposer et la réutilisation conduire à de belles réussites dans le domaine des procédés, les prochaines années devraient consacrer la formalisation comme ambition principale. Le vieux rêve des *environnements paramétrés par les procédés de développement*, tombé en désuétude durant une bonne décennie, devrait redevenir à la mode.

## 2.5 Les langages

### 2.5.1 Langages orientés étapes

Dans chaque étape d'un procédé de développement doit être réfléchi tout ou partie du système logiciel. Les développeurs ont donc cherché très tôt à se doter de formalismes pour matérialiser le logiciel à chaque étape d'un cycle de vie ; l'objectif étant de pouvoir asseoir la communication, la réflexion, et l'analyse sur des artéfacts concrets. Dans les années 70 et 80, plusieurs « méthodologistes » ont synthétisé et généralisé ces tentatives. Ils ont défini des *méthodes de modélisation* proposant parfois un découpage plus fin de certaines activités des cycles de vie (spécification, analyse, conception, etc.) mais surtout des langages utilisables pour représenter (on préfère souvent le terme « modéliser ») le logiciel en cours de construction.

On a très tôt admis que certains systèmes, à un niveau d'abstraction donné, ne pouvaient plus être humainement appréhendés à l'aide d'un unique modèle. Ces langages se sont très vite dotés de mécanismes de découpage spatial, souvent de type hiérarchiques, de confinement de la complexité en « parties » ; ce faisant il a fallu inclure des mécanismes permettant de regrouper (modules, classes, composants, packages), de masquer (visibilité privée et publique), de décrire les interfaces (contrats) puis de connecter ces parties dans un cadre favorisant découplage et la cohésion (appel par délégation, passage de paramètres par référence, connecteurs, etc.). Ce type de progression a en particulier conduit à l'avènement des paradigmes de développement par *objets* et par *composants logiciels*. C'est dans ce dernier paradigme de développement que se sont déroulés la plupart de mes travaux.

### 2.5.2 Langages orientés « préoccupations »

Lorsque le découpage spatial se révéla à son tour insuffisant, on dut se résoudre à introduire un second axe de découpage : un découpage selon les préoccupations à l'instar de ce qui est fait dans d'autres disciplines confrontées au développement de systèmes très compliqués. De la même manière qu'une habitation est décrite par différents plans et schémas (fonctionnel, façades, ossature, câblage électrique, etc.), un système logiciel est représenté, à un certain niveau d'abstraction, non par un seul modèle mais par une collection de modèles, chacun conforme à un langage particulier, aptes à capturer certains aspects, plus ou moins vaste du système (structure logique ou physique, comportement, vue fonctionnelle, sécurité, fiabilité, etc.).

La notion de vue, en offrant une vision logique et non plus physique, permit également de réifier des préoccupations « transversales » qui se trouvaient auparavant, de fait, éclatées dans de multiples parties. Ce fut l'avènement de la *programmation par aspect* puis plus récemment de la *modélisation par aspect* (Jézéquel, 2008). Cette dernière se veut plus générale. Elle ne se cantonne plus de réifier puis de tisser uniquement des préoccupations transversales sur un code dominant, mais tous types de préoccupation, d'égales importances les unes par rapport aux autres, qu'il faut ensuite agréger pour produire un système effectif. Malheureusement la multiplication des vues pose deux épineux problèmes : d'une part celui de la cohérence de l'ensemble de ces vues qui ne sont pas toutes indépendantes et d'autre part celui de les « composer » dans le but de produire un système complet. C'est dans la mouvance de la modélisation par aspect que se préparent les outils qui aideront demain les développeurs à résoudre ces deux types de problème.

### 2.5.3 UML

Le découpage temporel et celui par les préoccupations ont eu pour conséquence néfaste d'entraîner rapidement une explosion du nombre des langages de modélisation. Pour faire face aux problèmes de communication liés à la multiplication des jargons de modélisation, la décision fut prise dans le milieu des années 90 d'élaborer un référentiel universel pour la modélisation des logiciels : UML (Unified Modeling Language) porté par l'OMG.

Ce standard n'a cessé d'évoluer depuis sa première version en 1997 pour coller au mieux aux évolutions technologiques et méthodologiques. Il détaille une collection de 13 langages graphiques utiles pour représenter les différents aspects d'un système logiciel à tous les stades de son cycle de vie. Ces langages sont presque tous qualifiés de semi-formels dans la mesure où certaines parties de leur sémantique comportent des ambiguïtés voulues (points de variation) ou non.

En tant que langage généraliste, UML peut s'avérer inadapté à certains domaines d'application. Pour pallier cette difficulté, ses concepteurs ont défini des mécanismes permettant d'adapter dans une certaine mesure UML à un contexte applicatif particulier. UML offre ainsi le mécanisme de « profil » et de nombreux points de variation sémantique explicites et implicites facilitant son adaptation. On compte à ce jour une quinzaine de profils officiellement soutenus par l'OMG ; Cela va d'un profil pour cibler la plate-forme CORBA à des profils permettant d'introduire des concepts facilitant la spécification et des documents de test.

### 2.5.4 Des langages intégrant des mécanismes de réutilisation

Si le découpage spatial et par les préoccupations a orchestré nombre de progrès dans le monde des langages de programmation et de modélisation, la recherche perpétuelle d'un taux de réutilisation toujours plus élevé a également apporté son lot de contribution. Les trois types de découpages (temporel, spatial et selon les points de vue) en multipliant le nombre des artéfacts semblaient, en théorie, à même de multiplier proportionnellement les possibilités de réutilisation. Mais, il n'en fut rien. Les artéfacts restaient désespérément liés à un contexte applicatif particulier, devenant au contraire de plus en plus spécifiques diminuant d'autant les occasions de réutilisation.

Il fallait ouvrir deux nouvelles voies : non seulement faciliter l'acte de réutilisation mais également forcer l'émergence d'artéfacts réutilisables donc transcendant les projets. Au départ, pour le premier point, on mit sur pied des constructions langagières facilitant la réutilisation de fragments de codes et de modèles « standards » : *relations d'héritage* simples ou multiples (Meyer, 2000), *mixins* (Ancona, et al., 2000) et le mécanisme de *trait* (Ducasse, et al., 2006) au niveau du code, *opérateur de merge* au niveau des modèles par exemple. Sur le second point, profitant de techniques d'*analyse de domaine*, on identifia des artéfacts dont l'usage était récurrent. Ce fut l'avènement au niveau du code des *librairies* puis la constitution de *bibliothèques* de classes ou de composants technologiques ou métiers.

Cependant, pour aller plus loin, on ne pouvait se contenter de ne réutiliser que des artéfacts standards (concrets) qui n'offrent qu'une marge de manœuvre très faible en termes de variation (par exemple au travers de paramètres modifiant le comportement de méthodes). Cette faiblesse limite la taille de ce qui peut être réutilisé, mais également les occasions de réutilisation. Une piste possible consistait à user ou à définir des mécanismes langagiers plus puissants pour permettre la conception d'artéfacts (abstraits) comportant des points de variation sémantiquement plus nombreux et plus riches. Le paradigme objet au travers des classes et des méthodes abstraites, du sous-typage et du polymorphisme contribua à l'émergence d'artéfacts plus gros : les *frameworks*.

### 2.5.5 Langages pour exprimer la variation, langages dédiés et métalangages

La généralisation à un niveau de granularité encore plus élevé (au niveau applicatif) de l'approche consistant à distinguer des points de variation est désormais l'apanage des *lignes de produits* (Clements, et al., 2001). Avec les lignes de produit, des applications, partageant des points communs mais exhibant aussi des différences, sont développées à partir de composants communs dans un domaine déterminé. La dimension de l'espace couvert par une famille de produit dépend des éléments proposés dans la partie variable du modèle décrivant cette famille. Cette dimension est directement contrainte par la puissance du langage utilisé pour établir cette part variable. Cependant, plus on augmente la puissance de ces langages, plus on diminue les chances d'obtenir une génération automatique efficace et complète d'un produit de la famille. Un compromis doit donc être trouvé entre la taille de la famille et la part d'automatisation que l'on souhaite obtenir lors de la génération d'un représentant de cette famille.

Pour résoudre ce problème, une autre technique de réutilisation à grosse granularité a été proposée : celle des *développements dédiés* (Cook, et al., 2007) dont Microsoft est le chef de file. Cette technique s'appuie sur le concept de *langage dédié* (*Domain Specific Language* ou *DSL*, (Mernik, et al., 2005)). Un langage dédié est créé de toute pièce pour permettre la représentation de la partie variable d'un produit d'une famille de produit. Le langage dédié est défini pour une famille de produit et une seule. Des générateurs de code utilisent un modèle conforme à ce langage pour produire automatiquement le code d'un produit particulier de la famille. Grossièrement, un modèle conforme au DSL détaille la part variable et le générateur de code embarque « en dur » la part commune de la famille de produits qu'il réinjecte lors de la génération de code.

Les travaux sur les lignes de produits dessinent les formes de réutilisation à grosse granularité de demain. Leur mise en œuvre réclame cependant la définition de nouveaux types de langages : des langages pour exprimer la part variable d'une famille quelconque (dont le parangon est le *feature diagram* (Schobbens, et al., 2006)*)* et des *métalangages* qui permettent de construire de manière efficace et économiquement viable les langages dédiés (dont le parangon est le langage MOF de l'OMG).

Les exemples de généralisation que je viens d'évoquer peuvent laisser penser que seul le volet fonctionnel est concerné. Mais, la généralisation peut s'appliquer selon un autre axe : l'axe non fonctionnel. Cela consiste à identifier dans les modèles et les programmes des motifs récurrents auxquels on peut associer des propriétés non fonctionnelles. Cette réutilisation de squelettes de codes ou de modèles a comme représentants les plus célèbres les *patrons de conception* (Gamma, et al., 1995) et les *styles architecturaux* (Buschmann, et al., 1996). Là encore, la documentation de ces artéfacts nécessite la mise au point de langages particuliers. Ces langages partagent les exigences des langages pour décrire la variabilité. Les motifs qu'ils cherchent à représenter sont des modèles de systèmes incomplets. On peut donc considérer qu'un motif représente la classe de tous les modèles complets qui présentent ce motif. Tout ce qui manque est variable. Les techniques applicables pour définir les langages pour la variation sont réutilisables pour mettre au point les langages de description de motifs et inversement.

La tendance actuelle est donc la multiplication des langages. Les langages généralistes côtoient dans les projets des langages hyperspécialisés. Fait nouveau, aux langages universels, partagés par un grand nombre de développeurs, viennent s'ajouter des langages propriétaires, entièrement conçus pour les besoins propres d'une entreprise voire d'un développeur. Auparavant, les langages manipulés par les développeurs se comparaient deux à deux en usant de deux critères : le point de vue de description et le niveau d'abstraction (stade du développement d'un logiciel). Avec l'émergence d'outils de réutilisation puissants, les développeurs manipulent de nouveaux types de langages : des langages pour manipuler d'autres langages ou en créer de nouveaux. Désormais deux langages se comparent selon un troisième critère : leur distance langagière, leur différence de position dans la pile de définition des langages.

## 2.6 Les outils

### 2.6.1 Des environnements de plus en plus puissants

Désormais, suite aux trois types de découpages, un développement logiciel compliqué se décompose en une myriade d'activités. Chaque activité est menée à bien par un ou plusieurs intervenants capables de capitaliser et de mobiliser un savoir spécifique. Les intervenants tendent peu à peu à devenir des experts de domaines restreints. Ils peuvent, dès lors, faire l'usage de moyens efficaces (procédés, outils et langages) qui collent au plus près à leurs préoccupations et facilitent l'obtention de solutions pertinentes. Les experts d'un domaine peuvent avec ces moyens, communiquer efficacement entre eux, comprendre, valider, modifier, développer et réutiliser des modèles les aidant dans leurs tâches. Les outils, à l'image des langages et procédés qu'ils supportent, se spécialisent à leur tour.

Les langages de modélisation les plus connus, comme UML, ont dès leur apparition bénéficiés d'outils idoines. Cependant, la plupart des langages de modélisation généralistes utilisés dans l'industrie du logiciel ont un niveau de formalisme assez faible pour faciliter leur prise en main et permettre des adaptations locales. Les outils associés à ce type de langage doivent donc se cantonner au simple rôle d'éditeur, susceptibles de réaliser des contrôles de cohérence et des transformations assez basiques.

Avec les langages dédiés, la tendance est, au contraire, de renforcer le niveau de formalisation. De plus, comme leur sémantique est plus restreinte que celle des langages généralistes, la mise au point d'outils puissants est facilitée. Les fonctionnalités offertes par ces outils concernent :

- *la création de modèles* en bénéficiant d'une assistance plus ou moins évoluée (complétion de modèles, utilisation de métriques, application semi-automatique de patrons ou de styles architecturaux, etc.) ;
- *l'analyse statique et dynamique* des modèles créés (preuve de propriétés, exécution contrôlée, test, animation de modèles, etc.) ;
- *La gestion des versions et du travail collaboratif* (différence de modèles, composition de vues différentes d'un même modèle, archivage, etc.).

De plus, comme les projets de développement intègrent plusieurs langages offrant des vues interdépendantes, il faut disposer d'outils facilitant :

- *la transformation de modèles* conformes à un langage en des modèles conformes à d'autres langages pour réaliser des imports/exports inter-langages (génération de documentations et de code, rétro-conception, etc.) ;
- *la détection et la résolution d'incohérences* entre modèles dépendants ;
- *la composition de modèles hétérogènes* pour produire de nouveaux modèles.

### 2.6.2 Les méta-outils

La définition et l'instrumentation des langages dédiés ne peuvent se contenter des outils mis au point dans le cadre des langages généralistes. En effet, l'une des exigences majeures associées à ces langages dont l'usage est, par nature, le fait d'un nombre restreint d'experts est de limiter au maximum les efforts consentis pour leur développement, pour leur outillage et pour leur apprentissage. Le respect de cette exigence est vital pour garantir la viabilité économique de ce type de langage. En particulier, on ne peut se permettre de devoir développer entièrement un environnement pour chaque DSL. Il faut donc pouvoir réutiliser, d'un DSL à l'autre, le savoir mis en œuvre pour leur instrumentation et ainsi réaliser des économies d'échelles.

Dans l'idéal, il doit être possible de générer automatiquement depuis des *méta-outils*, partant d'une description précise de la syntaxe (concrète et abstraite) et de la sémantique d'un DSL, des

environnements de développement complets : éditeurs, outils de mise au point, outils de tests, simulateurs, outils de preuve, etc. Cela nécessite au préalable de définir des langages permettant de définir d'autres langages : les *métalangages* comme MOF, ECORE ou Kermeta. Les modèles définissant un langage sont alors exploités par le méta-outil pour générer plus ou moins automatiquement un environnement de développement complet dédié à ce langage.

Un méta-outil doit donc offrir plusieurs types de fonctionnalité.

- *Edition des modèles* définissant les différents compartiments d'un langage éventuellement dans plusieurs métalangages et si possible en partant de langages existants (par exemple en composant des métamodèles) : syntaxe concrète, abstraite, sémantique, domaine sémantique, projections syntaxe concrète / syntaxe abstraite et syntaxe abstraite / domaine sémantique) ;
- *Analyse des modèles* définissant un langage : par exemple pour déterminer qu'un langage admet au moins un mot ou qu'il exclut certains mots, pour vérifier que tout mot abstrait admet bien une sémantique ou une syntaxe concrète, etc. ;
- *Génération*, partant des modèles d'un langage, d'un environnement de développement offrant une suite d'outils performants ;
- *Test* de l'environnement généré ;
- Définition, avec précision, des *liens qu'entretiennent les langages* ;
- Définition ou génération partant des liens précédents des *mécanismes de manipulation, de gestion de la cohérence et de transformation* endogènes et exogènes des modèles de ces langages ;
- *Gestion de l'évolution* de tous les artéfacts manipulés (coévolution langage/modèles, coévolution syntaxe/sémantique, etc.).

Comme plusieurs langages ou parties de langages peuvent être modélisés à l'aide d'un même métalangage, il est possible de s'appuyer sur un autre type de langage détaillant des correspondances entre modèles issus du même métalangage : *les langages de transformation*. De plus, si le langage est exécutable, on peut réaliser automatiquement des traductions d'un langage vers un autre via un interpréteur. Plus généralement, on est capable de décrire et d'exécuter des procédés produisant des modèles partant d'autres modèles. Cette automatisation procure des gains en termes de qualité et de productivité. L'écriture de ces transformations est un moyen également, pour une communauté ou une entreprise, de capitaliser des pratiques de développement.

Toutes ces avancées dans le domaine de *l'ingénierie des langages* s'appuient sur la notion de « modèle ». Cette forme émergente d'ingénierie profite des travaux du monde de l'*Ingénierie Dirigée par les Modèles* (IDM). L'IDM est la discipline qui étudie l'usage des modèles et donc des langages dans les processus d'ingénierie. Dans l'IDM, les modèles sont la matière première en entrée et sortie des activités de développement. L'unique matière sur laquelle s'exercent le talent des développeurs et la puissance des machines et des algorithmes. L'objectif est d'automatiser le plus possible les procédés de développement pour diminuer les coûts et augmenter la qualité. L'ingénierie des langages est donc une réinterprétation et un prolongement des techniques de l'IDM. La première forme d'ingénierie se distingue de la seconde par le fait que les modèles représentent des langages et non des applications logicielles. Tout l'enjeu de cette nouvelle discipline est d'arriver à définir des métalangages capables de décrire aussi complètement et rigoureusement que possible les langages, particulièrement leur sémantique, et de concevoir des plates-formes qui seront capables d'user efficacement de ces descriptions.

# Chapitre 3 Quelques définitions

Ce chapitre se fixe comme objectif de donner toutes les informations utiles à la compréhension de mes activités de recherche ; activités que je détaillerai dans les chapitres qui suivront. Il donne les définitions et les références importantes touchant les architectures et les composants (section 1), la maintenance et l'évolution (section 2), l'ingénierie des langages (section 3).

Ne souhaitant pas noyer le lecteur sous une multitude indigeste de définitions et de références, j'ai délibérément choisi de privilégier la concision et j'espère la précision. Sur le plan bibliographique, je n'ai retenu que les écrits ayant indiscutablement marqué chaque domaine ou qui fournissent une synthèse de qualité.

## *3.1 Architecture et composant logiciel*

### 3.1.1 Architecture

L'avènement de la notion d'architecture d'un logiciel est le reflet de la prise de conscience que, dès le démarrage d'un projet, puis tout au long du cycle de vie, sont prises des décisions importantes qui concernent aussi bien la structure et les propriétés du futur logiciel, que les stratégies et les technologies à mettre en œuvre pour son développement. Il était donc important de fournir un support aux communications, réflexions et analyses nécessaires à une prise de décision éclairée.

Depuis plus de 20 ans, de nombreux articles et ouvrages se sont proposé de définir ce qu'est une architecture logicielle. On se reportera à (Shaw, et al., 2006) pour obtenir un historique de la discipline. Longtemps aucune définition n'a fait l'unanimité. Un ouvrage a contribué à l'émergence d'un consensus, celui de (Bass, et al., 2003). La définition que je retiens ici doit sans doute beaucoup à la vision de ces auteurs. Il s'agit de la définition donnée par la première norme dédiée à ce thème : ISO/IEC 42010 (« Recommended practice for architectural description of software-intensive systems », 2007). La définition proposée me semble être la plus générale et actuellement la plus consensuelle.

*The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.*

La *description* que l'on donne d'une architecture offre une vision à haut niveau et synthétique de la structure et du comportement interne d'un logiciel. Mais cette vision replace également le logiciel dans son environnement et dans le cadre des besoins qu'il cherche à combler. Contrairement à ce que l'on pense trop souvent, cette description ne peut être figée une fois pour toutes en début de projet, mais doit continûment refléter avec un niveau de synchronisme suffisant l'architecture réelle du logiciel, qui elle, évolue sans cesse. Car une architecture est la résultante d'une négociation permanente. La progression impose souvent des optimisations, des remises en cause que l'on ne peut interdire. La représentation à jour d'une architecture est, à chaque instant, le conseil d'une gouvernance éclairée ; gouvernance s'étendant autant sur les aspects techniques que sur la gestion des risques, des délais et des coûts, contrôlant toujours, se montrant inflexible ou au contraire compréhensive si nécessaire. La représentation est tour à tour table de loi ou table de travail. Elle est la marque des décisions passées et doit être le point d'entrée des décisions à venir.

### 3.1.2 Documenter l'architecture d'un logiciel

Comme le montre le diagramme de classes de la Figure 3-1, la description d'une architecture est une documentation qui prend concrètement la forme d'une collection de *modèles*. Ces modèles sont regroupés en *vues*. Un même modèle peut participer à plusieurs

vues. Une vue est une partie de la description de l'architecture qui se conforme à un certain *point de vue*. Les points de vue rencontrés en général portent sur la définition des modes d'utilisation du système, sur sa structure logique, sur son comportement interne, sa structure physique ou son déploiement. L'une des premières tentatives de classification des points de vue est la « 4+1 » de (Kruchten, 1995).

Un point de vue décrit la manière (langages, procédés, outils) de construire et d'utiliser une vue. Un point de vue répond à certains *centres d'intérêts* : les missions du système, les risques associés à son développement, etc. Le choix des points de vue (et donc des vues) à considérer est fonction des *intervenants* sur le projet de développement : un acquéreur, un utilisateur, un chef de projet, un architecte, un développeur, un évaluateur, un certificateur, etc. En effet, chaque type d'intervenant a, vis-à-vis de l'architecture, un ou plusieurs centres d'intérêts.

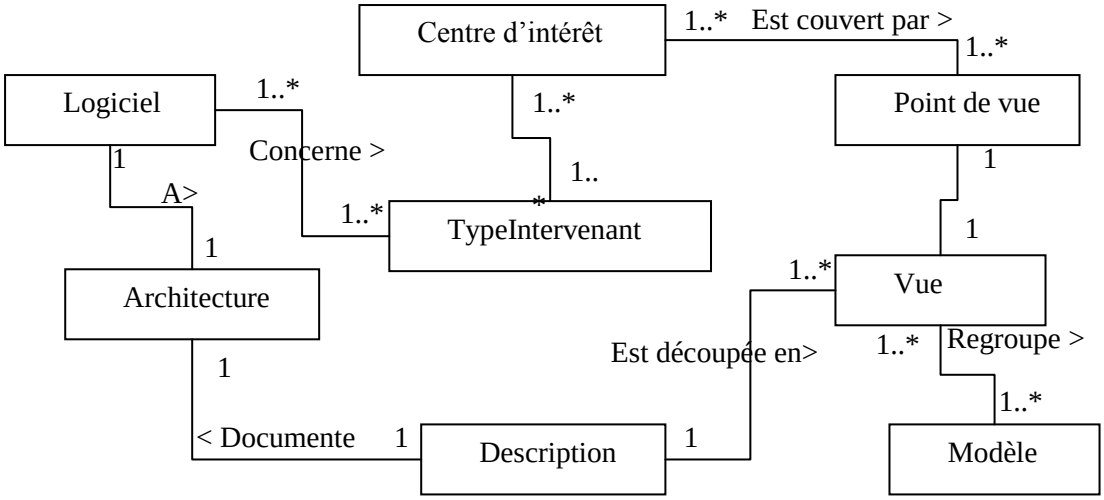


**Figure 3-1 Le domaine des architectures logicielles (inspiré de la norme ISO 42010)**

Les modèles utilisés peuvent être plus ou moins formels : du simple bout de texte, au diagramme informel dessiné à la main usant de quelques boîtes et flèches, en passant par un diagramme de composants UML, jusqu'à du code écrit dans un langage formel. Les langages utilisés pour produire ces modèles sont appelés : *Langages de Description d'Architecture*, acronyme LDA. On préfère souvent utiliser le terme anglais équivalent d'*ADL*.

On trouve dans la littérature de nombreux ADL. On se reportera à (Medvidovic, et al., 2000) pour une analyse comparative de plusieurs d'entre eux. Les ADL couvrent plus ou moins bien les différents points de vue d'une architecture. Cela explique, en partie, leur empreinte mitigée dans l'industrie. Il est difficile pour un ADL de répondre à l'ensemble des préoccupations même dans un domaine applicatif donné. De plus, l'usage de plusieurs ADL pour assurer cette couverture est difficile car ils sont souvent incompatibles deux à deux sur le plan de leur syntaxe et de leurs outils (Dashofy, et al., 2005).

Si on se limite au seul point de vue structurel d'une architecture, les ADL usent essentiellement :

- d'éléments représentant les unités de calcul et de sauvegarde des données dans ce système, appelés communément *composants* qui affichent leurs services et besoins au travers d'*interfaces*. Ces composants peuvent être dotés d'un état à l'instar d'un objet, mais ils se distinguent de ces derniers par leur granularité plus importante et par le fait qu'ils ne sont généralement pas créés ou détruits durant l'exécution d'un logiciel.

- d'éléments représentant les différents types d'interconnexions, potentiellement n-aires, existant entre les unités précédentes, appelés *connecteurs*. Les connecteurs affichent à chacune de leurs extrémités des *rôles*. Les connecteurs peuvent manifester des liens implicites (un appel de procédure) ou explicites (une connexion réseau). Ils peuvent être simples (une file d'attente de messages ou un point d'entrée dans une API) ou compliqués (un middleware). Dans ce dernier cas, la frontière avec la notion de composant devient floue. La différence repose sur le fait qu'un connecteur, s'il peut altérer la syntaxe des données qu'il véhicule, ne change en rien leur sémantique à un niveau applicatif.

Une vue structurelle prend alors la forme d'une *configuration* indiquant comment sont arrangées des *occurrences de composants* au travers de *liens* manifestant des *occurrences de connecteurs*.

Les langages associés au point de vue comportemental utilisent une gamme plus large de concepts : évènement, processus, état, opération, tâche, etc. Le point de vue déploiement s'appuie sur des concepts tels que les nœuds logiciels ou matériels, les liaisons point à point ou partagées, etc.

### 3.1.3 Décision architecturale

Très tôt, il a été noté l'importance de ne pas restreindre une architecture à la simple donnée de « ce qui est ». En effet, ce serait oublier qu'un modèle n'est que le point final d'un processus ayant connu une multitude de décisions manifestant des compromis. Des informations essentielles qui expliquent pourquoi tel ou tel motif apparaît dans une architecture sont invisibles dans les modèles finaux. Or, toutes ces informations sont essentielles à la compréhension, l'évaluation et à l'évolution d'une architecture. Leur importance a été soulignée très tôt par Perry et Wolf (Perry, et al., 1992) lorsqu'ils ont proposé leur équation :

*Software Architecture = {Elements, Forms, Rationale/Constraints}*

Cette équation mettait en évidence, il y a près de 20 ans, l'égale importance des éléments composant une architecture (composants, connecteurs), des motifs décrivant leur agencement (leur configuration) et des raisons et contraintes qui sont à l'origine des choix architecturaux réalisés. Mais comme le souligne (Kruchten, et al., 2009) dans un historique de la discipline, la documentation de ce troisième volet n'a véritablement intéressé les chercheurs que dans le milieu des années 2000. Ce manque d'intérêt est sans doute lié au fait que les acteurs principaux d'un projet, les clients et les développeurs, ne sont pas intéressés par ces détours. Ils ne réclament que des visions claires et précises concernant respectivement leurs besoins et les solutions à développer. Mais c'était oublier l'importance d'autres acteurs qui eux subissent l'incomplétude de ces modèles « instantanés ». Les évaluateurs et les personnels chargés de la maintenance souffrent de ne pas disposer du film complet. Ce film qui explique le pourquoi et les circonstances ayant conduit au modèle qu'ils ont sous les yeux et avec lequel ils doivent raisonner.

La prise de conscience s'opère en 2005, lorsque l'on reconnaît aux *décisions architecturales* un rôle central. On va même jusqu'à dire qu'une architecture est un ensemble de décisions architecturales (Jansen, et al., 2005). A la même époque, une étude met en valeur chez les architectes et les développeurs la prise de conscience de leur importance (Tang, et al., 2006). Dès lors, des dizaines d'articles vont paraître sur le sujet vantant les mérites de telle ou telle démarche de développement, de tel ou tel langage de documentation ou outil support (une étude comparative de 5 outils est faite dans (Tang, et al., 2010)) élevant au rang d'entité de première classe ce type de connaissance. C'est le sujet à la mode dans la communauté des architectures logicielles. Un workshop se consacre entièrement à ce thème depuis 2006 en marge de la conférence ICSE : « Workshop on SHAring and Reusing architectural Knowledge » (SHARK).

Mais cette connaissance de haut niveau qui mêle des informations aussi différentes que des « hypothèses », des « motivations », des « problèmes », des « alternatives », de multiples

« choix architecturaux », des « liens de dépendances » ou des « états » a longtemps échappé à toute définition universelle. Elle a été désignée sous de multiples termes : intention (rationale), décision architecturale (Architectural Decision), décision de conception (Design Decision), décision de conception architecturale (Architectural Design Decision), connaissance architecturale (Architectural Knowledge), etc. De plus, les auteurs ne mettent pas toujours derrière ces termes les mêmes informations. Le dernier terme, en particulier, semble le plus riche, englobant tous les autres.

Je retiendrai pour ma part la définition de décision architecturale proposée par (Jansen, et al., 2005).

*A description of the set of architectural additions, subtractions and modifications to the software architecture, the rationale, and the design rules, design constraints and additional requirements that (partially) realize one or more requirements on a given architecture.*

De nombreux formats de documentations des décisions architecturales ont été proposés. On citera en particulier (Tyree, et al., 2005), (Jansen, et al., 2005), (de Boer, et al., 2007), (Capilla, et al., 2007), (Cui, et al., 2008). Dans ces formats, une décision est décrite à l'aide de canevas comportant un nombre très variable de champs (par exemple 14 pour Tyree et Akerman) obligatoires ou optionnels. Certains champs facilitent la gestion des décisions : leur nom, leur état (en attente, décidé, approuvé, etc.), leur classification en groupes (celles concernant les données, l'interface homme machine, etc.), leur liens (variés) de dépendances. Mais le cœur de la description porte sur :

- l'*objectif* d'une décision (en particulier les propriétés non fonctionnelles dont elle cherche l'obtention) ;
- le détail du *contexte* dans lequel la décision fut prise (technologique, les standards à respecter, les contraintes de coûts et délais à respecter, les principes de conception en vigueur dans l'entreprise, etc.) ;
- les différentes *solutions* qui ont été examinées ;
- la présentation de *la solution retenue* (en particulier les motifs architecturaux relevant de la décision dans les modèles de l'architecture, plus généralement les contraintes qu'elle impose sur la forme de l'architecture, ses points forts et faibles) ;
- l'*argumentation* qui explique pourquoi cette solution fut retenue parmi toutes celles envisagées (éventuellement les critères de jugement et un classement des solutions) ;
- Les *conséquences* de la décision qui peut faire émerger de nouveaux objectifs (donc de nouvelles décisions) ;

Dans ce mémoire, j'userai du terme « décision architecturale » pour désigner un couple : *choix architectural* (le quoi, un motif mis en place dans une architecture), *propriété non fonctionnelle* (le pourquoi, la raison qui a conduit à intégrer le motif dans une architecture). Je reconnais cependant que cet usage, au demeurant correct sur un plan sémantique (une décision documente bien les raisons pour lesquelles le modèle est ce qu'il est), ne reflète pas toute la richesse du terme tel que le définit la littérature.

### 3.1.4 Composant

Il existe de très nombreuses définitions de la notion de composant logiciel. Les partisans de la réutilisation usent de cette notion pour désigner tout artéfact réutilisable. A l'inverse les promoteurs des composants sur étagères (Commercial Off-The-Shelf, COTS) limitent cette appellation aux seuls COTS. Les méthodologistes assimilent les composants à des unités de développement pouvant faire l'objet d'un livrable intermédiaire ou d'une gestion de configuration. Les architectes désignent avec cette notion des abstractions architecturales. Ce problème est en partie lié au fait que la notion de composant trouve matière à se projeter sur des éléments à la fois du monde de l'exécution (par exemple les objets, les processus), que du déploiement (des fichiers binaires, des classes), ou de la conception (des abstractions exprimant des contraintes architecturales).

Je retiendrai pour ma part la définition donnée par Clemens Szyperski (Szyperski, 2002).

*A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*

Cette définition suppose en particulier qu'un composant affiche les services qu'il rend au travers *d'interfaces fournies* et précise ses besoins au moyen *d'interfaces requises*. Un composant ne peut interagir avec l'extérieur (d'autres composants) qu'au travers de ces interfaces. Une interface regroupe des *opérations*. Une opération représente un comportement atomique que propose ou requiert un composant.

### 3.1.5 Documentation utilisateur d'un composant

Parce qu'un composant a souvent pour vocation d'être réutilisé « tel quel », comme une unité indivisible dont on ne connaît pas la structure interne, il doit être accompagné d'une documentation suffisante pour qu'une tierce personne puisse le comprendre et l'utiliser à bon escient. La complétude et la précision des informations contenues dans la documentation d'un composant conditionnent fortement sa capacité à être utilisé et remplacé si besoin.

Un composant offre à ses utilisateurs un certain nombre de *services* au travers de ses opérations. Les services fournis correspondent aux informations et comportements que peuvent attendre les clients. Ils manifestent ce que le composant est capable de faire pour eux, par exemple : imprimer une fiche de paie, afficher tous les trains reliant deux destinations à une date fixée, calculer les composantes connexes d'un graphe non orienté, etc. Un service est la sémantique dénotationnelle d'une opération : une fonction au sens mathématique. Les services fournis sont garantis sous réserve que le composant puisse lui-même s'appuyer sur des ressources externes qu'il doit clairement expliciter : les services requis. Les services fournis et requis constituent les *propriétés fonctionnelles* d'un composant.

Les propriétés fonctionnelles donnent d'un composant une image idéale : une collection de fonctions au sens mathématique. Mais lesdites fonctions ne sont pas des objets mathématiques définis par des équations. Elles sont, dans les faits, implantées par des programmes s'exécutant sur des machines. Un univers dans lequel les calculs exigent du temps et des ressources matérielles, dans lequel les algorithmes sont plus ou moins corrects, fiables et compliqués. Une opération ne peut donc être réduite à sa seule dénotation. Il faut également préciser la façon dont la fonction, que cette opération implante, est rendue (si elle est fournie) ou attendue (si elle est requise). Toutes ces informations replacent dans un contexte opérationnel les fonctions associées aux opérations. Elles constituent les *propriétés non fonctionnelles* d'un composant.

Les propriétés non fonctionnelles précisent les conditions dans lesquelles va opérer un composant, certaines de ses interfaces ou de ses opérations. Elles intéressent tout aussi bien les architectes (performance, fiabilité, convivialité, sécurité, etc.), les administrateurs (sécurité, configuration logicielle et matérielle requise, taille en mémoire disque, modes d'installation et de paramétrage, etc.), les acquéreurs (prix, garanties, contrats de maintenance, etc.). L'espace des propriétés non fonctionnelles est très vaste. Certaines de ces propriétés non fonctionnelles ne s'observent qu'à l'exécution (performance), lors d'une demande d'évolution ou de correction (maintenabilité et portabilité), d'autres s'apprécient sur les binaires (place en mémoire disque) et les documents associés (convivialité, prix, contrats de maintenance). Les propriétés non fonctionnelles entretiennent entre elles des liens variés. Certaines propriétés sont dépendantes les unes des autres (de manière covariante ou contra-variante). On trouve à ce titre, dans la littérature, des matrices de corrélation pour certaines d'entre elles.

Il est très important de ne pas confondre les propriétés que je viens d'évoquer, avec les *propriétés non fonctionnelles internes* qui sont, elles, perçues en parcourant uniquement le code et les documents de conception. Les seules propriétés utiles aux utilisateurs d'un composant en mode « boîte noire » sont les *propriétés non fonctionnelles externes* : toutes les propriétés observables en exécutant un logiciel ou en examinant les documentations fournies aux

utilisateurs. La partie de la documentation d'un composant à destination des utilisateurs d'un composant doit impérativement se limiter à tout ou partie des propriétés fonctionnelles et non fonctionnelles externes et ne doit, en aucun cas, laisser entrevoir des propriétés non fonctionnelles internes (sans intérêt). Plus encore, certaines propriétés fonctionnelles ou non fonctionnelles, pourtant externes, peuvent volontairement être exclues car elles sont la conséquence de choix d'implantation dont on ne veut pas faire dépendre les utilisateurs. On garde ainsi des possibilités d'évolution, sous réserve que les utilisateurs ne les découvrent et les usent sciemment ou n'en profitent sans même sans douter.

Idéalement, la documentation d'un composant doit donc détailler (Beugnard, et al., 1999) :

- sa *capacité fonctionnelle* : la signature de chaque opération fournie et requise (syntaxe d'appel), leur sémantique d'exécution (sa dénotation, par exemple sous la forme de pré-post conditions), les conditions d'activation et de parallélisme des services (sous la forme par exemple de machines à états, de code CSP de *Hoare* ou pi-calculus de *Milner*, etc.)
- sa *capacité non fonctionnelle* : en particulier les attributs qualité (performance, fiabilité, etc.) associés à ses artéfacts externes (interfaces et/ou opérations fournies et requises).

Parce que les composants sont souvent des boîtes noires exploitables dans des contextes non prévisibles, les volets « sémantique » et « non fonctionnel » prennent une importance considérable. Ils ne peuvent être négligés sans conséquences.

### 3.1.6 Prédiction et obtention des propriétés fonctionnelles d'un système

Tout système logiciel doit respecter des propriétés non fonctionnelles externes. Si le développement d'un système est le résultat d'une agrégation de composants, il n'est pas concevable de s'en remettre au hasard pour garantir ces propriétés. L'architecte doit en toute conscience choisir les composants idoines et les configurations à appliquer pour faire « émerger » les propriétés non fonctionnelles externes du système.

Il doit pour cela disposer :

1. d'informations précises sur les aptitudes non fonctionnelles externes des composants qui lui sont proposés ;
2. de mécanismes permettant de déduire de ces informations (plus ou moins complètes) et des configurations mises en place, les propriétés non fonctionnelles externes que l'on peut attendre du futur système ;

La proposition de langages permettant d'assurer le premier point a connu quelques avancées. Cependant l'expression de telles propriétés, déjà très difficile pour une application, est encore plus délicate pour un composant. En effet, par définition un composant est utilisable par des tiers : il est donc impossible de connaître son contexte d'exécution. La spécification de la plupart des propriétés non fonctionnelles externes devrait se faire sous la forme d'une fonction paramétrée par un contexte. Ce contexte identifie les valeurs ou des intervalles de valeur quantifiant l'impact des ressources de l'environnement sur la propriété du composant. On trouvera dans (Zschaler, 2009) un état de l'art assez complet sur la question ainsi que la présentation d'un langage dédié à ce type de documentation.

En revanche, les mécanismes de prédiction des propriétés non fonctionnelles externes nécessaires au second point ne sont encore que balbutiants. On distingue deux types d'approche complémentaires : celle qui repose sur l'hypothèse que les composants donnent des informations précises sur leurs propriétés et celle qui au contraire suppose que l'on ne dispose pas d'informations complètes ou fiables (Cheung, et al., 2008). A ce jour, les deux approches cantonnent leur prédiction à une propriété non fonctionnelle externe particulière (fiabilité, performance, etc.) pour des composants respectant des hypothèses fortes (composant sans état avec un flot de contrôle de type séquentiel par exemple). La seconde approche semble la plus

pragmatique des deux. Il est en effet fréquent que l'on ne dispose pas d'informations précises sur chaque composant.

### 3.1.7 Les propriétés qualité

Un sous ensemble des propriétés non fonctionnelles externes joue un rôle fondamental : les propriétés qualité. On parle également de qualité de service dans le monde des applications en réseaux. C'est à travers ces propriétés particulières que l'on peut spécifier ou évaluer le niveau de satisfaction d'une personne ou d'un groupe de personnes.

La norme de référence ISO 8402 définit la qualité comme étant :

« *L'ensemble des caractéristiques d'une entité qui lui confèrent l'aptitude à satisfaire des besoins exprimés et implicites* ».

Cette définition est doublement générique. Il faut préciser la qualité « de quoi » (d'un produit, d'un service, d'une activité de développement, etc.) et les besoins « de qui » (un utilisateur, un chef de projet, etc.). Ainsi, les caractéristiques identifiant la qualité ne sont pas les mêmes selon que l'on se place du point de vue d'un utilisateur, d'un programmeur, ou d'un décideur client. Alors que le premier considérera des caractéristiques portant sur tous les aspects visibles du logiciel livré tels que sa facilité d'utilisation ou ses performances (on parle de *qualité externe*), le second mettra en avant des caractéristiques telles que la testabilité ou la lisibilité du code qu'il conçoit (on parle de *qualité interne*). Le troisième s'intéressera pour sa part à des caractéristiques telles que les délais et les coûts de réalisation ou les garanties d'assurance de la qualité affichées par le fournisseur (certification ISO 9001 par exemple). Dans ce dernier cas, on note que les caractéristiques définissant la qualité portent sur le processus de développement du logiciel et non plus sur les produits issus de ce processus.

La notion de qualité diffère donc non seulement pour chaque individu ou groupe d'individus mais aussi selon l'entité que ces individus sont amenés à considérer. C'est pourquoi on trouve dans la littérature de multiples définitions de la qualité. Cette diversité se retrouve aussi au niveau des standards (IEEE, DoD, NAT, etc.) et des normes qualité (ISO, AFNOR, etc.). On distingue ainsi des normes et standards portant selon le cas sur le logiciel livré (qualité externe), sur les produits de la conception (qualité interne), sur des activités de développement, sur le processus de développement (ISO 12207) et sur le système qualité (ISO 9000-3 pour la déclinaison au logiciel d'ISO 9001, ISO 15504 « SPICE » et CMMI-DEV pour l'évaluation du niveau de maturité). Je me limiterai ici à la qualité afférente au produit logiciel.

Encore récemment, deux normes faisaient référence dans le domaine de la qualité du produit logiciel ISO 9126 (« Software product quality ») et ISO 14598 (« Software product evaluation »). Depuis 2005, ces deux normes complémentaires et parfois en désaccord ont été intégrées pour former la norme ISO 25000 SQuaRE (« Software product Quality Requirements and Evaluation »). Cette (famille de) norme propose un cadre unique pour spécifier la qualité d'un produit logiciel, la mesurer et l'évaluer. C'est une norme « 3 en 1 » répondant aux préoccupations, des acquéreurs, des développeurs et des évaluateurs.

Dans la suite de ce mémoire, lorsque j'évoquerai des propriétés qualité, je le ferai toujours en référence au modèle de la qualité ISO 9126 (repris par ISO 25000). Ce modèle propose de spécifier ou d'évaluer la qualité selon 3 niveaux : en usage, en externe ou en interne.

- La *qualité en usage* est celle qui est perçue par les utilisateurs dans le cadre de leurs activités quotidiennes.
- La *qualité externe* est celle qui peut être observée en exécutant le logiciel sur une plate-forme de test.
- La *qualité interne* est celle perçue par les développeurs en parcourant les artéfacts associés au logiciel.

L'objectif est, bien sûr, d'assurer au fil du développement le niveau de qualité en usage en garantissant un certain niveau de qualité externe, lui-même obtenu en respectant un certain

niveau de qualité interne. Pour des composants « boîtes noires », réutilisables dans des contextes inconnus, seul le second niveau est pertinent. A ce niveau, la qualité (externe) d'un composant (ou d'une de ses interfaces ou opérations) est spécifiée selon différents axes. Ces axes sont les nœuds d'un arbre de hauteur quatre. Sous le nœud racine, on distingue six *caractéristiques* : la capacité fonctionnelle, la fiabilité, la facilité d'utilisation, le rendement, la maintenabilité, la portabilité. Ces six caractéristiques sont affinées en *sous-caractéristiques*. Le standard en donne une liste indicative. Ces sous-caractéristiques peuvent à leur tour, si nécessaire, être découpées en *attributs*. Mais le standard laisse le libre choix de ces découpages aux entreprises.

Un axe va être spécifié quantitativement à l'aide d'une ou plusieurs *métriques*. Une métrique est une fonction qui prend en entrée un logiciel (au sens large, cela peut être un modèle de conception, un listing de code source, un programme en cours d'exécution) et fournit en retour une ou plusieurs valeurs tirées d'un ensemble muni d'une relation d'ordre totale. La spécification du niveau de qualité d'un composant selon un axe particulier consiste à indiquer, pour chaque métrique de l'axe, la ou les valeurs obtenues par le composant. Selon le niveau de qualité auquel la mesure s'opère on parle de *métrique en usage* (sur le système dans son environnement d'exploitation), *métrique externe* (sur le système en exécution) et *métrique interne* (sur les artéfacts du système). Pour un composant, seules les métriques internes et externes peuvent être utilisées.

### 3.1.8 Modèle et infrastructure de composants

Un composant peut être amené à interagir avec d'autres composants qui ne sont pas connus à l'avance. Mais pour que des composants quelconques puissent être composés les uns aux autres dans le but de concevoir de nouveaux services, il est nécessaire qu'ils puissent se comprendre, partager un « protocole » d'interaction commun. Cet ensemble de points d'accord est appelé un *modèle de composants*.

Un modèle de composants définit un ensemble de contraintes qui doit garantir que des composants développés en toute indépendance dans des lieux différents par des personnes différentes mais respectant tous ces contraintes pourront interagir. En particulier, un modèle de composants fixe des composants acceptables. Il serait donc plus correct de parler de « langage » plutôt que de « modèle » de composants. Mais le terme de « modèle » est maintenant passé dans le vocabulaire commun.

Cet aspect langage se retrouve d'ailleurs dans les informations dont font état les modèles de composants. Il détaille au minimum les informations qui suivent.

- La syntaxe et la sémantique des composants : la façon dont ils doivent être construits (par exemple qu'ils affichent obligatoirement certains interfaces) et décrits (en usant d'un langage de description des interfaces qui peut ou non être différent du langage de programmation du composant lui-même). La façon dont la syntaxe des composants doit être comprise, généralement il s'agit de fixer la signification de concepts tels que : interface, port, conteneur, etc. ;
- La syntaxe et la sémantique de la composition : localisation, mode de contrôle du flot d'exécution, protocole de communication, format d'encodage des données, etc.
- Les ressources universelles à disposition des composants et la manière d'en profiter.

On trouvera dans (Lau, et al., 2007) une analyse comparative de 13 modèles de composants.

Un modèle de composants n'est qu'une spécification. Une *infrastructure* fournit une implantation concrète d'un modèle de composants pour une architecture matérielle et logicielle cible. L'infrastructure va fournir le support aux composants pour qu'ils puissent remplir leurs missions et interagir dans le respect du modèle de composants.

### 3.1.9 Procédés de développement avec les composants

Le paradigme composant propose de construire un système à partir d'éléments faiblement couplés et pour la plupart déjà existants. L'un des nombreux intérêts de ce paradigme est

d'obtenir des taux de réutilisation beaucoup plus élevés que ceux constatés en usant des autres paradigmes de développement. Or, plus ces taux sont élevés, plus les délais et les coûts sont réduits. Simultanément, le niveau de qualité des applicatifs augmente. Cependant, l'obtention de taux de réutilisation rentables dans ce paradigme peut ne rester qu'un vœu pieux. En effet, il est indispensable à chaque étape du développement, dès qu'un besoin apparaît, de systématiquement chercher s'il n'existe pas, quelque part, un composant déjà écrit et testé, capable de le satisfaire tout en s'intégrant à moindre coût dans l'architecture logicielle en cours d'élaboration. Il s'agit ici d'adopter au sein des démarches de développement, des réflexes, plébiscitant résolument le développement par la réutilisation. Le développement d'une application en usant de composants logiciels diffère radicalement d'un développement « classique » (Crnkovic, et al., 2006).

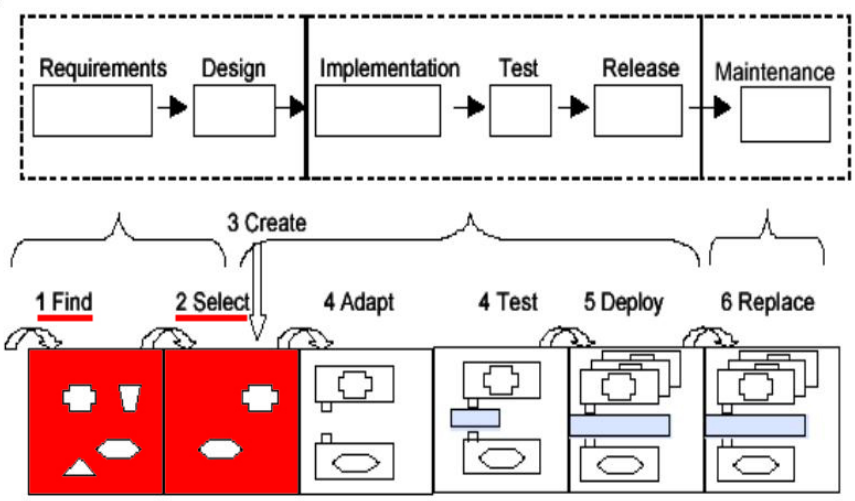


**Figure 3-2 Le procédé de développement d'un composant**

Lors de la modélisation de l'architecture, émergent un certain nombre de composants. Quelques-uns de ces composants ont été placés dans le modèle d'architecture sciemment car le concepteur sait qu'une implantation concrète existe. Les composants restants vont faire l'objet d'une recherche spécifique dont le but sera de sélectionner dans les marchés et bibliothèques un composant concret capable de se substituer intégralement à eux à coût d'adaptation et d'intégration le plus bas possible. L'intégration d'un composant concret à la place d'un composant conceptuel va, peut-être, imposer une modification du modèle d'architecture. Ces modifications peuvent concerner les composants conceptuels restants ou l'ajout de code supplémentaire (glue) pour permettre une collaboration effective avec des composants concrets. La construction d'une application à base de composants est donc incrémentale et itérative et implique la recherche d'un seul composant conceptuel à la fois[1].

On le voit avec ce type de procédé, les efforts liés à l'implantation s'effacent au profit de ceux liés à la localisation, à la sélection de solutions existantes, à leur éventuelle adaptation, à leur qualification et à leur intégration (Figure 3-2).

[1] Certains travaux voient au contraire l'élaboration d'une architecture comme un problème d'optimisation (NP-complet) consistant à identifier automatiquement depuis une base de composants, une configuration répondant aux besoins affichés tout en minimisant une certaine fonction de coût. Ce type d'approche globale a montré son efficacité lorsque l'architecte ne dispose d'aucune « intuition » concernant la solution. Mais son principal inconvénient est de ne pas tenir compte des propriétés non fonctionnelles. Celles-ci en effet ne peuvent être déduites avec les moyens actuels, en général, des propriétés non fonctionnelles des composants de la configuration.

## 3.2 Maintenance et évolution des logiciels

### 3.2.1 La maintenance des logiciels

La première loi de *Lehman*, issue de constatations sur le terrain, stipule qu'un logiciel doit évoluer, faute de quoi, il devient progressivement inutile (Lehman, et al., 1985). Bien qu'ancienne cette loi n'a jamais été démentie. La réactivité, toujours plus grande, exigée des applications informatiques toujours plus complexes, supports de processus métiers évoluant eux-mêmes de plus en plus vite, explique cette tendance. La maintenance est donc, plus que jamais, une activité aussi incontournable que coûteuse.

Parmi la vingtaine de normes évoquant la maintenance des logiciels, trois sont particulièrement importantes et informatives : ISO 14764 (« Software Maintenance ») de 2006, la norme IEEE 1219 de 1998 et la norme ISO 12207 (« Software Life Cycle Processes », 2008). Ces trois textes diffèrent très peu sur le sens qu'elles prêtent au terme de maintenance. En effet, la norme ISO 14764 reprend en grande partie, suite à un alignement, les éléments figurant dans la norme IEEE 1219. Cette dernière norme est donc devenue caduque. Enfin, la norme ISO 12207, la plus récente des trois, a été alignée sur la norme ISO 14764. On peut donc se contenter de présenter le sens donné par une seule d'entre elles. Voici, la définition proposée par la norme ISO 12207.

*La maintenance est le processus mis en œuvre lorsque le logiciel subit des modifications relatives au code et à la documentation correspondante. Ces modifications peuvent être dues à un problème, ou encore à des besoins d'amélioration ou d'adaptation. L'objectif est de préserver l'intégrité du logiciel malgré cette modification. On considère en général que ce processus débute à la livraison de la première version d'un logiciel et prend fin avec son retrait.*

Depuis plus de vingt ans, la littérature abonde de tentatives de référencement et de classification des activités de maintenance. Certains travaux ont d'ailleurs été repris dans les normes précédemment citées. Cependant, les mêmes termes ayant été utilisés parfois pour désigner des activités différentes, un certain flou a longtemps régné dans le domaine. Le critère de classification le plus souvent utilisé est celui de l'intention à l'origine de l'activité de maintenance. La classification usant de ce critère ayant le plus marqué le domaine est celle de Lientz et Swanson (Lientz, et al., 1980). Elle ne distinguait que trois types d'activités. La plus détaillée est celle de Chapin qui en dénombre pas moins de 12 (Chapin, et al., 2001). On trouvera également dans (Buckley, et al., 2005) une approche complémentaire de classification basée non plus sur les raisons qui poussent à les mettre en œuvre (le pourquoi) mais sur les moyens qu'elles mettent en œuvre (le comment). Je vais pour ma part me borner à définir 4 types d'activité, me semble-t-il, assez consensuelles. Elles expriment qu'une activité de maintenance opère un déplacement mesurable dans un espace comportant trois dimensions (Figure 3-3) : l'environnement matériel et logiciel, les fonctionnalités et les propriétés non fonctionnelles.

La mise en conformité d'un logiciel vis-à-vis de ses spécifications suite à la découverte d'un dysfonctionnement, ce que l'on appelle trivialement la correction d'une erreur, est généralement qualifiée de *maintenance corrective*. Elle peut concerner aussi bien des propriétés fonctionnelles que non fonctionnelles documentées dans la spécification.

La *maintenance adaptative* correspond à la modification d'un logiciel après livraison pour qu'il reste utilisable dans un environnement logiciel et matériel qui change. Les propriétés fonctionnelles et non fonctionnelles du système n'ont pas à être altérées ; seule son implantation s'adapte à de nouvelles conditions de fonctionnement (formats d'échange de données, système d'exploitation, etc.).

La *maintenance perfective* vise à maintenir la valeur ajoutée qu'offre le logiciel en lui permettant de coller au mieux aux nouveaux besoins fonctionnels et non fonctionnels exprimés par ses utilisateurs. Elle fait suite à une demande de modification de la spécification du système.

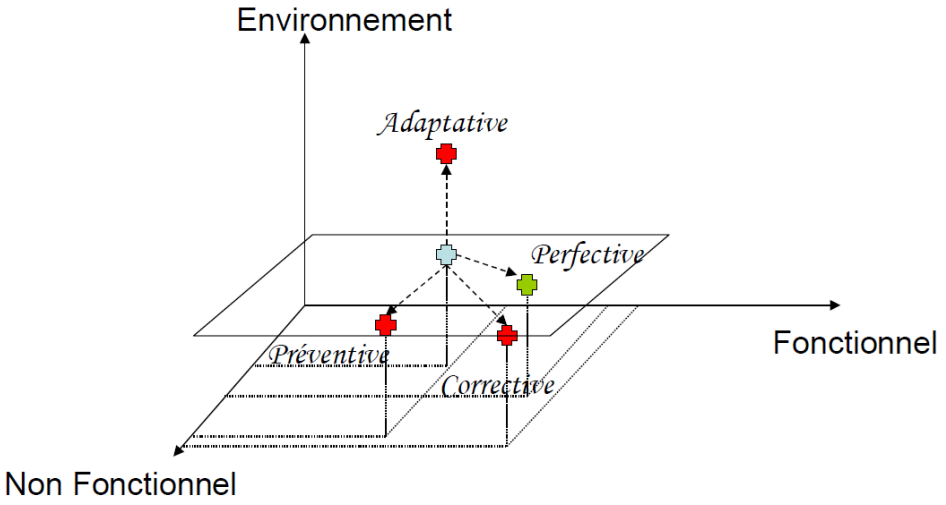


**Figure 3-3 Les différentes activités de maintenance selon leur intention**

Enfin, la maintenance des logiciels n'est pas seulement curative. Un volet préventif existe et cherche à réduire la probabilité de défaillance d'un logiciel ou la dégradation du service rendu. Elle fait suite à la constatation que l'on observe une dégradation de certains attributs qualité d'un logiciel au fil du temps. C'est le vieillissement du logiciel (Software Aging, (Parnas, 1994)). Ce type de maintenance, qualifiée de *maintenance préventive* est, soit systématique lorsque réalisée selon un échéancier établi, soit conditionnelle lorsque subordonnée à un événement prédéterminé révélateur de l'état du logiciel. Les principales actions conduites dans ce cadre visent le plus souvent, soit à accroître la robustesse d'une application, soit à renforcer son niveau de maintenabilité. Elles ne concernent donc que la dimension non fonctionnelle.

On peut constater que trois des quatre types d'activités opèrent dans le même espace plan. S'il est aisé de distinguer la maintenance corrective de la maintenance perfective sur la base de la constance (corrective) ou non (perfective) de la spécification du système. Il est au contraire plus difficile de distinguer la maintenance préventive des deux autres. Elle peut s'apparenter à de la maintenance corrective lorsqu'elle cherche à remettre en conformité un logiciel avec ses spécifications non fonctionnelles originales : rétablissant le niveau de fiabilité, de performance ou de maintenabilité éventuellement documenté à l'époque et mis à mal par un fonctionnement dans la durée et des évolutions successives. Plus encore, si l'on souhaite atteindre des valeurs différentes pour ces attributs documentés, la maintenance préventive s'apparente cette fois à de la maintenance perfective. Le distinguo n'est donc pas toujours évident à énoncer.

Un point fait encore l'objet de débats. La place des activités, dites préparatoires, conduites lors du développement initial pour préparer au mieux les maintenances à venir : choix de conception, évaluation d'architectures et de modèles de conception, mise en place de points de variation et de zones de paramétrage, planification des versions et de la logistique de maintenance, etc. En l'état, les textes normatifs, comme le montre la définition précédente, considèrent que ces activités ne relèvent pas de la maintenance.

De nombreuses études entreprises depuis le début des années 70 ont mis en évidence la part prépondérante des coûts liés aux activités de maintenance dans le budget des logiciels. Les chiffres estiment cette part entre 50 et 80% (Erlikh, 2000). Toutes confirment également que la maintenance perfective est majoritaire dans les coûts de maintenance (aux environs de 50%). La maintenance corrective se situe elle aux alentours des 20%, dans un intervalle comparable à la maintenance adaptative. Bien qu'encore réduit à quelques pourcents des budgets actuels, la maintenance préventive prend, avec l'élévation du niveau de maturité des entreprises, une place de plus en plus d'importante.

### 3.2.2 L’évolution des logiciels

Le terme évolution est, dans la littérature du Génie Logiciel, un terme éminemment ambigu. Les termes de maintenance et d'évolution sont souvent utilisés dans la littérature soit conjointement, l'un semblant compléter l'autre, soit indépendamment, l'un semblant pouvoir se substituer ou englober l'autre. Si le terme de maintenance dégage, nous l’avons vu, un certain consensus, il n'en est rien du second. Le seul point sur lequel tout le monde s'accorde et qui lui vaut un usage soutenu, est qu'il renvoie une image beaucoup plus positive. Il profite en cela d'une image favorable provenant de son usage dans les disciplines du vivant. Il évoque des aspects évolutionnistes et suggère l'existence d'une véritable théorie restant à découvrir pour les logiciels.

En parcourant la littérature, on peut distinguer trois écoles de pensée. La première et la plus ancienne de ces écoles date de la fin des années 1960. Les chefs de file de ce courant sont Lehman et Belady qui publièrent en 1976 une étude empirique, considérée comme fondatrice pour ce courant de pensée, mettant en évidence des phénomènes qui semblaient transcender les hommes, les organisations, les processus et les domaines applicatifs. Ces phénomènes ont été formulés en lois qui furent, par la suite, revisitées quelques 20 années plus tard (Lehman, et al., 1997). Le terme évolution désignait l'étude de la dynamique, au fil du temps, des propriétés d'un logiciel et des organisations impliquées (taille du logiciel, efforts, nombre des changements, coûts, etc.). Le lecteur intéressé trouvera dans (Kemerer, et al., 1999) une synthèse des travaux entrepris à l’époque dans le domaine.

La « filiation » scientifique avec les théories évolutionnistes du monde du vivant fut même consommée lorsque des travaux commencèrent à rechercher de nouvelles lois en s'appuyant, non sur une analyse de constatations empiriques, mais sur des analogies avec les lois du vivant dont on cherchait, seulement ensuite, à vérifier la validité dans les faits. Clairement cette école positionnait l'évolution comme une discipline nouvelle (Figure 3-4). En effet, l'étude d'une « dynamique » impose bien une nouvelle approche de recherche, il faut collecter des mesures à différents instants de la vie d'un logiciel puis tenter d'interpréter les fonctions discrètes obtenues pour chaque variable mesurée. Depuis cette mouvance n'a cessé de se développer profitant de la visibilité formidable de l'open-source et en particulier des projets comme Linux et Mozilla. On se reportera à (Herraiz, et al., 2008) qui référencent nombre de travaux relevant de cette mouvance.

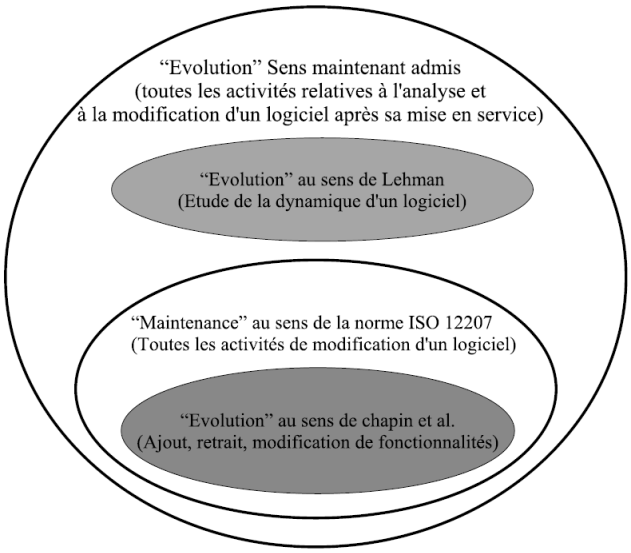


**Figure 3-4 Evolution vs Maintenance**

Par la suite, ce terme a été repris de façon opportuniste pour donner un nouvel attrait à certaines catégories déjà anciennes et particulières de maintenance : par exemple l'ajout, le retrait ou la modification de propriétés fonctionnelles. L'évolution est décrite par cette école, comme un sous-ensemble des activités de maintenance. Chapin *et al.* sont des représentants de cette école (Chapin, et al., 2001). Il semble aujourd'hui que cette vision soit devenue minoritaire dans la littérature.

La dernière école, qui semble s'imposer, considère que l'évolution est un terme plus général et plus approprié à décrire la problématique de la vie d'un logiciel après sa mise en service. Ce terme doit donc se substituer à celui de maintenance. Il étend ce dernier en incluant toutes les études sur la caractérisation dans le temps des propriétés d'un logiciel, des processus et des organisations. C'est cette définition que j'adopterai dans la suite de ce mémoire.

### 3.2.3 Le processus d'évolution et de maintenance

Dans la littérature, on distingue deux façons d'évoquer le processus de maintenance : une *vision macroscopique* et une *vision microscopique*. La vision microscopique se préoccupe de la manière dont est produite la version (n+1) d'un logiciel à partir de sa version (n). Elle décrit les activités verticales (analyse de la modification, compréhension du code, test, etc.) et supports (planification de la modification, gestion de la qualité, etc.) qui doivent être conduites pour ce faire. C'est la vision promue par les normes, celle intéressant le personnel technique. A l'inverse, la vision macroscopique s'intéresse à des intervalles de temps bien plus vastes. Elle cherche à étudier la nature des activités de maintenance d'un logiciel au fil du temps et de ses multiples versions. De grandes périodes de la vie d'un logiciel appelées phases, durant lesquelles le logiciel semble évoluer d'une manière bien particulière, émergent alors. Ce point de vue intéresse plutôt les décideurs confrontés au choix stratégiques.

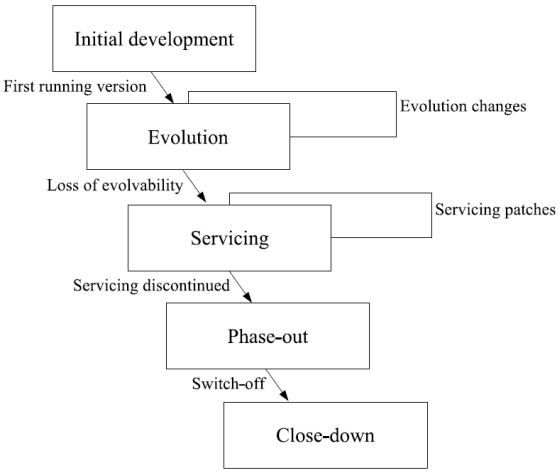


**Figure 3-5 Le modèle d'évolution de Rajlich**

La vision macroscopique est assez peu connue. Le modèle de Rajlich (Rajlich, et al., 2000) est le modèle le plus connu relevant de cette vision. Il affirme qu'un logiciel passe dans sa vie par 5 phases (Figure 3-5) : développement initial, évolution, service, fin de vie et arrêt d'exploitation. Ce modèle, dans son esprit, se place dans la droite ligne des travaux sur l'étude de la dynamique des logiciels initiés par Lehman. Dans chacune des phases de la vision macroscopique, les actions d'évolution semblent revêtir des propriétés identiques propres à la phase dans laquelle elles se placent ; stratégiquement les demandes d'évolution sont analysées de la même façon et les processus microscopiques mis en œuvre pour chacune d'entre elles se ressemblent. D'abord traitées rapidement et en toute confiance (phase « evolution »), les demandes de modification, même mineures, posent progressivement de plus en plus de problèmes (phase « servicing »), pour finalement ne plus être traitées (« phase-out »). Lorsque le logiciel, qui a cessé d'évoluer, devient complètement inutile, il est abandonné (« close-

down »). Ces phases manifestent que plus on avance dans la vie d'un logiciel, plus celui-ci devient difficile à maintenir du fait d'une part, de la dégradation de son architecture et d'autre part, d'une perte progressive de l'expertise le concernant.

L'objectif des chercheurs du domaine est bien sûr de prolonger au maximum la phase d'évolution ; c'est-à-dire de repousser le plus loin possible le moment où les coûts des modifications deviennent tels que celles-ci ne peuvent plus être conduites. Lors de la phase d'évolution, les mises à jour vont se succéder en respectant généralement le même processus. C'est ce processus, qualifié de microscopique, que je vais maintenant décrire.

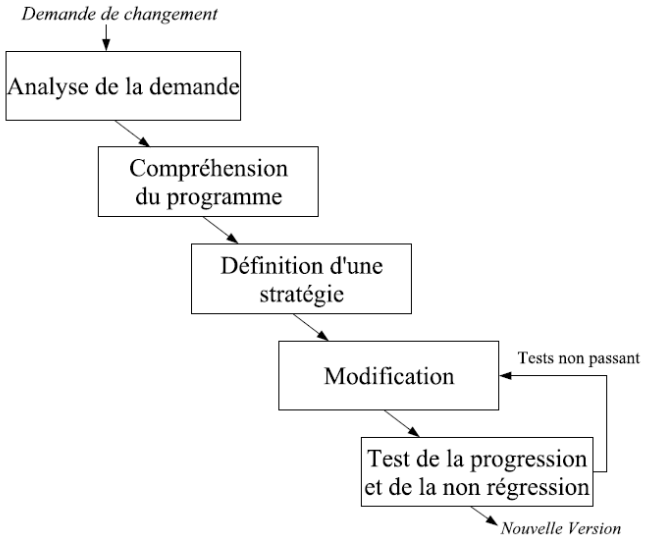


**Figure 3-6 Le microprocessus de maintenance**

La norme ISO 12207 donne une description consensuelle sur le plan microscopique de l'ensemble des processus et activités à suivre lors d'un acte de mise à jour d'un logiciel. La maintenance intègre des actions aussi bien techniques (tests, modification de code, etc.) que managériales (estimation de coûts, stratégie de gestion des demandes de modification, planification, transfert de compétences, etc.). Elle ne se résume donc pas aux seules actions de reprise de code faisant suite à la découverte d'anomalies de fonctionnement. Son champ d'action est bien plus vaste et fait par conséquent l'usage d'une multitude de techniques, d'outils, de méthodes et de procédures provenant de domaines variés. Des activités citées par ce texte, nous n'évoquerons pas ici les activités qualifiées de « support » par cette norme (gestion de configuration, assurance qualité, etc.), pour nous limiter à celles directement liées au processus de modification.

La réception d'une demande de changement est le point de départ de toute action d'évolution (Figure 3-6). Cette demande émane, soit d'un client, soit d'un acteur interne à l'entreprise. Elle fait suite à la constatation d'une anomalie de fonctionnement, au souhait de migrer l'application vers un nouvel environnement matériel ou logiciel, au désir de modifier les aptitudes fonctionnelles ou non fonctionnelles de l'application ou d'améliorer à titre préventif principalement sa maintenabilité et sa robustesse. Cette demande est tout d'abord évaluée pour déterminer l'opportunité de sa prise en compte. Des considérations stratégiques, d'impacts et de coûts vont prévaloir dans cette étape. Des outils, tels que des modèles analytiques d'efforts et de coûts, peuvent être utilisés. Si la demande est acceptée, on commence par se familiariser avec l'architecture de l'application pour développer une connaissance suffisante de sa structure et de son comportement. Cette étape de compréhension est très coûteuse. Elle occupe souvent, à elle seule, plus de la moitié du temps consacré à un cycle de maintenance (Corbi, 1989). On identifie ensuite différentes stratégies de modification. Ces stratégies vont être départagées en usant de critères, tels que le coût, les délais, le niveau de qualité garanti ou de compétence exigé, etc. Pour quantifier le coût des différentes stratégies, il est possible d'user d'outils évaluant l'impact des changements envisagés sur le reste du code (Arnold, 1996). La stratégie retenue va, ensuite,

être mise en œuvre par une modification effective des modèles, du code et des documentations associées. On use ici d'outils, de méthodes et de procédures identiques à ceux utilisés lors du développement de la version initiale de l'application. Une fois la modification faite, il faut s'assurer que celle-ci n'a pas, d'une part altéré ce qui ne devait pas l'être (non-régression) et d'autre part, qu'elle a bien ajouté les propriétés souhaitées (progression). Une fois la vérification faite, la nouvelle version peut être archivée, diffusée et installée.

### 3.2.4 Rétro-conception et compréhension de programme

Le microprocessus de maintenance fait usage de différentes techniques. Les techniques relevant de l'étape de compréhension, la plus coûteuse, sont souvent placées dans la littérature sous le sceau soit de la rétro-conception (*reverse engineering*) soit de la compréhension de programme (*program comprehension*).

La rétro-conception est le processus consistant à analyser un système pour identifier ses composants et les relations qu'ils entretiennent dans le but de créer des représentations de ce système sous une autre forme ou à des niveaux plus élevés d'abstraction : typiquement pour générer, depuis un exécutable du code ou depuis du code des modèles de conception (par exemple des diagrammes de classes ou de séquence UML). Ce processus compte deux approches qui se distinguent par la manière dont sont produites les informations facilitant la prise en main du logiciel à modifier :

- une approche statique, qualifiée aussi de « boîte blanche », exploite les sources et les modèles de conception du logiciel ;
- une approche dynamique, qualifiée de « boîte noire », s'appuie sur des traces d'exécution du logiciel.

Les deux approches de rétro-conception sont complémentaires. Les avantages de l'une sont les inconvénients de l'autre. L'approche dynamique a pour avantage la précision. Elle décrit le comportement réel du système, par exemple faisant fi de l'indéterminisme que pose, à l'analyse statique, la liaison dynamique dans le monde objet. Elle décrit ce qui est, à la différence d'une analyse statique qui, le plus souvent, ne peut produire qu'un sur-ensemble de ce qui peut être. Elle permet également de se focaliser sur les seuls scénarii d'exécution d'intérêt pour la modification envisagée. Ses deux principaux inconvénients sont d'une part l'incomplétude de l'analyse produite avec des traces qui ne peuvent refléter qu'une fraction de domaines en général infinis et la masse importante d'informations « brutes » générée par l'analyse qu'il faut pouvoir ensuite structurer. Dans les deux cas, c'est une ingénierie passive, elle ne change rien au système sur le plan de ses propriétés internes et externes. L'objectif est de fournir des « vues » offrant sur l'existant des informations à haute valeur ajoutée.

Les techniques de compréhension de programmes cherchent à produire des modèles mentaux de la structure d'un logiciel à différents niveaux d'abstraction (du code jusqu'au modèle du domaine). Ces modèles sont construits automatiquement depuis le code et les documentations qui lui sont associés. Cette définition est en apparence très proche de celle donnée pour la rétro-conception. Les techniques de compréhension se distinguent cependant des techniques de rétro-ingénierie par le fait qu'elles ne cherchent pas à reconstruire de l'information manquante ou oubliée dans un format usuel (diagrammes UML, graphes d'appels, etc.), mais à faire émerger, dans des formats mentalement parlants (donc le plus souvent propriétaires), une information à très haute valeur ajoutée ; une information qui n'a jamais été car ne pouvant pas être formulée dans les langages de modélisation utilisés par les informaticiens.

Il faut noter toutefois que cette distinction n'est d'une part, pas si évidente selon les travaux. Certains considèrent la rétro-ingénierie comme étant un cas particulier de la compréhension de programme et d'autres pensent l'inverse. Il est curieux de constater que ces deux types de techniques, qui tentent pourtant de résoudre le même problème, ont été portées pendant longtemps par deux communautés de chercheurs relativement indépendantes. Cela semble lié au fait que la seconde a toujours eu des préoccupations plus cognitives et didactiques que la

première. On trouvera une synthèse de ces deux types d'approches dans (Tonella, et al., 2007) et (Cornelissen, et al., 2009).

### 3.2.5 Vérification de la non régression

La vérification de la non-régression se fait en rejouant les tests non obsolètes de la précédente version du logiciel ; c'est-à-dire ceux qui vérifient les propriétés devant rester inchangées. Des outils du commerce permettent d'automatiser le jeu de ces tests. Cependant, la reconduction de l'ensemble des tests peut se révéler très coûteuse. Des algorithmes additionnels de *sélection*, d'*ordonnancement* et de *réduction* des tests ont été proposés pour éviter de rejouer tous les tests non obsolètes tout en maintenant un certain niveau d'efficacité (Rothermel, et al., 2004).

Deux types de techniques de sélection coexistent. Celles qui sont qualifiées de *sûres* (safe) garantissent que le sous-ensemble identifié inclut (sans égaler, l'égalité est indécidable dans le cas général) tous les tests conduisant à la détection d'une régression (Bible, et al., 2001). Les techniques non sûres permettent, elles, de rechercher un sous-ensemble optimisant certains paramètres (par exemple, le taux de couverture ou le nombre de fautes découvertes par unité de temps, etc.). Ces techniques, intéressantes sur un plan pratique pour tenir compte de contraintes de temps ou de ressources, peuvent cependant exclure des tests qui détecteraient des régressions. Ces travaux usent de techniques de comparaison de graphes de contrôle, avant et après modification, pour extraire de l'ensemble des tests le sous-ensemble de ceux qui sont potentiellement affectés par la modification de code réalisée. Le lecteur intéressé trouvera dans (Engström, et al., 2010) une liste récente et très détaillée de nombre de travaux sur ce thème.

Les techniques d'ordonnancement visent, quant à elles, à identifier parmi toutes les permutations possibles sur un ensemble de tests celle qui va maximiser une certaine fonction, typiquement, la vitesse de découverte des fautes en plaçant les tests les plus révélateurs en début de séquence (Rothermel, et al., 2001) (Li, et al., 2007).

Parce qu'un programme évolue, de nouveaux tests sont ajoutés pour valider de nouvelles fonctionnalités. Au fil du temps, l'ensemble des tests grandit, et certains tests peuvent présenter des intersections non vides en termes de parcours de code ou de vérification de certains comportements. Les techniques de réduction de tests identifient et suppriment définitivement les cas de tests redondants, diminuant la taille des tests à rejouer. Ces techniques se distinguent donc de celles de sélection, qui elles ne suppriment pas définitivement des cas de test, mais opèrent un filtrage temporaire. Un inconvénient potentiel de ces techniques est que la suppression de certains tests peut endommager la capacité de détection de fautes de la suite de test. On trouvera dans (Zhong, et al., 2008) une étude comparant quatre de ces techniques.

## *3.3 Les langages*

### 3.3.1 Définition théorique d'un langage

Lorsque l'on cherche à définir un langage de modélisation ou de programmation, il convient en premier lieu de préciser sa *syntaxe*. *La syntaxe d'un langage est l'ensemble des mots qu'il contient.* La syntaxe d'un langage est donc un ensemble (au sens mathématique du terme) comportant tous les mots appartenant à ce langage. Un *mot* est un ensemble de *symboles* repérés dans un espace généralement à une ou deux dimensions pris dans un *alphabet* fixé $\Sigma$ (fini en informatique). Les langages textuels usent souvent de symboles représentant les lettres de l'alphabet. Un langage graphique, comme le langage des diagrammes de classes UML, use pour sa part, non seulement d'un alphabet composé de symboles de type caractère mais également de symboles de type figure géométrique plane (rectangle, ovale, segment de droite, etc.). La finitude de l'alphabet est une contrainte forte de conception permettant la manipulation du langage par une machine. Mais ce n'est pas une contrainte limitant le pouvoir d'expression. On

peut toujours ramener, par des mécanismes de génération, un alphabet infini dénombrable à un alphabet fini.

Un mot est construit depuis les symboles de Σ en utilisant un ou plusieurs opérateurs *de placement*. Pour les langages textuels, il n'y a qu'un seul opérateur de placement : la *concaténation*. Un mot est une suite de symboles placés par concaténation les uns à la suite des autres. Pour les langages de programmation un mot est appelé un *programme*. Un mot est assimilable formellement à une application d'un intervalle de $\mathcal{N}$ dans Σ. La concaténation est un opérateur associatif permettant depuis deux mots d'en construire un troisième en « collant » les deux mots l'une derrière l'autre. Dans le cas des langages graphiques, il y a plusieurs opérateurs de placement pour exprimer la proximité, l'inclusion, la connexion, etc. Un mot est appelé un *modèle*.

On note Σ* l'ensemble des mots que l'on peut construire sur l'alphabet Σ en usant des opérateurs de placement. La syntaxe d'un langage est un sous ensemble de Σ*. $\mathcal{P}$(Σ*), l'ensemble des parties de Σ*, est donc l'ensemble de tous les langages que l'on peut construire sur l'alphabet Σ avec les opérateurs de placement. La syntaxe $\mathcal{S}$ d'un langage peut contenir un nombre très important de mots. Rien qu'avec le langage graphique diagramme de classes UML, il est possible de dessiner une infinité de modèles différents (mais néanmoins dénombrable). Toute la difficulté est de définir la syntaxe $\mathcal{S}$ sachant qu'en général le cardinal de $\mathcal{S}$ est infini. Toute définition *en extension* consistant à énumérer les mots qu'il contient est impossible. Il faut donc user d'une définition en *compréhension*.

Une définition en compréhension s'appuie sur un ou plusieurs ensembles déjà définis par ailleurs puis détaille un procédé rigoureux pour indiquer de quelle façon, depuis ces ensembles (généralement finis), on construit l'ensemble recherché. Construire des ensembles infinis partant d'ensemble fini passe nécessairement par un procédé incluant « une répétition » infinie de certaines étapes (par exemple l'étoile de Kleene pour les expressions régulières). Au contraire, la construction d'un langage infini partant d'autres langages infinis peut se faire à l'aide d'un procédé ne comportant aucune répétition infinie.

Comme une syntaxe est un ensemble, on dispose pour construire un langage infini partant d'autres langages infinis des opérateurs de la théorie des ensembles : *union*, *intersection*, *différence*, *produit cartésien*, *complémentaire*, *inclusion*. De plus, comme Σ* est un ensemble particulier (un monoïde avec la concaténation et le mot vide par exemple) d'autres opérateurs sont disponibles : *concaténation de langages*, *étoile de Kleene*, *morphisme* et *substitution*. Pour la définition partant d'ensembles finis de langages textuels, on utilise des *règles de réécritures* de chaînes. Ces règles, en nombre fini, indiquent comment produire une chaîne partant d'une autre chaîne. Formellement, cela revient à introduire une relation binaire sur Σ*. Un langage est alors défini par « induction » : toutes les chaînes dérivables depuis un mot racine en appliquant une séquence de taille quelconque (potentiellement infinie) de ces règles. Un langage est ainsi le codomaine d'un sous-ensemble de la fermeture réflexive et transitive de la relation binaire introduite par les règles. Les langages graphiques sont plus compliqués à définir mais ils s'appuient également sur une induction à base de *règles de réécriture de graphes* ou de l'*instanciation de métamodèles*.

Le problème est de proposer une définition en compréhension qui puisse satisfaire à la fois les utilisateurs de ces langages et les concepteurs d'outils manipulant ces langages. Il s'agit de trouver un juste équilibre entre une définition facilement appropriable tout en offrant un niveau de rigueur suffisant.

Il faut ensuite pouvoir donner un sens à chaque mot du langage. C'est-à-dire définir sa *sémantique* ; l'interprétation que doit en donner une machine ou un homme. Pour un langage de programmation comme Java, la sémantique doit permettre de déterminer « ce qui se passe » lorsqu'une machine virtuelle Java exécute le produit de la compilation d'un mot du langage. L'interprétation est dans ce cas appelée plus communément *exécution*. La sémantique du langage UML doit permettre à tout mot (modèle) d'associer les informations que l'on peut

déduire sur le système modélisé. Pour un langage de modélisation comme UML, l'interprétation est parfois le fait d'un homme (ou d'une femme). Une erreur fréquente consiste à croire que la sémantique associe nécessairement à un mot un comportement, une exécution. C'est vrai pour les langages de programmation ou pour des langages de modélisation comme les machines à états. Mais il existe une multitude de langages qui ne s'intéressent pas à la description de la dynamique d'un système, mais à sa structure (diagramme de classes, entité/association, etc.). Ces langages ont, eux aussi, une sémantique et cette sémantique n'associe pourtant pas un mot à un comportement.

La sémantique doit permettre de projeter tout mot (signifiant) vers la chose qu'il représente (le signifié). Décrire avec rigueur la sémantique d'un langage impose donc de décrire précisément non seulement cette fonction de projection mais également le domaine d'arrivée de cette projection (Harel, et al., 2004).

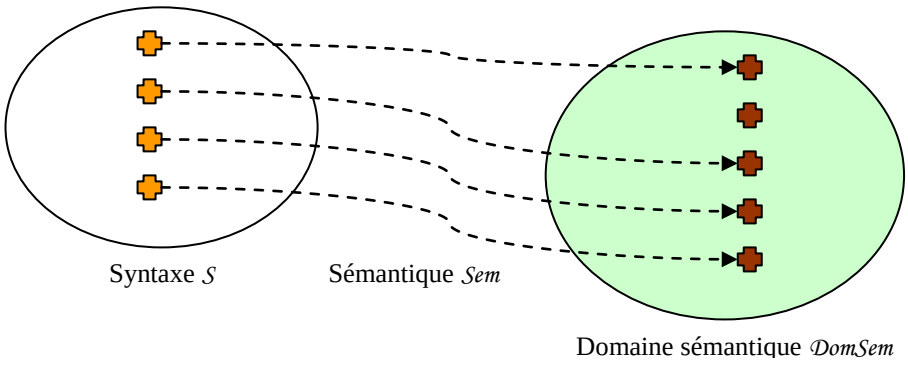


**Figure 3-7 Expression de la sémantique d'un langage**

La sémantique d'un langage est une fonction totale $Sem$ qui a pour domaine de définition l'ensemble des mots de ce langage $\mathcal{S}$ et pour domaine d'arrivée un ensemble appelé domaine sémantique $\mathcal{D}omSem$ (Figure 3-7).

$$Sem: S \rightarrow DomSem.$$

Il existe une grande variété de domaines sémantiques. Ils peuvent être de cardinal fini (par exemple $\mathcal{D}omSem=\{true, false\}$ pour le langage de la logique propositionnelle), infini dénombrable (l'ensemble des entiers relatifs pour le langage des expressions arithmétiques entières), infini non dénombrable (l'ensemble des fonctions de $\mathcal{N} \rightarrow \mathcal{N}$ pour un langage de programmation autorisant une entrée et une sortie entière). Ils peuvent être des ensembles mathématiques connus ou au contraire des ensembles composés d'abstraction de domaines particuliers (objets, évènements, transitions, etc.). Les ensembles les plus prisés sont bien sûr ceux pour lesquels il existe déjà une définition rigoureuse universellement connue.

Au final, un langage est formellement défini par la donnée d'un triplet : $\mathcal{L}=(S, Sem, \mathcal{D}omSem)$

### 3.3.2 Définition pratique d'un langage textuel

Pour des raisons d'ordre technologique, on adopte souvent une description en quatre étapes de la syntaxe d'un langage textuel. Ce découpage est en fait la conséquence des techniques algorithmiques particulières mises en œuvre lors de la détermination de l'appartenance ou non d'un mot à la syntaxe d'un langage.

- On décrit par simple énumération l'alphabet (fini) du langage ;
- On définit les constructions d'intérêts et primitives (identificateurs, nombres, mots-clés, etc.) qui peuvent se contenter pour leur description de techniques basiques de type grammaire linéaire (type 3 de la classification de Chomsky), expressions régulières, etc. Ces constructions syntaxiques sont reconnaissables de manière très efficace à l'aide d'automates finis déterministes.

- On défini des mots (phrases) du langage sur la base des constructions primitives en usant de techniques plus évoluées de type grammaires non-contextuelles particulières (LL, LR, etc.). Ces phrases sont reconnaissables avec des algorithmes relativement efficaces. A ce stade un sur-ensemble de la syntaxe souhaitée est décrite ;
- On restreint l'ensemble des phrases précédentes, à la syntaxe recherchée en incluant des aspects contextuels (visibilité, typage, unicité des identificateurs, etc.) qui ne pouvaient être décrits à l'aide des techniques non contextuelles précédentes.

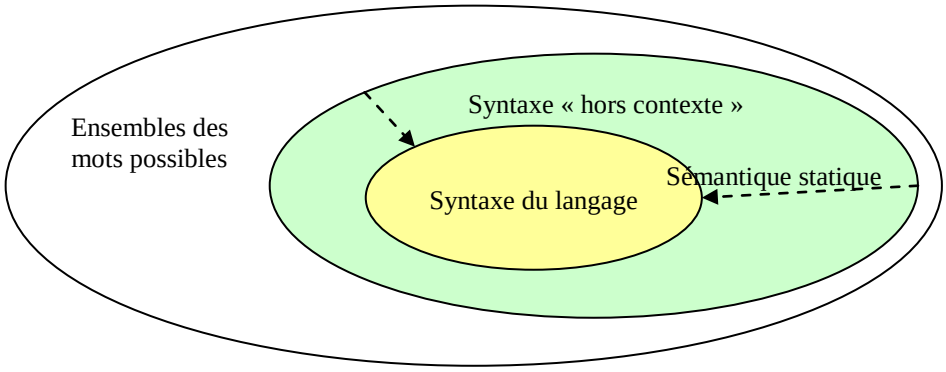


**Figure 3-8 La sémantique statique d'un langage**

Ce découpage est à l'origine d'une confusion historique sur la frontière entre syntaxe et sémantique dont la manifestation est la notion de *sémantique statique* (Meek, 1990). La troisième étape ne faisant l'usage, pour des raisons d'efficacité, que de langages non contextuels, la syntaxe décrite est un sur-ensemble de la syntaxe du langage. Il est impossible de préciser par exemple que deux variables de même portée ne peuvent pas avoir le même nom ou qu'une variable déclarée de type entier ne peut pas se voir affecter une chaîne de caractères. Pour exprimer ces contraintes, il faut appliquer une restriction à l'ensemble obtenu pour obtenir la syntaxe du langage (Figure 3-8). Cette restriction use de techniques qui ressemblent pour certaines à des techniques relevant de l'expression de la sémantique (règles d'inférences). Mais la sémantique statique n'a rien de sémantique. Elle ne concerne que la syntaxe. Elle ne fait qu'exprimer le « delta » existant entre la version non contextuelle de la syntaxe et la syntaxe réelle du langage. On retrouve cette confusion dans le monde MOF. Il est fréquent d'entendre parler de la partie « sémantique » d'un langage exprimé à l'aide d'invariants OCL sur un métamodèle. Ces invariants n'apportent rien de sémantique. Ils n'ont que pour seul objet de pallier le manque d'expressivité du langage MOF.

En l'espace de 50 ans, de nombreuses approches ont été proposées pour exprimer la sémantique d'un langage de programmation. Elles se classent en quatre catégories : opérationnelle, dénotationnelle, axiomatique et hybride. On en trouvera une synthèse dans (Zhang, et al., 2004) et (Mosses, 2006). Selon ces approches, la description partielle (mais suffisante pour l'usage courant) et rigoureuse du comportement d'un programme peut prendre une forme plus ou moins abstraite : la suite des états pris par la machine virtuelle après chacune des instructions (sémantique opérationnelle), la donnée de la fonction calculée par le mot (sémantique dénotationnelle), un ensemble de prédicats logiques liant l'état de la mémoire avant et après l'interprétation du mot (sémantique axiomatique). Au fil des ans, les approches ont progressé pour inclure les aspects temporels et surtout tenter de proposer une description modulaire de la sémantique pour faciliter la compréhension, l'évolution et la réutilisation de fragments de sémantique. L'objectif actuel est de rendre plus accessible ces approches encore très exigeantes sur le plan mathématique.

Au final un langage textuel est souvent défini par la donnée d'un quadruplet :

*L=(SyntaxeHorsContexte, SemantiqueStatique, Sem, DomSem).*

### 3.3.3 Définition pratique d'un langage graphique

Les langages graphiques se distinguent des langages textuels par le fait que :

- le placement des symboles est bidimensionnel nécessitant de s'appuyer sur plusieurs opérateurs de placement ;
- l'on use d'un panel plus vaste de symboles ayant en sus plusieurs propriétés caractérisant leur forme (taille, orientation, couleur, contour, etc.).

Pendant longtemps, la théorie des langages n'a pas accordé de traitement particulier aux langages graphiques (souvent appelés, à tort, langages visuels). En effet, leur complexité apparente peut toujours être maîtrisée en les projetant vers des langages textuels apportant les mêmes informations car incluant les données nécessaires pour mémoriser le placement et les propriétés de chaque symbole. Le langage SVG est, par exemple, un standard du W3C qui permet de décrire des ensembles graphiques en deux dimensions à l'aide du langage XML qui est lui textuel. Cependant avec l'émergence dans les années 90 de très nombreux langages graphiques avec lesquels on voulait assoir de manière précise et efficace la définition, de nombreux travaux se sont intéressés à ces langages particuliers.

Ces travaux ont révélé l'importance de séparer la syntaxe de « surface » d'un langage (on parle de *syntaxe concrète*), de sa syntaxe utile (on parle de *syntaxe abstraite*). La syntaxe concrète est très importante et ne doit jamais être négligée par le concepteur d'un langage. C'est avec elle que travaillent les développeurs et les machines. Elle doit donc privilégier selon les cas l'esthétique, la convivialité et une occupation structurante de l'espace de dessin ou l'efficacité du parcours et la taille du stockage. On trouvera dans (Moody, 2009) une discussion intéressante sur l'impact des choix visuels et sur les « qualités » que doit présenter un langage graphique.

Mais ce faisant, la syntaxe concrète pose des contraintes très fortes sur la forme d'un modèle qui sont sans rapport avec son fond. Or, il est de peu d'importance pour une machine de savoir comment (un rectangle, un segment, trois chaînes de caractères, etc.) et où (en haut à gauche du modèle) sont dessinés une classe `Etudiant` et ses deux attributs `nom` et `âge`. Ce qui importe, pour « raisonner » sur le modèle, c'est de savoir qu'une telle classe existe et qu'elle compte ces deux attributs. La syntaxe abstraite est donc une représentation minimale d'un modèle qui ne véhicule que l'information utile à une machine ou à un humain pour mener une analyse de conformité et une interprétation (association d'une sémantique). Elle oublie toutes les décorations qui ne véhiculent pas de structure et de sens.

Cette séparation offre deux avantages :

- construire une sémantique sur la base d'une syntaxe (abstraite) minimale épurée de tout élément inutile ;
- permettre d'associer à un même langage plusieurs syntaxes concrètes. Il est ainsi fréquent d'avoir pour un même langage une syntaxe concrète pour les utilisateurs et une autre pour le stockage en machine (on parle souvent de syntaxe de sérialisation car la syntaxe prend souvent la forme d'un fichier XML).

La définition d'un langage graphique passe donc le plus souvent par la définition (Figure 3-9) d'un 6-uplet :

1. domaine sémantique ;
2. syntaxe abstraite non contextuelle ;
3. sémantique statique ;
4. une fonction partielle projetant la syntaxe abstraite non contextuelle dans le domaine sémantique (la fonction est totale pour la syntaxe « contextualisée ») ;
5. plusieurs syntaxes concrètes dont au moins deux (une pour les utilisateurs et une pour le stockage) ;

6. de fonctions totales permettant de passer d'une syntaxe concrète vers la syntaxe abstraite et de la syntaxe abstraite vers une syntaxe concrète (produisant un modèle « par défaut »).

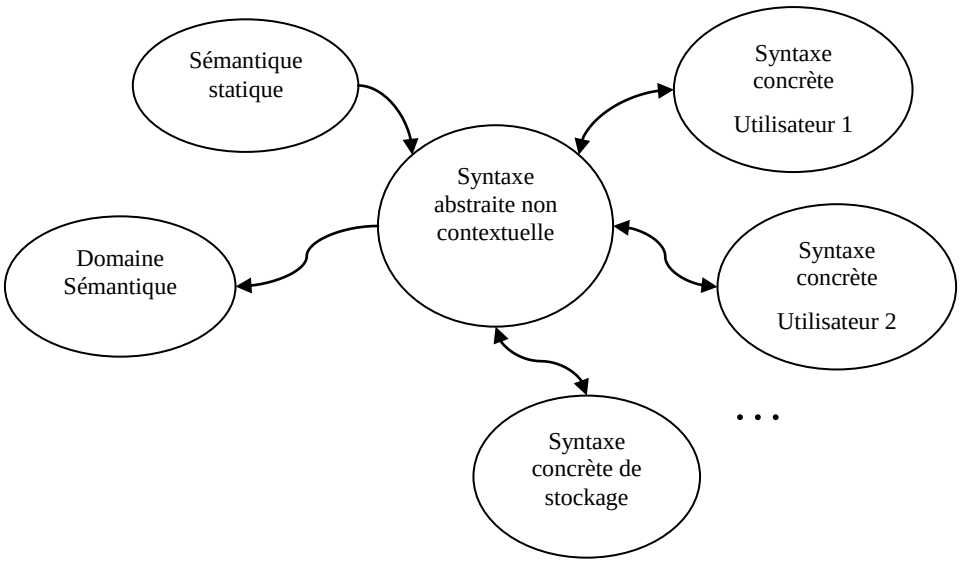


**Figure 3-9 Les différents compartiments d'un langage graphique**

Les techniques mises en œuvre pour le point 1 et le point 3 ne diffèrent pas de celles utilisées pour les langages textuels. Cependant, les langages graphiques sont nombreux à ne pas s'attacher à la dynamique de systèmes. Les domaines sémantiques sont donc plus variés que pour les langages textuels. Les grandes différences sont à mettre sur le compte des points 2, 4 et 5.

Il existe actuellement trois grandes approches pour définir la syntaxe d'un langage graphique.

- *les grammaires à positionnement* : XPGs (Extended Positional Grammars), RGs (Relationnal Grammars), CMGs (Constraint Multiset Grammars) et PLGs (Picture Layout Grammars) qui sont un cas particulier des grammaires à attributs dans lesquelles on embarque des informations sur la répartition spatiale des éléments. On trouvera des détails sur ce type de grammaire dans (Costagliola, et al., 2004)
- *les grammaires de graphes* : qui adoptent une représentation de la syntaxe à plus haut niveau comme SRs (Symbol-Relation Grammars), HG (Hypergraph Grammars), LCGs (Layered Graph Grammars). On trouvera un exemple d'utilisation dans (Minas, 2006).
- *Les métamodèles* : avec des métalangages comme ECORE, MOF.

Ces techniques diffèrent dans l'importance qu'elles accordent, dans le processus de définition du langage, à la syntaxe abstraite ou concrète. Certaines privilégient la syntaxe abstraite (les métamodèles) d'autres la syntaxe concrète (grammaires à positionnement). Ces deux attitudes s'opposent l'une à l'autre : la première attitude facilite la définition d'une sémantique et permet un découplage clair syntaxe concrète/syntaxe abstraite, mais à l'inverse de la seconde, elle pose le problème de la définition de mécanismes de projection vers une syntaxe concrète.

Pour chaque approche, on trouve des méta-outils capables de produire plus ou moins automatiquement des éditeurs depuis la syntaxe d'un langage :

- pour les grammaires à positionnement : VLDesk, Penguin, VPW, etc. ;
- pour les grammaires de graphes : DiaGen, GenGed, PROGRES, TIGER, etc. ;

- pour les métamodèles : ECLIPSE-Graphical Modeling Framework (GMF), XMF-MOSAIC, MetaEdit+, Generic Modeling Environment (GME), Microsoft DSL Tools, Jetbrains Meta Programming System (MPS).

### 3.3.4 Les langages dédié

Les *langages dédiés* (Domain Specific Language ou DSL) ont pour particularité d'ériger en qualité d'entité de première classe les concepts d'un domaine précis (Mernik, et al., 2005). Ils s'opposent en cela aux langages généralistes tels que UML, Java ou C++ qui eux, offrent une batterie de constructions générales souvent plus élémentaires mais capable par agrégation de répondre à un nombre important de problèmes. Les langages dédiés permettent une expression plus directe et précise de patrons ou d'idées, mais pour des classes plus limitées de problèmes. Ils offrent potentiellement une productivité dans leur domaine de prédilection supérieure à celle des langages généralistes.

Les langages dédiés ne sont pas récents. L'informatique a de tout temps produit et usé de ce type de langage. Le langage Backus–Naur Form (BNF) qui permet de décrire la syntaxe d'un langage textuel de type non contextuel peut être sans débat qualifié de langage dédié. Le langage HTML également. Ils apparurent cependant à la fin des années 50 pour le premier et au début des années 90 pour le second. Si le terme est apparu récemment, on lui trouve dans la littérature depuis près de 40 ans de nombreux synonymes : langages orientés applications, langages spécialisés, langages tâches-spécifiques, etc. Le fait que les langages dédiés tiennent autant l'affiche ces dernières années dans les travaux des chercheurs est lié à l'avènement de nouveaux outils permettant la définition et l'outillage véritablement industriel de langages dédiés.

Il n'est pas facile de tracer une frontière nette entre d'un côté les langages généralistes et de l'autre les langages dédiés. Les problèmes abordés par un langage peuvent être positionnés dans un espace comportant quatre dimensions : l'étape du cycle de vie dans laquelle se pose les problèmes (expression des besoins, spécification, conception, etc.), le domaine applicatif ciblé (temps réel embarqué, informatique de gestion, etc.), le point de vue de modélisation adopté (structurel, comportemental, fonctionnel, etc.) et le paradigme de développement (objet, composant, etc.). Un sujet de débat sans fin est alors de savoir quelle dimension maximale peut avoir le sous-espace associé à un langage pour être qualifié de « dédié ». Un langage dédié à la modélisation d'applications embarquées est plus « généraliste » qu'un langage dédié à la planification de trafics autoroutiers.

On peut invoquer d'autres critères pour tenter de déterminer avec certitude l'appartenance ou non d'un langage à la catégorie des langages dédiés, par exemple décider que le caractère Turing-Complet d'un langage force automatiquement son classement dans la catégorie des langages généralistes. Cela semble correspondre à la pensée usuelle pour des langages comme HTML ou SQL-92 (les versions plus récentes de SQL sont elles Turing-complète). Mais le langage de script Bourne Shell pour Unix ou le langage XSLT, considérés par beaucoup comme des langages dédiés, présentent eux, cette propriété. On peut aussi invoquer la productivité comparée d'un langage pour déterminer sa classe d'appartenance. Les langages dédiés sont par nature nettement plus productifs dans leur domaine de prédilection qu'un langage généraliste appliqué au même domaine. On peut par exemple comparer la productivité de SQL-99 considéré comme dédié à celle d'un langage généraliste représentatif comme Java. Sur ce cas, le critère semble pertinent. SQL-99 offre une productivité bien supérieure à celle de Java dans le domaine de la manipulation des données. Mais ce critère reste subjectif. Quel langage de référence utiliser et quelle différence constater pour placer un langage dans l'une ou l'autre des catégories. La meilleure approche consiste donc, non pas à adopter une partition des langages en deux classes, mais à considérer un continuum allant de langages clairement généralistes à des langages hautement spécialisés.

### 3.3.5 Définition pratique des langages dédiés

Les langages généralistes ont longtemps dominé le monde informatique et cela pour des raisons essentiellement pratiques relatives entre autres à leur coût d'outillage. La mise au point d'un outillage pour rendre effectif et agréable l'emploi d'un langage dédié dans un contexte industriel (éditeurs, compilateurs, débogueurs, etc.) réclamait auparavant trop d'efforts par rapport aux gains que l'on pouvait attendre de son usage occasionnel par une communauté réduite de développeurs. Le modèle économique pour ce type de langage n'étant pas rentable, les grands éditeurs s'en sont détournés et n'ont jamais investi dans l'un de ces langages. Mais les choses évoluent. Pour définir un langage dédié on dispose maintenant de trois approches viables sur le plan économique.

La première consiste à raffiner un langage généraliste existant en spécialisant certains de ces concepts. Cela suppose de disposer d'un langage généraliste capable de se « tordre » suffisamment vers le langage désiré via un mécanisme standard et rigoureux d'extension. Cette approche suppose une proximité importante entre la syntaxe abstraite et la sémantique du langage généraliste et celles du langage dédié. Son avantage est de permettre la réutilisation intégrale des outils existants et également la possibilité de basculer entre des vues « dialectes » et non « dialectes ». La notion de « profil » UML est représentative de cette approche.

La seconde possibilité consiste à étendre ou restreindre un langage existant en lui adjoignant des structures propres au domaine visé. Cette approche offre plus de flexibilité que la précédente. Elle permet également de capitaliser une partie de l'expertise acquise par les développeurs sur le langage généraliste. Les efforts d'apprentissage sont donc moindres. En contre partie, elle ne rend pas forcément possible la réutilisation de tous les outils. Le langage de modélisation de systèmes *SysML* est représentatif. En effet, il n'est pas un strict profil UML puisqu'il n'étend que 7 des 13 langages d'UML et en rajoute 2 nouveaux.

La troisième possibilité, enfin, propose de définir de toutes pièces le langage dédié. Cette approche offre une totale flexibilité. Il est possible de proposer un langage qui colle à la classe des problèmes soulevés, offrant les mécanismes les plus expressifs et puissants. Elle oblige cependant à développer de toutes pièces les outils associés. La courbe d'apprentissage est également moins rapide puisque les développeurs ne peuvent pas mobiliser leurs savoirs pour appréhender le langage. On peut limiter ces deux inconvénients d'une part en utilisant un méta-outil et d'autre part en réutilisant des patrons langagiers. Sur le plan de la sémantique, une approche intéressante pour réduire les efforts est d'user de la *translation*. C'est-à-dire de projeter les mots du langage dont on veut définir la sémantique sur les mots d'un autre langage dont la sémantique est connue et explicite. La définition de la sémantique est alors grandement facilitée.

Les deux premières approches conduisent à ce que (Atkinson, et al., 2007) qualifient de langages adaptés à un domaine (*Domain-Customized Language* ou DCL). Cette classe de langage dédié offre un bon compromis entre d'une part les coûts de définition et d'usage et d'autre part les gains procurés pour la résolution d'un problème précis. La troisième solution est de loin la plus coûteuse puisqu'elle nécessite de développer les outils correspondant.

### 3.3.6 Notion de modèle

Dans la littérature un modèle est souvent décrit comme : « une représentation simplifiée d'un système, une abstraction ». (Kühne, 2006) précise qu'un modèle, pour mériter ce titre, doit nécessairement être construit depuis un système selon un processus incluant comme première étape une *projection*. Cette fonction va restreindre syntaxiquement les informations présentes dans le modèle par rapport au système. La projection est :

- un morphisme (donc une fonction qui « préserve » les structures, c'est-à-dire les relations n-aires de toutes natures entre les éléments) ;

- partiel (certains éléments et liens entre ces éléments ne sont pas projetés, d'où la perte de certaines informations)
- injectif (il y a une relation « un vers un » entre les éléments du modèle et la partie non filtrée du système, donc rien n'est altéré).

Un modèle est une vue « filtrée » (mais fidèle sur la partie restante) d'un système.

Je trouve cette définition trop restrictive et marquée par l'hypothèse, commune dans d'autres disciplines (génie civil, physique, etc.), qu'un système est un objet du monde réel, par définition non représentable dans toute sa complexité et diversité à l'aide d'un modèle fini. Or, en informatique, le système modélisé est souvent un modèle lui-même et tous les modèles manipulés par une machine sont finis. Rien n'empêche donc un modèle d'offrir autant, voire plus, d'informations qu'un autre modèle, tout en usant du premier pour comprendre le second. On peut ainsi utiliser du code pour analyser un modèle de conception. On note au passage que le système et le modèle peuvent être conformes à deux langages différents. On nie également la possibilité pour un modèle d'ajouter de l'information « calculée » ou supplémentaire au système qu'il représente. La fermeture transitive d'un graphe peut être utile pour analyser un graphe. Elle peut jouer le rôle de modèle, bien que son contenu soit plus riche (sur le plan syntaxique s'entend). En informatique, on constate fréquemment qu'une entité peut au cours de sa vie jouer séquentiellement ou simultanément le rôle de modèle et de système. Il est important de noter qu'une même entité ne peut se voir attribuer un rôle de système ou de modèle à un instant donné que vis-à-vis d'une autre entité. Une entité n'est pas un modèle ou un système par lui-même. Mieux un modèle et un système peuvent intervertir leur rôle respectif à certaines occasions. Les rôles ne sont pas figés.

Je préfère donc une définition moins restrictive.

*Une entité joue le rôle de modèle pour un système si elle vérifie le principe de substituabilité : être suffisante pour permettre de répondre aux questions que l'on souhaite se poser en lieu et place du système qu'il est censé représenter*.

Les réponses obtenues doivent être les mêmes que si l'on s'était servi du système lui-même. Un modèle est construit dans un but précis et les informations qu'il contient sont choisies pour être pertinentes vis-à-vis de l'utilisation qui sera faite du modèle.

Un modèle selon l'usage qui lui est attribué à un instant donné peut être classé dans l'une des deux catégories qui suivent.

- *Descriptif* : le modèle offre une représentation d'un système existant, mais c'est le système qui tient lieu de référence. On fait en sorte que le modèle puisse se substituer à lui pour l'usage prévu. La compréhension en phase de maintenance, la vérification de propriétés en phase de test sont des exemples d'activités conduisant à produire des modèles relevant de cette catégorie.
- *Prescriptif* : le modèle est la référence, il représente un système qui n'existe pas encore ou qui doit évoluer. Le modèle pose des contraintes que le système devra respecter. L'élaboration des cas d'utilisation ou de l'architecture d'une application sont deux exemples d'activités générant des modèles appartenant à cette catégorie

Un modèle peut également offrir une vue d'un système sur un continuum comportant deux points de vue extrêmes : *instance* ou *type* (Kühne, 2006).

Un modèle est de nature « instance » s'il existe entre le système et le modèle un morphisme $f$ qui n'opère aucune classification : c'est-à-dire injectif (Figure 3-10). Il y a éventuellement un filtrage, des informations ajoutées et un changement de langage mais rien de plus. Les éléments et liens non ajoutées du modèle sont en « lien 1-1 » avec certains éléments du système et ils ont une sémantique comparable (au sens du sous-typage sémantique, une « route nationale » est projetée vers « une route ») dans le langage du modèle et dans celui du système. Ce type de vue a un caractère transitif. Si un modèle $\mathcal{M}$ est de nature instance d'un modèle $\mathcal{M}'$ (système pour $\mathcal{M}$)

de nature instance d'un système $\mathcal{S}$ alors $\mathcal{M}$ est, par transitivité, un modèle de nature instance de $\mathcal{S}$.

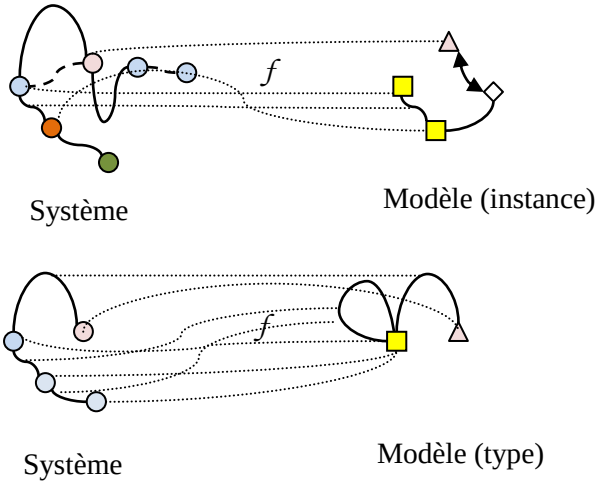


**Figure 3-10 Modèle offrant une vue "instance" ou "type" d'un système**

Un modèle complètement « type » capture, non pas les aspects singuliers des éléments d'un système $\mathcal{S}$, mais les propriétés universelles de ces éléments. Le lien entre le modèle et le système se manifeste par l'existence entre le système et le modèle d'un morphisme non injectif. Ce morphisme va induire une classification : il regroupe en « paquets », selon une nouvelle sémantique, des éléments et liens qui présentent des propriétés identiques. Ce morphisme va projeter parfois plusieurs éléments « différents » du système vers le même élément : par exemple « la route départementale X » et la « route départementale Y » vers l'élément « route départementale ». Comme toute fonction, le morphisme induit une relation d'équivalence $\mathcal{R}$ définie par $(x\mathcal{R}y \Leftrightarrow f(x)=f(y))$ sur les éléments du système $\mathcal{S}$. On peut voir un modèle de nature « type » comme l'ensemble quotient $\mathcal{S}$ /R. Les éléments du modèle sont les classes d'équivalence selon $\mathcal{R}$.

Entre ces deux extrêmes, il y a les modèles qui opèrent pour certaines parties du système des regroupements et pour d'autres non. Conservant des singularités à certains endroits et, au contraire, introduisant des propriétés universelles à d'autres. Un même modèle peut jouer le rôle de « type » ou de « token » pour deux systèmes différents. Par exemple, un diagramme de classes peut être le « token » d'un programme Java et le « type » d'une exécution de ce programme.

### 3.3.7 Métamodèle et métamétamodèle

Dans le monde de l'IDM, on distingue généralement trois niveaux de modélisation : le niveau « modèle » (référencé $M_1$), « métamodèle » (référencé $M_2$) et « méta-métamodèle » (référencé $M_3$). L'OMG, mentionne un quatrième niveau « instance » ($M_0$). Il correspond à la représentation d'un système en cours d'exécution. Cependant, la nature de la relation qui existe entre les modèles de niveaux $M_0$ et $M_1$ est différente de la nature des relations existant entre les modèles de niveaux $M_1$ et $M_2$, $M_2$ et $M_3$. Ces derniers entretiennent des liens de langage ($M_{i+1}$) à modèle ($M_i$) conforme à ce langage (l'un est une vue « type » de l'autre mais au sens *linguistique* (Kühne, 2006)). Alors qu'un modèle de niveau $M_1$ ne représente pas un modèle du langage d'un modèle de niveau $M_0$ mais une vue au sens *ontologique* : un élément de $M_1$ décrit les traits partagés de plusieurs éléments de $M_0$. C'est une vue « type » également mais avec un autre sens ontologique (Kühne, 2006).

A chaque niveau de cette architecture sont élaborés des modèles. C'est le système auquel est associé un modèle qui détermine son niveau d'appartenance. Il est donc de coutume pour

désigner un modèle de préciser son niveau de définition. On parle ainsi de «modèle des instances» pour un modèle défini au niveau M0, de « modèle » au niveau M1, de « métamodèle » au niveau M2 et de « métamétamodèle » au niveau M3.

Un *modèle* de niveau M1 est une représentation d'un système logiciel. Un modèle est un « mot » d'un langage de modélisation. Le langage de modélisation utilisé doit être, lui aussi, décrit. Un *métamodèle* (un modèle de niveau M2) est un modèle particulier dont l'objet est, le plus souvent, de modéliser la syntaxe abstraite du langage. Mais il peut également représenter la syntaxe concrète, voire la sémantique d'un langage. Un modèle est dit conforme à un métamodèle, s'il respecte les règles spécifiées par ce dernier. Pour pouvoir décrire les différents métamodèles associés à un langage, il faut disposer d'un langage. C'est un métalangage à l'instar d'EBNF qui permet de décrire les grammaires hors-contexte des langages. Dans le monde IDM, la description de ce métalangage prend lui aussi la forme de modèles de niveau $M_3$ désignés *métamétamodèles*. Ces modèles déterminent le paradigme utilisé dans les langages qui sont constructibles à partir d'eux. Pour éviter une progression infinie de ces niveaux méta, on fait en sorte que le métalangage s'auto-décrive. Le MOF et EBNF sont deux exemples de métalangage qui se décrivent eux-mêmes.

# Chapitre 4 Démarche de recherche

La première section de ce chapitre positionne mes travaux relativement aux domaines présentés dans le chapitre précédent et décrit les liens qu'ils entretiennent. Cette première section donne un aperçu synthétique de ce que furent mes activités pendant ces 7 dernières années. Elle s'attache à dessiner une esquisse de haut niveau du *quoi* et du *quand*. La section qui suit se focalise, elle, sur la manière dont j'ai travaillé. Elle aborde le *comment*.

## *4.1 Mon parcours thématique*

Mon domaine de recherche porte depuis 2003 sur la définition et l'usage de langages « annexes » (c'est-à-dire complétant les langages de développement) ; ceci dans le but de faciliter et d'améliorer le développement et la maintenance des applications logicielles, particulièrement celles conçues à l'aide de composants. Mes travaux se trouvent au carrefour de plusieurs disciplines du Génie Logiciel : le développement orienté composant, les architectures, la maintenance et l'évolution, la qualité et l'ingénierie dirigée par les modèles. Je vais retracer dans les sous-sections qui suivent mon parcours thématique au cœur de ces disciplines : le contrôle de l'évolution, la modélisation et l'exécution des bonnes pratiques et l'amélioration des concepts de l'ingénierie dirigée par les modèles.

### 4.1.1 Le contrôle de l'évolution

Fin 2003, mes activités de recherche concernaient la documentation rigoureuse et multi-langages des décisions architecturales (point 1 sur la Figure 4-1 et section 5.1 de ce mémoire). L'objectif était de faciliter la compréhension, lors des activités de maintenance, d'un modèle d'architecture en explicitant les liens unissant les motifs architecturaux implantés et les propriétés qualité du logiciel. Constatant que ces décisions, exprimées formellement, permettaient la mise en place d'un contrôle automatique des activités de maintenance, j'ai commencé à travailler fin 2004 sur la proposition d'un mécanisme multi-langages détectant l'altération potentielle de propriétés qualité (point 2 et section 5.2 du mémoire). Ces travaux constituent l'essentiel de la thèse de *Chouki Tibermacine* soutenue fin 2006.

Pour tenter d'étendre le contrôle précédent (limité à la perte de motifs architecturaux), aux modifications que peut subir un composant, une seconde thèse confiée à *Bart George* a débuté fin 2004 afin de définir un mécanisme de typage dans le monde des composants incluant les propriétés non fonctionnelles (point 3). L'objectif visé est d'alerter le concepteur en cas de modification ou substitution d'un composant ne respectant pas le principe de sous-typage et pouvant, en conséquence, altérer les propriétés non fonctionnelles du système.

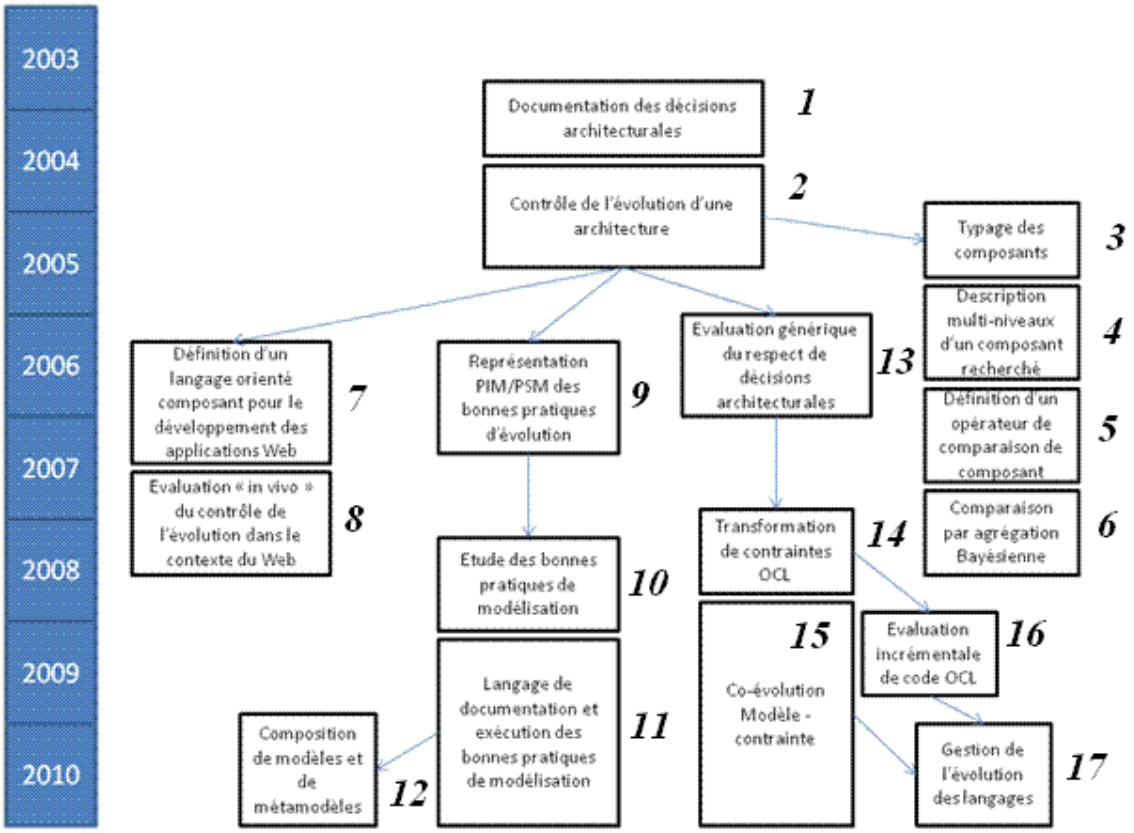


**Figure 4-1 Mon parcours thématique**

Nous avons rapidement constaté que la résolution pratique de ce problème dépassait le niveau de complexité que nous étions à même de gérer dans le temps d'une seule thèse. Nous nous sommes donc attelés à un travail moins ambitieux : la mise au point d'un procédé de recherche pour substituer un besoin (donné par l'architecte et non calculé) par un composant. Nous avons introduit pour cela le concept de « composant recherché » exprimé dans un langage ad hoc (point 4). Nous avons ensuite introduit un opérateur de comparaison automatique opérant sur ce langage basé sur les techniques de décisions multicritères (point 5). Il permet d'identifier dans un marché, le composant le plus proche du besoin exprimé. Ces deux derniers points ont constitué l'essentiel de la thèse de *Bart George* soutenue fin 2007. Une autre approche de comparaison de composants a été définie en 2008, en utilisant cette fois-ci les réseaux bayésiens (point 6). Les travaux relevant des points 3, 4, 5 et 6 sont détaillés dans la section 5.3 de ce mémoire.

Parallèlement à ce travail, une validation et un transfert industriel des travaux de *Chouki Tibermacine* a été entamé début 2006 sous la forme d'une thèse CIFRE avec pour doctorant *Réda Kadri*. La société partenaire *Alkante*, œuvrant dans le monde du web, voulait améliorer ses développements en migrant vers le paradigme composant. Un premier travail a défini un ADL spécifique (nommé `AlkoWeb`) pour ce domaine applicatif (point 7, section 5.2.6 du mémoire) sur la base duquel un contexte méthodologique et un outil support ont été développés. Nous avons ensuite entrepris, sur un cas d'étude interne à l'entreprise, d'établir l'intérêt et la pertinence du contrôle de l'évolution avec comme langage cible l'ADL développé (point 8, section 5.2.7 du mémoire). *Réda Kadri* a soutenu sa thèse début 2009.

### 4.1.2 La modélisation et l'exécution des bonnes pratiques

Courant 2006, dans le but, non pas d'améliorer la profondeur du contrôle de l'évolution (comme pour la thèse de *Bart George*) mais d'étendre le contrôle de l'évolution à d'autres bonnes pratiques, une thèse a été lancée sur la documentation et l'exécution des bonnes

pratiques d'évolution (thèse de *Soraya Sakraoui*). Les premiers travaux ont tenté de fournir un langage de documentation des bonnes pratiques identifiées par *Lehman* (Point 9). J'ai constaté l'impossibilité de donner vie à ces descriptions. En effet, ces bonnes pratiques sont trop générales et/ou impliquent l'orchestration de multiples outils. Cette thèse est désormais encadrée par *Chouki Tibermacine*.

Ce sujet a été poursuivi de mon côté mais dans un tout autre cadre début 2008 : les activités de modélisation. Ce domaine beaucoup plus restreint offrait de bien meilleures perspectives en termes d'automatisation. Une collaboration avec la société *Alkante* sous la forme d'une seconde thèse CIFRE a été lancée sur ce thème avec comme doctorant *Vincent Le Gloahec*. Son objectif est de proposer un langage de documentation des bonnes pratiques de modélisation indépendant des outils et une plate-forme pour leur contrôle dans les éditeurs sous ECLIPSE.

La validation prendra une importance toute particulière puisqu'il s'agit d'établir les moyens méthodologiques et technologiques pour qu'*Alkante* puisse, à terme, capitaliser et contrôler ses pratiques sur l'ADL `AlkoWeb`. Une importante étude sur la nature des bonnes pratiques de modélisation a été conduite (point 10). Des exigences ont pu être formulées. Elles ont conduit à un premier langage de documentation (point 11) et à un outil d'exécution associé (point 11). On trouvera le détail de ces travaux dans la section 6.1 de ce mémoire.

#### 4.1.3 Au cœur de l'ingénierie dirigée par les modèles

Dans le cadre des travaux précédents, j'étais un simple utilisateur des technologies de la mouvance de l'Ingénierie Dirigée par les Modèles, usant selon les besoins de langages et technologies de cette mouvance telles qu'OCL, MOF, UML, etc. Fort de l'expérience capitalisée sur le thème de la définition, de l'évolution et de l'outillage de DSL, une part croissante de mon activité s'oriente depuis 2009, vers l'ingénierie des langages de modélisation. D'utilisateur, je tente de devenir prescripteur.

Au cours de la thèse de *Chouki Tibermacine*, nous avions été confrontés à l'écriture et à la transformation de nombreux codes OCL (points 13 et 14) pour évaluer les décisions architecturales d'une manière générique. J'avais alors constaté les difficultés liées à l'évolution fréquente des modèles (la syntaxe abstraite des ADL ciblés et les architectures) sur lesquels reposaient nos codes OCL. J'ai donc proposé fin 2007 une thèse sur la coévolution code OCL/modèle (thèse en cours de *Kahina Hassam*). Ma réflexion s'est également portée sur le plan de l'évaluation incrémentale de codes OCL (point 16, sujet de Master Recherche de *Mohammed Said Belaid* en 2009). Plus généralement, je réfléchis désormais à l'impact de l'évolution des langages sur les plates-formes IDM et sur la définition de mécanismes facilitant la gestion de cette évolution (point 17).

La modélisation des bonnes pratiques introduit la notion de restriction langagière à certaines étapes d'un procédé de modélisation. La manière dont peut être formulée cette restriction m'a conduit à étudier les opérateurs de manipulation de métamodèles. Au travers d'un travail sur la sémantique de l'opérateur de composition de modèles, je réfléchis depuis fin 2009 à la proposition de mécanismes permettant de définir efficacement et à coût réduit de nouveaux langages partant de langages existants (point 12 et section 6.2 de ce mémoire).

### *4.2 Ma démarche de recherche*

Dans cette section, je commence par présenter ma vision sur la recherche en génie logiciel. Puis je mets en valeur l'importance fondamentale du facteur humain dans nos travaux et sa conséquence pour nous autres chercheurs que j'ai appelé « principe du compromis ». Je m'attarde ensuite sur l'épineux problème de la validation qui, à mon avis, est peu traité par nous autres chercheurs français. J'évoque successivement les aspects clés de la validation : définition du contexte, choix du type de validation et des techniques statistiques. Je conclus cette section par la description de ce que furent, en conséquence, les règles de conduite que j'ai respectées durant mon projet de recherche.

### 4.2.1 Le processus de recherche idéal

La recherche en Génie Logiciel peut prendre de multiples formes. Mais toute proposition doit, in fine, faire l'objet d'une preuve. Toute preuve inclut, à mon sens, la démonstration des lemmes qui suivent :

1. le problème abordé est effectivement constaté dans l'industrie ;
2. le problème ne connaît pas encore de réponse académique ou industrielle satisfaisante ;
3. la solution que l'on propose (concepts, langages, procédés et/ou outils) offre en théorie une réponse au problème ;
4. la solution est industrialisable et rentable *in vivo*.

La satisfaction des lemmes 2 et 3 est acquise a priori pour la grande majorité des travaux de la discipline par le simple fait qu'un article ou un mémoire de thèse est jugé au minimum sur sa capacité à se positionner dans l'espace des travaux du domaine et sur la démonstration de la pertinence de la proposition sur des scenarii typiques. Les points 1 et 4 sont, quant à eux, autrement plus délicats. Ce serait faire preuve de naïveté que de penser que l'on peut sans réserve faire confiance aux chercheurs pour garantir le point 1. Il est tentant de se laisser emporter, par quelques appétences personnelles, dans des méandres de peu d'intérêt pour les industriels du secteur. C'est un travers contre lequel tout chercheur de la discipline doit continuellement se prémunir. Le point 4 est, de loin, celui qui pose la plus grande difficulté dans notre communauté. Que de tourments pour un encadrant qui souhaite mettre sur pied avec un doctorant une validation rigoureuse. En effet, en dehors de quelques domaines relevant des techniques formelles pour lesquels la validation correspond à l'écriture d'une « preuve », la validation impose souvent la conduite d'une *expérimentation*.

Pour conduire une telle expérimentation :

- il est nécessaire d'avoir une idée précise de l'hypothèse que l'on souhaite confirmer et du contexte qui lui est associé ;
- il faut arrêter le type de validation à entreprendre ; ce choix est fonction de l'hypothèse et malheureusement aussi des ressources humaines, matérielles et financières dont on dispose ;
- il faut enfin que le protocole de validation évite les écueils de la « validité interne » (les résultats obtenus sont-ils fiables ?) et de la « validité externe » (les résultats sont-ils généralisables ?).

Plus généralement, il s'agit pour le chercheur de maîtriser certaines techniques de *l'Empirical Engineering* auxquelles, nous autres français, contrairement à nos homologues anglo-saxons, sommes peu formés.

Une expérimentation peut se faire en laboratoire ou en partenariat avec des entreprises. Le deuxième cas nécessite de convaincre un ou plusieurs partenaires industriels dont les préoccupations et l'échelle de temps sont bien différentes du monde universitaire.

Trop souvent éludé suite à ces difficultés, le quatrième lemme est pourtant l'unique moyen d'apporter la preuve indiscutable que ce qui est proposé est utile. Nos pairs l'ont d'ailleurs bien compris en instituant, dans la plus respectable des conférences de notre domaine (*ICSE*), comme critère d'acceptation (nécessaire mais non suffisant) l'existence dans l'article d'une validation digne de ce nom.

Ce point est suffisamment sensible pour que je m'y attarde en faisant part de mon expérience personnelle.

### 4.2.2 Le cinquième élément

Tout entier concentré vers son objectif, il est tentant pour un chercheur de se livrer à un exercice intellectuel entièrement tourné vers l'élégance d'une construction théorique, oubliant du même coup le contexte auquel devra se confronter la solution proposée. Or, il est fondamental de garder à l'esprit, qu'au bout du compte leur mise en œuvre repose sur la matière la plus noble et la plus instable qui soit : l'humain. C'est elle qui d'un côté prouve, résout, crée, innove mais de l'autre se montre craintive, rétive, se refuse à certains usages ou changements. De plus, l'homme adapte en permanence, sur la base de son expérience et de ses aptitudes, son comportement à l'environnement qu'il perçoit. Ainsi, les réactions face à tel ou tel langage, procédé ou outil de développement diffèrent d'un développeur à l'autre, d'un contexte à l'autre.

Il est donc plus facile pour un chercheur de produire une idée dont il peut prouver la valeur sur le papier qu'une idée utile dans la vraie vie. Il n'y a pas équivalence, mais implication. Tous les développeurs ne disposent pas du temps nécessaire pour découvrir les arcanes de tel ou tel catalogue de styles ou la documentation proustienne d'une infrastructure. Plus encore, tous les développeurs ne sont pas aptes à jongler avec une multitude de modèles (et donc de langages) multipliant les abstractions et nécessitant, pour être correctement utilisés, de disposer d'un solide bagage en mathématiques. La pertinence théorique d'une proposition est une condition nécessaire mais non suffisante. La théorie doit trouver des appuis dans un monde animé par des forces qui ne peuvent être mises en équation. Plus encore, la meilleure proposition sur le plan théorique ne sera pas forcément la meilleure *in vivo*.

Cette situation, je l'ai vécue à plusieurs reprises. Responsable qualité d'une PME souhaitant briguer un certificat ISO 9001 pendant ma thèse, j'avais à ma charge, comme l'exige la norme, de définir, documenter, de mettre en application et contrôler les pratiques de développement. J'ai rédigé, à l'époque, de multiples fiches de méthode pour des sujets aussi différents que la documentation des codes, la gestion des numérotations des versions et des configurations ou le suivi des demandes de modification client. Ces fiches me semblaient bonnes en « théorie », car conformes aux « règles de l'art » de la littérature qualité. Mais certains procédés décrits ne furent jamais appliqués. Les développeurs concernés se refusant à respecter des consignes qu'ils jugeaient, selon les cas, inutiles, disproportionnées ou mal expliquées. Plus récemment, confronté à la validation des travaux de deux doctorants, j'ai rencontré des difficultés analogues. Des techniques séduisantes sur le papier se sont révélées, pour certains aspects, (trop) exigeantes en pratique. Le choix d'OCL comme langage de documentation est un exemple de « bonne idée » sur le papier mais discutable en pratique.

J'ai mis du temps à l'admettre mais, le plus souvent, les développeurs avaient raison. L'être humain est ainsi fait qu'il recherche en permanence un *compromis* : retour sur investissement /effort. Tout effort supplémentaire demandé doit donner lieu à un gain dont le ressenti doit être en rapport avec les efforts consentis. La difficulté est grande sur ce point dans la mesure où le seuil optimal est variable d'un individu à l'autre, selon les projets, les entreprises et les domaines applicatifs. Tout l'enjeu de la recherche en Génie Logiciel est donc de pousser toujours plus loin les principes de découpage, de réutilisation et de formalisation tout en usant d'artifices pour en masquer l'âpreté ou tout du moins rendre « supportable » au plus grand nombre leur fréquentation. Les graines du progrès ne sont fertiles que si elles sont aptes à germer dans les sols qui leurs sont destinés. Le chercheur doit lui aussi déterminer ce point de compromis et vérifier qu'il est partagé par les opérateurs humains cibles de sa proposition.

### 4.2.3 Définir le contexte et l'hypothèse du travail

Il me semble donc qu'une condition nécessaire à la réussite d'un travail de recherche se proposant de définir de nouveaux moyens pour améliorer les développements est d'identifier au préalable et précisément :

- les *personnes concernées* par le problème que l'on cherche à résoudre : leur niveau de qualification, leur expérience, leurs motivations, etc. ;

- le *contexte* dans lequel ces personnes opèrent : le type d'entreprise, le type de projet, les domaines applicatifs, les outils et procédés utilisés, les techniques de gestion de projet et d'évaluation.

Ce n'est qu'une fois ces données connues, que le chercheur peut commencer à réfléchir à la définition d'une solution (langages, procédés et/ou outils). Cette solution devra être adaptée au public visé dans son contexte opérationnel. Elle doit pour cela maintenir dans un certain intervalle les efforts demandés tout en maximisant le retour sur investissement. Pour cela, deux armes complémentaires sont utilisables : l'automatisation de tout ce qui peut l'être pour masquer la complexité des traitements à entreprendre et pour ce qui ne peut pas l'être, offrir une présentation des données à saisir et des décisions à prendre aussi conviviale que possible.

La validation du travail de recherche doit, elle aussi, s'appuyer sur les deux types de données précédentes. Une validation n'est pertinente qu'à la condition de limiter ses contours au public concerné et cela dans leur contexte opérationnel.

Il faut ensuite définir avec la plus grande précision l'hypothèse que l'on souhaite étudier en conduisant une validation. Cela suppose de :

- bien préciser les termes employés ;
- faire correspondre aux phénomènes à manipuler (type de méthode), contrôler (expérience, formation, etc.) et observer (qualité, délais) des entités mesurables (nombre d'années de développement, nombre de fautes découvertes en test, temps de modélisation, etc.);
- formaliser la nature des relations que l'on souhaite valider entre les phénomènes (les corrélations ou plus difficilement les causalités, la comparaison, etc.).

Cette hypothèse est, dans l'idéal, formellement documentée sous la forme d'un ensemble de couples : (*hypothèse nulle* $H_0$, *hypothèse alternative* $H_a$). L'hypothèse $H_0$ correspond à la négation d'une quelconque relation entre certaines variables indépendantes et dépendantes. L'hypothèse alternative est son contraire. Une telle hypothèse $H_0$ peut être par exemple que la durée d'un développement avec la nouvelle méthode A soit supérieure à celle obtenue avec l'ancienne méthode B. Cette formalisation facilitera grandement la mise en place du protocole de validation sur le plan pratique et statistique.

### 4.2.4 Définir le type de validation à mettre en œuvre

Il existe plusieurs techniques de validation. Il est même souvent nécessaire d'utiliser non pas une seule technique mais une combinaison de techniques pour obtenir une preuve solide mêlant aussi bien des aspects *quantitatifs* (pour mettre en valeur des phénomènes) que *qualitatifs* (pour expliquer, donner du sens à ces phénomènes).

Toutes ces techniques ont des avantages et des inconvénients. Le choix des techniques à utiliser dépend de plusieurs paramètres et en particulier : de la nature de l'hypothèse, de la préexistence de théories, de la disposition de données antérieures, de l'accès à certaines ressources (par exemple, des étudiants, des professionnels, des entreprises partenaires, des annuaires d'entreprises) et à des moyens financiers et organisationnels adéquats.

Il est habituel de distinguer quatre types de validation.

- L'expérimentation contrôlée (*controlled experiment*) : souvent réalisée en laboratoire, elle offre le cadre le plus rigoureux pour vérifier une hypothèse sous réserve que celle-ci se limite à un nombre restreint de points. Il s'agit de manipuler des *variables indépendantes* (par exemple le type de technique d'inspection de code utilisé) et de mesurer leur effet sur des *variables dépendantes* (par exemple le nombre des fautes détectées dans le code) tout en se prémunissant des *variables parasites* (connues ou non) pouvant affecter les résultats (par exemple l'expérience des développeurs). Pour garantir la validité interne des résultats obtenus, il est nécessaire d'identifier et de contrôler un maximum de ces variables parasites. On

parle alors de *variables contrôlées*. Mais plus on multiplie les contrôles, plus il est nécessaire de restreindre le champ de l'expérimentation.

- Le cas d'étude (*case study*) : on réalise une étude dans une entreprise, sur un projet typique dans le but d'extraire des hypothèses nouvelles ou de confirmer une hypothèse. Cette approche s'utilise dans des situations où le contexte joue un rôle important (le stress des développeurs, le type de management) ou quand les phénomènes s'observent sur plusieurs plans et dans la durée ; c'est-à-dire dans des situations dans lesquelles on ne peut pas manipuler toutes les variables indépendantes et/ou contrôler toutes les variables parasites. La validité interne est donc beaucoup plus difficile à démontrer. Par contre, il est possible d'embrasser un champ d'étude plus large que pour une expérimentation contrôlée. On peut également tenter une analyse plus en profondeur des phénomènes et de leurs causes.
- Le sondage (*survey*) : il s'agit d'une étude conduite sur plusieurs projets et dans plusieurs entreprises que l'on choisit pour être aussi représentatifs que possible (pour éviter les biais liés à l'échantillonnage). Ce type d'étude permet d'acquérir des informations plus superficielles que pour les deux approches précédentes mais à une plus grande échelle. On utilise surtout des questionnaires et des interviews pour collecter des informations sur le déroulement passé de ces projets. Le choix des questions et leur formulation sont primordiaux pour être sûr que tous les intervenants comprennent les mêmes choses.
- L'analyse post-mortem (*post-mortem analysis*) : un projet typique est analysé rétrospectivement par exemple en étudiant ses documentations, des traces ou en réalisant des interviews. Une étude plus approfondie que dans le cadre d'un sondage est possible mais il se pose le problème de sa généralisation.

Ces quatre types de validation peuvent intégrer des volets quantitatifs ou qualitatifs. Les expérimentations contrôlées et les cas d'étude sont bien adaptés à des travaux prospectifs : étudier la pertinence de nouveaux outils, procédés ou langages. Les sondages et les analyses post-mortem sont utiles pour étudier les effets de techniques plus anciennes, déjà répandues dans l'industrie.

Mes propres travaux relèvent donc des deux premières techniques de validation. Les validations que j'ai réalisées ont toutes été des « cas d'étude ». Ce choix s'est imposé autant par manque de moyens que par ma crainte de ne pas maîtriser assez les techniques statistiques associées à la réalisation d'expérimentations contrôlées.

### 4.2.5 Bien utiliser les techniques statistiques

Une validation surtout si elle suit une approche quantitative impose toujours d'utiliser des techniques statistiques. L'analyse des données récoltées, souvent nombreuses et difficiles à obtenir, conduit à l'usage de deux types de statistique.

- *Statistique descriptive* : pour représenter les données sous une forme plus accessible avec des instruments comme la médiane, l'écart type, les histogrammes, les coefficients de corrélation, les régressions, etc. ;
- *Statistique inductive* : pour formuler en terme probabiliste un jugement sur une population à partir des résultats obtenus sur un échantillon extrait de cette population avec des tests d'hypothèse.

La première catégorie d'outils statistiques se manipule assez facilement. Par contre, la seconde pose plus de problèmes alors qu'elle se trouve être la plus utile en pratique. Le test d'hypothèse consiste à dire si, avec un risque d'erreur choisi, les données récoltées durant l'expérimentation sont incompatibles avec les hypothèses nulles $H_0$. Dans ce cas, on rejette les $H_0$, sinon on les accepte.

La réalisation d'un test d'hypothèse suppose le suivi d'un protocole précis :

- choisir le test statistique approprié ;

- spécifier un niveau de signification et la taille de l'échantillon ;
- trouver la distribution d'échantillonnage du test sous $H_0$ ;
- définir la région de rejet ;
- calculer la valeur du test statistique à l'aide des données.

Chaque étape est un piège potentiel pour le profane. Mais l'étape la plus délicate me semble être le choix du test statistique. Ce choix doit être rationnel et guidé par trois critères : la façon dont l'échantillon a été réalisé, la nature de la population de laquelle a été tiré l'échantillon et la nature des mesures réalisées. Les tests paramétriques (Student, ANOVA, Walsh, etc.) sont plus puissants que les tests non paramétriques (Fisher, Khi-deux, Wilcoxon, etc.) mais ils imposent des conditions particulières sur les données (des distributions normales par exemple).

Si les techniques à mettre en œuvre sont routinières pour des statisticiens, leur emploi par des informaticiens est beaucoup plus délicat (Kitchenham, et al., 2002). L'idéal serait de pouvoir constituer des binômes informaticien/statisticien. Choses délicates dans la mesure où nos collègues n'ont pas d'intérêt scientifique à aider dans l'utilisation de techniques « communes ».

### 4.2.6 Ma démarche de recherche

Je ne prétends pas avoir complètement et parfaitement satisfait les 4 lemmes évoqués au début de cette section pour l'ensemble de mes travaux. J'ai par contre eu le souci permanent, modulo mes compétences et les moyens dont je disposais, d'établir des preuves aussi « optimales » que possible. Pour ce faire, mes activités de recherche durant ces 7 dernières années ont fait l'objet d'une démarche respectueuse de deux grands principes.

Le premier d'entre eux est que les problématiques à l'origine de mes travaux devaient être issues ou confirmées par des préoccupations industrielles. J'ai pour cela puisé dans les préoccupations universellement admises dans la littérature mais également dans celles d'une société partenaire de mon équipe de recherche : la société *Alkante*, une PME Rennaise leader français dans le domaine des applications Web à base de Systèmes d'Information Géographique. Ce principe garantissait, a priori, la preuve du premier lemme.

Le second principe est que l'établissement d'une démonstration du type de celle décrite précédemment pouvait profiter d'une approche en deux temps. Une première thèse devait s'attacher à vérifier le point 2 puis dégager les bases théoriques d'une solution dont on cherchait à montrer la pertinence *a minima* sur le papier (point 3). Une seconde thèse, à vocation industrielle cette fois-ci (idéalement en convention CIFRE), légèrement décalée dans le temps (environ deux ans), assurait son transfert en déterminant un cadre méthodologique opérationnel et en développant un outillage support adéquat. Elle se chargeait ensuite d'évaluer l'intérêt de la solution proposée *in vivo* dans le cadre de projets ad hoc (point 4) et enfin prolongeait la réflexion en proposant des axes d'amélioration. Ces axes pouvant devenir le point de départ d'un nouveau cycle de recherche.

Je trouve ces deux principes particulièrement féconds et adaptés à la recherche dans le domaine du Génie Logiciel dont l'objet est la proposition de méthodes et outils capables (dans les faits et non simplement sur le papier) d'améliorer la qualité des (vrais) développements.

J'ai eu la chance au gré des financements de thèse de pouvoir respecter ces deux principes pour deux de mes trois propositions : pour le contrôle de l'évolution (thèse de *Chouki Tibermacine* soutenue en 2006 et *Réda Kadri* soutenue en 2009), et la modélisation des bonnes pratiques (thèse de *Soraya Sakhraoui* débutée en 2006 et de *Vincent Le Gloahec* débutée en 2008). Le travail sur la sélection de composants (thèse de *Bart George* soutenue en 2007) ne put faire malheureusement l'objet d'un transfert industriel, ni même d'une validation *in vivo*. Une validation significative bien que non industrielle fut cependant réalisée. Un transfert sur ce thème pourrait cependant être réalisé à moyen terme avec la société MGDIS de Vannes dans le cadre de la thèse de *Allier* débutée fin 2008.

# Chapitre 5 Les travaux sur les architectures et les composants

La section 1 présente une technique de documentation multi-langages des décisions architecturales. La section 2 détaille une méthode facilitant la préservation de propriétés non fonctionnelles lors de l'évolution « à froid » d'une architecture à base de composants. Cette méthode s'appuie sur la documentation précédente. Enfin, la section 3 introduit une technique automatique de recherche de composants basée sur un langage dédié permettant de décrire à différents niveaux d'abstraction le composant recherché.

## *5.1 Documentation des décisions architecturales*

Début 2003, avec un collègue MCF (*Salah Sadou*), profitant de la restructuration complète de notre laboratoire de rattachement, le VALORIA, nous avons décidé de nous intéresser au thème de la maintenance dans le cadre particulier des applications conçues à l'aide de composants. Ce collègue apportait des connaissances sur l'adaptation dynamique des composants. De mon côté, je pouvais réinvestir mes compétences sur les processus de développement et la qualité développées lors de ma thèse de doctorat tout en m'intéressant aux techniques de métamodélisation émergentes à l'époque. Une équipe de recherche fut crée. Elle se proposait de définir des méthodes, des techniques et des outils facilitant l'activité de maintenance tout en diminuant ses coûts dans le domaine des logiciels à base d'objets et de composants.

Un sujet de Master Recherche sur ce thème fut déposé et pourvu par *Chouki Tibermacine* dans la formation doctorale « Système Distribué » de l'université *Paris VI* en 2003. Ce stage de Master permis de déterminer plus finement nos ambitions à court terme en plaçant nos efforts sur une étape particulière du processus de maintenance : l'étape de compréhension d'une application. Ce positionnement semblait judicieux dans la mesure où plusieurs études avaient établi que cette étape est l'une des plus coûteuses de ce processus.

### 5.1.1 Le problème posé par l'étape de compréhension

Avant d'ajouter, de retirer ou de modifier des propriétés fonctionnelles et non fonctionnelles à une application, il est nécessaire d'acquérir sur sa structure un niveau de connaissance suffisant. Il s'agit, non seulement de reconstruire une image de ses réelles aptitudes fonctionnelles et non fonctionnelles (ce que fait dans sa version actuelle cette application), mais également de la manière dont ces aptitudes ont été obtenues (quelles ont été les décisions architecturales prises). Cette étape clé de tout processus de maintenance est d'autant plus facile que la documentation fournie avec le logiciel est de qualité. Cette documentation doit, en particulier être complète, pertinente, à jour et non ambiguë. Or, les documentations jointes aux applications ne respectent que très rarement ces propriétés. En effet, les différentes versions d'un logiciel se répartissent sur des périodes suffisamment longues pour que toutes les informations qui ne sont pas dûment archivées soient perdues et cela, d'autant plus facilement que plusieurs développeurs, souvent soumis à une pression importante sur leurs délais d'action, se succèdent. Pour tenter de pallier les déficiences des documentations, on peut faire l'usage de techniques issues de la mouvance de la rétro-ingénierie (*Reverse Engineering*) et de la compréhension des programmes (*Software Comprehension*).

Quoi qu'il en soit, toutes ces techniques se cantonnent pour le moment à dégager des visions (graphes des appels, diagrammes de classes, etc.) ou des abstractions (patrons de conception) très (trop) proches du code source. De telles vues ne permettent pas de dégager des traits architecturaux de plus haut niveau, préalable essentiel à la bonne compréhension d'une application. En supposant même que l'on dispose de techniques offrant les niveaux de

visualisation adéquats, la reconstruction automatique du lien unissant le «pourquoi» (l'objectif recherché au travers d'un choix architectural) au «comment» (le choix architectural constaté) semble un vœu pieux. La documentation reste donc le seul outil fiable capable de maintenir à chaque étape du développement le lien entre une spécification et son implantation. Tout le problème est, alors, de garantir non seulement la présence d'une telle documentation, mais également sa mise à jour lorsque nécessaire.

### 5.1.2 L'étape de compréhension dans le monde des composants

Dans le cadre des applications conçues à l'aide de composants, l'étape de compréhension entrelace invariablement deux types de découverte :

- celle de chaque composant considéré individuellement ;
- celle de leur structure d'interconnexion ; ce que l'on appelle l'architecture de l'application.

Toute la difficulté est alors de redécouvrir les propriétés d'un système au regard des choix de conception réalisés pour les mettre en œuvre en usant des informations glanées lors de ces deux types de parcours. Il faut substituer à une vision physique (des composants liés via des connecteurs), une vision logique (un ensemble de décisions architecturales) (Bosch, 2004). Les outils facilitant la découverte d'un composant particulier, essentiellement des techniques documentaires, donnaient lieu à de nombreux travaux en particulier autour de la notion de *contrat*. Il nous est par contre apparu que les concepts et outils utiles à la découverte d'une architecture étaient un domaine peu traité. Avec le recul et comme le confirme (Kruchten, et al., 2009) qui positionnent les premiers travaux sur la documentation explicite des décisions architecturales sur la période 2004-2008, nous étions dans les tous premiers à aborder ce thème.

Ce manque d'intérêt nous semblait d'autant plus préjudiciable que l'activité de compréhension d'une application conçue à l'aide de composants est essentiellement guidée par l'architecture. En effet, c'est l'architecture qui détermine l'éclairage à porter sur chaque composant. Sur un plan fonctionnel, une architecture place tout composant dans un « espace de sollicitation » donné qui fixe en conséquence son « espace d'action » ; à l'instar d'une fonction mathématique $f$ dont on viendrait restreindre, par composition avec une fonction $g$, le domaine de définition et du même coup l'ensemble image. La difficulté est bien sûr plus grande dans le monde des composants car d'une part cette composition est potentiellement bidirectionnelle de par l'existence de connecteurs qui le sont ($f$ influence $g$ en retour) et d'autre part car elle ne se limite pas à une chaîne d'influence ($f \circ g \circ h \circ \ldots$) mais prend la forme d'un graphe d'influence (plus de deux fonctions peuvent interagir).

Plus encore, l'architecture porte en elle-même des informations que ne sauraient révéler, un examen individuel attentif des parties qui la compose. Les motifs apparaissant dans une architecture, par exemple ceux insérant de la réplication ou des composants façades, sont dictés par la recherche de propriétés non fonctionnelles telles que la fiabilité, la portabilité, etc. Une architecture véhicule, en elle-même, une part non négligeable des propriétés d'un système (Bass, et al., 2003). Or, la mise à jour de motifs et plus encore des propriétés qui leur sont associées est une tâche ardue en l'absence de documentations ad hoc. Un même motif peut adresser plusieurs propriétés, certaines pouvant être effectivement recherchées et d'autres non. Plus encore certains motifs peuvent n'être que le fruit du hasard.

La Figure 5-1 illustre ce propos. Elle présente l'architecture d'un composant gérant le contrôle d'accès à un bâtiment. Son architecture n'a pas été seulement dictée par les simples faits d'implanter telle ou telle fonctionnalité ou de disposer de tel ou tel composant sur étagères. Des choix architecturaux ont été faits pour obtenir certaines exigences qualité. Ainsi, le composant `DataAdminRetrieval` a été introduit pour découpler les formats de données pris en charge par le composant de ceux usités à l'intérieur de celui-ci. Ce composant joue le rôle de « façade » au sens des patrons de conception. Ce découplage permet au composant de respecter l'exigence qualité qui avait été formulée dans sa spécification initiale : « Le composant devra pouvoir

facilement prendre en charge de nouveaux formats de données ». De même, la séquence de sous-composants `AccessCtlDup` et `LoggingDup` est une duplication architecturale de la fonction d'authentification. En effet, si la séquence normale `AccessCtl` et `Logging` n'est pas utilisable du fait, par exemple, d'une absence de données en entrée via le port `DataManagement`, il sera toujours possible pour une liste restreinte de personnes (dont les autorisations sont stockées localement dans le sous-composant `AccessCtlDup`) d'accéder au bâtiment. Cette duplication architecturale fut introduite pour garantir un haut niveau de disponibilité (même dans un mode dégradé) telle que réclamé dans la spécification du composant.

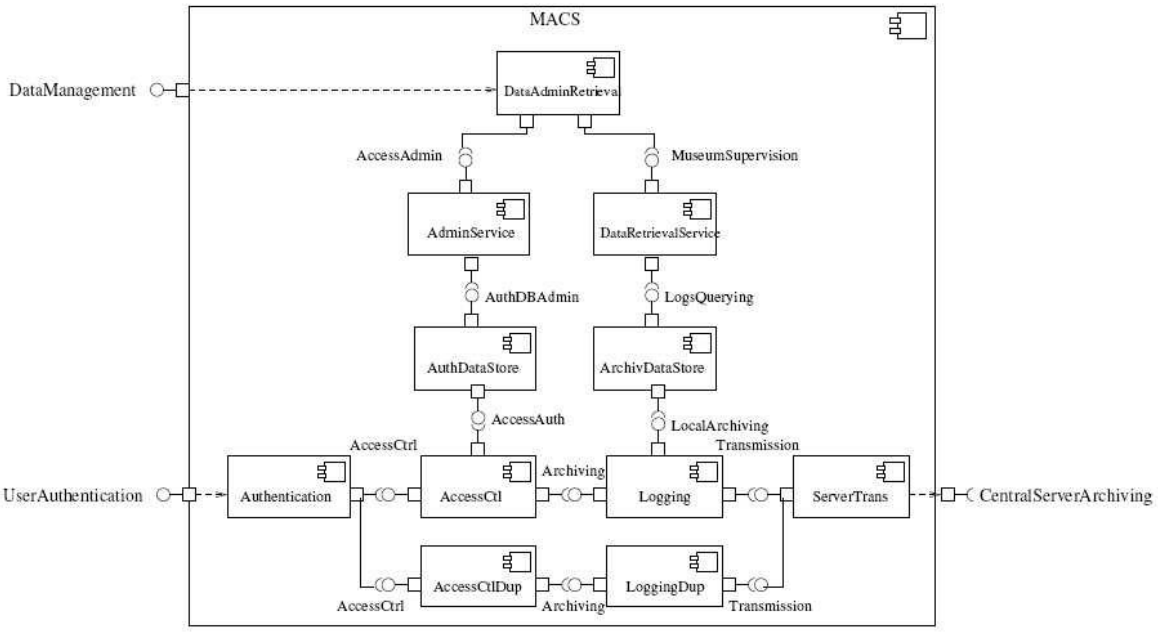


**Figure 5-1 Un exemple d'architecture**

Cet exemple illustre bien l'importance des aspects qualité sur les choix architecturaux réalisés lors de la conception d'une application. L'identification de l'apport réel d'un composant particulier ne peut donc être entreprise qu'au travers du filtre que constitue l'architecture dans laquelle il prend place. Et cette architecture ne peut être comprise qu'à la lumière des intentions, tout particulièrement non fonctionnelles, qui en sont à l'origine.

### 5.1.3 Importance des décisions architecturales

La connaissance des liens unissant propriétés non fonctionnelles et choix architecturaux est donc du plus grand intérêt pour les personnes en charge de la maintenance. Si l'architecture est bâtie sur la base d'une recherche de certaines propriétés, la construction d'une image mentale suffisante de cette architecture, préalable à toute modification, passe nécessairement par la reconstitution de cette connaissance. Faciliter cette reconstitution, c'est diminuer d'autant les efforts à consentir lors de la coûteuse phase de compréhension de la structure existante.

Si la notion d'intention (« Design Rationale ») était connue dans le monde de l'architecture depuis de nombreuses années (Perry, et al., 1992), rares étaient en 2003 les travaux portant explicitement sur sa documentation. On se concentrait à l'époque essentiellement sur la définition de langages (les ADL) permettant de décrire l'effet des décisions, l'architecture résultante, et non les décisions elles-mêmes. Nous avons donc décidé de nous concentrer dans un premier temps sur la mise en place d'une documentation rigoureuse explicitant les décisions architecturales : les liens unissant les motifs architecturaux aux propriétés non fonctionnelles dont ils visent l'obtention.

Il nous fallait donc concevoir un langage pour décrire :

1. des motifs architecturaux ;
2. des propriétés non fonctionnelles ;
3. des liens unissant les motifs architecturaux et les propriétés non fonctionnelles.

La rigueur souhaitée pour lutter contre toute forme d'ambiguïté ne devait cependant pas conduire à un langage à la syntaxe et à la sémantique trop éloignées des compétences des acteurs qui devront en faire usage (les architectes et les développeurs). Nous avons tenté autant que possible de résoudre cette quadrature du cercle.

Je vais dans un premier temps détailler la solution proposée pour la description de motifs. Je m'intéresserai ensuite aux deux derniers points.

### 5.1.4 Un langage de documentation de motifs architecturaux

Nos premières réflexions portèrent sur la proposition d'un langage de description de motifs architecturaux. Ces réflexions furent conduites fin 2004 durant les premiers mois de la thèse de *Chouki Tibermacine*. A l'époque, de très nombreux ADL existaient. Aucun ne semblait véritablement s'imposer et plus encore ils offraient tous une certaine complémentarité notamment en termes de couverture du cycle de vie. Nous nous étions donc fixé comme hypothèse de travail que le langage proposé pour décrire des motifs architecturaux devait pouvoir être applicable indifféremment à tous les langages. Pour répondre à ce besoin de généricité, j'ai proposé de définir non pas un langage mais une famille de langages. Cette famille fut désignée par l'acronyme *ACL* (pour *Architecture Constraint Languages*). Elle comporte potentiellement autant de langages que d'ADL utilisables dans le cadre d'un développement logiciel. Elle a été construite un peu dans un esprit « ligne de produit ». Les éléments communs à tous les langages (les opérateurs de contraintes et de navigation) ont été regroupés et séparés des éléments qui les distinguent (les concepts langagiers propres à chaque ADL).

Dans le but de faciliter la définition et plus encore de diminuer autant que possible les temps d'appropriation de ce langage, j'ai tenu à créer cette famille sur la base de langages qui m'apparaissaient du niveau de compétence des utilisateurs cibles : les langages OCL et MOF portés par l'OMG. L'idée était de considérer un modèle comportant un motif architectural particulier comme appartenant à un langage imposant à tous ses mots (modèles) le respect du motif en question. Spécifier un motif revient donc à définir le sous-langage de l'ADL considéré dont tous les mots (modèles) comportent le motif. Définir ce sous-langage, c'est opérer une restriction langagière sur le langage de l'ADL et vérifier le respect d'un motif par un modèle revient à établir que ce modèle appartient bien au (sous-) langage imposant ce motif. Or, MOF est un langage permettant de décrire la syntaxe abstraite de langages et OCL le moyen de restreindre le nombre des mots d'un langage défini en MOF. Il était donc envisageable d'user de ces deux langages pour définir des sous-langages et donc des motifs.

Si un métamodèle MOF décrit la syntaxe abstraite d'un ADL, une contrainte OCL qui lui est adjointe, exprime que seules certaines architectures (i.e. modèles) sont dérivables (i.e. instanciables dans l'univers MOF) dans ce langage. Par exemple, on peut imposer que, dans toute architecture, les composants aient moins de 10 interfaces requises, en posant cette contrainte dans le contexte d'une métaclasse `Composant`. Cette contrainte est exactement du type de celles que nous souhaitons exprimer. Cependant avec la sémantique d'OCL sa portée est globale. Elle doit être respectée par tout modèle et dans chaque modèle par tout composant et non à un composant particulier d'un modèle particulier comme nous souhaitons le faire. Il fallait donc trouver le moyen de restreindre la portée d'une contrainte OCL. Ce mécanisme de restriction fut défini sur la base d'une modification de la syntaxe et de la sémantique du champ contextuel d'OCL (introduit par le mot clé *context*).

=

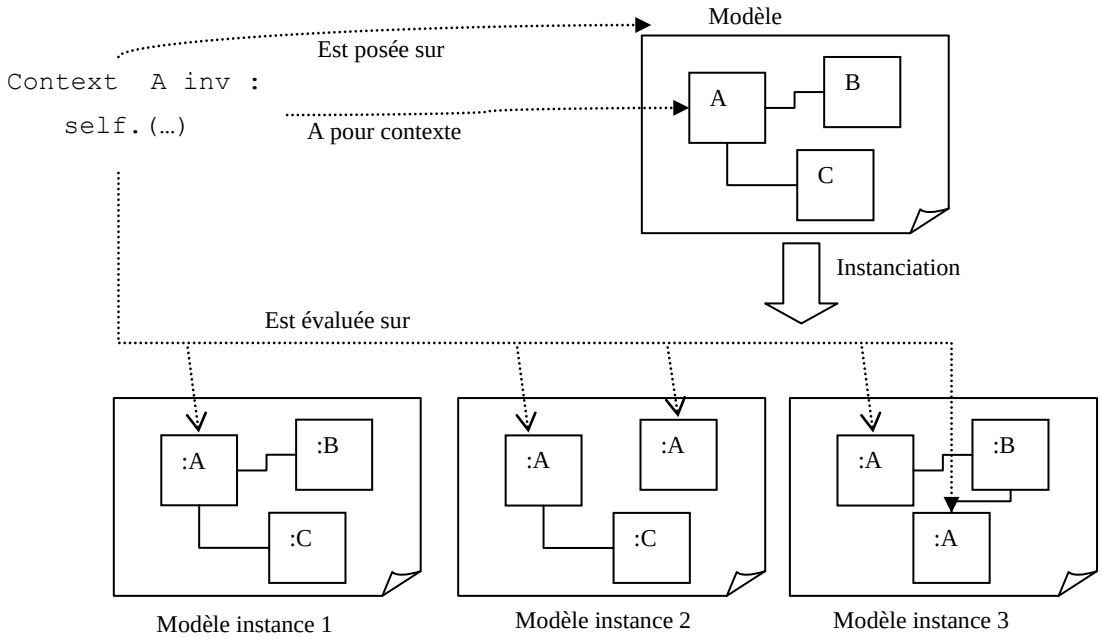


**Figure 5-2 Définition et évaluation d'une contrainte OCL**

Une contrainte OCL est posée sur un modèle (typiquement un modèle MOF ou un diagramme de classes) dans lequel elle navigue (Figure 5-2). Dans ce modèle elle est écrite explicitement dans le contexte d'un `classifier` du modèle (typiquement une métaclasse d'un modèle MOF ou une classe d'un diagramme de classes). Son évaluation est effective à tout instant et s'applique à toutes les instances du `classifier` dans tous les modèles instances du modèle qui porte la contrainte. J'ai donc proposé de modifier la syntaxe d'OCL pour permettre une évaluation comportant un double filtrage :

- Un premier filtrage limite l'évaluation de la contrainte à un seul modèle instance ;
- Un second filtrage permet, si on le souhaite, de limiter l'évaluation dans ce modèle instance à quelques instances seulement du `classifier`.

Le premier niveau de filtrage est obtenu en posant la contrainte non plus sur le modèle dans lequel elle navigue mais sur le modèle instance auquel on veut limiter l'évaluation. Le second niveau de filtrage exploite la possibilité de donner un nom dans le champ *context* d'OCL pour désigner une instance dans la contrainte (en l'absence de nom, le mot clé utilisé est *self*). J'ai proposé de donner dans ce champ la liste des instances du `classifier` auxquels on souhaite limiter l'évaluation.

Pour mieux comprendre ce mécanisme de double filtrage, voici un petit exemple. On supposera que le métamodèle jouet $\mathcal{MM}_j$ de la Figure 5-3 est celui associé à un ADL quelconque. $\mathcal{MM}_j$ comprend 3 métaclasses (`Composant`, `Interface` et `Opération`) et 3 associations qui décrivent l'univers suivant : un composant affiche des interfaces, une interface a un nom et regroupe un ensemble d'opérations. On supposera également que nous voulions contraindre un modèle d'architecture $\mathcal{M}$ appartenant au langage de cet ADL contenant plusieurs composants dont deux d'entre eux s'appellent `Toto` et `TiTi`. Si nous posons la contrainte suivante sur ce modèle $\mathcal{M}$.

```
context Toto, Titi : Composant inv:
(self.interface->select(i|i.nom="Maison")->size()=1)
and
(self.interface->forAll(i|i.operation->size()<7))
```

La contrainte est compilée comme si elle était posée dans le métamodèle. Si l'on ne tenait pas compte de notre mécanisme de restriction de contexte, elle exprimerait que l'on ne peut pas produire de modèles de composants dans lesquels : un composant ne disposerait pas d'une et une seule interface de nom `Maison` et comporterait plus de 6 opérations par interface. Cependant avec notre mécanisme de réduction, nous réduisons la portée de cette contrainte au modèle $\mathcal{M}$ (qui porte la contrainte) et plus encore à seulement deux instances de la métaclasse `Composant` (les composants `Toto` et `Titi`) de ce modèle. De tous les composants apparaissant dans $\mathcal{M}$, seuls ces deux composants devront la respecter.

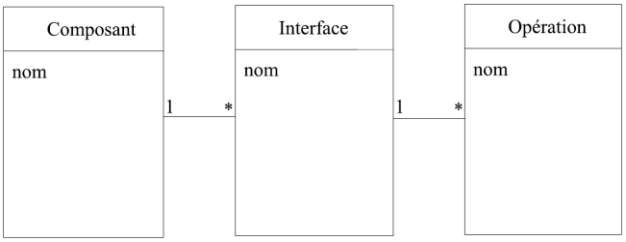


**Figure 5-3 Un métamodèle jouet MMj**

Il était difficile de prévoir tous les types de motifs et donc de contraintes que les développeurs pourraient être amenés à exprimer. Néanmoins, il est certain que des contraintes imposant des styles architecturaux, des patrons de conception, des règles de modélisation et de codage émanant de plans qualité devaient pouvoir être écrites. Ces contraintes devaient pouvoir être exprimées avec ou sans passage à l'échelle. Il est en effet différent de dire : « les trois sous-composants respectent un style en *pipeline* » et « quel que soit le nombre des sous-composants, ils doivent respecter un style pipeline». La première contrainte ne résistera pas à l'ajout d'un quatrième sous-composant préservant le style pipeline, la seconde oui. Le langage OCL offrait nativement un panel d'opérateurs dont la puissance d'expression semblait répondre à l'ensemble de ces besoins. Notre langage reprend donc la totalité de la librairie des opérateurs OCL.

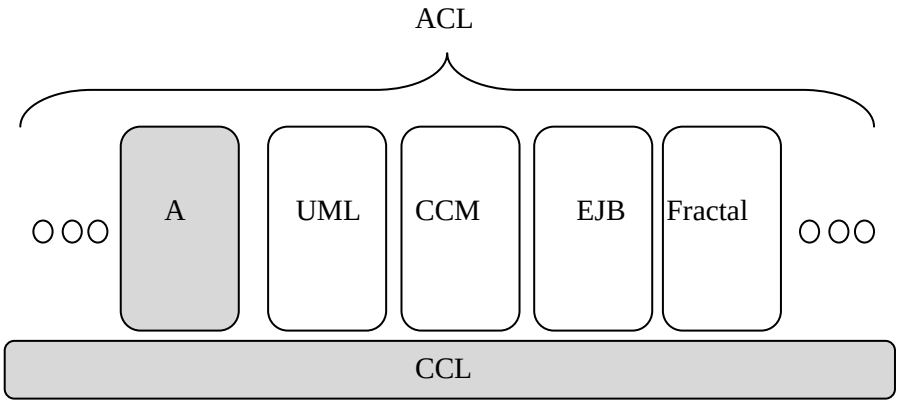


**Figure 5-4La famille de langage ACL**

Au final, la description de motifs dans un modèle $\mathcal{M}$ conforme à un ADL $\mathcal{A}$ consiste à écrire des invariants OCL sur le métamodèle MOF $\mathcal{MM}_{\mathcal{A}}$ représentant la syntaxe abstraite de $\mathcal{A}$. Invariants dont les domaines d'action se limiteront par un double filtrage au modèle $\mathcal{M}$ sur lequel ils sont posés.

Selon cette approche, un langage de la famille ACL dédié à la description de motifs pour l'ADL $\mathcal{A}$ est donc défini entièrement sur le plan syntaxique et sémantique par la donnée (Figure 5-4) :

- d'une version modifiée d'OCL 1.5 incluant le filtrage de contexte qui permet l'expression de contraintes sur des modèles de type MOF. Il offre donc les opérations

de navigation nécessaires sur ce type de modèle, des opérateurs ensemblistes, les connecteurs logiques et les quantificateurs usuels. Il a été baptisé CCL (Core Constraint Language) ;

- d'un modèle MOF 2.0 décrivant les concepts langagiers de l'ADL $\mathcal{A}$, dédiés à l'expression de la partie statique uniquement d'une architecture.

CCL est la partie commune à tous les langages de la famille. A l'inverse, le modèle MOF est propre à chaque langage de la famille.

Le langage CCL est un exemple type de construction d'un langage partant de langages existants : un *Domain Customized Language* (DCL) tel que le définissent (Atkinson, et al., 2007). Grossièrement, la syntaxe du langage est une version modifiée de celle d'OCL opérant tour à tour une restriction (certaines phrases OCL sont interdites comme les pré et post-conditions) et une extension langagière (possibilité de citer dans le contexte d'un invariant un élément non du modèle contraint, mais d'une instance de ce modèle). La sémantique d'OCL est également modifiée en introduisant un double filtrage du domaine d'évaluation d'une contrainte OCL : la contrainte n'est évaluée que sur un seul modèle instance et sur ce modèle, éventuellement une partie seulement des éléments doit vérifier l'invariant exprimé (ceux cités explicitement dans la partie *context*). Ce détour peut être vu comme un moyen d'introspecter un élément de modèle en navigant dans le métamodèle.

### 5.1.5 Description des propriétés non fonctionnelles

Une propriété non fonctionnelle (que j'appelle une NFP) est une des exigences qualité formulées dans le document de spécification d'un composant logiciel. Dans le but d'ouvrir au maximum notre langage à tout type de pratiques industrielles et pour faire face à une absence flagrante de consensus théorique, nous avons choisi d'externaliser, en partie, leur description. Une propriété non fonctionnelle est décrite sous la forme d'un triplet : (Artéfact Porteur, Attribut Qualité, Enoncé de l'exigence).

Toute propriété non fonctionnelle est classée selon un modèle de qualité. Le modèle de qualité que nous utilisons accepte comme attribut qualité, les caractéristiques et les sous-caractéristiques du modèle ISO 9126 (par exemple, la maintenabilité, la portabilité, etc.). La propriété doit être une exigence « atomique », c'est-à-dire n'être associée qu'à un et un seul attribut qualité.

Un attribut qualité est porté par un artéfact architectural externe. Un artéfact externe représente un concept architectural public d'un composant pour l'ADL considéré ; c'est-à-dire un artéfact visible par les autres composants. Pour ne pas avoir à décliner autant de langages qu'il y a d'ADL, nous avons considéré qu'il était possible de se limiter dans la pratique à trois types d'artéfacts externes : composant, interface et opération. Il est toujours possible, moyennant une perte limitée, de raccrocher toute propriété à l'un de ces types d'artéfact.

En l'état, nous ne faisons aucune hypothèse sur le format d'écriture de l'énoncé détaillant la propriété non fonctionnelle. Il peut être formulé sous la forme d'un texte libre (c'est le cas le plus fréquent) ou en usant d'un langage propriétaire dédié à l'expression de certaines propriétés non fonctionnelles. Dans le but de gérer le maximum de cas possible, la seule contrainte que nous imposons, pour des raisons de stockage et d'affichage, est que ce format permette un stockage sous la forme de chaînes de caractères. Bien sûr, cette flexibilité se paye par l'absence de toute structure pour la troisième composante du triplet définissant une NFP ; absence préjudiciable à la mise sur pied de mécanismes de contrôle plus puissants.

Voici deux exemples de NFP qui pourraient être construites depuis la documentation d'un composant :

- (maintenabilité, composant C, « le composant C doit supporter facilement d'autres formats de fichier ») ;

- (performance, opération `o()` de l'interface `I`, « L'opération `o()` doit garantir un temps de réponse inférieur à 10ms »).

Le développeur a donc pour obligation de découper la spécification non fonctionnelle d'un composant en autant de triplets de la sorte.

### 5.1.6 Description des liens unissant propriétés non fonctionnelles / motifs

Il nous restait à proposer un moyen pour documenter les liens unissant les motifs (que j'appellerai dans cette sous-section des CA pour Choix Architectural) aux propriétés non fonctionnelles (NFP). Dans notre approche un tel lien, un couple (CA, NFP), est appelé *tactique non fonctionnelle* (NFT). Les relations qu'entretiennent les concepts de CA, NFT et NFP sont illustrées dans la Figure 5-5.

Les motifs (CA) sont construits selon un modèle hiérarchique. Un CA peut donc être construit sur la base d'autres CA. Le motif CA1 a été construit sur les motifs CA2 et CA3. Un motif est une contrainte écrite en CCL navigant dans le métamodèle de l'ADL visé et posé sur un modèle d'architecture donné.

Une NFT manifeste que le développeur a fait le choix d'implanter le motif CA dans le but de satisfaire tout ou partie de l'exigence NFP. Un même motif peut apparaître dans plusieurs NFT. Ainsi le motif CA3 se retrouve dans deux NFT (NFT1 et NFT2). Un même motif peut donc être lié à plusieurs NFP. Ainsi le motif CA3 participe à l'obtention des exigences NFP1 et NFP2. Inversement, une même NFP peut être associée à plusieurs motifs. Ainsi NFP2 est liée aux motifs CA3 et CA6 (via respectivement NFT2 et NFT3).

Concrètement, nous avons fait le choix de stocker les NFT associées à un composant dans un unique document XML conforme à une DTD reprenant la structure que nous venons d'évoquer. Les motifs architecturaux exprimés en CCL sont également embarqués dans ce document XML.

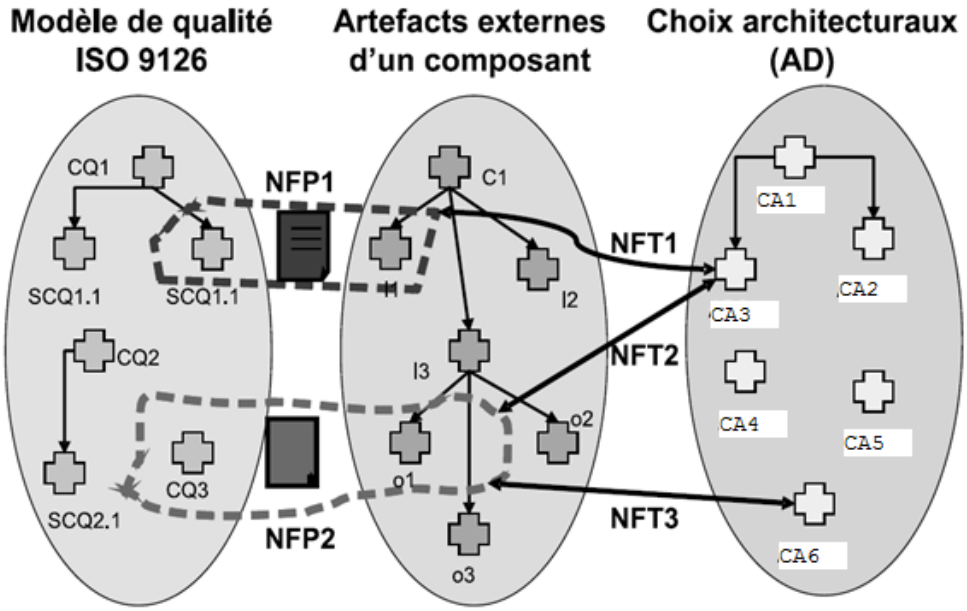


Figure 5-5 Lien NFP/CA

L'intégralité de ce travail sur la documentation de décisions architecturales pour faciliter l'étape de compréhension lors d'un cycle de maintenance fut présenté à la conférence sur les architectures WICSA (Tibermacine, et al., November, 2005b). Il a fait également l'objet d'une description beaucoup plus en profondeur dans la revue JSS (Tibermacine, et al., 2010).

## *5.2 Le contrôle de l'évolution d'une architecture*

### 5.2.1 Difficultés posées par la vérification de la non régression

La vérification de la non-régression et de la progression se fait traditionnellement a posteriori, une fois la modification entérinée en constatant in vivo ses effets, par exemple au travers des tests de non-régression. Des outils du commerce permettent d'automatiser le jeu de ces tests. Des algorithmes additionnels de sélection de tests ont également été proposés pour limiter le nombre des tests à rejouer tout en maintenant le même niveau d'efficacité. Par contre, on constate que la vérification de la progression et de la non-régression sur la composante non fonctionnelle se prête généralement mal à une automatisation. En effet, elle se fait à l'aide de plusieurs logiciels ad hoc ou du commerce, indépendants et très spécialisés (analyseur de code pour les aspects qualité par exemple). A cette difficulté, on doit également ajouter le fait que cette approche de vérification a posteriori génère d'inévitables allers et retours entre la phase de test et celle de développement. Un test non passant va nécessiter une reprise d'un cycle de modification qui sera suivi à nouveau d'une phase de test. Ces allers-retours s'avèrent d'autant plus coûteux que le nombre des tests à rejouer est important.

Il serait donc judicieux de promouvoir, en complément, une vérification au plus tôt, « a priori ». Cette vérification peut être manuelle et prendre la forme de revues, ou de manière plus rigoureuse d'inspection des codes et des documentations. Si l'efficacité de ce type de vérification est prouvée, elle présente l'inconvénient d'être très coûteuse en temps et en homme. Il serait donc intéressant de proposer un mécanisme approchant mais automatisable. Un tel mécanisme pourrait, par exemple, au moment où l'on exécute la modification sur le code ou sur la documentation, alerter des conséquences de celle-ci. Ces contrôles a priori peuvent permettre de réduire significativement le nombre des erreurs détectées (tardivement) lors des tests et en conséquence diminuer le nombre des allers-retours nécessaires à leur résolution. La non-régression a posteriori est de loin la mieux maîtrisée, ayant fait l'objet d'un nombre très important de travaux. L'automatisation a priori, bien qu'utile et complémentaire, ne faisait l'objet en 2004 et à notre connaissance d'aucune recherche.

### 5.2.2 Contrôle de l'évolution

Avec la famille de langage ACL nous disposions d'une documentation non ambiguë capturant des décisions architecturales. L'existence d'une telle connaissance est un atout essentiel pour appréhender une architecture, préalable indispensable à toute maintenance. En soit, cette documentation offrait déjà des perspectives intéressantes pour diminuer le coût des activités de maintenance. Je pressentais cependant que ce type de documentation « papier », utile en théorie, souffrirait à l'usage des mêmes maux que tous les autres types de documentation : la non obligation de sa mise à jour la rendait dans la durée inefficace. Il était donc essentiel de lui donner une place dans le procédé de maintenance qui impose sa mise à jour. Il fallait la rendre « productive » au sens de l'ingénierie dirigée par les modèles, c'est-à-dire obtenir de sa manipulation par les outils une certaine valeur ajoutée. Or, remettre en cause un motif architectural en altérant un modèle d'architecture, c'est se poser la question du devenir des attributs qualité dont ce motif visait l'obtention. Il est dès lors possible, partant de la liste des modifications faites, de déterminer les motifs mis à mal et par là les risques potentiels d'altération de certaines propriétés. La documentation ACL est le lien qui potentiellement rend possible cette analyse d'impact.

Le type de documentation retenu pour décrire les motifs architecturaux se révéla ici primordial. Non seulement cette description prenait la forme d'un ensemble d'invariants (le motif est-il présent ou non) mais en plus elle était formelle car évaluable automatiquement. On disposait à travers la documentation ACL d'un moyen automatique pour alerter des conséquences de modifications. Il suffisait de lancer l'évaluation des contraintes associées à un motif pour détecter son altération. Il était possible de mettre en place une démarche de vérification *a priori* selon la composante non fonctionnelle (Figure 5-6). Notifier

automatiquement, au moment même d'une modification, la perte potentielle de propriétés non fonctionnelles pour aider l'architecte à déterminer s'il doit ou non maintenir sa décision et déterminer les prochaines décisions à prendre.

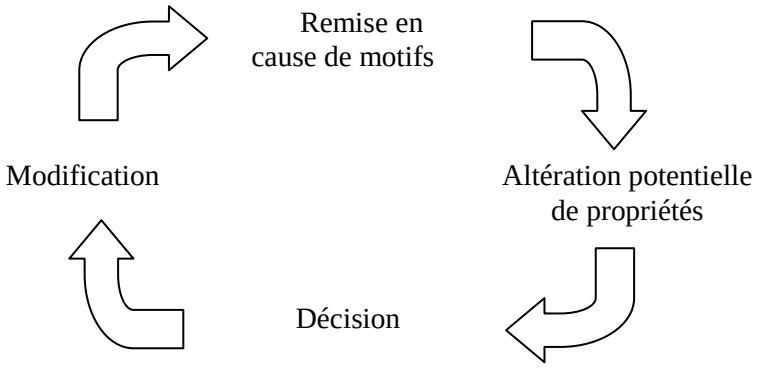


**Figure 5-6 Le cycle de vérification**

Sous réserve de réglementer les actions possibles suite à ces alertes, on peut même se retrouver dans un système contribuant d'une part à la mise à jour de la documentation des décisions architecturales et d'autre part à la vérification (a priori) de la non-régression. Ce type de système augmente grandement les chances d'aboutir à une solution correctement documentée, remplissant les nouvelles exigences, tout en préservant les attributs qualité qui ne devaient pas être altérés. Il vient compléter, sans pour autant la remplacer, la vérification classique par test (a posteriori). En détectant, au plus tôt, l'occurrence (potentielle) de certaines erreurs, on diminue d'autant le nombre des tests non-passants et donc les coûteux allers-retours qui leur seraient associés.

Un procédé de maintenance usant de la documentation ACL a donc été mis au point au cours de l'année 2005. Il s'agissait de promouvoir, à travers lui, une assistance suffisamment flexible pour ne pas limiter la créativité des architectes, tout en bornant les activités dans un intervalle garantissant la mise à jour effective des documents manipulés. Dans ce procédé, la documentation ACL décrivant les décisions architecturales devient un *contrat d'évolution*. Nous avons introduit ce terme, très connoté dans notre communauté, car il documente à nos yeux les droits et devoirs de deux parties : le concepteur de la précédente version du logiciel qui s'engage à garantir certaines propriétés non fonctionnelles, sous réserve du respect, par le développeur de la nouvelle version, des choix architecturaux que le premier avait établis.

A tout modèle d'architecture on attribue un contrat d'évolution. La première version de ce contrat est définie lors de la conception de la version initiale de ce modèle. Le contrat est évalué à la demande par le développeur lors de ses activités de modification du modèle. Comme par principe une évolution peut avoir pour objet de modifier certaines propriétés ou la façon dont elles sont obtenues, un contrat peut être amendé. Mais les avenants sont strictement encadrés par des règles précises évitant de conduire le logiciel dans un état incohérent. Par exemple, il n'est pas concevable qu'une propriété non fonctionnelle, devant être maintenue dans la nouvelle version du logiciel, se retrouve dans celle-ci sans aucun choix architectural associé en fin de cycle. Au mieux cela manifeste une erreur de documentation (des choix architecturaux ont été faits pour garantir l'obtention de la propriété mais ils n'ont pas été documentés), au pire, cela traduit une possible régression (aucun choix architectural n'a été fait pour garantir l'obtention de la propriété dans la nouvelle version du logiciel).

Au final, le mécanisme proposé assiste le développeur tout au long de ses activités. Comme le montre la Figure 5-7, une activité de maintenance est perçue comme une trajectoire dans un espace à trois dimensions (temps, fonctions offertes, et propriétés non fonctionnelles présentes). L'assistance est une forme de contrôle discret de la trajectoire limitant les risques de dérives non

voulues dans le plan non fonctionnel. Cette assistance, en alertant au plus tôt des conséquences possibles des modifications entreprises, contribue au suivi d'une trajectoire potentiellement plus à même de conduire à un maintien des propriétés non fonctionnelles. De plus, si l'on souhaite une évolution selon l'axe non fonctionnel, l'assistance force la mise à jour en fin de cycle de la documentation.

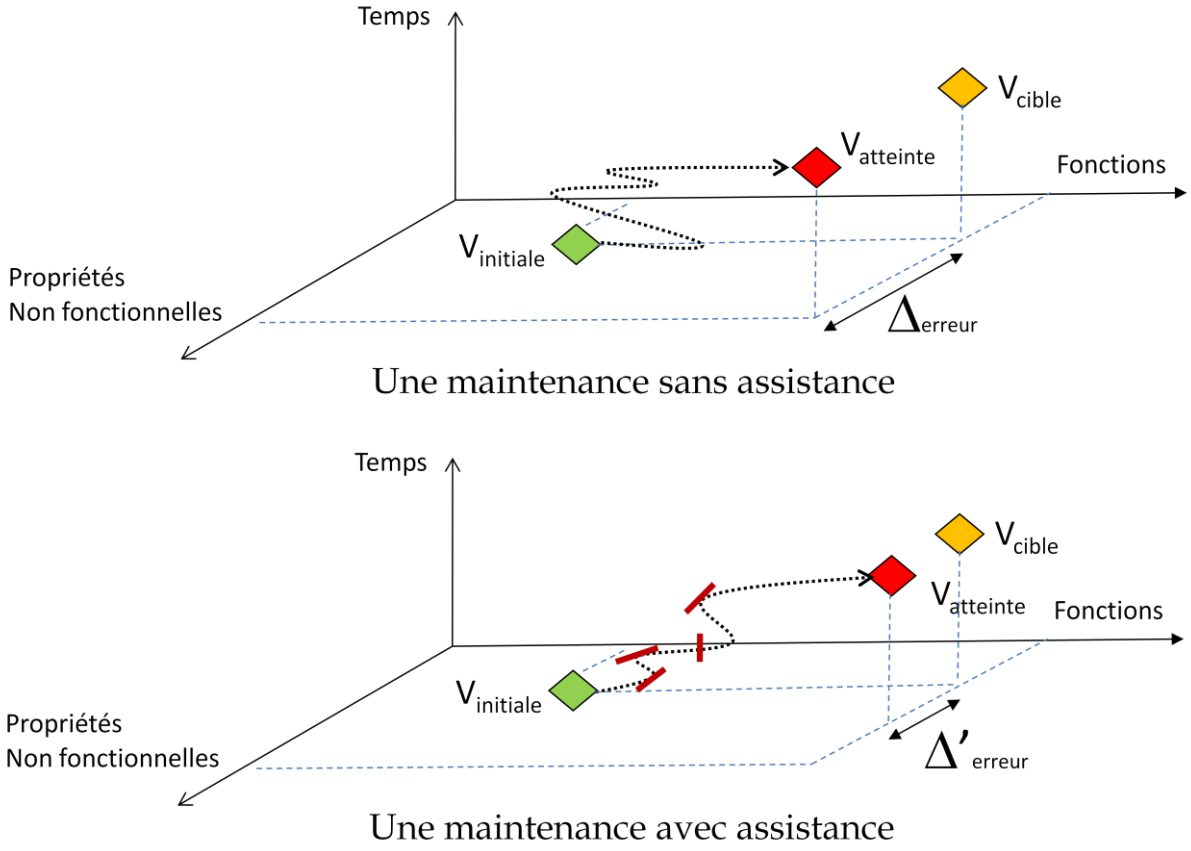


**Figure 5-7 Une version imagée de l'impact d'une assistance lors d'une évolution**

Une première version de ce mécanisme d'assistance a été présentée à LMO (Fleurquin, et al., Mars, 2005) ; une version un peu plus aboutie a suivi dans la conférence ASE (Tibermacine, et al., November, 2005a) sous la forme d'un papier court. Un article dans la revue L'objet (Fleurquin, et al., 2007) en donne tous les détails.

### 5.2.3 Extension de CCL pour la documentation de contraintes d'évolution

Avec ce nouvel usage de la famille ACL, nous avons rapidement constaté que de nouveaux types de décisions architecturales et donc de motifs devaient pouvoir être exprimés. Jusqu'à présent, les décisions architecturales n'impliquaient que la version en cours de modification. Or il est parfois utile de poser des décisions architecturales traduisant des contraintes sur la façon dont peut évoluer dans le temps une architecture. Ce cas se présente lorsque certaines décisions veulent réglementer, selon un mode différentiel, les structures acceptables. Par exemple, on peut vouloir interdire, pour des raisons de fiabilité, l'ajout à un composant d'une interface fournie supplémentaire lors d'une évolution. Il faut donc être capable d'évaluer des « différences » entre deux versions successives de ce composant. Ces décisions, qui sont de véritables contraintes d'évolution, ne sont pas inutiles comme l'indique (van Ommering, et al., 2000).

Nous avons donc ajouté au langage CCL un marqueur qui permet de désigner l'ancienne version d'un artéfact architectural auquel il s'applique : le marqueur `@old`. Considérons l'exemple suivant s'appliquant à notre métamodèle jouet.

```
context C : Composant inv:
```

```
(self.interface->size())< ((self@old.interface->size())+2)
```

Cette contrainte précise que le composant `C` ne peut pas augmenter de plus d'une unité le nombre de ses interfaces d'une version à sa suivante. Notons qu'elle ne dit rien sur les modifications éventuellement subies par les anciennes interfaces de ce composant.

Si nous voulions imposer, qu'au plus une seule de ces anciennes interfaces soit altérée d'une version à l'autre, nous devrions comparer les structures avant et après évolution de toutes les interfaces. Ce type de contrainte serait pénible à écrire avec les opérateurs OCL usuels et le seul marqueur `@old`. Nous avons donc introduit également dans CCL les opérateurs de collection qui suivent :

- `modified():Collection(T)` qui retourne tous les éléments de la collection à laquelle il s'applique qui ont subi une modification entre l'actuelle et la précédente version (seulement pour ceux existant dans les deux versions) ;
- `added():Collection(T)` qui fait de même pour les éléments qui n'existaient pas dans la précédente version ;
- `deleted():Collection(T)` qui fait de même pour les éléments qui ont disparus de la nouvelle version.

La contrainte totale précédente s'écrirait donc :

```
context C : Composant inv:
(self.interface->added()->size())<2) and
(self.interface->deleted()->size())=0) and
(self.interface->modified()->size())<2)
```

### 5.2.4 Une plate-forme logicielle pour l'assistance

Un outil, nommé *AURES*, a été développé par *Chouki Tibermacine* pour supporter l'algorithme d'assistance évoqué à la sous section précédente. L'outil a été développé en JAVA. D'un point de vue fonctionnel cet outil offre les mécanismes qui suivent.

*Edition de contrats* : il permet une édition assistée de contrats d'évolution. Il demande de choisir le langage de description d'architecture cible ; choix qui se résume à charger le format XMI du métamodèle associé au profil sélectionné. Dans sa version actuelle, l'éditeur est à même, en parcourant le métamodèle, d'offrir des mécanismes de complétion automatiques des instructions de navigation CCL saisies ;

*Compilation de contrats* : il vérifie la correction syntaxique d'un contrat dans le cadre d'un langage particulier. A l'instar de ce qui se passe dans les compilateurs OCL, cette vérification inclut deux passes : la première vérifie que le code est un code CCL valide, la seconde que les parcours font bien référence à des parcours valides dans le métamodèle MOF associé au langage. Une troisième passe, propre à notre mécanisme de double filtrage, s'assure que les instances de `classifier` citées en partie contexte sont bien des artéfacts de la description architecturale du composant sur lequel est posé le contrat. La description architecturale est un fichier XML produit par l'ADL cible (par exemple le descripteur Fractal généré par le modèle Fractal). Si la compilation est concluante, l'outil produit une archive composée de la description de l'architecture et de son contrat. Le contrat est stocké sous la forme d'un fichier XML.

*Assistance à l'évolution* : cet outil vérifie à la demande de l'architecte le respect ou non d'un contrat pour une architecture qui lui est soumise. Pour ce faire, il récupère le fichier XML contenant le contrat concerné et les deux descripteurs de l'architecture (avant et après évolution). Si l'évaluation du contrat retourne la valeur vraie, la description architecturale après évolution est associée au contrat. Le tout fait l'objet d'un archivage par un gestionnaire de versions. Dans le cas contraire, les choix architecturaux altérés et les propriétés non-fonctionnelles associées sont indiqués au développeur. Le développeur a alors la possibilité de

corriger la description architecturale, directement à travers l'interface d'AURES et de sauvegarder les changements. Il peut aussi modifier le contrat tout en préservant la règle imposée par l'algorithme d'assistance stipulant que pour chaque NFP dans un contrat, il doit exister au moins un choix architectural associé. Une fois les modifications faites, il peut tenter une nouvelle évaluation du contrat.

### 5.2.5 Effet collatéral : la définition d'un ADL générique

Le langage ACL n'est pas un langage mais une famille de langages. La difficulté était donc de proposer autant d'évaluateurs différents dans l'outil *AURES* qu'il y a de langages supportés (actuellement 5). Nous avons donc trouvé plus judicieux, de profiter des ressemblances que présentaient les métamodèles de chacun des langages que nous supportions, pour définir : un métamodèle intermédiaire auquel on associe un unique évaluateur et autant de traducteurs des autres langages vers ce langage intermédiaire. Ce métamodèle intermédiaire forme avec le langage CCL un langage baptisé *ArchMM*.

Avec ce mode de fonctionnement, pour évaluer un contrat $\mathcal{C}$ sur un modèle d'architecture $\mathcal{A}$ exprimé dans un langage $\mathcal{L}$ de la famille ACL :

- on traduit $\mathcal{A}$ via XSLT en un modèle $\mathcal{A}'$ conforme au (schéma XML du) langage ArchMM ;
- le code CCL du contrat $\mathcal{C}$ associé à $\mathcal{A}$ est également traduit dans le langage ArchMM et fournit un nouveau contrat $\mathcal{C}'$.
- Le contrat $\mathcal{C}'$ est évalué sur le modèle $\mathcal{A}'$.

Toute la difficulté a été de proposer un métamodèle intermédiaire suffisamment riche et abstrait pour fournir un espace de traduction sans perte depuis chacun de nos 5 langages. Mais ce métamodèle a été conçu avec une ambition plus grande, sur la base d'une étude comparative plus large comportant deux types de langages. Les langages de description d'architecture et les modèles de composants. L'étude sur les ADL s'est principalement focalisée sur Acme (Garlan, et al., 2000), Koala (van Ommering, et al., 2000) et xADL (Dashofy, et al., 2005). Nous nous sommes basés également sur un autre état de l'art de (Medvidovic, et al., 2000). Pour la deuxième catégorie de langages nous avons analysé les technologies de composants : *EJB* de Sun Microsystems, *CCM* de l'OMG, *Fractal* du consortium ObjectWeb et la partie concernée par les composants du métamodèle UML. Le métamodèle élaboré unifie et abstrait les concepts architecturaux représentés par ces langages. Il constitue une synthèse intéressante de la partie structurelle des principaux langages à base de composants. On en trouvera le détail dans la thèse de *Chouki Tibermacine* qui a été soutenue à *l'Université de Bretagne Sud* en Octobre 2006.

### 5.2.6 Une incartade dans le monde du Web

La solution proposée est, dans son principe et dès 2005, confrontée à l'avis de nos pairs. Cependant, conformément à la structure de preuve que j'ai décrite précédemment, nous devions encore trouver le moyen de vérifier in vivo sa viabilité. Courant 2005, nous trouvons une société du secteur logiciel prête à collaborer sur le thème de l'évolution : la société *Alkante* une PME Rennaise leader français dans le domaine des applications Web à base de Systèmes d'Information Géographique.

Cette entreprise faisait évoluer depuis plusieurs années une infrastructure Web, appelée Alkanet, dédiée au développement de sites nécessitant l'usage de systèmes d'information géographiques. Cette infrastructure, développée en PHP, Perl et JAVA était organisée sous forme de modules interconnectés. Or, les développeurs de cette entreprise constataient une progression importante des coûts liés à l'évolution de certains modules. La société était donc désireuse de revoir en profondeur ses procédés de développement pour remédier à ces difficultés.

L'idée est de monter une collaboration sur un mode gagnant-gagnant avec ce partenaire industriel. Nous leur offrons une expertise méthodologique pour repenser, dans leur domaine applicatif de prédilection, leur procédé de développement. En contrepartie, ils nous offrent un terrain d'expérimentation pour tester les contrats d'évolution. En Juin 2005, une collaboration est lancée sur la base d'une thèse en convention CIFRE.

Dans le cadre de cette convention, le doctorant recruté (*Réda Kadri*) s'est vu confier les missions qui suivent.

- Mettre sur pied au sein de la société un nouveau procédé de développement s'inspirant des approches par composants permettant de maintenir le patrimoine code et le savoir faire technologique de la société. Ce procédé de développement a été décrit dans (Kadri, et al., 2006).
- Développer un ADL dédié à la représentation de l'architecture des applications Web et son outil support. L'ADL qui a été élaboré (`AlkoWeb`) est défini sous la forme d'un profil UML. Il propose une vision à base de composants hiérarchiques des applications web. On trouvera une présentation de cet ADL dans (Kadri, et al., 2007). L'outil support (`AlkoWeb-Builder`) est un plugin Eclipse. Il est présenté dans (Kadri, et al., 2008).
- Evaluer sur un projet pilote de l'entreprise, l'intérêt des contrats d'évolution.
- Sur la base du retour d'expérience du projet pilote, intégrer dans le procédé de développement de la société et dans l'outil `AlkoWeb-Builder` les contrats d'évolution.

Mon intervention sur cette thèse a essentiellement concerné les trois derniers points. Dans ce mémoire, je n'évoquerai que le troisième. On se reportera au mémoire de thèse de *Réda Kadri* pour avoir plus de détail sur les trois autres.

### 5.2.7 Un cas d'étude pour évaluer les contrats d'évolution

Cette validation a été faite dans la société *Alkante*. Il s'agissait de montrer qu'un développement utilisant les contrats d'évolution conduit effectivement à une réduction des coûts et des délais de maintenance. Le public visé comprend les architectes et les développeurs de niveau ingénieur dans un contexte de production soumis à des délais relativement tendus et dans des entreprises connaissant un taux de roulement sur les projets représentatif des sociétés de services. La société *Alkante* répondait largement à ces critères.

Il semblait difficile d'exiger de la société *Alkante* deux développements réels différents pour un même cahier des charges. De la même façon, la démarche consistant à étudier des projets répondant à des cahiers des charges différents n'offrait pas une garantie suffisante sur le plan de la validité interne. Nous avons donc choisi une solution intermédiaire : rejouer (virtuellement) avec les contrats d'évolution un projet déjà terminé qui lui n'avait pas fait l'usage des contrats d'évolution. Toute l'étude repose sur la comparaison des résultats obtenus pour ce même cahier des charges en utilisant deux méthodes de développement.

L'étude s'est déroulée en deux temps. Dans un premier temps nous avons collecté le maximum d'information sur le cycle de vie d'une application de la société appelée `GéoConv` (`GeoConv` est un convertisseur de format cartographique). Ce composant a été choisi car il avait connu de nombreuses modifications sur une période de 14 mois : 2 nouvelles versions majeures notées $V1_{observée}$ (72 heures*homme), $V2_{observée}$ (342 heures*homme) et deux mineures $V1.1_{observée}$ (132 heures*homme) et $V2.1_{observée}$ (186 heures*homme). Il comportait également trois décisions architecturales importantes adressant la portabilité et la maintenabilité. Ces décisions étaient invariantes et non documentées. Elles avaient pour motifs : un style en couche, un composant façade et une séparation du flux des données en entrée.

Nous avons récupéré un historique détaillé couvrant la période concernée : logs CVS, e-mails échangés par les développeurs entre eux et ceux échangés avec les clients. Nous

disposions également de données concernant les temps de développement déclarés pour `GéoConv` émanant du logiciel de gestion de projet de l'entreprise dans lequel sont enregistrées les heures consacrées par les développeurs sur chaque tâche. De plus, en examinant les pratiques de développement de la société, nous avons pris comme base de travail qu'un microcycle de maintenance comporte 3 étapes : compréhension, conception d'une solution, codage et test de la solution. Au sein d'un microcycle la charge se répartit habituellement dans la société de la manière suivante : 25% pour la compréhension, 25% pour la conception d'une solution et 50% pour son codage et son test.

Nous avons constaté que les évolutions mineures $V1.1_{observée}$ et $V2.1_{observée}$ étaient en fait des correctifs appliqués aux versions majeures pour rétablir des décisions architecturales perdues et dont le non respect compromettait l'utilisation ou l'évolution du composant. Un examen attentif des données en notre possession a montré que le temps de développement de la version $V1_{observée}$ (72 heures*homme) a été complètement perdu car l'architecture et le code proposés dans la version correcte $V1.1_{observée}$ ont dû être bâtis sur la base de $V0_{observée}$. A l'inverse une partie importante du code réalisé pour $V2_{observée}$ a pu être réutilisée dans $V2.1_{observée}$ (environ les ¾). Nous avons estimé en examinant les modifications faites dans $V2.1_{observée}$ que la production d'une version correcte de $V2_{observée}$ aurait coûté environ 390 heures*homme, soit une perte d'environ 140h par rapport au temps mis pour mener à bien $V2_{observée}$ et $V2.1_{observée}$. Au total, sur ce projet nous avons constaté une charge supplémentaire de 40% par rapport à un développement idéal ; charge liée à la détection et à la correction *a posteriori* de pertes de propriétés non fonctionnelles.

Nous devions déterminer ensuite les charges de maintenance qui auraient été observées si l'application avait été développée par les mêmes développeurs en utilisant les contrats d'évolution : c'est-à-dire la charge qui aurait été consacrée au développement d'une version $V1_{contrat}$ correcte et identique à $V1.1_{observée}$ et d'une version $V2_{contrat}$ identique à $V2.1_{observée}$.

Pour déterminer ces deux charges nous nous sommes appuyés sur l'estimation des charges des deux évolutions idéales : $V1_{idéale}$ est en fait V1.1 (132 heures*homme) et $V2_{idéale}$ estimée précédemment à 390 heures*homme.

Nous avons supposé que les développeurs ne consultaient pas la documentation associée aux contrats (donc que la charge liée à chaque étape de compréhension ne changeait pas) mais qu'ils soumettaient en fin d'étape de conception leur solution à l'outil de contrôle de l'évolution (qui lui vérifie le respect des décisions). Cette hypothèse d'une part simplifiait notre expérimentation (on peut supposer identique les charges de compréhension avec et sans contrat) et d'autre part permettait de ne mesurer que l'effet du seul contrôle indépendamment de la documentation.

Ainsi, nous obtenons :

$V1_{contrat}$ = charge de compréhension de $V1.1_{observée}$
+ charge de conception de $V1_{observée}$
+ charge d'évaluation des contrats
+ charge de reprise et de production de la conception de $V1.1_{observée}$
+ charge d'évaluation des contrats
+ charge de codage et de test de $V1.1_{observée}$

$V2_{contrat}$ = charge de compréhension de $V2_{idéale}$
+ charge de conception de $V2_{observée}$
+ charge d'évaluation des contrats
+ charge de reprise et de production de la conception de $V2.1_{observée}$
+ charge d'évaluation des contrats
+ charge de codage et de test de $V2_{idéale}$

Les charges liées à l'évaluation des contrats sont négligeables par rapport aux autres postes. En effet, l'outil réalise une évaluation en quelques secondes et on peut considérer que la mise en place d'une évaluation ne prend que quelques minutes. Les charges consacrées à la reprise et à la production d'une nouvelle conception sont approchées en utilisant les charges de conception des versions observées correspondantes. Les charges de compréhension, et de codage et test de $V2_{idéale}$ se calculent en utilisant la répartition indiquée plus haut. Au final, les charges de $V1_{contrat}$ et $V2_{contrat}$ sont respectivement de 150 heures*homme et 425 heures*homme. Si l'on ajoute les temps de rédaction des contrats (5h) qui ont été rédigés par un développeur d'Alkante nous obtenons 580 heures*hommes pour un développement usant des contrats à comparer aux 732 heures*homme observées ; soit un gain d'environ 20%.

Dans cette étude nous avons volontairement écarté le temps de formation du personnel car il peut être amorti sur plusieurs projets. La validité interne est perfectible car quelques approximations ont été faites. Plus encore, cette validation est loin de constituer une expérimentation totalement satisfaisante sur le plan de la validité externe. Elle ne concerne qu'un unique applicatif dont on connaissait par avance le vécu tumultueux (cela majore les résultats), dans un domaine applicatif très sujet à de la maintenance en urgence et pour un type d'entreprise particulier (PME qui n'affiche pas les certificats qualité des grandes SSII). Je la prends donc pour ce qu'elle est : une tendance qu'il conviendrait de confirmer par d'autres cas d'étude ou par une expérimentation contrôlée (pour étudier en particulier l'impact du choix d'OCL).

### 5.2.8 Bilan du thème

Ce thème de recherche s'est étendu sur une période d'un peu plus de 5 ans. Deux thèses dont j'ai assuré le co-encadrement ont été soutenues sur cette période : *Tibermacine* en Octobre 2006 et *Kadri* en Janvier 2009. Cinq années durant lesquelles j'ai eu la chance de mettre en pratique la démarche de recherche que je défends (cf. sous-section 4.2.6). Ces travaux ont conforté ma conviction, acquise lors de ma propre thèse, qu'un travail dans notre discipline ne peut être considéré comme véritablement abouti qu'une fois confronté aux conditions du monde réel. Le passage en entreprise est un puissant révélateur : les zones de flou, les failles, les hypothèses hasardeuses sont très rapidement levées.

Sur ce thème, je retiens de l'expérimentation entreprise essentiellement deux choses :

- l'intérêt de la documentation des décisions architecturales et du contrôle de l'évolution que permet cette documentation pour diminuer les délais et coûts de maintenance ;
- l'inadéquation pour un usage industriel d'une proposition langagière basée exclusivement sur OCL.

Sur le premier point, il est heureux de constater dans la réalité ce que le bon sens nous pousse à espérer : le coût de correction d'une erreur est d'autant plus faible que celle-ci est découverte tôt. L'efficacité du contrôle proposé tient à la mise en place, dès la phase de conception, d'une détection d'erreurs (potentielles). Celles-ci ne seraient détectées autrement qu'en toute fin du microcycle de maintenance (lors des tests), voire comme il s'agit ici de propriétés non fonctionnelles telles que la maintenabilité ou la portabilité dont l'utilité ne se révèle que dans la durée, plusieurs microcycles plus tard.

Sur le second point, le langage OCL est bien adapté à l'expression de contraintes documentaires simples, c'est-à-dire navigant sur un nombre réduit de `classifiers` et dont l'évaluation impose la prise en compte d'un petit nombre d'instances. Il est par contre délicat d'écrire et de comprendre des contraintes OCL vérifiant la présence de motifs architecturaux globaux pourtant visuellement simples (pipeline, couche, etc.). De tels motifs nécessitent, pour être exprimés, des parcours dans plusieurs `classifiers` et un usage forcené d'opérateurs manipulant des collections. L'usage d'OCL pour exprimer de tels motifs était donc indéfendable sur le plan industriel quel que soit le niveau de compétence de l'architecte.

L'idée que nous proposons est bonne sur le fond et perfectible sur la forme (langage et outillage associé). Nous avons potentiellement deux pistes à explorer pour remédier à ce problème : soit une librairie de contraintes OCL pré écrites pour les motifs architecturaux usuels (solution légère) soit un second langage graphique, de préférence complémentaire à OCL, s'inspirant des langages utilisés pour décrire des patrons ou des points d'action dans le monde des aspects qui permettrait une représentation graphique intuitive des motifs (solution lourde).

## 5.3 La sélection de composants

### 5.3.1 Le problème de la modification des composants logiciels

Courant 2005, nous constatons que si l'assistance que nous proposons est bien adaptée à la détection de pertes de propriétés non fonctionnelles ayant pour origine la remise en cause de motifs architecturaux, elle est par contre de peu de secours lorsqu'une modification concerne non pas la configuration elle-même (l'ajout, la suppression de connecteurs et composants) mais un composant ou un connecteur particulier élément de cette configuration. En effet, à configuration constante, un composant/connecteur peut subir des modifications et donc évoluer en version, voire être remplacé par un autre composant/connecteur.

Les ADL dotés d'un système de types vont vérifier quelques propriétés de compatibilité syntaxiques (covariance des interfaces fournies et contra-variance des interfaces requises), afin d'assurer que la modification n'invalide pas les « compositions » associées au composant/connecteur concerné. Mais, en l'état notre approche ne permettait pas d'inférer quoi que ce soit d'utile pouvant compléter ce qu'offrent les ADL typés sur les conséquences de ces modifications « locales ». Nous ne pouvions les relier aux propriétés non fonctionnelles du système tout entier. Il nous fallait pour cela adjoindre à nos motifs architecturaux des informations portant également sur les propriétés non fonctionnelles des composants/connecteurs impliqués dans un motif.

Nous décidons d'approfondir cet aspect et obtenons un financement de la région Bretagne pour un sujet de thèse sur le thème fin 2004. *Bart George* sera recruté en Octobre 2004. Il est important de noter qu'il ne s'agissait pas de réfléchir au moyen de prédire des propriétés non fonctionnelles d'un système partant des propriétés non fonctionnelles de ses parties. Ce sujet, très difficile est toujours ouvert en 2010 et fait l'objet de nombreux travaux. Il s'agissait au contraire de considérer que cette information est connue de l'architecte et qu'il la documente : il précise que telle propriété non fonctionnelle de tel composant contribue à telle propriété non fonctionnelle du système. On veut, sur la base de cette documentation, pouvoir déterminer si la modification subie par un composant est de nature à remettre en cause les propriétés non fonctionnelles du système.

Je propose à *Bart George* d'entamer le sujet sous l'angle du typage non fonctionnel de composant. En effet, une information à forte valeur ajoutée lors de la modification d'un composant logiciel est de savoir si sa nouvelle version (ou le composant qui se substitue à lui) est un sous-type. Si c'est le cas, la modification est a priori sans conséquence sur les propriétés non fonctionnelles du système. Dans le cas contraire, une alerte doit être levée. Les premiers mois de la thèse se concentrent donc sur le problème de la définition d'un système de typage non fonctionnel pour les composants logiciels.

Il nous apparaît rapidement que le sujet est aussi vaste que complexe (George, et al., 2007). Nous rencontrons plusieurs obstacles. Un tour d'horizon de la littérature montre que les ADL restreignent fortement aussi bien qualitativement que quantitativement les informations de typage. Ils ne permettent qu'une définition incomplète d'un type, omettant tout particulièrement les aspects non fonctionnels. De plus, certaines propriétés non fonctionnelles sont souvent données sous la forme d'une fonction ayant en paramètre l'environnement du composant. Or, avec les ADL actuels cet environnement n'est pas totalement explicite. Il est difficile de connaître au-delà du niveau syntaxique les services d'un composant effectivement consommés et plus encore les services dont il pourra bénéficier. Dans le monde des composants, le typage

classique présente moins d'intérêt. Il ne s'agit pas, d'étudier des substitutions composant/composant mais besoin (pour l'architecture considérée) / composant. Or, l'identification automatique du besoin auquel répond effectivement un composant dans une architecture particulière est hors de portée avec les ADL actuels.

Face à ces difficultés, nous décidons pour ne pas perdre nos efforts, en particulier bibliographiques, de réorienter la thèse de *Bart George* sur la sélection (pragmatique) de composants. Ce thème présente l'intérêt de reposer sur les mêmes fondamentaux que ceux étudiés jusqu'alors : représentation des propriétés non fonctionnelles, comparaison de composants à un besoin, etc.

### 5.3.2 Importance de l'étape de sélection

La recherche d'une solution existante à un besoin précis, doit pouvoir être entreprise aussi souvent que nécessaire et de manière efficace. Cette exigence d'ordre technique est une condition nécessaire sans laquelle il n'est pas possible d'assoir une quelconque politique de réutilisation rentable.

En premier lieu, l'efficacité d'un processus de sélection doit se manifester sur le plan des coûts. En effet, une sélection trop consommatrice en temps et en ressources, peut finir par menacer les gains conférés par la réutilisation, rendant du même coup caduque voire néfaste toute politique de réutilisation.

Mais si l'aspect coût est une condition nécessaire, il ne peut prétendre à être un caractère suffisant. Un processus de sélection peu coûteux mais fournissant des réponses loin d'être optimales dans l'espace de recherche, génère des coûts supplémentaires liés à l'adaptation des composants trouvés ou au développement de nouveaux composants. La seconde qualité d'un processus de recherche est donc son aptitude à proposer le meilleur candidat possible dans l'espace de recherche au vue des exigences formulées.

Toute la difficulté est donc de proposer un processus de recherche offrant un bon compromis coût/justesse. C'est la solution de ce problème qui fait l'objet de la section qui suit.

### 5.3.3 Le processus de sélection et les difficultés qu'il pose

La Figure 5-8 récapitule l'ensemble des activités que l'on rencontre lors de la recherche d'un composant, en tenant compte de l'état actuel des marchés et des bibliothèques de composants.

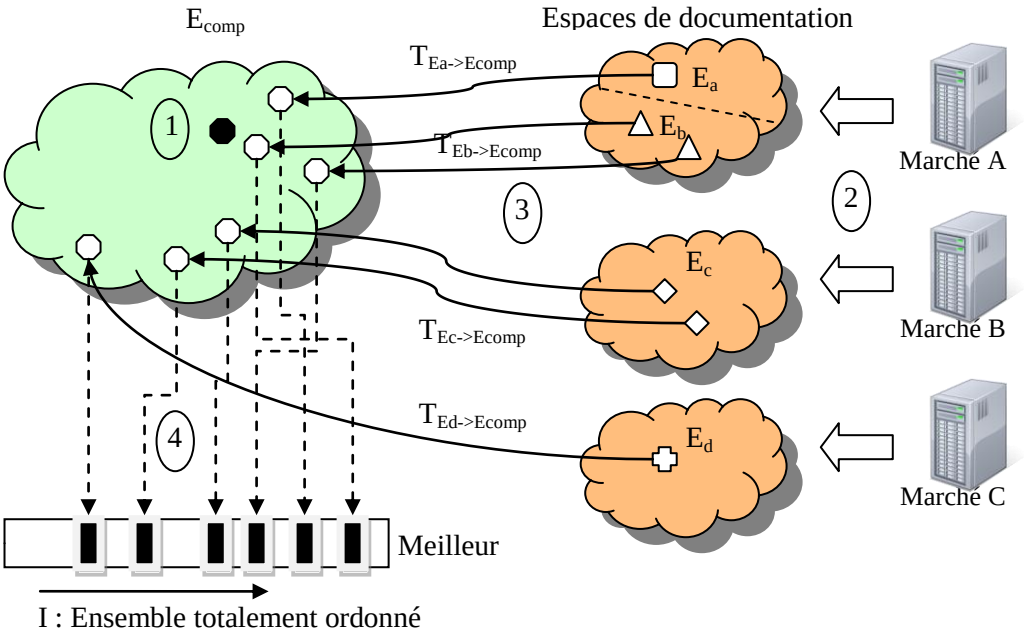


**Figure 5-8 Le processus de sélection**

La première étape d'un processus de sélection consiste à décrire les propriétés du composant recherché (étape 1 sur Figure 5-8). Or, à ce jour, il n'existe pas de format de documentation de composants faisant consensus. Chaque marché utilise son propre format aussi bien pour documenter les aspects fonctionnels (opérations fournies et requises) que non-fonctionnels (niveau de fiabilité, sécurité, technologie support, prix, origine...). De plus, un marché peut fournir des informations qu'on ne retrouvera pas dans un autre marché. Pire, au sein d'un même marché peuvent coexister plusieurs formats différents. Cette hétérogénéité est liée suivant les cas : à la présence au sein d'un même marché de composants développés dans des technologies différentes qui supposent donc la mise à disposition d'informations propres à l'une ou l'autre de ces technologies ou au caractère laxiste du marché qui autorise l'enregistrement de composants sans pour autant contraindre la complétude ou le format des informations fournies.

*Le premier problème qui se pose est donc celui du choix du langage à utiliser pour réaliser cette description.*

En supposant documenté le composant recherché dans un langage $L_{comp}$ ad hoc (dont l'ensemble des mots constitue ce que nous appelons l'espace de comparaison $E_{comp}$), il faut ensuite parcourir plusieurs marchés et bibliothèques pouvant comporter des milliers de composants (étape 2 sur la Figure 5-8). Malheureusement, rien qu'en se limitant aux marchés de composants disponibles sur internet, on constate que l'accès à leur documentation se fait selon des modes et des critères très différents. Au niveau des modes, l'utilisateur peut disposer de mécanismes de recherche usant de requêtes simples (par exemple une liste de mots clés), une navigation arborescente (respectant un plan de classement) ou une navigation hypertextuelle (navigation dans un réseau de nœuds et de liens créés par des associations entre des mots et entre des documents). Quoi qu'il en soit, les modes d'accès proposés restent primitifs. Ils permettent au mieux de circonscrire la recherche à quelques éléments.

L'hétérogénéité prévaut également au niveau des critères de filtrage offerts. Les composants sont regroupés dans certains marchés par thème de manière arborescente et totalement propriétaire. Ainsi, le marché *ComponentSource*[2] permet de filtrer ses composants selon leur

[2] Pour plus d'informations: http://www.componentsource.com/index.html

domaine d'application : image, programmation, communication internet, etc. D'autres marchés, au contraire, livrent les composants tels quels, sans classement particulier. Si on peut supposer que certains marchés offriront dans l'avenir des mécanismes de recherche plus ou moins complets pour faciliter la découverte de leurs composants, il y a peu d'espoir qu'à terme un protocole standard unique de consultation, respecté de tous, émerge. Il est donc nécessaire, si l'on souhaite interroger plusieurs marchés, de prévoir pour chacun d'entre eux une procédure d'accès spécifique permettant de tirer profit des mécanismes et critères de filtrage mis à disposition.

*Cela impose, d'une part d'extraire depuis le modèle du composé recherché (conforme à $L_{comp}$) des informations utiles et, d'autre part, la constitution sur la base de ces informations d'une « requête » permettant d'user des mécanismes de consultation propres à chaque marché.*

Sous réserve d'avoir su jongler avec des mécanismes de consultation différents, il a sans doute été possible, pour chaque marché, de restreindre l'espace des composants à examiner. Un travail important reste encore à faire : comparer et classer tous les candidats prometteurs isolés dans chacun des marchés. Tâches d'autant plus ardues qu'elles imposent, entre autres, l'analyse de documentations souvent écrites dans des formats disparates. Les comparaisons devant se faire par rapport aux besoins exprimés, elles doivent prendre place tout naturellement dans l'espace de comparaison $E_{comp}$. Il est donc nécessaire de « projeter » dans cet espace l'intégralité des descriptions des composants identifiés dans les divers marchés lors de l'étape précédente (étape 3 de Figure 5-8).

*On doit disposer, pour chaque langage $L_i$ de description de composant utilisé dans un marché (auquel correspond un espace des modèles conformes $E_i$), d'un mécanisme de traduction vers une description conforme au modèle de comparaison.*

Formellement, cette traduction est une fonction que l'on souhaite totale, que nous notons $T_{Ei->Ecomp}$, de signature $E_i \rightarrow E_{comp}$. En théorie, deux descriptions différentes d'un espace $E_i$ peuvent être projetées sur la même description de $E_{comp}$ (la traduction est alors non injective). A l'inverse certaines descriptions de l'espace de comparaison ne sont pas atteignables depuis l'espace $E_i$ (la traduction est non surjective). Ces phénomènes sont liés respectivement au manque de puissance du langage de comparaison $L_{comp}$ qui peut omettre certaines informations existantes dans le langage de $L_i$ ou à l'inverse à une puissance supérieure du langage de comparaison par au rapport au langage $L_i$.

D'un point de vue pratique, il est nécessaire de proposer autant de fonctions de traduction qu'il existe de formats de description dans les bibliothèques et marchés existants. C'est un travail important qui ne saurait prétendre à l'exhaustivité, d'autant plus délicat que les formats en question ne sont que rarement explicités et encore moins formalisés. Dans le cas de `ComponentSource`, mis à part les pages Web officielles qui contiennent certaines informations, les documentations proposées par les fournisseurs sont très inégales d'un composant à l'autre, certains d'entre eux n'en possédant même pas. Comme il est illusoire d'automatiser la construction de ces formats en partant de la description des composants, on doit se résoudre à écrire ces traductions à la main au cas par cas, par paquet de composants à peu près respectueux d'un même format documentaire.

*Une fois l'ensemble des composants décrits dans un même langage $L_{comp}$, la dernière étape consiste à user d'un opérateur de comparaison pour classer les composants candidats.*

Celui qui possède le "score de comparaison" le plus élevé est considéré comme le meilleur candidat. Pratiquement, les opérateurs de comparaison de la littérature utilisent des techniques de prise de décision multicritères (Multi-Criteria Decision Methods, MCDM). Il en existe beaucoup. Les plus utilisées dans le monde des composants sont WSM (Weighted Scoring Method) et AHP (Analytic Hierarchy Process). La technique WSM est la plus simple. Elle consiste à utiliser la formule suivante pour chaque composant candidat : score=$\sum_{i=1}^{n} w_i * s_i$. Le poids $w_i$ représente l'importance du i-ème critère par rapport aux autres. Le score local $s_i$ évalue le degré de satisfaction du critère numéro i par le candidat étudié. Un critère est une propriété

exprimée par le candidat recherché. Le score total, agrégation des scores locaux, représente la valeur globale d'évaluation d'un composant. AHP est une technique qui met en œuvre un procédé d'agrégation des scores locaux plus sophistiqué. Ce procédé s'appuie sur un arbre hiérarchique de critères et de sous-critères d'évaluation, dont les feuilles sont les candidats à évaluer. A l'intérieur de chaque critère-nœud, on détermine l'importance de chaque sous-critère par rapport aux autres.

### 5.3.4 Nécessité d'automatiser le processus de sélection

On le voit, la mise en place d'un processus de sélection efficace suppose que l'on ait su relever plusieurs défis :

1. choisir un langage de documentation approprié pour décrire le composant recherché ;
2. réussir l'accès à des marchés supportant des protocoles variés ;
3. traduire les documentations disparates des composant candidats dans le langage de comparaison ;
4. calculer un classement optimal sur la base des critères de choix retenus.

Il existe plus d'une dizaine de méthodes de sélection de composants. On en trouvera une synthèse dans (Land, et al., 2008). Si elles diffèrent dans les étapes qu'elles préconisent, les critères de sélection qu'elles mettent en avant et les mécanismes d'agrégation, elles ont toutes comme point commun de ne pas automatiser le calcul des scores locaux pour chaque candidat. Or, on se retrouve fréquemment avec un nombre de candidats dépassant la dizaine. A titre d'exemple, la seule section communication internet de `ComponentSource` comporte plus de 131 composants. On doit ainsi parcourir des dizaines de documentations pour extraire manuellement les informations permettant de calculer ces scores locaux dont vont ensuite pouvoir user les outils de calcul MCDM. Une activité manuelle vite rédhibitoire avec un nombre élevé de candidats et de critères. La sélection reste fastidieuse et risquée par le fait qu'elle s'appuie sur des mécanismes exigeant une intervention humaine fréquente.

On se retrouve alors devant un paradoxe. L'une des clés de la réussite du paradigme composant tient à l'existence d'une base de composants la plus large possible. On est donc en droit de se féliciter et même de réclamer la constitution de nombreux et vastes référentiels de composants. Or, avec les moyens actuels, la montée en charge de ces référentiels ne fait qu'accroître les coûts des processus de sélection, menaçant du même coup l'intérêt d'une politique de réutilisation. La seule solution viable est donc de disposer d'un mécanisme de recherche et de sélection de composant qui soit autant que possible automatisé.

Les travaux que nous avons entrepris ont donc cherché à lever certains des verrous que nous venons d'évoquer, plus particulièrement les points 1 (langage de comparaison) et 4 (classement optimal automatique) précédents. Pour le point 4 nous avions fait l'hypothèse que les points 2 et 3 avaient trouvé une solution. C'est-à-dire que la procédure de classement pouvait s'opérer sur un nombre quelconque de composants tous disponibles dans le langage de comparaison retenu. Cette hypothèse ne constitue en rien la négation des difficultés (considérables) afférentes aux points 2 et 3. Elle doit être interprétée plutôt comme la volonté d'obtenir une première ébauche de réponse couvrant tout le processus de sélection ; ébauche applicable telle quelle dans certaines situations d'intérêt pratique. Le reste de cette section détaillera les travaux concernant les points 1 et 4. Cependant conscient de la criticité des points 2 et 3 pour la mise sur pied d'une solution générale, des pistes suivies pour apporter un début de réponse aux points 2 et 3 seront évoquées dans la sous-section bilan.

### 5.3.5 Caractéristiques souhaitables du langage de comparaison

L'objectif étant de pouvoir comparer le besoin d'une application avec des composants documentés selon des formats potentiellement très différents, il était impératif d'user pour sa description d'un format suffisamment riche et flexible aussi bien sur le plan fonctionnel que non fonctionnel. Les formats des composants commerciaux (ActiveX, JavaBeans, .NET...)

dépendent fortement de technologies qui leur sont propres. Il n'était donc pas envisageable d'user de l'un d'entre eux. Il n'y avait donc pas d'autre solution que de *proposer un modèle plus abstrait pour atteindre le niveau de généricité requis*. Il fallait toutefois veiller, pour faciliter l'écriture des traductions, à ce qu'il garde une certaine proximité aussi bien syntaxique que sémantique avec les formats concrets existants.

Il y avait là, un vrai défi, celui de la définition d'un format de description du besoin qui soit suffisamment abstrait pour embarquer un maximum d'informations présentes dans tout format de composant existant, mais pas trop pour ne pas réclamer l'écriture de mécanismes de traduction trop complexes. D'un point de vue plus formel, il s'agissait de veiller à ce que l'espace de comparaison associé au langage de description puisse autant que possible être (quasi) isomorphe aux espaces concrets.

Lorsque l'on analyse l'activité de sélection, on constate qu'elle relève souvent d'une démarche d'exploration du patrimoine par tâtonnement. Une sélection peut alterner des phases de recherche à grain plus ou moins fin et cela en fonction du nombre des candidats trouvés, du niveau de détail dont on dispose ou de contraintes que l'on se fixe. Des tentatives de sélection usant d'un nombre important de propriétés très précises (comparaison de signatures de méthodes, comparaison d'interfaces, comparaison de valeurs de métrique, etc.) peuvent par exemple conduire à des temps de calcul rédhibitoires, à un résultat vide ou trop restreint, imposant de relâcher ou d'amender progressivement certaines exigences. A l'inverse, une recherche usant d'un filtrage modéré usant de mots-clés peut permettre de filtrer rapidement un grand nombre de candidats pour isoler ou découvrir certains groupes de composants intéressants et obtenir une vision en largeur des composants à disposition. La prise en compte de ce mécanisme exploratoire *impose de disposer pour le composant recherché d'une description potentiellement multi niveaux*.

Les mécanismes de recherche nécessitent le recours fréquent à des termes ou à des structures qui dénotent des sens ou des propriétés particulières. Or plusieurs termes ou structures différentes (les signifiants) peuvent dénoter les mêmes objets ou des objets sémantiquement proches (les signifiés). Ainsi un même service (au sens spécification de la sémantique d'un comportement) peut être proposé dans des composants concrets par des opérations ayant des noms différents. Un service de création d'un répertoire pourra être offert sous les traits d'une opération de nom `CreateDirectory` ou `MakeDirectory` ou encore `CreerRepertoire`. Plus subtilement, un service complexe peut apparaître sous la forme d'opérations se distinguant non seulement par leur nom mais également par la liste ordonnée des types de leurs paramètres. Il peut également être réalisable au travers d'une séquence de plusieurs opérations élémentaires ou d'une méthode plus abstraite moyennant un paramétrage adéquat. De la même façon, une interface peut se retrouver éclatée en plusieurs interfaces dont l'union englobe l'interface recherchée. La prise en compte systématique de tous les cas de figure est impossible car elle mettrait en œuvre des calculs pour certains indécidables. Des équivalences architecturales, des ontologies ou plus prosaïquement la gestion de dictionnaires de synonymes ne peuvent apporter au mieux qu'une réponse partielle au caractère protéiforme des « signes » associés aux propriétés abstraites recherchées. Il faut cependant *offrir autant que possible des mécanismes langagiers permettant de tenir compte de ces zones de variabilité*.

Dans le monde des composants, il est rare de trouver un candidat qui réponde parfaitement à toutes les attentes du concepteur. Le concept de composant recherché doit donc décrire, non seulement les propriétés fonctionnelles et non fonctionnelles attendues, mais aussi la manière dont sont traités les « écarts » avec les propriétés des candidats. En effet, le simple calcul d'une différence ensembliste entre la liste des propriétés listées par le composant recherché à un certain niveau de granularité et celles offertes par un candidat concret est insuffisant pour permettre un classement pertinent. Se limiter à une telle approche revient à admettre que la présence, l'absence ou le respect partiel de deux propriétés ont les mêmes conséquences. Or l'absence ou le respect seulement partiel de certaines propriétés peut être plus ou moins pénalisant. L'absence d'un élément dans un candidat peut être pardonnable si cela ne demande pas trop d'effort de le rajouter ou s'il n'est pas essentiel, tandis que l'absence d'un élément

important que l'on ne peut ajouter vaudra le rejet du candidat. Par conséquent, le composant recherché ne doit pas seulement englober une description de ses propriétés, mais aussi *une structure détaillant la manière de traiter les disparités entre sa description et celles des candidats*.

Un langage dédié à la représentation d'un composant recherché doit donc intégrer les constructions permettant de résoudre différents problèmes :

1. un (quasi) isomorphisme avec les concepts des langages de description de composants existant aussi bien sur le plan fonctionnel que non fonctionnel ;
2. une description à plusieurs niveaux d'abstraction ;
3. la reconnaissance de zones de variabilité ;
4. la description de la structure d'interprétation des divergences.

Je vais dans la sous-section qui suit détailler la syntaxe abstraite et la sémantique du DSL que nous avons proposé ; DSL qui tente de relever une partie des 4 défis qui précèdent.

Le langage que nous avons défini n'a pas à ce jour de syntaxe concrète autre que celle émergeant de l'outil utilisé pour saisir un modèle de composant recherché. La syntaxe concrète étant la vision utilisateur d'un langage, elle présente bien sûr à l'usage une grande importance en termes de lisibilité et d'ergonomie. Mais nos travaux ne se sont pas focalisés sur ce point en l'état.

### 5.3.6 Syntaxe abstraite et sémantique du langage de comparaison

La syntaxe abstraite du langage de comparaison a été donnée à l'origine sous la forme d'un métamodèle plat (un seul package) exprimé en UML (Figure 5-9). Le choix d'un modèle « plat » est un point critiquable sur lequel je reviendrai dans la partie bilan. Dans sa structure même, la version actuelle du métamodèle entrelace donc les constructions langagières relatives aux quatre aspects précédemment évoqués. Cette syntaxe a été voulue pragmatique dans le sens où les concepts intégrés sont à la fois le reflet des concepts en vigueur dans les marchés actuels mais également de notre souci de proposer une réponse effective (automatisable) et non théorique. Il aurait été en effet possible d'injecter des informations de nature sémantique et/ou plus formelle. Mais, les marchés sont très loin d'user de ce type d'information.

Pour le premier point, une partie du métamodèle (partie située à droite de la Figure 5-9) réalise une synthèse des artéfacts présents dans la grande majorité des langages de description de composants concrets du moment. Trois types d'artéfacts ont été retenus : les composants, constitués de deux sortes d'interfaces (fournies et requises), chaque interface est elle-même composée d'un ensemble d'opérations. Une opération est définie par sa signature (liste ordonnée des types en entrée et du type de retour).

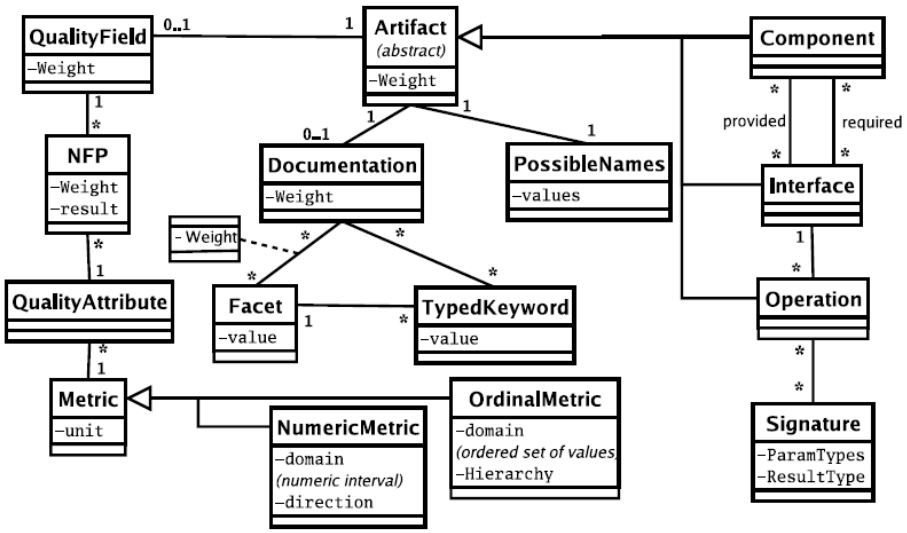


**Figure 5-9 Syntaxe abstraite du langage de comparaison**

Sur le plan non fonctionnel, tout artéfact (composant, interface ou opération) peut se voir associer un champ qualité. Ce champ regroupe l'ensemble de ses propriétés non fonctionnelles (partie en haut à gauche de la Figure 5-9). L'idée est inspirée des langages de contrats de qualité de service tels que QML (Frolund, et al., 1998) et QoSCL (Defour, et al., 2004). Une propriété non fonctionnelle (NFP sur la figure) représente une valeur (notée `result`) obtenue suite à la mesure du niveau d'une caractéristique ou sous-caractéristique qualité (au sens ISO 9126) sur un artéfact. Cette mesure est faite au moyen de l'unique métrique associée à l'élément (caractéristique ou sous caractéristique) qualité concerné. Cette structure s'inspire des modèles de qualité pour composants sur étagère comme CQM (Alvaro, et al., 2006).

Nous avons choisi les métriques pour représenter et comparer les propriétés non fonctionnelles, car contrairement à d'autres méthodes qui se concentrent sur une propriété ou une famille particulière de propriétés comme la qualité de service, elles nous ont semblé l'outil d'évaluation le plus pragmatique, à ce jour, pour spécifier les propriétés fonctionnelles relatives à la qualité. Il n'existe pas dans la littérature de données systématiques permettant de comparer les valeurs obtenues pour un même élément qualité avec des métriques différentes. En conséquence, nous avons imposé qu'un attribut qualité ne puisse être mesuré que par une seule métrique, même si une métrique peut mesurer plusieurs éléments.

Une métrique peut être numérique ou ordinale. Le domaine d'une métrique numérique est un sous-ensemble des nombres réels positifs (entiers, pourcentage...). Le domaine d'une métrique ordinale est, pour sa part, un ensemble fini totalement ordonné. Une métrique numérique possède un attribut supplémentaire appelé direction, qui permet d'interpréter le résultat d'une métrique. Les directions utilisables sont `croissante` et `décroissante`. Une direction croissante signifie que la qualité est d'autant meilleure que la valeur de la métrique est élevée. Une métrique ordinale possède un attribut supplémentaire appelé hiérarchie. La hiérarchie regroupe toutes les valeurs possibles de la métrique en leur associant une clé qui symbolise leur rang. C'est ce rang qui définit la relation d'ordre total sur le domaine de la métrique.

La capacité de décrire un composant à plusieurs niveaux d'abstraction est obtenue par la conjonction d'une description hiérarchique (on arrête la description au niveau souhaité) et d'un ensemble d'informations (au centre de la Figure 5-9) qui tiennent lieu de documentation pour tout artéfact. Chacune des informations contenues dans cette documentation est appelée « mot-clé typé ». Il s'agit d'une paire de chaînes de caractères `(facette, valeur)`. La valeur (classe `TypedKeyword`) est typée par la facette (classe `Facet`). Une facette définit un domaine fini de valeurs. Pour un composant donné, une facette si elle est documentée, est associée à une valeur particulière tirée du domaine de la facette. Il est ainsi possible d'associer à un artéfact une documentation stipulant que son concepteur est la société NBOS et qu'il a été développé avec la technologie EJB. Il suffit pour cela de lui associer une documentation comportant deux mots-clés typés. Le premier aura pour facette `Publisher` et pour valeur `NBOS`. Le deuxième aura pour facette `Technology` et pour valeur `EJB`. Le domaine de valeur de la facette `Technology` pourrait inclure d'autres valeurs comme : `Corba`, `JavaBean`, etc.

La prise en compte du caractère protéiforme de certaines propriétés (point 3 précédent) est, de loin, le point le plus délicat. La version actuelle du langage est sur ce point perfectible car minimaliste. Le langage se limite à une gestion simplifiée de la variabilité par nommage en incluant des dictionnaires de synonymes. En effet, tout artéfact a la possibilité de se voir adjoindre un ensemble de noms possibles (champ `values` de la classe `PossibleNames`). Pour des raisons de cohérence, on imposera qu'un ensemble de noms possibles ne contienne pas deux fois le même nom. Pour chaque opération, il est également possible d'associer plusieurs signatures. Deux autres types de variabilité d'ordre structurelle sont prises en charge mais à un niveau non plus langagier mais outil. L'opérateur de comparaison que je présenterai dans la section suivante vérifie ainsi l'inclusion d'une interface dans d'autres interfaces et le sous-typage des signatures.

Pour résoudre le problème des discordances entre les propriétés d'un composant recherché et celles d'un composant candidat, nous avons choisi d'associer au format de description une

pondération. Elle consiste à attacher un poids (attribut `Weight` dans le diagramme de classes de la Figure 5-9) à chaque élément du format de description. Ce poids exprime l'importance relative de chaque élément par rapport aux autres éléments de même niveau. Ainsi, à l'intérieur d'un même champ qualité, le poids de chaque NFP représente son importance par rapport aux autres NFP. Le sens qu'on donne à ces poids varie selon la sémantique de comparaison choisie. Lorsque chaque élément d'un composant candidat est comparé à un élément correspondant dans le composant recherché, le poids de ce dernier élément influe sur la comparaison. Il permet d'estimer l'importance réelle de cette comparaison locale dans l'évaluation globale du composant candidat.

### 5.3.7 Caractéristiques souhaitables pour un opérateur de comparaison

Un opérateur de comparaison que je noterai $comp(E_1,E_0)$ est une fonction à deux opérandes : $E_1$ est la description d'un composant candidat et $E_0$ est la description du composant recherché. Cette fonction totale d'arité deux a pour ensemble de définition le produit cartésien de l'espace de comparaison $E_{comp}$ avec lui même. A l'aide de cette fonction, on doit pouvoir classer tous les composants candidats. Cela suppose en particulier que l'on puisse comparer tout couple de résultat, donc que l'ensemble d'arrivée $I$ de la fonction soit muni d'une relation d'ordre totale $\leq_I$. $I$ peut être, par exemple, un intervalle de l'ensemble des réels.

Sémantiquement, on cherche à induire pour chaque composant recherché $E_0$ une relation de pré-ordre totale $\leq_{comp,E0}$ sur $E_{comp}$ à l'aide de la fonction de comparaison et de l'ordre total existant sur $I$. Ce pré-ordre induit est formellement défini par : $E_1 \leq_{comp,E0} E_2 \Leftrightarrow comp(E_1,E_0) \leq_I (E_2,E_0)$. La relation de pré-ordre $\leq_{comp,E0}$ doit, au minimum, respecter le principe de substitution : $E_1$ sous-type de $E_0 \Leftrightarrow comp(E_1,E_0) = comp(E_0,E_0)$. En effet, pour les descriptions sous-types, un classement ne présente aucun intérêt puisqu'elles répondent toutes parfaitement au besoin exprimé. Elles ne peuvent être départagées par l'opérateur à moins de renforcer les exigences présentes dans la description du besoin. L'opérateur `comp` donne pour tous les sous-types de $E_0$ la même valeur égale à $comp(E_0,E_0)$. On peut décider sans perte de généralité que cette valeur est une constante pour tout besoin $E_0$ (par exemple 100% pour manifester une couverture totale du besoin). On peut de même, considérer que cette constante est la borne sup de $I$. $I$ est alors l'intervalle des nombres réels $[0,1]$.

Cette première propriété ne définit encore que très partiellement la sémantique d'un opérateur de comparaison pour les composants. Il reste encore à définir la sémantique que nous voulons associer à $E_2 \leq_{comp,E0} E_1$ pour deux descriptions candidates $E_1$ et $E_2$. On veut que cette relation manifeste le fait que $E_1$ répond « mieux » au besoin décrit par $E_0$ que $E_2$. Cependant, cette notion de "mieux" contient une ambiguïté qui rejaillit sur la sémantique à donner à `comp`. On peut avoir un candidat qui possède un grand nombre des éléments exigés par $E_0$, mais qu'on ne peut pas adapter pour les quelques éléments restants ou alors, au prix d'un effort important. A l'inverse, on peut avoir un candidat auquel il manque de nombreux éléments, mais que l'on peut facilement adapter. Il y a donc au moins deux façons pertinentes dans le monde des composants pour définir la fonction de comparaison : soit sous l'angle de la satisfaction fonctionnelle et non-fonctionnelle (plus un candidat satisfait de propriétés mieux il est noté), soit sous l'angle de l'effort à consentir pour intégrer un composant candidat à la place du besoin exprimé. Cette notion d'effort est préférable mais malheureusement beaucoup plus difficile à calculer automatiquement que la satisfaction. L'objectif est donc d'utiliser la première approche pour tenter d'approcher au mieux la seconde.

### 5.3.8 Deux opérateurs de comparaison

Une analyse attentive de la syntaxe abstraite du langage de comparaison nous permet de distinguer une structure hiérarchique. Dans ce langage, un composant est décrit par un arbre dont la racine est un artéfact de type composant et les nœuds enfants sont potentiellement : des

interfaces, une documentation, un ensemble de noms possibles et un champ qualité. Parmi eux, un nœud interface peut avoir pour nœuds enfants (Figure 5-10) : des opérations, une documentation, un ensemble de noms possibles et un champ qualité. La description du composant recherché et d'un composant candidat prennent donc la forme de deux arbres dont les nœuds sont typés (Interface, opération, documentation, etc.).

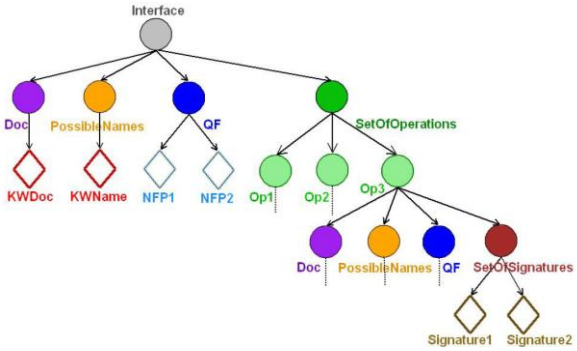


**Figure 5-10 Forme arborescente d'une description de composant**

Nous devions donc proposer un opérateur capable de comparer deux arbres. Nous avons décidé de définir un opérateur de comparaison qui parcourt récursivement les deux arbres de manière synchrone selon un mode en profondeur suffixé. Il retourne une valeur comprise dans l'intervalle des réels [0,1]. Si la comparaison retourne 1, le composant candidat est sous-type du composant recherché. Si la comparaison retourne 0, le composant candidat ne présente aucune des propriétés exprimées dans le composant recherché.

Pour comparer un nœud interne $E_0$ de l'arbre du composant recherché (Figure 5-11) avec un nœud $E_1$ de même profondeur du composant candidat, nous comparons deux à deux chaque nœud fils (de gauche à droite) de $E_0$ avec tous les nœuds fils de $E_1$. Ainsi pour comparer $E_0$ et $E_1$ nous comparerons dans l'ordre : ($e_{00}$ et $e_{10}$), ($e_{00}$ et $e_{11}$), ($e_{00}$ et $e_{12}$), ($e_{00}$ et $e_{13}$), ($e_{01}$ et $e_{10}$), ($e_{01}$ et $e_{11}$), etc. Le fait de comparer chaque nœud fils du recherché avec tous les fils du nœud candidat offre plusieurs avantages : l'ordre d'apparition des nœuds est sans importance et surtout on cherche le meilleur substitut chez le candidat. Notez qu'un même nœud fils du candidat peut être désigné comme meilleur substitut pour plusieurs nœuds du recherché.

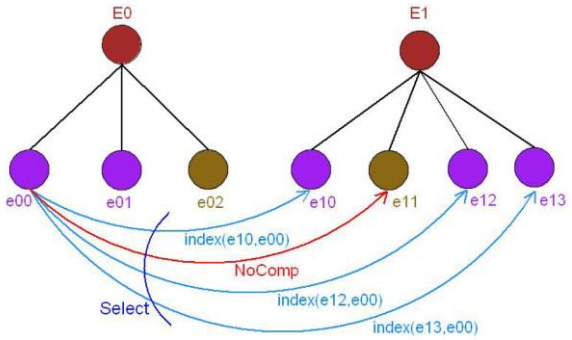


**Figure 5-11 Comparaison de deux nœuds internes $E_0$ et $E_1$**

Lors de ces comparaisons entre nœuds fils trois cas peuvent se présenter.

1. On compare deux nœuds de types différents. Cette comparaison n'ayant aucun sens, la comparaison retourne la valeur 0. Ainsi la comparaison des nœuds fils $e_{00}$ et $e_{11}$ retournerait 0 (on suppose sur la figure que la couleur d'un nœud manifeste son type).
2. La comparaison de deux nœuds feuilles de même type va demander le calcul d'une comparaison propre à ce type. Nous avons défini quatre opérateurs de comparaison, un par type de feuille possible (opération, NFP, mot-clé typé et ensemble des noms possibles). Ces 4 opérateurs retournent une valeur comprise entre 0 et 1. La valeur 0

correspond à un besoin non pris en compte et 1 à sa prise en compte complète. Les valeurs intermédiaires manifestent selon l'ordre croissant une prise en compte partielle du besoin.

3. On compare deux nœuds de même type et qui ne sont pas des feuilles de l'arbre. Dans ce cas on compare de manière récursive les sous-arbres dont ils forment la racine.

La comparaison de la signature des opérations tient compte du sous-typage (covariance du retour et contra-variance des paramètres pour une opération fournie).

Ce protocole connaît une exception : les interfaces. Dans la mesure où un composant candidat peut regrouper au sein de plusieurs interfaces fournies les opérations recherchées, la comparaison d'une interface recherchée se fait avec l'union des interfaces fournies par le candidat. Cela introduit une zone de variabilité supplémentaire qui n'est pas langagière mais procédurale (implantée dans l'opérateur). Les interfaces requises procèdent d'une manière analogue aux interfaces fournies mais en inversant le paramétrage des formules de calcul : le nœud $E_1$ devient le nœud $E_0$ et vice et versa car c'est $E_0$ qui doit être sous-type de $E_1$. En effet, il faut que $E_1$ requière moins de propriétés que $E_0$.

La prise en compte ou non des interfaces requises dans le processus de sélection a fait l'objet de nombreux débats entre nous. Certains pensant qu'une recherche ne devait pas les intégrer, d'autres qu'il fallait au contraire les considérer soit de bout en bout, soit à certaines étapes uniquement du processus de sélection. En l'état, une sorte de consensus mou est respecté. Les formules définissant l'opérateur de comparaison les intègrent (cf. la thèse de *Bart George*). Par contre l'outil *Substitute* implantant l'opérateur ne les gère pas…

Une fois toutes les comparaisons effectuées entre les nœuds fils, le calcul du score d'un nœud père est réalisé en « agrégeant » les scores obtenus au niveau des nœuds fils. Cette agrégation se fait en utilisant une pondération par distribution sur les nœuds du composant recherché. Chaque nœud se voit affecté un poids. La somme des poids des descendants directs d'un nœud père doit être égale à 1. Le score de comparaison de deux nœuds pères $E_0$ et $E_1$ est donc la somme pondérée des meilleurs scores de comparaison des nœuds fils de $E_0$. On trouvera plus de détails sur cet opérateur de comparaison dans (George, et al., 2008).

Ce sont bien entendu ces poids qui déterminent la sémantique de la comparaison. Dans la thèse de *Bart George* deux sémantiques de pondération ont été proposées :

- estimation de la « ressemblance » entre le candidat et le recherché en mettant en valeur les éléments selon leur niveau d'importance ;
- estimation de l'effort à fournir pour adapter le candidat en s'appuyant sur les coûts de développement « partant de rien » de chaque élément du composant recherché.

Avec une pondération portée par le composant recherché et donc identique pour tout composant candidat, il n'est pas possible de s'approcher plus de la notion d'effort d'adaptation. Un second opérateur de comparaison, qui utilise en entrée les résultats d'une comparaison par ressemblance, permet d'aller plus loin. Il évalue pour un candidat particulier son coût d'adaptation. Le calcul ne s'opère que sur les éléments discordants (détectés à l'aide de l'opérateur précédent) et sur la base d'une pondération renseignée par l'utilisateur propre, cette fois-ci, au composant candidat. Le poids d'un élément représente l'effort à consentir pour remédier à la discordance mesurée. Cet opérateur nécessite de saisir une importante quantité d'information pour chaque composant candidat, il ne peut donc être utilisé que sur un nombre très restreint de candidats.

### 5.3.9 Une démarche de sélection, un outil et deux expérimentations

Sur la base des deux opérateurs de comparaison précédents, une démarche de sélection comportant plusieurs étapes a été proposée. Cette démarche promeut, en particulier, une

recherche itérative utilisant des comparaisons alternant aussi bien les niveaux d'abstraction que les sémantiques de pondération. Classiquement, trois itérations peuvent être proposées :

1. Un premier filtrage des candidats est réalisé sur la base de critères uniquement de haut niveau (en utilisant le premier opérateur avec une sémantique de pondération de type ressemblance) ;
2. Un second filtrage utilisant tous les éléments du composant recherché est conduit (avec le même opérateur et la même sémantique de pondération que dans l'étape précédente) ;
3. Un classement des quelques candidats restant est effectué en usant d'une estimation de l'effort d'adaptation à fournir (avec le second opérateur estimant l'effort d'adaptation de chaque candidat).

Pour valider cette démarche et les deux opérateurs qui la portent, nous avons réalisé deux expérimentations sur un marché réel : *ComponentSource*. Elles ont montré la faisabilité pratique et l'intérêt de notre proposition. Toutes deux sont des cas d'étude menés en laboratoire. Ils furent conduits par *Bart George*. Nous nous étions placés dans le contexte suivant : un concepteur recherche pour son application un composant réseau parmi tous les candidats disponibles dans la section « communication internet » du marché aux composants *ComponentSource*. Cette section contenait 131 composants au moment où nous avons effectué nos deux expérimentations. Nous avons préalablement produit (semi automatiquement), pour chaque composant de cette section, une description conforme à notre langage de comparaison. C'est sur la base de ces 131 descriptions que nos deux cas d'étude ont été conduits. Différents types de filtrage ont été réalisés mélangeant les niveaux d'abstraction et les sémantiques de comparaison. Les calculs ont été réalisés à l'aide de notre outil appelé *Substitute*. Cet outil prend en paramètre les fichiers XML décrivant le modèle de qualité choisi et l'ensemble des descriptions des composants utilisés (le composant recherché comme les composants candidats). Il peut retourner aussi bien les indices de ressemblance locaux pour les éléments fils (opérations, NFP...), que l'indice global de ressemblance de chaque composant candidat.

Les deux cas d'étude ont mis en évidence le principal avantage de l'automatisation de la sélection : les indices ont été calculés sur plusieurs dizaines de composants en quelques secondes, alors qu'une évaluation manuelle de chaque candidat pour chaque critère aurait pris plusieurs jours. De plus, il a été possible d'interpréter graphiquement les résultats à l'aide d'un outil de visualisation tiers, offrant une autre forme d'agrégation : une agrégation visuelle dans laquelle plusieurs mesures sur un même composant sont représentées par des attributs graphiques (taille, couleur, orientation). Cette méthode permet de repérer intuitivement et en un coup d'œil les meilleurs candidats. Elle permet également à l'utilisateur d'effectuer ses propres compromis de sélection en ayant accès aux indices locaux de tous les candidats en même temps. Cette étude montre enfin que notre approche de sélection multi niveaux permet des comparaisons fines, qui exploitent dans une large mesure les propriétés fonctionnelles et non fonctionnelles des composants.

### 5.3.10 Bilan du thème

Une des qualités de ce travail est d'être en totale adéquation avec l'état actuel des marchés aux composants. Il est applicable tel quel *in vivo*. Ils existent des travaux dans la littérature ouvrant, en théorie, des perspectives bien plus séduisantes, proposant des recherches usant de critères sémantiques (Mili, et al., 1997) (Zaremski, et al., 1997) (Morel, et al., 2004), gérant mieux la variabilité ou offrant une représentation beaucoup plus fine des aspects non fonctionnels (Zschaler, 2009). Mais toutes les approches que j'ai pu étudier posent de telles hypothèses sur les documentations (souvent formelles) des composants ou sur l'existence d'hypothétiques modèles « universels » (par exemple des ontologies complètes sur un vaste domaine) qu'elles sont inapplicables dans les marchés actuels et pour plusieurs années encore. Notre travail est moins ambitieux mais en prise avec la réalité.

Il nous reste cependant à progresser sur plusieurs points. La traduction des descriptions des composants candidats dans notre format documentaire est un obstacle important. Les deux cas d'études sur le marché *ComponentSource* ont nécessité un travail laborieux pour traduire les 131 composants concernés. Ce travail a occupé une personne pendant plusieurs jours malgré l'usage d'un outil ad hoc permettant d'extraire automatiquement depuis des documents webs de l'information respectant certains formats. Nous avons commencé à explorer la piste de la transformation de modèles, mais on se heurte à la trop grande diversité des formats et à l'absence de métamodèles décrivant les formats utilisés. Ce sont aux marchés de progresser sur ces deux derniers points pour permettre le développement d'outils de recherches multi-marchés.

Le premier opérateur de comparaison pourrait intégrer des mécanismes plus puissants pour les mots-clés et les noms d'opération, par exemple s'appuyant sur des distances calculées dans des ontologies. Les systèmes d'agrégation des scores des nœuds fils pour obtenir le score du nœud père par pondération et visualisation peuvent être utilement complétés par d'autres types de pondération. Nous avons étudié une agrégation bayésienne qui permet de représenter le caractère indispensable de certaines propriétés. Cette pondération ouvre aussi la voie à des mécanismes d'inférence. Le second opérateur pourrait se connecter aux travaux estimant l'effort d'adaptation d'un composant. Mieux, il y a sans doute des ponts à tirer entre ce travail et les travaux proposant d'automatiser l'adaptation d'un composant (Bracciali, et al., 2005) qui prennent en entrée les différences constatées.

Enfin, il aurait été préférable de séparer dans le métamodèle du langage de comparaison : le langage décrivant la structure des composants, l'expression de la variabilité et les informations nécessaires à l'agrégation. Il serait ainsi plus facile de faire évoluer les uns indépendamment des autres. Plus encore on pourrait envisager de « composer » à la demande le langage de description en « tissant » ces trois préoccupations.

# Chapitre 6 Les travaux autour des langages

Ce chapitre détaille mes activités de recherche actuelles. La première section de ce chapitre aborde la modélisation et le contrôle des bonnes pratiques de modélisation. Ce travail est mené avec un doctorant en convention CIFRE depuis le milieu de l'année 2008. La seconde section présente un travail récent, débuté fin 2009, sur la caractérisation des opérateurs de composition de modèles. La troisième section est prospective. Elle présente quelques sujets qui pourraient donner lieu à des travaux dans l'avenir.

## *6.1 Modélisation et exécution de bonnes pratiques*

### 6.1.1 Origine du projet

Courant 2007, la thèse de *Réda Kadri* a conduit à la définition d'un ADL dédié à la représentation des applications web riches et d'un outil support. Très vite, il s'est posé lors de son intégration dans la société *Alkante*, des questions concernant son environnement méthodologique. En effet, la société disposait d'une expérience de plusieurs années de développement dans son domaine applicatif, elle souhaitait que ce capital méthodologique (essentiellement des règles de documentation, des architectures types et des règles simples de structuration des applicatifs) ne disparaisse pas et soit disponible lors de l'usage de cet ADL. *Alkante* reconnaissait ainsi ses bonnes pratiques comme un capital précieux pour garantir la satisfaction de ses clients, se distinguer ; briguer éventuellement des labels, des certificats et faire face à la concurrence. Elle rejoignait en cela une opinion partagée dans la communauté qualité (Gratton, et al., 2005).

Deux options s'offraient alors : intégrer ce capital « en dur » dans l'outil support de l'ADL ou encadrer l'usage de l'ADL au travers d'un procédé documenté sous une forme papier. La première solution garantissait le respect des consignes mais limitait fortement les possibilités d'évolution de ce patrimoine. La seconde, à l'inverse, offrait une grande flexibilité dans la durée, mais posait les mêmes problèmes que toute documentation qualité papier : le respect des consignes ne repose que sur la compétence et la bonne volonté du personnel (Shull, et al., 2005).

*Alkante* était confrontée aux difficultés que rencontrent nombre d'entreprises conscientes de la valeur de leur capital méthodologique. Faute de moyens qu'elles jugent adéquats, beaucoup d'entreprises ne rationalisent pas la gestion de ce type de connaissance. Elles ont, en effet, le sentiment qu'en absence de moyens dédiés, le coût de cette gestion est déraisonnable au regard du retour sur investissement escompté. Le savoir-faire ne se manifeste, ne perdure et ne s'enrichit qu'aux détours des modes opératoires et de l'expérience de quelques individualités. Il est donc non partagé, potentiellement volatil car soumis aux aléas des mouvements de personnel et à une hypothétique transmission. Cette volatilité est encore plus délicate pour les Petites et Moyennes Entreprises du secteur informatique soumises à des fluctuations importantes de leur personnel.

J'ai, donc proposé, début 2008, à la société *Alkante* de travailler sur la documentation et l'exécution de *bonnes pratiques de modélisation*. Dans le reste de ce mémoire, j'emploierai les acronymes *BP* pour désigner des bonnes pratiques quelconques et *BPM* pour désigner le sous-ensemble des bonnes pratiques qui concernent les activités de modélisation. Ce thème était pour moi dans la continuité de travaux débutés en 2006 à l'aide d'une doctorante (*Soraya Sakhraoui*, enseignante à l'université de *Sétif*) dont l'encadrement a été confié depuis à *Chouki Tibermacine* une fois celui-ci recruté en qualité de maître de conférences à Montpellier. L'objectif de sa thèse est d'étendre l'algorithme de contrôle de l'évolution proposé dans la thèse de *Tibermacine* pour intégrer des pratiques d'évolution autres que la seule vérification de la cohérence spécification non fonctionnelle / motifs architecturaux

Les premiers travaux se sont concentrés sur une représentation, indépendante de toute plate-forme logicielle à l'aide de modèles MOF, des bonnes pratiques identifiées par *Lehman* (Lehman, et al., 1997). Si la modélisation de ces BP semblait en partie possible, très vite, nous nous sommes heurtés à la difficulté de leur exécution dans les outils : non seulement ces BP sont très abstraites mais en plus leur contrôle implique l'orchestration d'activités relevant de plusieurs outils souvent propriétaires donc incapables de communiquer. J'avais conclu à l'époque que ce problème, pour être traité en profondeur, réclamait plus de ressources humaines (doctorants, ingénieurs) que notre petite équipe de recherche ne pouvait en dégager. Si nous voulions apporter des réponses concrètes, il fallait impérativement nous restreindre à des BP qui ne concernent qu'un seul outil. Les bonnes pratiques de modélisation (BPM), dont on peut confiner le contrôle dans des éditeurs, apparaissaient comme une occasion intéressante de poursuivre ces travaux.

Un sujet de thèse sur ce thème en convention CIFRE est signé en Février 2008. Le doctorant recruté (*Vincent Le Gloahec*) s'est vu confier la mission de proposer :

- un langage de documentation des BPM indépendant des outils ;
- un mécanisme permettant de rendre effectives (productives) dans les outils ces BPM.

### 6.1.2 Etat de l'art

La réponse à la problématique de la capitalisation des BP est un sous-domaine du "Knowledge Management" ; domaine qui s'intéresse à la mise en place de moyens permettant à une entreprise d'identifier, de collecter et de capitaliser toutes les connaissances (dont les bonnes pratiques) utiles à l'amélioration de sa performance (Stewart, 2001). Les activités identifiées dans ce domaine s'appliquent, tout naturellement, à l'univers des BP. On retrouve des préoccupations identiques, relatives à la recherche, la collecte, l'archivage, la diffusion, l'adaptation, l'utilisation, l'évaluation et l'amélioration. Dans le domaine des BP, deux de ces activités ont donné lieu à de nombreux travaux : l'archivage et le contrôle de l'utilisation des BP. Nous allons présenter successivement ces deux types de proposition.

Des travaux proposent d'introduire des processus et des outils informatiques qui facilitent le stockage, le partage et la diffusion de BP au sein des entreprises (Fragidis, et al., 2006). Ils prônent en particulier, l'usage de véritables dépôts informatisés offrant des modes de consultation variés, facilitant la recherche et la découverte de BP. Des systèmes de classification multicritères rendent possible le parcours du dépôt en usant de mots clés relatifs à un domaine applicatif, au type et à la taille des projets, à l'étape du cycle de vie concerné, au nom du problème rencontré, etc. Cependant, les BP référencées par ces systèmes, ne sont consultables au final qu'au travers de documents textuels et informels. Ce manque de formalisation pose des problèmes d'interprétation. Ils augmentent le risque d'un usage inadéquat et donc potentiellement inefficace des BP.

A ma connaissance, il n'existe pas de travaux abordant frontalement la documentation rigoureuse des BPM. Les travaux qui s'en approchent le plus sont le langage SPEM de l'OMG et les tentatives liées à la modélisation des méthodes de développement (*method engineering*) (Henderson-Sellers, 2003) (Gonzalez-Perez, et al., 2007) (Rolland, 2009). Cependant, les langages proposés ambitionnent la documentation de pratiques (méthodes) bien plus générales. Ils ont essentiellement une vision tournée vers la représentation de procédés à gros grain : des étapes et leurs liens de précédences. Ils n'accordent pas d'importance à la forme des artéfacts manipulés et aux langages ou aux parties de langage utilisés. Ils laissent donc de côté certaines spécificités importantes des activités de modélisation, en particulier les restrictions langagières et le respect de règles de documentation imposées par certaines étapes.

De toute façon, même correctement documentées, placées et adaptées dans le cadre d'un processus de développement particulier, les BPM ne sont pas nécessairement appliquées. Leur application reste encore soumise à la bonne volonté des personnes qui en ont la charge. Or, force est de constater que fréquemment, in situ pendant les projets, les développeurs ne les

appliquent pas. Ce faisant, les personnels concernés, avec souvent l'assentiment de leurs responsables opérationnels, pensent à tort pouvoir répondre plus efficacement à des impératifs de délais ou de coûts.

Pour limiter ce genre de dérive, des travaux proposent au travers des outils de Génie Logiciel, d'inciter, voire d'imposer, le respect de certaines BPM. Ces travaux tentent de pallier les déficiences des outils du commerce tels que *Together*, *MagicDraw* ou *Objecteering* qui au mieux permettent de lancer à la demande l'exécution de codes OCL ou de scripts pour vérifier que les modèles respectent certaines propriétés. Le domaine ayant obtenu les résultats les plus probants ces dernières années est, sans conteste, celui qui s'intéresse à la gestion des incohérences lors des activités de modélisation. On citera, en particulier, les travaux de (Egyed, 2007) (Blanc, et al., 2008) (Biehl, et al., 2009). Ils proposent de greffer sur des outils de modélisation comme *Eclipse* ou *Rationale* des extensions capables d'intercepter les actions des développeurs et d'inspecter le système d'information de l'outil pour détecter l'apparition de certains types d'incohérence.

Les incohérences traitées sont de natures très variées. Certains travaux vont au-delà de l'analyse, déjà très complexe, de la consistance syntaxique et sémantique de modèles exprimés dans un ou plusieurs langages de modélisation. Ils abordent explicitement le problème de ce qu'ils appellent les "incohérences méthodologiques" ; c'est-à-dire de la détection du non respect de directives qui ne sont pas liées à ce qu'est le langage, mais à la façon dont il doit être utilisé. En cela, ils fournissent un premier éventail d'outils potentiellement utiles à la vérification de certaines BPM.

Au final, on ne peut que constater la totale déconnexion actuelle de deux types de travaux. Il y a ceux qui se préoccupent de l'archivage et de la diffusion des BP et ceux qui proposent la vérification automatique in situ du respect des BP. Notre objectif est donc, à terme, de bâtir un pont entre, d'une part l'univers des dépôts des BP et d'autre part celui des outils offrant des mécanismes permettant la vérification du respect de BPM. Nous proposons, pour cela, d'adopter une démarche d'ingénierie dirigée par les modèles (IDM) en rendant les BPM productives.

### 6.1.3 Nécessité d'une documentation indépendante des outils

Il y quatre façons d'implanter dans les outils de Génie Logiciel, la vérification du respect de BPM (Figure 6-1). L'implantation peut être externalisée (un outil tiers dialogue avec l'éditeur pour « contrôler » son fonctionnement) ou interne (c'est l'éditeur lui-même qui contrôle le respect des bonnes pratiques). L'implantation peut également permettre de faire évoluer ou non les BPM prises en charge. Au final, ce sont quatre possibilités d'implantation des BPM qu'il faut examiner :

1. les coder dans le code source de l'éditeur ;
2. les coder dans le code source d'une extension logicielle dialoguant avec l'éditeur au travers de son API ;
3. les exprimer dans le langage de paramétrage offert par l'éditeur (si un tel langage existe), par exemple le langage *J* de l'atelier *Objecteering* ;
4. les exprimer dans un langage « interprétable » par une extension de l'éditeur (par exemple Prolog dans le cas de (Blanc, et al., 2008)).

Chacune de ces possibilités présente des avantages et des inconvénients. La première va permettre la représentation (propre à l'éditeur) de toutes les BPM, profitant d'un accès complet au système d'information de l'éditeur. Elle suppose par contre que le code source de l'éditeur soit disponible. La deuxième possibilité ne présente pas cet inconvénient, mais elle repose sur l'existence d'une API (plus ou moins puissante) donnant accès au système d'information de l'outil. Les possibilités 3 et 4 permettent, sans recompilation, d'étendre à volonté la liste des BPM prises en compte par l'outil. Elles sont, en conséquence, bien adaptées à la capitalisation au fil de l'eau. La troisième exige la présence d'un langage de paramétrage suffisamment puissant. Mais la description des BPM reste propriétaire. La quatrième possibilité est celle qui a

la plus grande chance de fournir un langage véritablement adapté à la description de BPM indépendantes de tout outil cible.

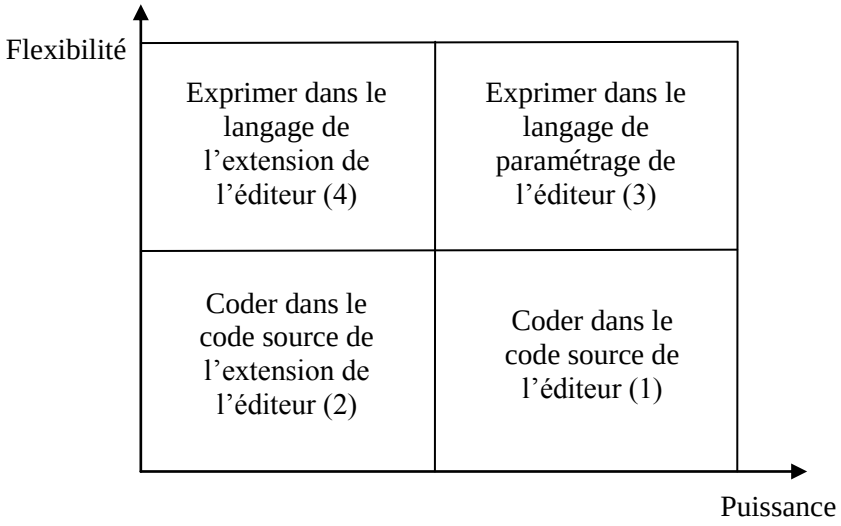


**Figure 6-1 Les 4 implantations possibles**

Nous avons opté pour la quatrième possibilité ; la seule permettant d'envisager une capitalisation indépendante de tout outil. A l'image de la terminologie que prône l'OMG, j'ai identifié deux niveaux de description d'une BPM (Figure 6-2) :

- PIM (*Platform Independent Model*) qui offre une représentation indépendante de tout outil ;
- PSM (*Platform Specific Model*) qui donne une représentation adaptée à un outil particulier

Ce faisant, je ne faisais qu'entériner le fait qu'une BP dépend uniquement du langage dont elle organise l'usage et non des outils utilisés. La capitalisation s'opèrerait donc au niveau PIM, avec un langage dédié aussi puissant et accessible que possible. Ce langage doit, bien sûr, être acceptable par les développeurs et les personnels en charge dans l'entreprise de la capitalisation des bonnes pratiques (les responsables qualité).

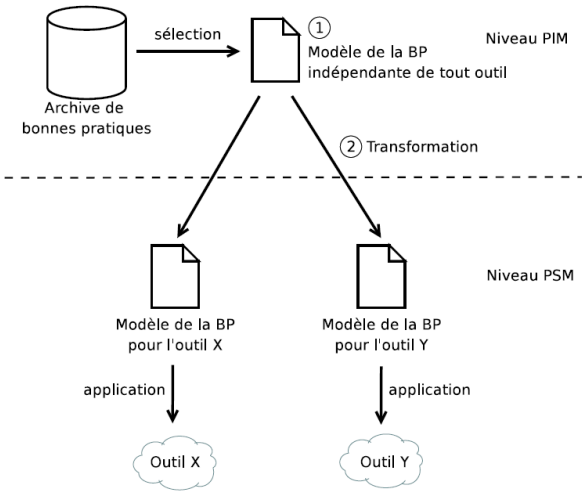


**Figure 6-2 Passage PIM vers PSM des BPM**

L'encodage nécessaire pour passer d'une formulation de niveau PIM à un niveau PSM, exprimée dans un langage propriétaire interprétable par l'extension d'un éditeur ou par l'éditeur lui même, est une tâche qui peut s'avérer lourde et ardue ; d'autant plus que l'on peut devoir cibler plusieurs outils et qu'un outil peut lui-même évoluer et donc remettre en cause une

description existante de niveau PSM. Le caractère productif des modèles de niveau PIM va permettre de répondre à ce problème. En effet, il devient envisageable de transformer, à la demande, de manière aussi automatique que possible les modèles PIM en modèles de niveau PSM conformes aux langages interprétables par les extensions de chaque outil.

### 6.1.4 Nature des bonnes pratiques de modélisation

Avant de proposer un langage de modélisation des BPM nous devions déterminer précisément leur nature. Nous avons posé comme hypothèse de travail que les procédés associés aux BPM à un niveau PIM sont séquentiels, sur le constat qu'un développeur ne mène qu'une seule activité à la fois. Ce choix, raisonnable au niveau PIM (les méthodes de modélisation sont toutes séquentielles), est cependant discutable au niveau PSM. En effet, les éditeurs permettent parfois de lancer en tâches de fond des activités tout en laissant le développeur poursuivre son travail.

Comme les BPM incluent potentiellement des aspects « procédés » et/ou « styles de modélisation», nous avons étudié des situations relevant de ces deux extrêmes. Pour l'aspect « procédé » nous avons analysé les procédés associés aux :

- méthodes de modélisation (on en trouve une synthèse dans (Ramsin, et al., 2008)) ;
- démarches d'application de patrons de conception ou de styles architecturaux ;

Pour les aspects styles nous avons étudié :

- le référentiel des BPM de la *modélisation agile* (Ambler, et al., 2002). Ce référentiel contient un peu moins de 300 pratiques pour le langage UML relevant, pour leur grande majorité, de consignes de documentation ;
- la littérature sur la modélisation des patrons de conception aussi bien sur le plan de leur contexte d'application (Kim, et al., 2007) que de la solution qu'ils préconisent (Bayley, et al., 2010) ;
- les métriques associées au modèles UML (Yi, et al., 2004).

Cette étude a mis en avant la nécessité de disposer des moyens qui suivent.

1. Définir, d'une part des activités dans lesquelles les développeurs sont libres d'agir comme ils le souhaitent et, d'autre part, des activités entièrement automatiques ou semi-automatiques (dans lesquelles s'instaure une *modélisation collaborative* Homme/Editeur).
2. *Restreindre*, dans certaines activités, les concepts du langage de modélisation manipulables par le développeur ; manipulations elles-mêmes limitées, pour chacun des concepts, à certaines classes d'actions parmi : création, destruction, lecture et modification.
3. Assurer que l'on ne puisse entrer ou sortir de certaines activités de modélisation que si certaines *conditions* sont vérifiées. A titre d'exemples un pattern n'est applicable que dans un contexte particulier et un style de modélisation s'apprécie une fois l'activité de modélisation terminée. Une partie de ces conditions sont vérifiées automatiquement au travers de *fonctions* usant d'informations présentes dans la syntaxe abstraite ou dans la représentation graphique du modèle ; les conditions restantes ne sont appréciables que par le développeur car elles intègrent des notions trop abstraites relatives à l'esthétique des modèles ou à leur sémantique (pertinence, complétude, etc.).
4. Organiser les activités de modélisation d'une BPM selon des procédés comportant des *séquences*, des *itérations*, des *étapes facultatives* et des *points de décision* (lorsque plusieurs choix de poursuite sont envisageables en sortie d'une étape). Une décision peut être prise automatiquement sur la base de l'état courant du modèle et du procédé ou laissée à l'appréciation du développeur.
5. Permettre d'orchestrer l'utilisation au même instant de plusieurs BPM ; un développeur devant par lui-même exécuter une tâche figurant dans une BPM

peut décider de lancer une autre BPM dont il connaît l'efficacité pour la tâche considérée. Il peut ainsi « empiler » les appels à des BPM ; la fin de la BPM « appelée », implique un retour au contexte de la BPM « appelante ».

Les aspects précédents sont relatifs à la définition du contenu d'une BPM. Mais une BPM ne s'exprime pleinement que dans un environnement adéquat. Leur usage peut varier selon deux axes orthogonaux.

- *Domaine de définition* : Tous les projets, les domaines applicatifs, les développeurs ou les organisations ne profitent pas des mêmes BPM. Il doit être possible d'associer ou non les BPM à certains types de projets, projets particuliers, voire à certains développeurs ;
- *Domaine de variation* : Les BPM peuvent profiter d'adaptations « locales ». Un développeur expérimenté peut choisir de « sauter » plusieurs étapes d'un procédé, élémentaires à ses yeux, et produire d'un seul trait un modèle respectueux des conditions en entrée d'une étape plus lointaine. Plus encore, une BPM ne peut pas décrire tous les cas de figure. Elle décrit le scénario le plus probable. Face à certains événements exceptionnels, il doit être possible de suivre des chemins dans le procédé qui n'ont pas été prévus (retour en arrière en cas de blocage avec un mécanisme transactionnel, etc.).

Toute la difficulté est donc d'offrir un cadre permettant la définition et l'exécution de BPM répondant à deux visions qui ne sont pas antinomiques au sein des entreprises disposant d'un système qualité.

- Une *vision systémique de la qualité* : des BPM sont capitalisées au niveau de l'entreprise. Elles sont ensuite imposées sur certains procédés. Leur respect strict est exigé de tous les développeurs. On se situe dans l'univers régi par des référentiels normatifs tels qu'ISO 9000-3 et CMMI. Dans cette vision le domaine de définition des BPM est fixé par les responsables qualité et le domaine de variation est très réduit.
- Une *vision individuelle de la qualité* : chaque développeur capitalise ses propres BPM dont il apprécie la valeur ajoutée dans certaines situations. Il les utilise, lorsque bon lui semble, comme autant de « macros sur étagères » facilitant son quotidien. Le domaine de définition des BPM est adapté au fil de l'eau par chaque développeur. Le domaine de variation de chaque BPM est aussi large que possible dynamiquement et statiquement. Cette vision se place dans la mouvance de l'amélioration permanente de la qualité des individus telle que PSP (Humphrey, 1997).

La première vision pose clairement les BPM en contraintes : les BPM sont des procédés idéaux dont les procédés réels doivent s'écarter le moins possible. La seconde vision, au contraire fait des BPM des « macros » méthodologiques dont les développeurs peuvent adapter les contours en temps réel pour gagner en efficacité. La coexistence de ces deux visions conduit à des procédés réels de modélisation dont la trame émerge de la mise en séquence et parfois de « l'empilement » non anticipé de BPM pour certaines imposées et pour d'autres désirées. La cohérence de cette « agrégation dynamique » de BPM est du ressort des responsables qualité et de chaque développeur. Elle est partiellement encadrée par le biais des conditions d'entrée et de sortie posées sur les activités de modélisation. Enfin, il semble judicieux de prévoir un mode d' « agrégation statique » des BPM : c'est-à-dire permettant de concevoir de nouvelles BPM en « composant » à l'aide d'opérateurs ad hoc (séquence, itération, substitution d'une activité d'une BPM par une autre BPM, etc.) des BPM existantes.

### 6.1.5 Un premier langage exploratoire : le langage *GooMod*

Sur la base des propriétés que nous avons identifiées, un premier langage de niveau PIM, nommé *GooMod*, a été proposé. Il sera présenté à la conférence internationale QSIC'2010.

Ce langage est encore loin de satisfaire tous les besoins évoqués dans la section précédente. En particulier, il n'intègre pas les concepts nécessaires à la représentation des domaines de définition. Il ne permet pas non plus l'expression explicite de domaines de variation. Ces deux points sont pour le moment fixés par l'unique implantation PSM existante qui tient donc lieu de référence. Le langage sera donc amené à évoluer dans les mois qui viennent. Il permet cependant d'explorer l'espace des solutions et de tester l'impact de certains choix sur le plan méthodologique et technologique. La syntaxe abstraite de ce langage à la date de rédaction de ce mémoire (Avril 2010) est présentée dans la Figure 6-3 sous la forme d'un modèle MOF.

Dans ce langage, le procédé associé à une BPM est décrit syntaxiquement sous la forme d'un graphe orienté faiblement connexe. Un sommet du graphe (`Step`) représente une activité de modélisation cohérente que nous appelons une « étape ». Les arcs (`Bind`) lient des paires de sommets. Les boucles (arcs dont la tête et la queue coïncident) sont autorisées, mais pas les multi-arcs (arcs ayant la même queue et la même tête). Les graphes intègrent la séquence, l'itération, la notion d'étape optionnelle et implicitement de point de décision automatique ou non. Le point 4, identifié dans la sous-section précédente, est ainsi satisfait. Sémantiquement, sur l'aspect procédé, le langage *GooMod* est assez proche du diagramme d'activités UML sans les concepts relatifs au parallélisme (nœuds de bifurcation et d'union).

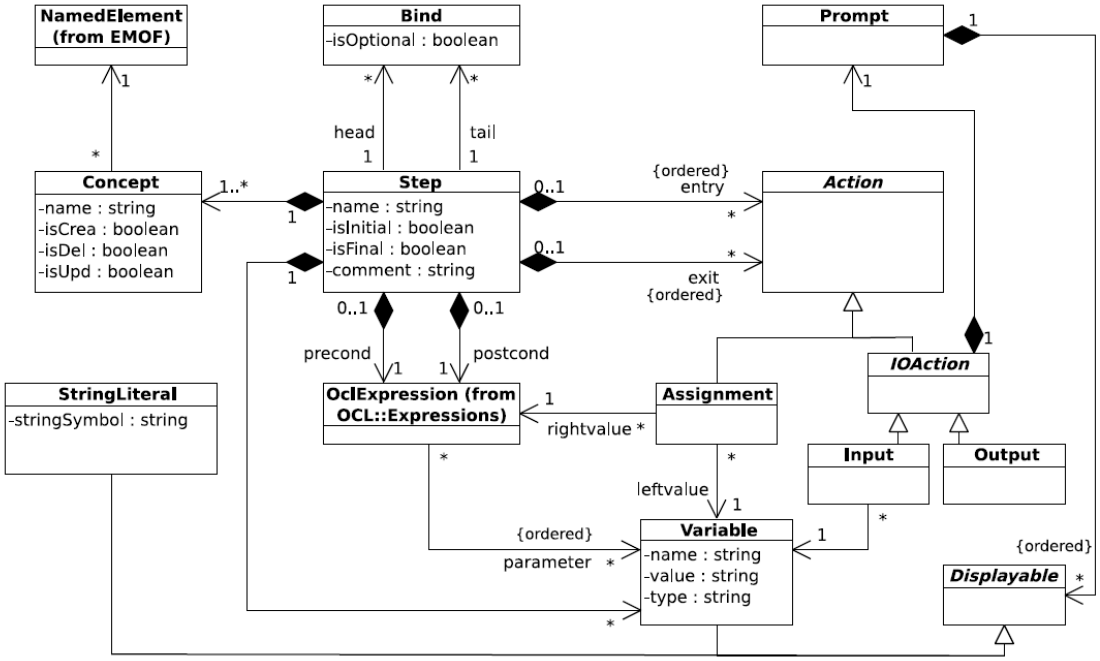


**Figure 6-3 La syntaxe abstraite du langage *GooMod***

Pour répondre au point 1, nous proposons de définir les procédés associés aux BPM en usant de deux types d'étape : les étapes libres et les étapes semi-automatiques.

Dans *une étape libre*, le développeur est seul à modifier le modèle mais son activité est cependant contrainte. Cette étape comporte trois phases :

- dans la première phase, une condition d'entrée sur le modèle détermine si on peut rentrer dans l'étape (pré-condition) ; le lancement de l'évaluation de cette condition est automatique dès que l'étape est atteignable ; Si l'évaluation est positive et que cette étape est la seule atteignable, on passe dans la seconde phase (point de décision automatique) ; si plusieurs étapes sont atteignables et présentent des pré-conditions positives, le développeur doit indiquer l'étape qu'il souhaite emprunter (point de décision non automatique) ;

- dans la seconde, le développeur modifie le modèle comme il l'entend mais en utilisant uniquement les concepts du langage qui ont été autorisés lors du déroulement de cette étape ;
- dans la troisième phase, le développeur demande l'évaluation d'une condition de sortie sur le modèle (post-condition) ; cette condition détermine si le développeur peut sortir ou non de l'étape. Si elle n'est pas vérifiée, on retourne dans la deuxième phase, sinon on lance l'évaluation des pré-conditions des étapes successeurs.

Dans la version actuelle du langage, les pré et post-conditions prennent la forme de contraintes *OCL* évaluées sur la syntaxe abstraite du modèle. C'est une réponse au point 3. Le choix d'*OCL* est temporaire. L'expérience acquise avec la description de motifs architecturaux a clairement montré qu'un tel langage, efficace pour certaines propriétés, se révélait insupportable pour d'autres. Cependant, dans une phase exploratoire, il permet la réutilisation d'outils de compilation et d'évaluation et limite ainsi les efforts de développement.

En réponse au point 2, il est possible de préciser les concepts utilisables dans une étape. Ces concepts sont un sous-ensemble des métaclasses non abstraites du métamodèle du langage ciblé par la BPM. Leur manipulation peut être réduite au cas par cas à un sous-ensemble de : création, modification, destruction, lecture.

Une étape semi-automatique se compose, elle aussi, de trois phases dont une est optionnelle (la deuxième). La première phase est identique à celle d'une étape libre. Elle correspond à l'évaluation d'une pré-condition. Les deux phases suivantes sont :

- deuxième phase (optionnelle) : le développeur doit renseigner de manière ordonnée une liste de variables ;
- troisième phase : une séquence d'actions entièrement prises en charge par l'outil est réalisée : ajout, retrait, modification d'éléments du modèle, calcul et affichage d'une valeur. Une fois ces actions réalisées, les pré-conditions des étapes successeurs sont évaluées.

Nous explorons trois approches complémentaires pour exprimer les deux phases précédentes :

- une description sous la forme d'une transformation de graphes avec une partie gauche incluant éventuellement des éléments indéterminés (ce sont les variables qui devront être renseignées par le développeur lors de la deuxième phase) et une partie droite qui décrit (classiquement) les modifications à réaliser ;
- un ensemble ordonné (éventuellement vide) de saisies de valeurs dans des variables, suivi d'un ensemble ordonné d'actions élémentaires MOF (ajout d'un élément, suppression, etc.) ;
- un ensemble ordonné (éventuellement vide) de saisies de valeurs dans des variables, suivi de l'évaluation d'une fonction OCL sur le modèle usant de ces variables, puis l'affichage de la valeur de cette fonction.

Ces trois approches correspondent à trois types d'étapes semi-automatiques. La première correspond à une substitution de motif multi-occurrences (partout ou il apparaît), la deuxième à une substitution de motif mono-occurrence, la troisième au calcul d'une métrique ou à la vérification du respect d'un style.

Une étape libre est interruptible sur la demande du développeur pour lancer une autre BPM (libre ou semi-automatique) sous réserve que la pré-condition de la BPM appelée soit satisfaite. A l'instar d'un appel de procédure, le contexte de la BPM appelante est sauvegardé. Il sera rétabli une fois terminée la BPM appelée. Les étapes semi-automatiques ne sont pas interruptibles.

### 6.1.6 Outillage et premières expérimentations

Pour évaluer l'aptitude du langage *GooMod* à décrire des BPM et étudier les problèmes que pose l'activation de BPM écrites avec ce langage, *Vincent Le Gloahec* a développé une plate-forme sous Eclipse. Elle comprend deux outils : le premier (*BP Definition Tool*) permet une description en *GooMod* (donc de niveau PIM) d'une BPM à l'aide d'un éditeur graphique, le second (*BP Activation Tool*) est capable d'imposer à tout éditeur généré depuis *Eclipse-GMF* le respect d'une BPM décrite en *GooMod*. Ce second outil est un outil « deux en un ». Il réalise implicitement une traduction PIM vers PSM (dans l'univers des éditeurs *GMF*) d'une BPM et il « interprète » la description PSM obtenue pour piloter l'éditeur cible.

Actuellement, c'est l'outil *BP Activation Tool* qui fixe le domaine de définition et de variation des BPM dans la mesure où, le langage *GooMod* ne permet pas une représentation explicite de ces deux aspects.

Les premiers tests montrent qu'avec cette plate-forme, il est d'ores et déjà possible de définir puis d'exécuter des BPM ayant un volet procédé dominant :

- une BPM décrivant une partie de la méthode *OMT* (les étapes portant sur l'élaboration d'un diagramme de classes) avec pour langage cible *UML* et pour outil cible l'éditeur *UML* de *Eclipse* ;
- une BPM qui portait sur l'élaboration de l'architecture d'une application web avec pour langage cible l'ADL *AlcoWeb* et pour outil cible *AlCoWeb-Builder*. Cette expérimentation fera l'objet d'une communication dans la conférence internationale QOSA'2010.
- une BPM qui portait sur la définition de BPM avec comme langage cible *GooMod* et outil cible le *BP Definition Tool*…

### 6.1.7 La suite de ce travail

Ce travail est à mi-parcours. Les premiers résultats sont encourageants mais il reste encore plusieurs points à régler à court terme.

Il faut essayer de documenter d'autres types de BPM : applications de patrons de conception, vérification du respect de règles de documentation et calcul de métriques.

L'expression des conditions d'entrée et de sortie d'étapes ne doit pas reposer uniquement sur OCL. Il serait utile de le compléter par un autre formalisme, idéalement graphique, qui permettrait par exemple de représenter des fragments de syntaxe abstraite (ou mieux concrète) de modèles représentant ce que l'on veut trouver dans le modèle. Les langages utilisés dans les grammaires de graphes ou les *model-snippets* (Ramos, et al., 2007) sont des pistes à explorer.

Il faut également clarifier la transition PIM vers PSM ; notamment identifier les contours précis de l'API idéale que devrait offrir tout éditeur pour permettre le pilotage de BPM. La transition PIM/PSM pourrait se faire alors en deux étapes : tout d'abord générer un code d'activation de la BPM selon cette API puis adapter le code obtenu pour un éditeur particulier en profitant d'un mapping entre cette API et l'API de l'éditeur cible. Des travaux actuellement en cours au sein de l'équipe *Triskell* à travers l'outil *Kompose* abordent le problème de la génération (semi) automatique de code glue entre API (ICSE'2010 à paraître)

Un travail important reste à entreprendre sur la prise en charge par le langage du domaine de définition et du domaine de variation des BPM. Pour le second point, il serait intéressant d'étudier le procédé d'une BPM sous l'angle d'un système dont l'architecture peut évoluer dynamiquement mais dans le respect de certains invariants. L'utilisation des techniques à base d'aspects développées au sein de l'équipe *Triskell* pour piloter et contrôler au niveau modèle l'adaptation dynamique d'une architecture réelle est une piste possible (Morin, et al., 2009).

A moyen terme, d'autres chantiers sont à entreprendre. Le premier d'entre eux porte sur la définition puis la réalisation d'une évaluation rigoureuse de l'approche. Il faudra procéder en plusieurs temps :

- Montrer la puissance du langage de modélisation proposé ;
- Montrer la faisabilité et quantifier l'intérêt du découplage PIM/PSM ;
- Quantifier l'impact de l'approche sur l'élévation du niveau de maturité des développeurs (vision individuelle) et des entreprises (vision systémique).

Pour le premier point on peut s'inspirer des critères énoncés par (Moody, 2009). En l'état j'entrevois une preuve s'intéressant à trois aspects.

- Expressivité : Le pouvoir d'expression du langage *GooMod* est-il suffisant pour représenter une classe « utile » de BPM (cela suppose bien sûr de définir rigoureusement les membres de cette classe « utile ») ?
- Efficacité : les BPM sont-elles décrites en *GooMod* d'une manière concise, précise, claire, etc. ?
- Adéquation : le langage proposé est-il adapté à ses utilisateurs ?

Le second chantier porte sur la définition de l'environnement technologique et méthodologique associé à l'introduction et à l'usage de BPM. Il faut répondre à de multiples questions portant sur la manière de gérer ces connaissances : archivage, recherche, droits d'accès, création de nouvelles BPM partant de BPM existantes, etc.

## 6.2 Une sémantique pour la composition de modèles

Ce travail sur la sémantique de la composition de modèles s'inscrit dans la lignée des travaux menés dans l'équipe *Triskell* autour des mécanismes de composition (essentiellement d'aspects). L'idée de départ est que les 4 opérateurs actuellement portés par l'équipe *Triskell Geko* (Morin, et al., 2008), *smartAdapter* (Morin, et al., 2008), *Kompose* et dans une moindre mesure le mécanisme d'aspect de *Kermeta* sont, à l'origine, des réponses apportées à une classe particulière de besoins. On a tenté par la suite d'étendre les fonctionnalités des ces outils pour répondre à d'autres classes de besoin qui apparaissaient au fil des projets portés par les uns et les autres. Approche par généralisation qui a le mérite de proposer une réponse pratique immédiate mais qui ne garantit en rien la pertinence des extensions qui ont été proposées. On se retrouvait donc face à des outils, véritables couteaux suisses, dont :

- on a du mal à cerner la couverture du domaine des besoins (peut-on tout faire, si non, quelles sont les classes de besoin adressées ?);
- on peut s'interroger sur les éventuels recouvrements fonctionnels des outils deux à deux ;
- on peut discuter la cohérence sémantique des fonctionnalités regroupées au sein d'un même outil, la disposition de certaines semblant affaiblir, voire remettre en cause des propriétés souhaitables pour d'autres.

J'ai donc décidé fin 2009 de réfléchir à une sémantique de la composition et à travers elle d'un moyen de classer les différentes formes de composition.

### 6.2.1 Pourquoi une sémantique de composition

Composition (Dictionnaire Robert) : « *Action, manière de former un tout en assemblant plusieurs parties, plusieurs éléments.* »

La composition de modèles est un concept flou à double titre :

- d'une part parce que la notion même de composition est très générale (que signifie formellement « former un tout en assemblant » ?) ;

- d'autre part parce que les parties considérées ici, les « éléments » sont des Modèles (quand j'use du sens le plus général de ce mot, je mets une majuscule). Or, on sait que les Modèles sont de différentes natures. Si l'on se restreint à la pile de l'OMG, on peut déjà identifier 4 sortes de Modèles selon la nature du système qu'ils représentent : métamétamodèles, métamodèles, modèles, instances de modèles.

Il est donc tentant de qualifier d'opérateur de composition toute fonction agissant sur plusieurs Modèles et produisant un autre Modèle. En conséquence de nombreuses fonctions sur les modèles de la littérature sont associées par leurs auteurs à la composition :

- la définition modulaire et progressive de Modèles (*merge* de UML) ;
- la gestion du développement séparé en permettant de fusionner des Modèles issus d'un développement collaboratif, gérant en particulier les incohérences selon certains protocoles paramétrables ou non ;
- la séparation des préoccupations en usant du tissage d'aspects au niveau des Modèles
- la comparaison (recherche de correspondances ou de points de divergence) et plus particulièrement la recherche de mappings et la production de différences entre Modèles ;
- la gestion des incohérences entre Modèles ;
- la génération de code partant de modèles représentant la structure et la dynamique d'une application ;
- la transformation ayant pour objet d'aller vers un palier d'abstraction plus riche si l'on considère qu'une transformation de ce type est une composition « dénaturée » avec l'un de ses arguments manquants (les choix relatifs à la projection vers un environnement plus concret, codés dans la transformation).

Cependant, il semble assez évident que les fonctions agissant sur des métamodèles ont sans doute d'autres objectifs (la « concaténation » de langages, leur intersection, leur union, etc.) et donc opèrent des manipulations au niveau syntaxiques différentes des fonctions opérant sur les modèles qui elles visent la fusion de domaines, la mise en commun des produits d'un développement séparé, la génération de code ou la recherche d'incohérences.

Une bibliographie sur la composition m'a permis d'identifier plusieurs dizaines de travaux connexes, relevant ou se réclamant de la composition. On trouvera une synthèse d'une partie de ces travaux dans (Jeanneret, et al., 2008). S'il n'est en rien préjudiciable de voir se développer une batterie de fonctions adaptées à tel ou tel besoin, on comprend cependant que ce flou rende difficile la compréhension du domaine et plus encore rende impossible toute démarche visant à structurer et à analyser l'espace des fonctions proposées. Comment dans ces conditions : positionner un outil dans cet espace et évaluer sa puissance, comparer deux outils, identifier les éventuels déficits ou la surabondance dans certains secteurs.

Il est donc nécessaire de décrire avec plus de rigueur ce que l'on entend par « composition de Modèles ». Deux voies sont possibles. La plus simple consiste à limiter ce terme à une classe suffisamment réduite de fonctions existantes dont les contours peuvent faire très facilement consensus. Cette approche exclue nombre de travaux plus qu'elle ne structure l'espace. C'est la voie qui fut prise par (Herrmann, et al., 2007) en limitant ce titre à un opérateur équivalent au *merge* d'UML. Son intérêt est de permettre l'identification de propriétés algébriques relativement fines pour cette classe de fonctions.

L'autre voie, que je défends, consiste au contraire à établir une classe plus vaste de fonctions. L'intérêt de cette approche est d'offrir un espace beaucoup plus important dans lequel établir des « structures » : relations d'équivalence (donc partitions) et/ou bases de fonctions, etc. Cet espace offre un cadre dans lequel il va être possible de comparer les travaux du domaine. La difficulté est alors de trouver une liste de propriétés communes minimales et consensuelles dont doivent faire état les fonctions membres de cet espace.

### 6.2.2 Quelques hypothèses et définitions de travail

On peut choisir de se placer dans un unique espace technique, par exemple celui de la pile à 4 niveaux de l'OMG dont la racine est MOF.

Soit **V un alphabet** *fini* de symboles recouvrant au minimum l'intégralité des éléments graphiques du langage UML (donc MOF).

Soit **O un ensemble d'opérateurs de positionnement** *fini* dans un espace 2D (à l'intérieur de, connecté à, etc.) suffisant pour construire partant depuis V tout modèle de l'espace technique

Soit **M l'ensemble de tous les modèles** *dénombrable* construit depuis V en usant des opérateurs de O.

Soit **S l'ensemble de tous les systèmes**. On suppose que $M \subset S$ pour permettre à tout modèle de jouer le rôle de système pour d'autres modèles.

**La syntaxe d'un langage de modélisation** L que nous noterons $Synt_L$, c'est-à-dire l'ensemble des modèles qui lui sont conformes, est défini de manière ensembliste (Figure 6-4) comme un sous-ensemble de M : $Synt_L \subseteq M$.

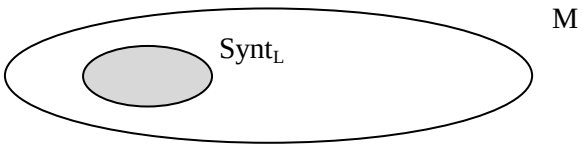


Figure 6-4 Syntaxe d'un langage

**La sémantique d'un langage de modélisation** L est définie par la donnée d'une fonction totale $Sem_L$.

$Sem_L : Synt_L \rightarrow P(S)$

A tout modèle $m \in Synt_L$, cette fonction associe le sous-ensemble de S des systèmes qui respectent les propriétés décrites dans m (Figure 6-5).

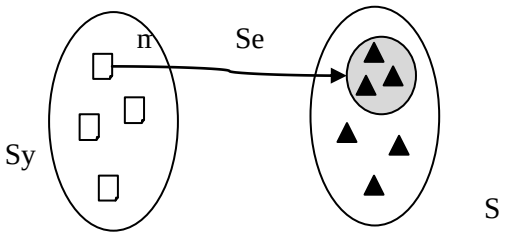


Figure 6-5 Sémantique d'un langage

On utilise ici une sémantique « relâchée » déjà utilisée par (Herrmann, et al., 2007). Cette sémantique est dite relâchée car on suppose que tout ce qui n'est pas dit par le modèle est potentiellement tolérable dans les systèmes ciblés. On ne pointe donc pas vers un unique système, mais un ensemble de systèmes. Cette sémantique est la version fonctionnelle de la relation binaire $\rho(s,m)$ introduite par Kühne (Kühne, 2006) qui détermine si un modèle m décrit de manière intentionnelle un système s.

Ce type de sémantique semble tout à fait approprié pour travailler sur la notion de composition qui sous-entend syntaxiquement une « mise en commun » au sein d'un unique modèle des propriétés en provenance de plusieurs modèles. Ce qui se traduit assez

grossièrement sur un plan syntaxique par une forme d'union des propriétés et sur un plan sémantique par une forme d'intersection. En effet, les propriétés décrites par un modèle $m_1$ et par un modèle $m_2$ se conjuguent dans un modèle résultat $m_r$. La sémantique de $m_r$ est alors bien l'intersection des systèmes respectant les propriétés de $m_1$ et de $m_2$.

**Le domaine sémantique d'un langage** L noté $DomSem_L$ est l'ensemble des systèmes qui respectent les propriétés exprimables depuis tous les modèles de L : $DomSem_L = \bigcup_{m \in Synt_L} Sem_L(m)$.

Il faut noter qu'un langage L correspond nécessairement à la donnée de deux éléments si l'on suppose S et M implicites.

$L=(Synt_L, Sem_L)$.

C'est-à-dire aux modèles qui lui sont conformes et à la manière d'interpréter ces modèles. La donnée de la seule syntaxe ne suffit pas à définir un langage.

L'ensemble de tous les langages sur M est bien sûr un ensemble non dénombrable (comme ensemble des parties d'un ensemble dénombrable).

### 6.2.3 Définir formellement la composition

Toute fonction sur l'ensemble M des Modèles a $n \in \aleph$ arguments et renvoie $m \in \aleph$ résultats. Sa signature est donc de la forme :

$$\prod_{i=1}^{n} A_i \rightarrow \prod_{j=1}^{m} B_j$$

Une telle signature doit vérifier : $\forall i \in [1,n]$, $A_i \subseteq M$ et $\forall j \in [1,m]$, $B_j \subseteq M$.

Une fonction prend donc un n-uplet de modèles et produit un m-uplet de modèles. Cet ensemble infini non dénombrable de fonctions est noté F.

La question est donc de déterminer quelles sont les restrictions à apporter à cette définition pour caractériser précisément le sous-ensemble des fonctions qui sont, à nos yeux, des compositions.

Ce sous-ensemble, nous le noterons C (Figure 6-6). Dans la mesure où le nombre des langages de modélisation est potentiellement infini, le nombre des fonctions de composition est lui aussi infini. Cet ensemble ne peut donc être décrit en extension. Il doit l'être *en compréhension*, c'est-à-dire qu'on le définit par une propriété Q caractéristique parmi les éléments de F : $C = \{c \in F | Q(c)\}$. Toute la difficulté est de cerner le prédicat Q.

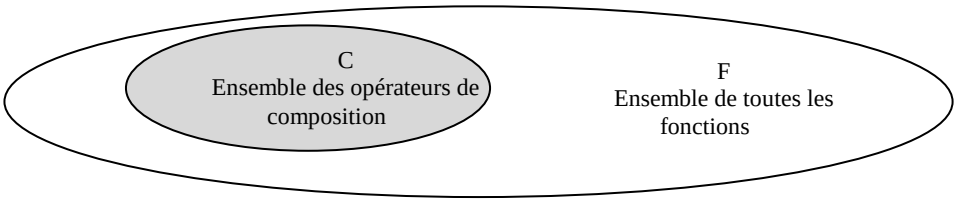


**Figure 6-6 Les opérateurs de comparaison**

Les questions à résoudre sont alors pour un opérateur de composition aussi sur le plan de sa signature que de ses propriétés algébriques :

1. Quel est le nombre de ses arguments (n=2 ?) ;
2. La nature des arguments : les ensembles $A_i$ sont-ils deux à deux disjoints, d'intersections non vide ou égaux pour certains ? L'égalité manifestant une composition en milieu homogène (de modèles conformes au même langage) ;

3. La fonction est-elle totale ou partielle ?
4. Le nombre des ensembles d'arrivée (m=1 ?) ;
5. Les $B_i$ sont-il identiques à l'un des Ai pour distinguer des compositions endogènes (le résultat est conforme au même langage que l'un des arguments) ou exogènes (le résultat est conforme à un langage différent de ceux des arguments) ?
6. Quelles sont les propriétés algébriques de l'opérateur : associativité (l'ordre de composition n'importe pas), commutativité (rôle symétrique des arguments), etc. ?
7. Quelle est la sémantique de l'opérateur, c'est-à-dire les liens qu'entretiennent la sémantique des modèles composés et la sémantique des résultats obtenus ?

Voilà un florilège minimal de questions pour lesquelles des réponses me semblent nécessaires.

### 6.2.4 Une première ébauche de formalisation

**Nombre des paramètres en entrée** : je propose de prendre $n \geq 2$ pour permettre la prise en compte de fonctions de composition réclamant plus de deux paramètres en entrée comme : le tissage (modèle de base, pointcut , advice), merge « intelligent » (modèle 1, modèle 2, mapping, politique de gestion de conflit), etc.

**Nature des paramètres en entrées** : pour qu'il y ait bien composition, il faut que deux (au moins) des n paramètres soient des parties de langage, c'est-à-dire des modèles avec une sémantique associée : $\exists(k,l)\in[1,n]^2$, $\exists L$ , $\exists L'$, ($A_k \subseteq Synt_L$ et $A_l \subseteq Synt_{L'}$). On n'impose pas que L=L', mais on ne l'interdit pas non plus. Comme L peut être différent de L', il faut, pour que la composition ait du sens, que ces deux parties de langages partagent une partie de leur domaine sémantique, soit formellement : $DomSem(L)\cap DomSem(L')\neq\emptyset$.

**Totalité** : Dans la mesure où nous considérons potentiellement des parties de langage dans le point précédent, on peut prendre comme hypothèse que la fonction est partielle sur les langages mais totale sur les parties de langage considérées.

**Nombre des résultats fournis** : je propose de prendre $m \geq 1$ pour permettre la prise en compte de fonctions de composition retournant plusieurs modèles : (modèle résultat, liste de conflits), etc.

**Nature des résultats fournis** : la composition doit-elle toujours atteindre l'un des langages de départ ? Je propose que non. La composition de modèles issus de un ou plusieurs langages doit pouvoir donner lieu à la création d'un modèle d'un autre langage. Par exemple diagramme à états+diagramme de classes+diagramme de séquences donne le code Java d'une classe. Par contre, si les langages diffèrent, le domaine sémantique du résultat doit avoir des points communs avec les modèles de départ.

- un (au moins) des m paramètres doit être un langage : $\exists t\in[1,m]$, $\exists L''$ , $B_t = Synt_{L''}$.
- ce langage doit partager une partie de son domaine sémantique avec les 2 langages en entrée, soit formellement : $DomSem(L'')\cap DomSem(L')\cap DomSem(L)\neq\emptyset$.

**Propriétés algébriques** : Rien pour le moment…

**Sémantique de l'opérateur de composition** : La composition doit permettre de produire en résultat un modèle dont la sémantique est un entrelacement de la sémantique des modèles en entrée.

- $\forall(m,m')\in A_k\times A_l$, $Comp(\ldots,m,\ldots,m',\ldots)=(\ldots,m'',\ldots)\Rightarrow Sem_L(m)\cap Sem_{L''}(m'')\neq\emptyset$ et $Sem_{L'}(m')\cap Sem_{L''}(m'')\neq\emptyset$

### 6.2.5 Quelle suite à ce travail

La proposition précédente est hautement perfectible. Je compte en particulier prendre un à un les opérateurs les plus significatifs de la littérature pour déterminer si oui ou non, ils respectent les contraintes exprimées et affiner en conséquence ma proposition. Je compte ensuite, sur la base des figures ensemblistes types de ces opérateurs sur le domaine sémantique S (intersection vide ou non, inclusion des sémantiques de modèles en entrée et en sortie de l'opérateur), profitant du domaine sémantique que j'ai choisi, déterminer une relation d'équivalence sur l'ensemble des opérateurs et par là une partition en classes d'équivalence de l'espace des opérateurs relevant de ma définition. Il sera dès lors possible d'identifier selon ces classes, l'abondance ou la pénurie d'opérateurs.

## *6.3 Quelques thèmes de recherche en attente d'un avenir*

Dans cette section, je présente quelques réflexions personnelles relatives à des thèmes sur lesquels je pourrais, à l'avenir, poursuivre mes travaux. Ces thèmes ne sont encore qu'au stade de l'intuition. Il s'agit ici d'évoquer d'autres pistes potentielles de recherche que les travaux présentés dans les deux sections précédentes.

### 6.3.1 Des opérateurs spécifiques pour la définition de DCL

Les langages dédiés nécessitent pour leur définition d'user de techniques à faible coût. L'une des voies à explorer pour diminuer ces coûts est d'appliquer l'un des principes clé du Génie Logiciel : la réutilisation. Construire la syntaxe et la sémantique de langages dédiés sur la base d'autres langages déjà définis (dédiés ou non). Pour ce faire, on peut distinguer deux types d'approche complémentaires :

- *réutilisation anticipée* : dès la conception des langages on prévoit de les découper en « parties » cohérentes et faiblement couplées les unes aux autres ; on peut ensuite archiver ses parties, éventuellement les généraliser, pour permettre leur réutilisation dans de nouvelles combinaisons pour définir de nouveaux langages. On retrouve l'exacte réplique des deux cycles pour et par la réutilisation du monde logiciel. Les artéfacts réutilisés étant ici des « bouts » de langages. Il faut pouvoir décrire les parties avec précision (ce qu'offrent et requièrent ces fragments de langage) à l'instar des composants logiciels.
- *réutilisation non anticipée* : les langages existent et sont ce qu'ils sont ; ils n'ont pas nécessairement été conçus dans l'esprit de servir à la définition d'autres langages ; il faut donc prévoir des opérateurs pour les adapter, les découper et les composer.

La première approche est ambitieuse et préférable mais de plus longue haleine et moins pragmatique que la seconde en l'état actuel des langages. Ma réflexion s'inscrit dans le cadre de la seconde approche et sur le volet syntaxique dans un premier temps.

Dans la théorie des langages textuels, il est habituel de construire la syntaxe d'un langage sur la base de la syntaxe d'un ou plusieurs langages déjà définis. On utilise pour cela des opérateurs hérités de la théorie des ensembles (*union*, *intersection*, *différence*, *complémentaire*, *différence symétrique*) et des opérateurs propres aux monoïdes qu'est $\Sigma^*$ (*concaténation* de deux langages, *étoile de Kleene* d'un langage, *homomorphisme* ou *substitution* appliqué à un langage). L'intérêt de ces opérateurs est double : non seulement ils ont une sémantique de niveau « utilisateur » (leur usage individuel a un intérêt pour les concepteurs de langages) mais en plus ils fournissent un système d'opérateurs en apparence « complet » pour construire des langages.

On trouve dans la littérature des traductions des opérateurs stables (c'est-à-dire maintenant le résultat dans la classe des langages de départ) dans le monde des grammaires formelles non contextuelles. Ces traductions impliquent des « transformations » (modification du vocabulaire des notions, des règles de production et du symbole initial) sur les grammaires formelles des langages opérandes pour déterminer la grammaire formelle du langage résultat.

Or, les métamodèles (conformes à des métalangages comme MOF, ECORE, KERMETA, etc.) partagent de nombreuses similitudes avec les grammaires formelles de type 2. Ces sont toutes deux des approches de définition par induction. Il est également assez facile de traduire toute grammaire BNF en un métamodèle équivalent (Kunert, 2008), c'est-dire-produisant les mêmes arbres syntaxiques : sous réserve d'ajouter un ordre explicite de « lecture » car la notion de séquence n'est pas naturelle dans les graphes (même si les graphes MOF ont une racine imposée). L'inverse n'est malheureusement pas toujours possible car les métalangages sont plus puissants (sur le plan langagier) que les grammaires formelles non contextuelles. Cependant leur ressemblance est peut-être suffisante pour tenter de projeter les opérateurs de la théorie des langages textuels vers les métamodèles.

L'idée est de profiter non seulement de ce catalogue d'opérateurs pour étudier sa pertinence et sa complétude dans le monde des métamodèles mais également de s'inspirer des traductions existantes pour proposer des mécanismes syntaxiques concrets donnant corps à ses opérateurs dans le monde des métamodèles.

Cette projection va être évidente pour certains opérateurs (union par exemple), mais beaucoup plus délicate pour d'autres (concaténation). La concaténation de deux langages textuels s'exprime très simplement au niveau des grammaires formelles par le simple ajout d'une nouvelle règle de production racine mettant en séquence les symboles de départ des deux langages opérandes. Cette simplicité est liée au fait que la concaténation textuelle ne « colle » deux mots qu'à un seul endroit (le bout de l'un au début de l'autre).

La concaténation au niveau des langages graphiques est plus subtile. Elle correspond au fait de pouvoir mettre sur un même schéma deux schémas initialement distincts mais qui doivent être cohérents et éventuellement connectés graphiquement. Par exemple ajouter à un langage représentant des classes la possibilité d'associer à chacune d'entre elles une machine à états. Les métamodèles correspondant à ces deux langages doivent être « associés » : il faut tirer des associations entre les concepts des deux métamodèles (par exemple entre les métaclasses *Classe* et *Machine à états* et entre *Transition* et *Méthode*). On note au passage pour cet exemple que ce n'est pas toujours une composition au sens habituel, c'est-à-dire pilotée par les points communs (des points de « matching »). Dans les métamodèles dont les mots sont des graphes, la concaténation suppose des collages sur plusieurs « sites ». A la différence des grammaires formelles, les sites doivent être explicitement donnés.

On voit qu'il est nécessaire de proposer des opérateurs de manipulation sur les graphes. Pour résoudre les difficultés, on doit sans doute pouvoir user d'opérateurs de la théorie des graphes et d'idée déjà exploitées dans les grammaires de graphes.

### 6.3.2 La gestion de l'évolution des langages

Ce thème fait suite à des travaux portant sur la transformation automatique de contraintes OCL après la modification du modèle sur lequel ces contraintes s'appliquent (thèse en cours de *Kahina Hassam*) et sur l'évaluation des mécanismes de vérification incrémentale de contraintes OCL (Master Recherche de *Mohammed Said Belaid*). Ces travaux montrent à quel point l'approche IDM pose, en contrepartie de sa puissance, de nouvelles difficultés lorsque les langages utilisés évoluent.

La puissance de l'approche IDM est liée à la réification explicite sous la forme de modèles de nombreux éléments qui étaient auparavant implicites ou absents : les données décrivant les langages de développement. Il est par exemple possible de générer automatiquement depuis ces données un éditeur capable d'aider à l'écriture de mots pour ce langage. Mais ces données, jusqu'alors immuables, deviennent du même coup modifiables.

On peut modifier la syntaxe abstraite non contextuelle d'un langage, l'une de ses syntaxes concrètes, sa sémantique statique, sa fonction sémantique ou son domaine sémantique. Une modification même mineure d'un seul de ces aspects peut avoir des conséquences importantes sur la plate-forme de développement (les outils générés plus ou moins automatiquement pour ce

langage de type éditeur, analyseur statique, simulateur) et sur les modèles existants de ce langage (souvent en très grand nombre et de grande taille). Des outils d'édition ou de transformation peuvent se retrouver incapables de manipuler les modèles d'un langage qu'ils sont sensés supporter. Des modèles anciens peuvent être invalidés et ne plus être conformes à leur langage d'origine.

Il est donc important sur la base des liens qu'entretiennent les différents artéfacts définissant un langage (métamodèles, sémantique statique, sémantique, mappings, etc.) et les artéfacts qui leur sont associés (transformations, modèles, etc.) :

- d'identifier de manière exhaustive l'ensemble des scénarii d'évolution possibles pour chaque artéfact définissant un langage et d'étudier leur impact (autres artéfacts concernés, type et gravité de l'impact) ;
- d'apporter des réponses aussi bien préventives que curatives pour limiter ou résoudre les problèmes que posent les scénarii.

Le concept de *type de modèle* va jouer un rôle fondamental pour le second point,fournissant un critère (syntaxique uniquement) permettant de décider de l'impact ou non d'une modification. Ainsi, sous réserve d'inférer ou de disposer du type de modèle associé à une contrainte OCL ou à une transformation, on peut déduire si respectivement, la contrainte est toujours évaluable, ou si la transformation est toujours exécutable.

### 6.3.3 Un nouvel OCL

Actuellement nombre de travaux de recherche font usage du langage OCL. Quelques-uns, comme nous l'avons fait pour documenter les motifs architecturaux, aménage la syntaxe ou la sémantique d'OCL pour mener à bien l'écriture de certains types de contraintes ou mettre en place certains contrôles. En particulier, il est fréquent que l'on veuille écrire des contraintes OCL dont on souhaite limiter le domaine d'évaluation dans l'espace ou le temps.

On peut souhaiter qu'une contrainte posée sur un modèle, ne soit évaluée que pour certaines instances de ce modèle et non sur toutes : par exemple pour contraindre la forme d'une famille de procédés écrits en SPEM sans pour autant vouloir restreindre le langage SPEM tout entier ou pour profiter de techniques d'évaluation incrémentale de contraintes OCL (Cabot, et al., 2006) de pouvoir limiter à certaines instances et non à toutes celles du `classifier` associé au contexte de la contrainte. On peut également vouloir qu'une contrainte soit temporairement rendue inactive à l'instar de ce qui se fait avec les contraintes Oracle (mots clés `enable`/`disable`) en fonction de l'état du modèle sujet à l'évaluation ou du processus de développement : par exemple pour un modèle en cours de construction ou de modification, pour un langage dont la syntaxe varie en fonction de l'étape courante du procédé de développement.

L'idée est, sur la base de ce qu'est une contrainte OCL, d'identifier des axes de généralisation, de proposer en conséquence des modifications à la syntaxe et à la sémantique d'OCL et enfin de montrer comment, avec ces amendements, on résout élégamment les problèmes cités plus haut et mieux encore on ouvre OCL à de nouveaux usages.

Une première réflexion rapide m'a conduit à envisager les axes de généralisation qui suivent.

- Généralisation de la zone porteuse de la contrainte : Avoir des contraintes OCL qui ne se posent plus sur un seul modèle mais sur une famille de modèles. Formellement parlant, la contrainte est associée à un type de modèle et se trouve applicable à tous les modèles conformes à ce type ou à l'un de ses sous-types.
- Généralisation du contexte de définition : une contrainte pourrait avoir pour contexte de définition un `classifier` d'un type de modèle qui concrètement rend effective cette contrainte dans tous les `classifiers` apparaissant dans un fragment du modèle conforme au type ou à l'un de ses sous-types. Il s'agit de rechercher tous les isomorphismes de sous-graphes (du type vers le modèle). Une même contrainte a ainsi du sens pour plusieurs `classifiers` d'un même modèle.

- Généralisation du niveau d'évaluation : en temps normal une contrainte définie à un niveau N est évalué sur le niveau N-1 de la pile d'instanciation, on pourrait réfléchir au sens et à l'utilité de contraintes traversant plus d'un niveau d'instanciation, c'est-à-dire évaluées à un niveau M (M<N) quelconque.
- Restriction temporelle : l'évaluation d'une contrainte ne pourrait être effective qu'à certains instants, instants par exemple déterminés par une condition de garde (portant sur l'état du modèle ou du procédé)
- Restriction en largeur du domaine d'évaluation : l'évaluation ne concerne qu'un sous ensemble (éventuellement un seul modèle instance) des modèles instances du modèle portant la contrainte.
- Restriction en profondeur du domaine d'évaluation : l'évaluation sur un modèle instance ne porte que sur un sous-ensemble des instances du `classifier` apparaissant en contexte de la contrainte.

Il est intéressant de noter que les 3 dernières généralisations répondent à des problèmes déjà évoqués car rencontrés dans la littérature.

# Chapitre 7 Conclusion

Mes travaux de recherche ont eu pour ambition de contribuer à l'amélioration du développement et de la maintenance des logiciels en introduisant dans les procédés des langages que je qualifierai de « support » que ce soit pour documenter les décisions architecturales ou pour décrire un composant recherché ou enfin capitaliser et exécuter des bonnes pratiques de modélisation. Ces langages permettent la représentation rigoureuse de connaissances importantes jusqu'alors perdues ou informelles. Dès lors, des outils peuvent exploiter ces connaissances pour automatiser certaines tâches et ainsi enrichir « l'éther » des développeurs : les aider à mieux utiliser les langages de modélisation et de programmation qui constituent le cœur de leur métier.

Le chapitre six a évoqué plusieurs perspectives pouvant tenir lieu de suite logique à mes travaux. Je compte poursuivre ma réflexion sur la documentation et l'exécution des bonnes pratiques de modélisation. Je suis convaincu de l'importance de ce travail. Un langage n'est véritablement compréhensible et efficace que lorsqu'il s'inscrit dans un contexte méthodologique explicite. Plus encore, je plaide en faveur d'une extension du doublet classique syntaxe et sémantique pour inclure systématiquement le volet procédé trop souvent externalisé, voire négligé. Les méta-outils doivent, eux aussi, évoluer pour intégrer ce troisième volet.

Mes activités de recherche épousent les tendances actuelles de la discipline. J'ai participé à la constitution de langages ouvrant de nouveaux points de vue. Toutefois l'explosion du nombre des langages pose maintenant des problèmes en termes de cohérence, de vérification et de génération de code. Il est nécessaire de proposer des mécanismes pour les résoudre. La composition de modèles m'apparaît comme un thème d'une grande importance pour l'avenir ; thème dans lequel je compte m'investir. La définition et l'évolution de langages dédiés par les développeurs eux-mêmes, dans le but d'accroître plus encore le niveau de réutilisation dans les projets, soulèvent également des difficultés. La proposition d'outils efficaces pour concevoir un langage partant d'autres langages et l'utilisation du mécanisme de typage de modèle pour réglementer l'évolution des langages sont deux aspects auxquels je souhaite également contribuer.

Les thèmes que je viens d'évoquer manifestent clairement mon souhait de poursuivre mes activités de recherche au cœur même de l'ingénierie dirigée par les modèles. Après avoir été, plusieurs années durant, utilisateur des technologies développées par les acteurs de ce domaine, je souhaite désormais apporter ma propre contribution à ce thème fondamental.

J'espère ainsi participer à la proposition d'outils, procédés, langages et concepts qui permettent à notre discipline de répondre aux attentes que ne cessent de nous poser les utilisateurs. Je compte pour cela, autant qu'il me sera possible, conserver une démarche scientifique à l'écoute des préoccupations industrielles et confronter mes propositions au contexte industriel : avec l'espoir, ce faisant, d'arriver à proposer des choses autant élégantes qu'utiles.

# Chapitre 8 Bibliographie

**Alvaro Alexandre, de Almeida Eduardo Santana et Meira Silvio Lemos** A Software Component Quality Model: A Preliminary Evaluation [Conférence] // 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO '06). - [s.l.] : IEEE Computer Society, 2006. - pp. 28-37.

**Ambler Scott W. et Jeffries Ron** Agile modeling: effective practices for extreme programming and the unified process [Livre]. - [s.l.] : John Wiley & Sons, Inc., 2002.

**Ancona D., Lagorio G. et Zucca E.** Jam-a smooth extension of Java with mixins. [Conférence] // ECOOP. - [s.l.] : Springer-Lecture Notes in Computer Science., 2000. - Vol. 1850. - pp. 145-178.

**Arnold Robert S.** Software Change Impact Analysis [Livre]. - [s.l.] : IEEE Computer Society Press, 1996.

**Atkinson Colin et Kühne Thomas** A tour of language customization concepts [Section du livre] // Advance in computers / éd. Zelkowitz Marvin. - [s.l.] : Elsevier, 2007. - Vol. 70.

**Bass Len, Clements Paul et Kazman Rick** Software Architecture in Practice [Livre]. - [s.l.] : Addison-Wesley Educational Publishers Inc, 2003.

**Bayley Ian et Zhu Hong** Formal specification of the variants and behavioural features of design patterns [Article] // J. Syst. Softw.. - [s.l.] : Elsevier Science Inc., 2010. - 2 : Vol. 83. - pp. 209-221.

**Beugnard Antoine [et al.]** Making Components Contract Aware [Article] // Computer. - [s.l.] : IEEE Computer Society Press, 1999. - 7 : Vol. 32. - pp. 38-45.

**Bible John, Rothermel Gregg et Rosenblum David S.** A comparative study of coarse- and fine-grained safe regression test-selection techniques [Article] // ACM Trans. Softw. Eng. Methodol.. - [s.l.] : ACM, 2001. - 2 : Vol. 10. - pp. 149-183.

**Biehl Matthias et Löwe Welf** Automated Architecture Consistency Checking for Model Driven Software Development [Conférence] // 5th International Conference on the Quality of Software Architectures (QoSA'09). - [s.l.] : Springer-Verlag, 2009. - pp. 36-51.

**Blanc Xavier [et al.]** Detecting model inconsistency through operation-based model construction [Conférence] // 30th international conference on Software engineering (ICSE'08). - [s.l.] : ACM, 2008. - pp. 511-520.

**Bosch J.** Software Architecture: The Next Step [Conférence] // EWSA 2004 : First european workshop on software architecture. - [s.l.] : Springer, 2004. - Vol. LNCS 3047. - pp. 194-199.

**Bracciali Andrea, Brogi Antonio et Canal Carlos** A formal approach to component adaptation [Article] // J. Syst. Softw.. - [s.l.] : Elsevier Science Inc., 2005. - 1 : Vol. 74. - pp. 45-54.

**Briand Lionel C., Daly John W. et Würst Jürgen** A Unified Framework for Cohesion Measurement in Object-Oriented Systems [Article] // Empirical Softw. Engg.. - [s.l.] : Kluwer Academic Publishers, 1998. - 1 : Vol. 3. - pp. 65-117.

**Briand Lionel C., Daly John W. et Wüst Jürgen K.** A Unified Framework for Coupling Measurement in Object-Oriented Systems [Article] // IEEE Trans. Softw. Eng.. - [s.l.] : IEEE Press, 1999. - 1 : Vol. 25. - pp. 91-121.

**Brooks F.P.** No Silver Bullet Essence and Accidents of Software Engineering [Revue] // Computer / éd. Computer IEEE. - April 1987. - 4 : Vol. 20. - pp. 10-19.

**Buckley Jim [et al.]** Towards a taxonomy of software change [Article] // J. Softw. Maint. Evol.. - [s.l.] : John Wiley and Sons, 2005. - 5 : Vol. 17. - pp. 309-332.

**Buschmann Frank [et al.]** Pattern-Oriented Software Architecture, Volume 1 : A System of Patterns [Livre]. - [s.l.] : John Wiley & Son Ltd, 1996.

**Cabot J. et Teniente E.** Incremental evaluation of OCL constraints [Conférence] // 18th Int. Conf. on Advanced Information Systems Engineering (CAiSE'06). - [s.l.] : Springer, 2006. - Vol. LNCS, vol. 4001. - pp. 81-95.

**Capilla Rafael, Nava Francisco et Duenas Juan C.** Modeling and Documenting the Evolution of Architectural Design Decisions [Conférence] // Second Workshop on SHAring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent (SHARK'07). - [s.l.] : IEEE Computer Society, 2007.

**Chapin N [et al.]** Types of software evolution and software maintenance [Revue] // Journal of Software Maintenance and Evolution. - [s.l.] : John wiley & Sons, 2001. - 3 : Vol. 13. - pp. 3-30.

**Cheung Leslie [et al.]** Early prediction of software component reliability [Conférence] // 30th international conference on Software engineering (ICSE'08). - [s.l.] : ACM, 2008. - pp. 111-120.

**Clements P. et Northrop L.** Software Product Lines : Practices and Patterns [Livre]. - Boston : MA: Addison Wesley, 2001.

**Cook Steve [et al.]** Domain-Specific Development With Visual Studio Dsl Tools [Livre]. - [s.l.] : Addison-Wesley Educational Publishers Inc, 2007.

**Corbi T. A.** Program understanding: Challenge for the 1990s [Article] // IBM Systems Journal. - 1989. - 2 : Vol. 28. - pp. 294-306.

**Cornelissen Bas [et al.]** A Systematic Survey of Program Comprehension through Dynamic Analysis [Article] // IEEE Trans. Softw. Eng.. - [s.l.] : IEEE Press, 2009. - 5 : Vol. 35. - pp. 684-702.

**Costagliola Gennaro, Deufemia Vincenzo et Polese Giuseppe** A framework for modeling and implementing visual notations with applications to software engineering [Article] // ACM Trans. Softw. Eng. Methodol.. - [s.l.] : ACM, 2004. - 4 : Vol. 13. - pp. 431-487.

**Crnkovic Ivica, Chaudron Michel et Larsson Stig** Component-Based Development Process and Component Lifecycle [Conférence] // International Conference on Software Engineering Advances (ICSEA'06). - [s.l.] : IEEE Computer Society, 2006.

**Cui Xiaofeng, Sun Yanchun et Mei Hong** Towards Automated Solution Synthesis and Rationale Capture in Decision-Centric Architecture Design [Conférence] // Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008). - [s.l.] : IEEE Computer Society, 2008. - pp. 221-230.

**Dashofy Eric M., Hoek André van der et Taylor Richard N.** A comprehensive approach for the development of modular software architecture description languages [Article] // ACM Trans. Softw. Eng. Methodol.. - [s.l.] : ACM, 2005. - 2 : Vol. 14. - pp. 199-245.

**de Boer Remco [et al.]** Architectural Knowledge: Getting to the Core [Conférence] // QoSA'07. - [s.l.] : Springer-Verlag, 2007. - Vol. LNCS 4880. - pp. 197-214.

**Defour O., Jézéquel J.M. et Plouzeau N.** Extra-functional contract support in components [Conférence] // 7th Int. Symp. on CBSE (CBSE'04). - [s.l.] : Springer, 2004. - Vol. LNCS 3054.

**Ducasse Stéphane [et al.]** Traits: A Mechanism for Fine-grained Reuse [Revue] // Transactions on Programming Languages and Systems (TOPLAS) . - [s.l.] : ACM, March 2006. - 2 : Vol. 28. - pp. 331-388.

**Egyed Alexander** Fixing Inconsistencies in UML Design Models [Conférence] // 29th international conference on Software Engineering (ICSE '07). - [s.l.] : ICSE '07the, 2007. - pp. 292-301.

**Engström Emelie, Runeson Per et Skoglund Mats** A systematic review on regression test selection techniques [Article] // Inf. Softw. Technol.. - [s.l.] : Butterworth-Heinemann, 2010. - 1 : Vol. 52. - pp. 14-30.

**Erdogmus Hakan** Architecture meets agility [Revue] // IEEE Software. - [s.l.] : IEEE Computer Society Press, September 2009. - 5 : Vol. 26. - pp. 2-4.

**Erlikh Len** Leveraging Legacy System Dollars for E-Business [Article] // IT Professional. - [s.l.] : IEEE Educational Activities Department, 2000. - 3 : Vol. 2. - pp. 17-23.

**Fleurquin Régis et Tibermacine Chouki** Une assistance pour l'évolution des logiciels à base de composants [Revue] // L'objet. - Paris : Hermes, Janvier 2007. - 1 : Vol. 13. - pp. 9-44.

**Fleurquin Régis, Tibermacine Chouki et Sadou salah** Le contrat d'évolution d'architectures. Un outil pour le maintien de propriétés non fonctionnelles [Conférence] // Conference sur les Languages et Modèles à Objets (LMO). - Bern, Suisse : Hermes, Mars, 2005. - pp. 209-222.

**Fragidis G. et Tarabanis K.** [Conférence] // International Conference on Management of Innovation and Technology. - [s.l.] : IEEE, 2006. - Vol. 1. - pp. 370-374 .

**Frolund Svend et Koistinen Jari** Quality of services specification in distributed object systems design [Conférence] // 4th conference on USENIX Conference on Object-Oriented Technologies and Systems (COOTS'98). - [s.l.] : USENIX Association, 1998.

**Gamma E. [et al.]** Design patterns: Elements of reusable object-oriented software [Livre]. - [s.l.] : Addison-Wesley, 1995.

**Garlan David, Monroe Robert T. et Wile David** Acme: architectural description of component-based systems [Section du livre] // Foundations of component-based systems. - [s.l.] : Cambridge University Press, 2000.

**George B. [et al.]** Un mécanisme de sélection pour les composants logiciels. [Article] // RTSI-L'Objet. - [s.l.] : Hermès, 2008. - 1 : Vol. 14. - pp. 139-163.

**George Bart, Fleurquin Régis et Sadou Salah** A Component Selection Framework for COTS Libraries [Conférence] // 11th International Symposium on Component-Based Software Engineering (CBSE'08). - [s.l.] : Springer-Verlag, 2008. - pp. 286-301.

**George Bart, Fleurquin Régis et Sadou Salah** A Methodological Approach to Choose Components in Development and Evolution Processes [Article] // Electron. Notes Theor. Comput. Sci.. - [s.l.] : Elsevier Science Publishers B. V., 2007. - Vol. 166. - pp. 27-46.

**Gonzalez-Perez Cesar et Henderson-Sellers Brian** Modelling software development methodologies: A conceptual foundation [Article] // J. Syst. Softw.. - [s.l.] : Elsevier Science Inc., 2007. - 11 : Vol. 80. - pp. 1778-1796.

**Gratton L. et Ghoshal S.** Beyond Best Practices », , vol. 46, n° 3, [Article] // Sloan Management Review. - [s.l.] : MIT, April 2005. - 3 : Vol. 46.

**Harel D. et Rumpe B.** Meaningful modeling: what's the semantics of "semantics"? [Article] // Computer . - [s.l.] : IEEE Computer Society, 2004. - 10 : Vol. 37.

**Henderson-Sellers Brian** Method engineering for OO systems development [Article] // Commun. ACM. - [s.l.] : ACM, 2003. - 10 : Vol. 46. - pp. 73-78.

**Herraiz Israel, Gonzalez-Barahona Jesus M. et Robles Gregorio** Determinism and evolution [Conférence] // MSR '08: Proceedings of the 2008 international working conference on Mining software repositories. - Leipzig, Germany : ACM, 2008. - pp. 1-10.

**Herrmann Christoph [et al.]** An Algebraic View on the Semantics of Model Composition [Conférence] // Third European Conference on Model Driven Architecture- Foundations and Applications (ECMDA-FA). - Haifa, Israel : Springer, 2007. - Vol. LNCS 4530/2007. - pp. 99-113.

**Herrmannsdoerfer Markus, Benz Sebastian et Juergens Elmar** Automatability of Coupled Evolution of Metamodels and Models in Practice [Conférence] // 11th international conference on Model Driven Engineering Languages and Systems (MODELS'08). - [s.l.] : Springer-Verlag, 2008.

**Humphrey Watts S.** Introduction to the personal software process [Livre]. - Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1997. - ISBN 0-201-54809-7.

**Jansen Anton et Bosch Jan** Software Architecture as a Set of Architectural Design Decisions [Conférence] // 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05). - [s.l.] : IEEE Computer Society, 2005. - pp. 109-120.

**Jansen Anton, Bosch Jan et Avgeriou Paris** Documenting after the fact: Recovering architectural design decisions [Article] // J. Syst. Softw.. - [s.l.] : Elsevier Science Inc., 2008. - 4 : Vol. 81. - pp. 536-557.

**Jeanneret Cédric, France Robert et Baudry Benoit** A reference process for model composition [Conférence] // AOSD workshop on Aspect-oriented modeling (AOM'08). - [s.l.] : ACM, 2008.

**Jézéquel Jean-Marc** Model Driven Design and Aspect Weaving [Revue] // Journal of Software and Systems Modeling (SoSyM). - [s.l.] : Springer, May 2008. - 2 : Vol. 7. - pp. 209-218.

**Jhala Ranjit et Majumdar Rupak** Software model checking [Article] // ACM Comput. Surv.. - [s.l.] : ACM, 2009. - 4 : Vol. 41.

**Kadri R. [et al.]** AlkoWeb : Un outil pour modéliser l'architecture des applications Web riches. [Conférence] // 2ème Conférence Francophone sur les Architectures Logicielles (CAL'2008). - [s.l.] : Cépaduès-Éditions, 2008. - Vol. RNTI-L-2. - pp. 107-118.

**Kadri Reda, Merciol Francois et Sadou Salah** CBSE in small and Medium-Sized Enterprise: Experience Report [Conférence] // 9th ACM SIGSOFT International Symposium on Component-Based Software Engineering (CBSE'06). - [s.l.] : Springer-Verlag, 2006. - Vol. LNCS 4063. - pp. 154-165.

**Kadri Reda, Tibermacine Chouki et Le Glohaec Vincent** Building the Presentation-Tier of Rich Web Applications with Hierarchical Components. [Conférence] // International Conference of Web Information Systems Engineering (WISE07). - [s.l.] : Springer, 2007. - Vol. LNCS 4831. - pp. 123-134.

**Kemerer Chris F. et Slaughter Sandra** An Empirical Approach to Studying Software Evolution [Article] // IEEE Trans. Softw. Eng.. - [s.l.] : IEEE Press, 1999. - 4 : Vol. 25. - pp. 493-509.

**Kim Dae-Kyoo et El Khawand Charbel** An approach to precisely specifying the problem domain of design patterns [Article] // J. Vis. Lang. Comput.. - [s.l.] : Academic Press, Inc., 2007. - 6 : Vol. 18. - pp. 560-591.

**Kitchenham Barbara A. [et al.]** Preliminary guidelines for empirical research in software engineering [Article] // IEEE Trans. Softw. Eng.. - [s.l.] : IEEE Press, 2002. - 8 : Vol. 28. - pp. 721-734.

**Kruchten P.B.** The 4+1 View Model of Architecture [Article] // IEEE Softw.. - [s.l.] : IEEE Computer Society Press, 1995. - 6 : Vol. 12. - pp. 42-50.

**Kruchten Philippe, Capilla Rafael et Duenas Juan Carlos** The Decision View's Role in Software Architecture Practice [Article] // IEEE Softw.. - [s.l.] : IEEE Computer Society Press, 2009. - 2 : Vol. 26. - pp. 36-42.

**Krueger Charles W.** Software reuse [Article] // ACM Comput. Surv.. - [s.l.] : ACM, 1992. - 2 : Vol. 24.

**Kühne Thomas** Matters of (Meta-) Modeling [Revue] // Software and Systems Modeling. - [s.l.] : Springer, Décembre 2006. - 4 : Vol. 5. - pp. 369-385.

**Kunert Andreas** Semi-automatic Generation of Metamodels and Models From Grammars and Programs [Article] // Electron. Notes Theor. Comput. Sci.. - [s.l.] : Elsevier Science Publishers B. V., 2008. - Vol. 211. - pp. 111-119.

**Land Rikard [et al.]** COTS Selection Best Practices in Literature and in Industry [Conférence] // 10th international conference on Software Reuse (ICSR '08). - [s.l.] : Springer-Verlag, 2008. - pp. 100-111.

**Lau Kung-Kiu et Wang Zheng** Software Component Model [Article] // IEEE Trans. Softw. Eng.. - [s.l.] : IEEE Press, 2007. - 10 : Vol. 33. - pp. 709-724.

**Lehman M M. [et al.]** Metrics and Laws of Software Evolution - The Nineties View [Conférence] // METRICS '97: Proceedings of the 4th International Symposium on Software Metrics. - [s.l.] : IEEE Computer Society, 1997.

**Lehman MM et Belady LA** Program Evolution - Processes of Software Change [Livre]. - London : Academic Press, 1985.

**Li Zheng, Harman Mark et Hierons Robert M.** Search Algorithms for Regression Test Case Prioritization [Article] // IEEE Trans. Softw. Eng.. - [s.l.] : IEEE Press, 2007. - 4 : Vol. 33. - pp. 225-237.

**Lientz BP et Swanson EB** Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organization [Livre]. - [s.l.] : Addison-Wesley, 1980.

**Medvidovic Nenad et Taylor Richard N.** A Classification and Comparison Framework for Software Architecture Description Languages [Article] // IEEE Trans. Softw. Eng.. - [s.l.] : IEEE Press, 2000. - 1 : Vol. 26. - pp. 70-93.

**Meek Brian** The static semantics file [Article] // SIGPLAN Not.. - [s.l.] : ACM, 1990. - 4 : Vol. 25. - pp. 33-42.

**Mens Tom et Tourwé Tom** A Survey of Software Refactoring [Article] // IEEE Trans. Softw. Eng.. - [s.l.] : IEEE Press, 2004. - 2 : Vol. 30. - pp. 126-139.

**Mernik Marjan, Heering Jan et Sloane Anthony M.** When and how to develop domain-specific languages [Revue] // ACM Computing Survey. - New-York : ACM, 2005. - 4 : Vol. 37. - pp. 316-344.

**Meyer Bertrand** Object-Oriented Software Construction [Livre] / éd. Hall Prentice. - 2000. - p. 1296. - 978-0136291558.

**Mili Rym, Mili Ali et Mittermeir Roland T.** Storing and Retrieving Software Components: A Refinement Based System [Article] // IEEE Trans. Softw. Eng.. - [s.l.] : IEEE Press, 1997. - 7 : Vol. 23. - pp. 445-460.

**Minas Mark** Syntax Definition with Graphs [Article] // Electron. Notes Theor. Comput. Sci.. - [s.l.] : Elsevier Science Publishers B. V., 2006. - 1 : Vol. 148. - pp. 19-40.

**Mohagheghi Parastoo et Conradi Reidar** An empirical investigation of software reuse benefits in a large telecom product [Article] // ACM Trans. Softw. Eng. Methodol.. - [s.l.] : ACM, 2008. - 3 : Vol. 17.

**Moody Daniel** The ``Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering [Article] // IEEE Trans. Softw. Eng.. - [s.l.] : IEEE Press, 2009. - 6 : Vol. 35. - pp. 756-779.

**Morel Brandon et Alexander Perry** SPARTACAS Automating Component Reuse and Adaptation [Article] // IEEE Trans. Softw. Eng.. - [s.l.] : IEEE Press, 2004. - 9 : Vol. 30. - pp. 587-600.

**Morin Brice [et al.]** A generic weaver for supporting product lines [Conférence] // 13th international workshop on Early Aspects (EA'08). - [s.l.] : ACM, 2008. - pp. 11-18.

**Morin Brice [et al.]** Managing Variability Complexity in Aspect-Oriented Modeling [Conférence] // 11th international conference on Model Driven Engineering Languages and Systems (MODELS'O8). - [s.l.] : Springer-Verlag, 2008. - pp. 797-812.

**Morin Brice [et al.]** Taming Dynamically Adaptive Systems using models and aspects [Conférence] // 31st International Conference on Software Engineering (ICSE'09). - [s.l.] : IEEE Computer Society, 2009. - pp. 122-132.

**Mosses Peter D.** Formal Semantics of Programming Languages [Article] // Electron. Notes Theor. Comput. Sci.. - [s.l.] : Elsevier Science Publishers B. V., 2006. - 1 : Vol. 148. - pp. 41-73.

**Paige Richard F., Brooke Phillip J. et Ostroff Jonathan S.** Metamodel-based model conformance and multiview consistency checking [Article] // ACM Trans. Softw. Eng. Methodol.. - [s.l.] : ACM, 2007. - 3 : Vol. 16.

**Parnas D.L.** On the criteria to be used in decomposing systems into modules [Revue] // Communications of the ACM / éd. ACM. - [s.l.] : ACM press, December 1972. - 12 : Vol. 15. - pp. 1053-1058.

**Parnas David Lorge** Software aging [Conférence] // ICSE '94: Proceedings of the 16th international conference on Software engineering. - Sorrento : IEEE Computer Society Press, 1994. - pp. 279-287.

**Perry Dewayne E. et Wolf Alexander L.** Foundations for the study of software architecture [Revue] // SIGSOFT Softw. Eng. Notes. - [s.l.] : ACM, 1992. - 4 : Vol. 17. - pp. 40-52.

**Pierce Benjamin C.** Types and Programming Languages [Livre]. - [s.l.] : MIT Press, 2002. - p. 645.

**Rajlich V. T. et Bennett Keith H.** A Staged Model for the Software Life Cycle [Article] // Computer. - [s.l.] : A Staged Model for the Software Life Cycle, 2000. - 7 : Vol. 33. - pp. 66-71.

**Ramos Rodrigo, Barais Olivier et Jézéquel Jean-Marc** Matching Model-Snippets [Conférence] // Model Driven Engineering Languages and Systems (MODELS'2007). - [s.l.] : Springer, 2007. - pp. 121-135.

**Ramsin Raman et Paige Richard F.** Process-centered review of object oriented software development methodologies [Article] // ACM Comput. Surv.. - [s.l.] : ACM, 2008. - 1 : Vol. 40. - pp. 1-89.

**Rolland Colette** Method Engineering: State-of-the-Art Survey and Research Proposal [Conférence] // conference on New Trends in Software Methodologies, Tools and Techniques. - [s.l.] : IOS Press, 2009. - pp. 3-21.

**Rothermel Gregg [et al.]** On test suite composition and cost-effective regression testing [Article] // ACM Trans. Softw. Eng. Methodol.. - [s.l.] : ACM, 2004. - 3 : Vol. 13.

**Rothermel Gregg, Untch Roland J. et Chu Chengyun** Prioritizing Test Cases For Regression Testing [Article] // IEEE Trans. Softw. Eng.. - [s.l.] : IEEE Press, 2001. - 10 : Vol. 27. - pp. 929-948.

**Schobbens Pierre-Yves, Heymans Patrick et Trigaux Jean-Christophe** Feature Diagrams: A Survey and a Formal Semantics [Conférence] // 14th IEEE International Requirements Engineering Conference (RE'06). - [s.l.] : IEEE Computer Society, 2006. - pp. 136-145.

**Shaw Mary et Clements Paul** The Golden Age of Software Architecture [Article] // IEEE Softw.. - [s.l.] : IEEE Computer Society Press, 2006. - 2 : Vol. 23. - pp. 31-39.

**Shull F. et Turner R.** An empirical approach to best practice identification and selection: the US Department of Defense acquisition best practices clearinghouse [Conférence] // International Symposium on Empirical Software Engineering. - [s.l.] : IEEE, 2005.

**Stewart Thomas A.** The Wealth of Knowledge: Intellectual Capital and the Twenty-first Century Organization [Livre]. - [s.l.] : Doubleday, 2001.

**Szyperski Clemens** Component Software: Beyond Object-Oriented Programming [Livre]. - [s.l.] : Addison-Wesley, 2002. - p. 624.

**Tang A. [et al.]** A comparative study of architecture knowledge management tools [Article] // Journal of Systems and Software. - 2010. - 3 : Vol. 83. - pp. 352-370.

**Tang Antony [et al.]** A survey of architecture design rationale [Article] // J. Syst. Softw.. - [s.l.] : Elsevier Science Inc., 2006. - 12 : Vol. 79. - pp. 1792-1804.

**Team CMMI Product** CMMI for Development, Version 1.2 [En ligne] // Software Engineering Institute. - August 2006. - 13 October 2009. - http://www.sei.cmu.edu/reports/06tr008.pdf.

**Tibermacine Chouki, Fleurquin Régis and Sadou Salah** Nfrs-aware architectural evolution of component-based software [Conference] // 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05) / ed. ACM. - Long Beach, California, USA : [s.n.], November, 2005a. - pp. 388-391.

**Tibermacine Chouki, Fleurquin Régis et Sadou Salah** A Family of Languages for Architecture Constraint Specification [Revue] // Journal of Systems and Software. - [s.l.] : Elsevier, 2010.

**Tibermacine Chouki, Fleurquin Régis et Sadou Salah** On-Demand Quality-Oriented Assistance in Component-based Software Evolution. [Conférence] // 9th ACM SIGSOFT International Symposium on Component-Based Software Engineering (CBSE'06). - Västerås : Springer-Verlag, 2006. - Vol. LNCS 4063. - pp. 294-309.

**Tibermacine Chouki, Fleurquin Régis et Sadou Salah** Preserving Architectural Choices throughout the Component-based Software Development Process [Conférence] // 5th Working IEEE/IFIP Conference on Software Architecture (WICSA) / éd. Society IEEE Computer. - Pittsburgh, Pennsylvania, USA : [s.n.], November, 2005b. - pp. 121-130.

**Tonella Paolo [et al.]** Empirical studies in reverse engineering: state of the art and future trends [Article] // Empirical Softw. Eng.. - [s.l.] : Kluwer Academic Publishers, 2007. - 5 : Vol. 12. - pp. 551-571.

**Tyree Jeff et Akerman Art** Architecture Decisions: Demystifying Architecture [Article] // IEEE Softw.. - [s.l.] : IEEE Computer Society Press, 2005. - 2 : Vol. 22. - pp. 19-27.

**Tyree Jeff et Akerman Art** Architecture Decisions: Demystifying Architecture [Article] // IEEE Softw.. - [s.l.] : IEEE Computer Society Press, 2005. - 2 : Vol. 22. - pp. 19-27.

**van Ommering Rob [et al.]** The Koala Component Model for Consumer Electronics Software [Article] // Computer. - [s.l.] : IEEE Computer Society Press, 2000. - 3 : Vol. 33. - pp. 78-85.

**Woodcock Jim [et al.]** Formal methods: Practice and experience [Article] // ACM Comput. Surv.. - [s.l.] : ACM, 2009. - 4 : Vol. 41. - pp. 1-36.

**Yi Tong, Wu Fangjun et Gan Chengzhi** A comparison of metrics for UML class diagrams [Article] // SIGSOFT Softw. Eng. Notes. - [s.l.] : ACM, 2004. - 5 : Vol. 29.

**Zaremski Amy Moormann et Wing Jeannette M.** Specification matching of software components [Article] // ACM Trans. Softw. Eng. Methodol.. - [s.l.] : ACM, 1997. - 4 : Vol. 6. - pp. 333-369.

**Zhang Yingzhou et Xu Baowen** A survey of semantic description frameworks for programming languages [Article] // SIGPLAN Not.. - [s.l.] : ACM, 2004. - 3 : Vol. 39. - pp. 14-30.

**Zhong Hao, Zhang Lu et Mei Hong** An experimental study of four typical test suite reduction techniques [Article] // Inf. Softw. Technol.. - 2008. - 6 : Vol. 50. - pp. 534-546.

**Zschaler Steffen** Formal specification of non-functional properties of component-based software systems [Article] // Software and Systems Modeling (SOSYM). - [s.l.] : Springer, February 2009.