République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université Mentouri Constantine
Faculté des Sciences de l'Ingénieur
Département d'informatique

N° ordre :
Série :

Thèse

pour obtenir le diplôme de
Doctorat en sciences en Informatique

Titre

Utilisation des design Patterns et des méthodes formelles dans le développement des systèmes d'information

Présentée par :

ZITOUNI Abdelhafid

Soutenue le 02/12/2008, devant le Jury :

M. BENMOHAMED	Professeur à l'université Mentouri - Constantine	Président du Jury
M. BOUFAIDA	Professeur à l'université Mentouri - Constantine	Directeur de thèse
M. AHMED NACER	Professeur à l'USTHB - Alger	Examinateur
Z. BOUFAIDA	Professeur à l'université Mentouri - Constantine	Examinatrice
D. MESLATI	M.C. à l'université Badji Mokhtar- Annaba	Examinateur

Remerciements

Enfin, l'heure est venue de rédiger ces fameux remerciements, but ultime de chaque doctorant puisqu'il s'agit bel et bien de l'événement qui marque la fin de la thèse.

En premier lieu, je tiens à exprimer ma profonde reconnaissance au Professeur *M. BOUFAIDA* qui m'a accueilli au laboratoire LIRE, m'a formé à la recherche et a assuré avec beaucoup de gentillesse et d'attention l'encadrement de ma thèse de doctorat. Nombre des travaux réalisés ici sont le fruit de ses conseils et de sa patience. J'ai pu à plusieurs reprises apprécier et profiter de sa haute compétence scientifique et pédagogique que tout le monde lui connaît. Qu'il trouve dans ces lignes si courtes, la sincère expression de ma gratitude et le témoignage de ma reconnaissance.

Je tiens à remercier profondément le Professeur *M. Benmohamed* qui a su, malgré ses lourdes responsabilités et son emploi du temps extrêmement chargé, m'apporter son soutien et me faire l'honneur de présider le jury de cette thèse.

C'est avec un grand plaisir que je compte le Professeur *Mohamed. AHMED NACER* de l'USTHB parmi les membres de ce jury. Qu'il trouve l'expression de ma sincère reconnaissance pour l'attention avec laquelle il a examiné mes travaux, et pour être venu de si loin pour participer à ce jury.

Je suis également tout à fait honorée de la présence à ce jury du Professeur *Mme Zizette BOUFAIDA*, et je la remercie vivement pour l'intérêt qu'elle a accordé à mon travail, pour son aide, surtout moral dans les moments difficiles, et d'avoir accepté de participer à ce jury.
(je vous remercie beaucoup et permettez moi de vous dire ceci: Merci ma SŒUR, sincèrement vous m'avez beaucoup aidé).

J'adresse mes sincères remerciements au docteur *Djamel Meslati*, Maitre de Conférence à l'université Badji Mokhtar de Annaba, pour m'avoir fait l'honneur d'étudier mes travaux de thèse et de participer à ce jury en qualité d'examinateur.

Merci à tous mes collègues (anciens et présents) du département d'Informatique pour leur soutien et leurs précieux encouragements, surtout Ilhem Labed, Nadia Chellali, Naima Benabdelaziz , Mohammed Chihoub, Hacene Sebih, Nasro Zarour, Mohamed Gharzouli, Djamel Benmerzoug et les collègues du labo LIRE, surtout ceux de l'équipe SIBC……...

Je n'aurai garde d'oublier de remercier tous ceux que j'ai eu le plaisir de côtoyer durant mes stages à Paris et Lille. En particulier, Le Professeur Lionel SEINTURIER, avec qui j'ai beaucoup appris.
Je le remercie de l'intérêt qu'il a manifesté pour mes travaux, de l'accueil qu'il m'a réservé au sein de son laboratoire et pour la coopération agréable et fructueuse qu'on a eu ensemble.

Enfin, ce n'est pas sans émotion que j'évoque ici le soutien moral permanent des membres de ma famille. Je ne peux m'empêcher de les citer tous : *ma famille*, ma belle famille. Qu'il me pardonne de ne pas savoir traduire par les mots appropriés l'ampleur de mes sentiments à leur égard ; et qu'ils sachent que c'est avec un grand plaisir que je leur dédie à tous, cette thèse.

Résumé

Cette thèse s'inscrit dans le cadre général de l'ingénierie de systèmes d'information. Elle concerne plus précisément, l'utilisation et l'intégration des design patterns (patrons de conception) et les méthodes formelles, dans le but de spécifier, modéliser et de développer les systèmes d'information.

Dans ce but, nous proposons une approche formelle basée contrat, permettant au développeur de logiciels de modéliser et intégrer des composants pendant la face de conception. Pour ce faire, nous décrivons une méthode de spécification de patterns, intégrant deux paradigmes, les méthodes UML (semi- formelle) et le langage de spécification formelle Lotos.
Nous fournissons une spécification formelle basée sur des constructeurs de la logique de description. Avec ces constructeurs, on peut facilement représenter, détecter, instancier, évoluer et composer les patrons de conception.

Nous proposons aussi, un langage de description d'architecture ADL-LOTOS, permettant la description d'architectures logicielles abstraites. Le langage ADL-LOTOS permet la spécification des aspects statiques et dynamiques des systèmes de façon unifiée. Il est expressif grâce à une syntaxe simple. Il favorise la définition de composants avec différents types de communication (synchrone et/ou asynchrone) à un haut niveau d'abstraction. La description du langage est illustrée par quelques exemples.

Mots clés : Design Pattern, méthodes formelle, LOTOS, Systèmes d'information, langages de description.

Abstract

This thesis falls within the global framework of information systems engineering. More precisely, it concerns the use and integration of design patterns coupled with formal methods. Toward this direction, we provide a contract formal approach, enabling developers to modelize components at the conception stage.

In order to achieve this goal, we describe a method of specification of patterns, integrating the two paradigms, the UML method (semi formal) and the LOTOS language of formal specification.

In addition, we propose a language for architecture description ADL-LOTOS. The ADL-LOTOS language allows one to specify system static and dynamic aspects in a unified way. It is expressive, thanks to its simple syntax. Beside, it promotes the components definition with various way of communication (synchronous and/or asynchronous) at a high level of abstraction. The description of the language is illustrated by some examples and a case study.

Key words: Design Pattern, formal methods, LOTOS, Information Systems, description languages.

ملخص

تندرج هذه الأطروحة ضمن النظام العام لاستعمال الطرق الرياضية لهندسة الأنظمة المعلوماتية. وتهتم هذه الأطروحة بصفة خاصة بإدماج و استعمال موصفات النماذج بواسطة الطرق نصف و كاملة القطعية بواسطة LOTOS و UML.

في هذا الاتجاه وفرنا مقاربة قطعية على أساس العقد كما إقترحنا لغة وصف LOTOS-ADL. و تعد هذه اللغة مميزة بنحو بسيط يحتوي على رسوم بيانية. إضافة إلى ذلك فان هذه اللغة تسمح بوصف مركبات الأنظمة ومختلف الاتصالات فيما بينها على درجة عالية من التجريد.

كلمات مفتاح:

موصفات النماذج، الطرق القطعية، **LOTOS** ، أنظمة المعلوماتية، لغة وصف.

Table des Matières

Introduction

Présentation et contexte de la thèse

Le domaine de l'ingénierie des systèmes d'information a toujours été un secteur très demandeur en techniques et en méthodes nouvelles permettant d'améliorer aussi bien la qualité des produits que la performance des processus utilisés pour les élaborer. Ces besoins ont fait émerger dans le passé des méthodes et des outils innovants qui ont été largement adoptés et utilisés. Cette pratique se traduit aujourd'hui par l'existence d'un savoir-faire important en matière d'ingénierie des systèmes d'information permettant ainsi d'envisager une nouvelle approche de développement basée sur la réutilisation de composants. Il est maintenant possible d'envisager une approche de développement basée sur la réutilisation de composants existants et éprouvés.

Une telle approche doit permettre de réduire le temps de conception des systèmes d'information, d'en améliorer la qualité et d'en faciliter la maintenance. Afin de rendre systématique le développement par réutilisation, nous pensons qu'il est essentiel de fournir des outils, méthodes et techniques permettant de développer des systèmes de réutilisation en vue de concevoir des systèmes d'information par réutilisation de composants.

Cet aspect « composant réutilisable » est apparu dans un premier temps dans les phases de conception détaillée et d'implantation. La réutilisation dans les phases amont du processus de développement des SI est quant à elle, apparue avec les approches à base de patrons (ou patterns) [Gam et al. 95] dont nous rappelons brièvement les principaux aspects : Un patron est généralement défini comme une **solution** à un **problème** dans un **contexte**. La solution d'un patron est constituée d'un petit nombre de classes représentées de manière semi-formelle par des diagrammes type UML.

Pour résumer, les patrons capitalisent un savoir-faire consensuel sur un domaine donné. Dans le cadre de la conception des systèmes d'information, tout l'enjeu est d'adapter ce savoir-faire à un contexte applicatif afin de pouvoir passer de la solution générale à son imitation selon des contraintes établies.

Plus récemment, un intérêt croissant pour les patrons s'est développé au sein de la communauté du génie logiciel et en particulier parmi les personnes s'intéressant aux approches orientées objet et à la réutilisation et les méthodes formelles.

En effet, les spécifications formelles s'imposent progressivement comme une condition essentielle a un bon développement logiciel. Leurs avantages principaux sont la rigueur dans la spécification du problème, la preuve de propriétés et une conception méthodique, automatisable en partie.

L'ajout de formalisme aux méthodes d'analyse et conception à objets permet une meilleure structuration et une organisation plus naturelle des spécifications. D'un autre coté, les objets seront plus sûrs et la conception plus rigoureuse.

Cet intérêt pour la formalisation des patrons a pris un essor grandissant avec pour objectif d'améliorer la programmation de logiciels. L'évolution a ensuite consisté à appliquer le principe des patrons à des étapes de plus en plus en amont de la programmation.

Malgré cette évolution vers les phases les plus en amont du cycle de développement des systèmes, il n'existe pas, à notre connaissance, de travaux sur des patrons qui portent sur la définition, l'organisation et le développement de systèmes d'information.

Pour notre part, nous nous intéressons à l'ingénierie de systèmes d'information basée sur la réutilisation. Nous nous inscrivons dans une approche reposant sur le paradigme «*composant conceptuel*». Un pas, décisif dans ce domaine, est donc d'introduire des aspects formels dans le cycle de développement des systèmes d'information.

L'objectif de cette thèse consiste à utiliser et à intégrer les patrons de conception (composants conceptuels) et les méthodes formelles, dans le but *de spécifier, modéliser et de développer les systèmes d'informations*.

Notre objectif est de proposer un environnement de développement des systèmes d'information, intégré pour lui ajouter les assistants logiciels permettant :

i) de représenter un catalogue de patrons ;
ii) d'instancier les patrons de ce catalogue pendant le développement ;
iii) de reconnaître ces patrons dans du code source existant.

Problématique et motivation

Notre problématique s'inscrit dans le contexte du développement de systèmes d'information par réutilisation. Le principal problème rencontré lors de la réutilisation est de retrouver les composants conceptuels les plus pertinents possibles pour élaborer un système d'information particulier.

La maîtrise et l'exploitation de ces systèmes posent dès lors le défi de la construction d'infrastructures logicielles adaptées et sûres. Même si la plupart des difficultés liées à la construction de ce type d'application ne sont pas encore complètement résolues, de nombreuses avancées permettent actuellement de créer efficacement des applications réparties à grande échelle.

Dans ce sens, l'arrivée du paradigme de composant CBSE, pour (Component Based Software Engineering [Szy02]) et sa prise en compte à tous les niveaux du cycle de développement permettent une meilleure réutilisation, et une expression simple des dépendances entre modules logiciels. Le composant est généralement défini comme entité indivisible, composable, déployable et identifiable par les services qu'il offre et qu'il requiert.

Malgré les nombreux atouts de ce concept, des questions très anciennes en génie logiciel se posent avec une acuité nouvelle, compte-tenu de la complexité des systèmes concernés et de leur besoin d'adaptabilité à leur environnement. Les problèmes de la construction de systèmes de grande taille par composition de modules logiciels sont pour

le moment peu pris en compte dans les modèles de composants existants. Cet objectif de qualité doit se retrouver tout au long d'un cycle étendu de développement de logiciel, depuis l'analyse des besoins jusqu'à la maintenance.

Actuellement, une des limitations de la CBSE vient du fait que la composition logicielle est principalement traitée lors des phases d'assemblage de composant ce qui entraîne notamment des dépendances non spécifiées entre composants ou encore des «adaptations» des composants. Ces dépendances ou adaptations sont nécessaires pour intégrer les composants dans le système car ils ne sont pas toujours conçus pour prendre en compte leur compossibilité et leur réutilisabilité. Des problèmes se posent alors non pas sur les composants, mais sur leur composition.

Cette conception par composition pose cependant plusieurs problèmes. Les services non uniformes impliquent que les messages échangés entre les composants le soient selon un dialogue bien précis, qui doit être respecté.

La faiblesse sémantique des représentations actuelles des patrons (patterns) entraîne des interprétations ambiguës et limite leur application. La spécification formelle s'avère être un mécanisme très utile permettant l'adaptation de solutions à un problème d'architecture ou de conception d'un système.

Le pouvoir d'expression d'un langage est une des choses primordiales à étudier pour notre cas. Le fait de pouvoir représenter les composants d'un point de vue structurel et comportemental est une chose importante. Mais le fait de pouvoir vérifier que cette représentation est correcte par rapport au modèle choisi est aussi important.

Il est par conséquent indispensable d'obtenir des systèmes sûrs, dont le fonctionnement a pu être vérifié avant leur mise en œuvre. La spécification d'un système présente tout d'abord des avantages quant à la description et la compréhension du système considéré :
-L'écriture de la spécification permet au concepteur de mieux comprendre le fonctionnement attendu de son système ainsi que les interactions entre les différents composants. L'écriture du modèle conduit à remettre en cause des choix conceptuels d'une part et à une meilleure compréhension du système développé d'autre part.
-Le modèle constitue une documentation précise lors de la maintenance.

Les modèles formels permettent aussi, de prouver formellement (i.e.mathématiquement) le bon fonctionnement du système développé. On est alors complètement sûr que quelle que soit l'évolution du système, son comportement sera celui attendu.

De plus, le développement de systèmes complexes demande des approches bien établies qui facilitent la robustesse des produits. La phase de conception d'un système prend alors de l'ampleur. La modélisation d'un système permet de raisonner sur ce dernier afin de prévenir les erreurs avant l'implémentation. Elle permet aussi une meilleure compréhension du système et une meilleure organisation de l'équipe de développement.

Pour cela, un des éléments primordiaux est le cycle de vie d'une application. Ce cycle de vie, va de l'analyse des besoins à la réforme d'une application, en passant par différentes phases qui peuvent s'organiser de manières différentes en fonction du processus de développement utilisé.

Contribution et démarche adoptée

La thèse que nous présentons est qu'il est possible d'adopter une démarche de développement des systèmes d'information à la fois rigoureuse et pragmatique, utilisant les composants conceptuels (décrits avec la notation UML), et reposant sur des techniques formelles éprouvées.

Pour atteindre cet objectif ambitieux, il faut donner aux représentations des composants conceptuels (Les design Patterns) un sens, c'est-à-dire une sémantique.

C'est donc par son biais que les techniques formelles doivent être mises à la disposition des concepteurs. Ces techniques formelles doivent s'intégrer dans l'environnement de développement.

Le premier objectif de cette thèse de compléter cette notation graphique par les bases formelles et les outils complémentaires, afin de rendre enfin accessibles aux développeurs de logiciels des techniques formelles permettant d'améliorer la qualité de leurs réalisations. Plutôt que d'attendre que les développeurs viennent aux techniques formelles, il nous semble plus réaliste et efficace d'amener doucement les techniques formelles aux développeurs, sans brusquer ces derniers et surtout sans révolutionner leurs habitudes et leurs environnements de développement.

Nous décrivons une méthode de spécification de patrons, intégrant deux paradigmes, les méthodes UML (semi- formelle) et le langage de spécification formelle LOTOS [Bol89] (Langage Of Temporal Ordering Specification). Nous allons donner une interprétation précise sur l'aspect structure, comportement et communication entre patrons. La démarche consiste à enrichir par une spécification formelle en LOTOS les patrons décrits en UML, en vue de leur vérification.

Le second objectif, est de proposer aux architectes des systèmes d'information un cadre formel d'analyse et de conception de ces systèmes. Nous considérons dans cette thèse qu'une bibliothèque de composants est un cas particulier de système d'information. Elle est au centre du processus de réutilisation et doit être capable de gérer toutes les informations qui se rapportent aux composants et aux processus de réutilisation et de capitalisation des connaissances et du savoir-faire de l'équipe.

Nous adoptons une approche descriptive car nous pensons qu'il est aussi important de gérer une représentation des composants plutôt que les composants eux mêmes. L'approche descriptive garantit à notre bibliothèque de composants la possibilité de s'intégrer dans un environnement de développement déjà existant avec le minimum de modifications dans la façon de travailler de l'équipe de développement.

L'approche proposée est une approche orientée problème et centrée utilisateur (un utilisateur est un concepteur de systèmes d'information utilisant un processus de recherche de composants). Cette approche propose aux concepteurs de formuler des problèmes de conception et de décrire la situation dans laquelle ils souhaitent les résoudre.

Nous nous intéressons dans notre démarche à l'étape de construction et de modélisation des applications à base de composants réutilisables. Cette étape consiste à construire l'architecture de l'application, c'est-à-dire la modélisation des composants participant à la

réalisation d'un service et leurs interactions. Pour cela nous définissons notre propre démarche de construction d'application basée sur les modèles de composants abstraits permettant de spécifier, construire, déployer et superviser une architecture logicielle typée. Ces modèles permettent de séparer la spécification des invariants du système et de sa dynamique.

Cette démarche inclue les processus de spécification, de sélection, d'instanciation, d'intégration et de validation, ainsi que l'analyse de leur composition à base de contrats. Ceci permet aux composants conceptuels d'être réutilisable. Ainsi, cette démarche permet au concepteur (non seulement) de modéliser les composants (qu'il sélectionne) d'une manière claire et précise, et aussi de détecter les erreurs d'interaction entre les composants et pouvoir les corriger avant leur mise en œuvre.

La spécification d'un assemblage de composants doit répondre à un double défi:
- elle doit être assez riche pour permettre une analyse de l'architecture et une configuration de celle-ci sur une plate-forme cible,
- elle doit rester légère et facilement compréhensible, i.e. elle doit abstraire uniquement les concepts nécessaires à cette analyse et à cette reconfiguration.

Nous nous plaçons à un niveau abstrait, dans lequel l'architecture du système d'information est une description abstraite et modulaire du système. A ce niveau, l'architecture est perçue comme une collection de composants (au sens d'entités logicielles), une collection de connecteurs (pour décrire les interactions entre composants) et des configurations, c'est-à-dire des assemblages de composants et de connecteurs [All97]. On utilise des ADLs (*Architecture Description Language* pour spécifier ces architectures [MT00].

Un autre objectif de cette thèse est de proposer une solution aux limitations des langages de description des architectures. Nous proposons un modèle simple, formel et outillé de description d'architecture logicielle dans lequel on puisse à la fois concevoir simplement les architectures et disposer de conditions d'assemblage riches et flexibles. Ce modèle est nommé LOTOS-ADL comme le langage de spécification.

Plan de la thèse

Ce mémoire est organisé en 08 chapitres

Le chapitre 1, motive l'importance de la modélisation conceptuelle et l'utilisation des méthodes formelles dans le développement des systèmes d'information. L'analyse de l'état de l'art dans le domaine de la spécification de systèmes d'information montrera que deux aspects complémentaires sont à considérer : la spécification du système en lui-même, qui s'appuie actuellement sur UML mais aussi parfois sur l'utilisation des méthodes formelles afin de capturer sans ambiguïté les aspects du système d'information

Le chapitre 2 présente un état de l'art sur les approches à composants. Cette étude permet l'identification des différents types de composants susceptibles d'être utilisés lors du processus de développement de systèmes d'information. Nous abordons aussi, les différentes formes de réutilisation en ingénierie du logiciel. Nous faisons un état de l'art portant sur les techniques de recherche de composants, la réutilisation et le principe de découplage entre composants qui constituent deux des paradigmes principaux sur lesquels s'appuient nos travaux.

Le chapitre 3 présente un état de l'art sur les langages de description d'architecture, nous étudions la notion d'architecture logicielle et plus particulièrement les styles architecturaux.

Dans les chapitres 4, nous abordons l'étude détaillée du langage LOTOS. Nous présentons les concepts de base de ce langage. Cette étude sera illustrée par des exemples simples.

Le chapitre 5, sera consacré à la présentation des patrons de conception. Nous y décrivons ainsi le formalisme, nous citerons leurs différentes catégories, dont nous détaillerons le contenu renforcées avec des exemples types.

Dans les chapitres 6, 7, 8, nous donnerons nos propres contributions dans cette thèse

Dans le chapitre 6, nous proposons un cadre pour l'utilisation des Designs Pattern par intégration du langage de spécification LOTOS. Nous donnons une interprétation précise sur l'aspect structure et communication entre patrons. L'objectif est d'utiliser des patrons existants pour spécifier des systèmes de manière à bénéficier aussi bien des avantages des approches semi-formelles que de ceux des approches formelles. Notre démarche s'effectue en deux temps: compléter par une spécification formelle en LOTOS le comportement des patrons de conception existants, puis les utiliser Notre objectif est de donner une interprétation précise sur l'aspect structure et communication entre patterns.

Dans le chapitre 7, nous proposons une approche basée contrat pour analyser les composants conceptuels. Dans ce chapitre, nous proposons une approche formelle, permettant au développeur de logiciels de modéliser et intégrer des composants pendant la face de conception. Pour ce faire, nous fournissons une spécification formelle basée sur des constructeurs de la logique de description. Nous enrichissons les patrons de conception par des constructeurs. Avec ces constructeurs, que nous définissons, on peut facilement représenter, détecter, instancier, évoluer et composer les patrons de conception.

Enfin, le chapitre 8, définit notre proposition de langage de description d'architecture ADL-LOTOS. Ce dernier est un langage formel permettant la description d'architectures logicielles abstraites. Il permet de prendre en considération l'évolution fondée sur les termes architecturaux, i.e., *comportements, connections et structures* à travers des mécanismes que nous présentons également. *ADL-LOTOS* permet l'évolution progressive d'une architecture abstraite en une architecture concrète prévue pour l'implémenter. Nous définissons notre méta-modèle afin d'obtenir un langage de description aussi concis que possible dont chacun des constituants puisse être aisément traduit dans un langage de programmation. Nous illustrerons ses concepts dans une application.

Une conclusion propose une synthèse et un bilan du travail effectué, ainsi qu'un ensemble de perspectives liées à la continuation du travail, aux nouvelles applications et aux nouveaux thèmes de recherche.

Des annexes complètent ce document pour en illustrer certaines parties un peu abstraites.

CHAPITRE 1

Ingénierie des systèmes D'information et méthodes formelles

Introduction

Dans le cadre de la modélisation des Systèmes d'Information (SI), de l'expression des besoins à la conception et à la réalisation de solutions logicielles, de nombreuses représentations souvent hétérogènes sont utilisées. Elles se différencient pour exprimer différents niveaux (externe, conceptuel, logique, etc.) ou différents points de vue (statique/dynamique/fonctionnel, comportement externe, contexte, architecture du logiciel,...). Ces représentations constituent des modèles (ou un modèle) du système d'information. Ces modèles ont différents objectifs selon le moment où on les écrit ou celui où on les utilise : on parlera de modèle de spécification, de conception, d'implantation, etc. Les nombreux acteurs qui interagissent (décrivent, évaluent, utilisent, etc.) dans cette modélisation ont des compétences et souvent aussi des objectifs différents.

Les concepts et les notations utilisés par ces représentations sont variés. Certaines représentations s'appuient sur des langages libres ou un peu structurés, d'autres sur des langages précis et formels. Dans le domaine des systèmes d'information, ce sont essentiellement des représentations graphiques qui sont utilisées ; elles sont qualifiées de langages semi-formels. Ce chapitre motive l'importance de la modélisation conceptuelle, et les méthodes formelles dans la méthodologie moderne de développement de systèmes d'information.

Dans la première section, nous abordons, le développement des SI, la modélisation conceptuelle, les modèles de système d'information, ainsi que, les méta-modèles pour la modélisation des SI. Dans la seconde section nous nous intéressons aux méthodes formelles dans la spécification des logiciels.

I Ingénierie des systèmes d'information

L'utilisation de la modélisation conceptuelle dans le développement des systèmes d'information a pour objectif une prise en compte plus adéquate des besoins des applications dans leur environnement d'utilisation. La modélisation conceptuelle consiste à représenter de manière abstraite, c'est-à-dire en termes de concepts familiers aux domaines d'application et indépendamment des technologies d'implémentation.

Les spécifications formelles s'imposent progressivement comme une condition essentielle a un bon développement logiciel. Leurs avantages principaux sont la rigueur

dans la spécification du problème, la preuve de propriétés et une conception méthodique, automatisable en partie.

I.1 Développement des systèmes d'information

I.1.1 Crise du Logiciel

Malgré les progrès constants et considérables des technologies de l'informatique, les systèmes d'information complexes répondent souvent bien moins que parfaitement aux besoins qu'ils sont supposés satisfaire. Le terme de « génie logiciel » (software engineering), discipline qui peut se définir comme l'ensemble des processus méthodologiques et des produits pertinents au développement de logiciels à grande échelle, a d'ailleurs été inventé pour susciter une recherche méthodologique plus appropriée.

La crise du logiciel a plusieurs causes:
- La complexité toujours croissante des systemes d'information modernes: les progrès constants de la technologie informatique suscitent et rendent possibles des applications toujours plus ambitieuses;

- La sous-estimation traditionnelle de la difficulté et donc, du cout du développement de logiciels (les méthodes largement intuitives de la programmation individuelle ne s'étendent pas à la construction de grands systemes complexes), avec comme conséquence un processus de développement qui se révèle être souvent complexe, long et couteux.
- une fiabilité et une maturité plus faibles pour les technologies du logiciel que pour celles du matériel, alors que l'importance relative du logiciel dans la réalisation des fonctions de systèmes complexes ne cesse de croitre.

I.1.2 Méthodologie de développement

Les questions méthodologiques sont plus complexes que dans des disciplines plus anciennes et donc plus mûres. En informatique, il est possible de commencer la programmation avant que la phase de conception soit terminée, parce que des modifications incrémentales aux programmes sont moins onéreuses que les modifications correspondantes dans des productions plus tangibles (génie civil ou fabrication mécanique, par exemple).

Les méthodes modernes de conception de logiciel s'accordent sur une définition générale des phases de ce qu'on appelle le cycle de vie du logiciel:
- L'analyse précise des besoins à satisfaire par le futur système, on parle aussi de phase d'ingénierie des besoins; il s'agit d'appréhender le domaine du problème posé et de spécifier le comportement externe (boite noire) d'un système qui peut résoudre ce problème.

Cette phase implique une modélisation adéquate des concepts pertinents du monde réel, la modélisation conceptuelle;
-La conception d'une solution au problème posé et de la structure générale d'un système qui va le résoudre;
-L'implémentation de l'architecture à l'aide de la technologie choisie (typiquement, un langage de programmation);

-Le test et la validation de l'implémentation, et la mise en opération du système;
-L'adaptation, ou maintenance, du système opérationnel aux changements des besoins et de l'environnement d'utilisation.

Les idées sur les qualités souhaitables des méthodes de développement d'applications ont beaucoup évolué. Par exemple, un apport important de la technologie des objets est que la structure des logiciels développes avec cette technologie peut refléter bien mieux celle du problème posé, ce qui simplifie le développement et sa compréhension.

L'évolution, toujours en cours, des idées sur la méthodologie de développement des systèmes d'information tend donc à consacrer une part plus grande de l'effort global aux phases en amont du processus (c'est-`a-dire l'analyse des besoins et la modélisation conceptuelle). C'est un changement important par rapport aux pratiques, où, souvent, ces phases en amont ne sont considérées que comme des aides servant à maitriser une phase de programmation complexe, considérée comme centrale au processus de développement.

I.2 La modélisation conceptuelle

L'idée qui sous-tend la modélisation conceptuelle est très simple: il s'agit de développer un modèle d'un domaine d'application particulier en termes des concepts familiers aux acteurs de ce domaine.

Un bon modèle conceptuel implique un processus d'abstraction, de simplification d'une situation perçue comme complexe dans le monde réel et de suppression des détails dont l'effet sur la solution au problème posé est minime ou inexistant. La solution au problème est donc aussi simplifiée, et son implémentation rendue plus performante.

Les modèles conceptuels pourraient remplir plusieurs fonctions dans le cycle de vie d'un produit logiciel:
-améliorer la compréhension des structures et fonctions d'un domaine du monde réel, grâce à un modèle basé sur des concepts clairs et intuitifs proches de ceux utilisés par les acteurs du domaine. Un modèle conceptuel peut être utile pour mieux comprendre la complexité;
-fournir un vecteur de communication précis entre modélisateurs et développeurs du système d'une part, et spécialistes du domaine d'application et utilisateurs finaux d'autre part;
-spécifier et guider les phases avales du cycle de vie du système telles l'implémentation,

I.3 Applications aux systèmes d'information

La modélisation conceptuelle consiste donc à construire des représentations abstraites de certains aspects des systemes physiques et sociaux et de leur environnement dans le monde qui nous entoure. La modélisation est une entreprise interdisciplinaire, ou les informaticiens se doivent d'étudier les domaines d'application suffisamment pour en comprendre les concepts importants et les termes dans lesquels se posent les problèmes, et les spécialistes des domaines d'application se doivent de pouvoir appréhender les possibilités de la technologie du moment.

La modélisation conceptuelle nécessite des notations, outils et techniques de représentation de l'information et des traitements concernant un domaine d'intérêt. Ces techniques de modélisation peuvent également être intègres dans des outils d'aide à la conception d'applications informatiques, ou ateliers de génie logiciel.

Des progrès en modélisation conceptuelle consistent à définir des langages de modélisation simples, puissants et de haut niveau, qui permettent de réduire la distance entre les concepts familiers aux acteurs dans les domaines d'application et leur représentation dans les modèles.

I.4 Modèle de système d'information

Un modèle de SI est une représentation formelle ou semi-formelle représentant la structuration de l'information dans le SI et son évolution au cours du temps. Ce niveau est décrit à travers des diagrammes qui représentent la structure du SI (aspects statiques) et son comportement (aspects dynamiques et règles d'intégrité).

Dans ce travail, nous nous intéressons à la représentation et à la modélisation de l'information dans les SIs. Un modèle de SI demeure une composante essentielle du « développement durable » du SI de l'entreprise ou de l'institution : bien construit, il constitue une démarche précieuse pour limiter les risques dans le développement d'un SI.

Les modèles de SI garantissent d'une part la traçabilité des évolutions que subit le SI, et d'autre part d'appréhender de nouvelles situations qu'aurait à supporter le SI.

I.4.1 Modèle

Un modèle est une représentation abstraite utilisée pour rendre compte d'un ensemble de phénomènes qui possèdent entre eux certaines relations. Son caractère abstrait facilite la compréhension du système étudié: il réduit sa complexité, permet de le simuler, le représente et reproduit ses comportements. Concrètement, un modèle (décompose) la réalité, dans le but de disposer d'éléments de travail exploitables par des moyens mathématiques ou informatiques.

I.4.2 Modèle de système d'information (SI)

Un modèle de SI est une représentation des concepts du domaine supporté par le SI et de leur organisation qui décrit de manière formelle les informations qui sont manipulées.

Un concept est une «représentation générale et abstraite d'un objet ou d'un ensemble d'objets ». C'est une «représentation d'un aspect de la réalité, isole par l'esprit, et une unité de pensée constituée d'un ensemble de caractères attribues a un objet ou a une classe d'objets ».

Dans un modèle de SI, il ne s'agit pas de modéliser les objets du « monde réel » mais l'information utilisée et échangée pour pouvoir accomplir les activités du métier. Les objets informationnels doivent en même temps pouvoir être dérivés en objets informatiques.

Un modèle informationnel représente la manière dont les informations sont structurées et la manière avec laquelle elles évoluent. Un modèle de SI est une représentation non ambigüe (i) des activités humaines et institutionnelles dans leurs dépendances avec le SI, (ii) des différents composants d'un SI et de leurs interactions, (iii) et des interactions entre ce monde des activités et celui du SI.

Dans ce travail, nous adoptons le paradigme orienté objet pour la modélisation des SIs. « Les méthodes objets introduisent la notion d'objet regroupant les structures de données, les contrôles et les traitements et proposent des spécifications globales d'un SI par l'intermédiaire de spécifications complémentaires statiques et dynamiques ». Un modèle de SI prend la forme de plusieurs diagrammes représentant la structure du SI (aspects statiques) et son comportement (aspects dynamiques).

I.4.2.1 Aspects statiques

Les diagrammes de classes constituent une expression semi-formelle des propriétés statiques du SI. Ils décrivent les classes du SI, leurs attributs, leurs identifiants, les méthodes des classes, les liens qui relient ces classes.

I.4.2.2 Aspects dynamiques

L'état d'un objet d'une classe évolue suite à l'exécution d'opérations (création suppression, mise-a-jour) ou de méthodes de classes. Les modèles dynamiques ont pour objectif de décrire les règles d'évolution des objets au cours du temps. Dans les méthodes objets, la dynamique des objets est traditionnellement décrite à l'aide de diagrammes d'états/transitions (dérivés des machines d'états finis) ou des réseaux de Petri.

I.4.2.3 Méta-modèle pour la modélisation des Sis

Un méta-modèle est un modèle qui définit le langage pour l'expression d'un modèle. Il définit: (i) les éléments du modèle (les concepts manipulés), et (ii) la sémantique de ces éléments (leur définition et le sens de leur utilisation).

Un modèle est un outil de communication permettant à plusieurs acteurs de confronter leur vision du système et de sa retranscription. Il est essentiel que les modèles utilises respectent au mieux des normes reconnues ou, a tout le moins, des standards partages par les divers acteurs. La définition d'un méta-modèle fixe un langage commun pour la production et l'exploitation de modèles de SI.

II Méthodes formelles

De nombreux progrès ont été faits ces dernières années dans le domaine des méthodes d'analyse et de conception. Ces progrès visent à assurer la qualité des documents produits et du logiciel et à rentabiliser les efforts de développement. De nouvelles méthodes sont apparues, les méthodes formelles.

L'ajout de formalisme aux méthodes d'analyse et de conception à objets permet une meilleure structuration et une organisation plus naturelle des spécifications. D'un autre

cote, les objets seront plus surs et la conception plus rigoureuse. Ce mariage doit se faire impérativement en respectant certaines contraintes industrielles qui sont généralement difficiles a tenir tout en respectant la production d'un logiciel de qualité. Des outils, ou mieux un environnement complet de développement, sont indispensables.

II.1 Méthodes de développement du logiciel

Une méthode est une technique de résolution de problèmes [Lau86]. Le terme de méthode recouvre plusieurs notions. C'est à la fois une philosophie dans l'approche des problèmes, une démarche, un formalisme ou des normes.

II.1.1 Processus de développement.

Tout système a un cycle de vie comprenant principalement sa conception, son exploitation, sa maintenance et sa mort. Le processus de développement d'un système d'information couvre l'intégralité de son cycle de vie. Il comporte les activités suivantes :

1. L'analyse des besoins définit les services du système, ses contraintes et ses buts en consultant les utilisateurs du système. L'analyse est la construction d'un modèle du système à partir de l'analyse des besoins. Ce modèle sert à établir plusieurs scenarios et les comparer dans une étude de faisabilité.

2. La conception est une proposition de solution au problème spécifié dans l'analyse. Elle définit la solution retenue par prise en compte des caractéristiques logiques d'usage du futur système d'information et des moyens de réalisation, humains, techniques et organisationnels. Conception système et conception détaillée sont parfois séparées. La première a pour objectif de donner l'architecture globale du système (i.e. les différentes parties) et la seconde décrit chaque partie du système. Cette spécification reste indépendante de tout moyen de réalisation.

3. La réalisation produit la solution exécutable en termes de programmes. Pour les logiciels complexes, deux phases sont requises : d'une part l'implantation des différentes parties et leur validation par des tests unitaires, et d'autre part l'implantation du système complet par intégration des parties et tests systèmes validant la spécification des besoins.

4. L'installation du logiciel règle les problèmes de mise en place dans l'organisation.

5. La maintenance adapte la solution conceptuelle aux changements organisationnels et aux évolutions technologiques (évolutive). Elle corrige aussi les erreurs accumulées dans les phases précédentes (curatives). La maintenance est un processus itératif dont l'activité répétée est un cycle de développement complet.

Validation : L'étude des facteurs de coûts du logiciel de [Boe82] et [Cal90] montre l'importance de la validation. Plus une erreur est détectée tardivement, plus sa correction

est chère. C'est pourquoi l'effort doit être porté sur la spécification plutôt que sur la conception.

De plus la validation doit être réalisée par une équipe pluridisciplinaire, comprenant des informaticiens certes, mais aussi des utilisateurs et des intervenants extérieurs au projet. La confrontation des points de vue devrait aboutir a un compromis raisonnable. De ce fait, la lisibilité des documents est un point clé de la validation. Elle se traduit par un langage unique et sans ambigüités (langage formel) et des formalismes graphiques.

II.1.2 Introduction aux méthodes formelles:

Nous nous intéressons ici aux modèles formels dans la spécification du logiciel, parfois appelée étape de spécification formelle.

II.1.2.1. Spécification formelle

Une spécification formelle est exprimée dans un langage à syntaxe et sémantique précises, construite sur une base théorique et qui permet des validations automatisées (système formel notamment). Cette base est généralement mathématique. Les réseaux de Petri, les grammaires formelles, les automates à états finis, la logique formelle, l'algèbre de processus, la théorie des graphes,... sont autant d'exemples de telles techniques [Bra88].

II.1.2.2. Intérêts d'utilisation des méthodes formelles

L'étape de spécification du logiciel est universellement reconnue comme étant cruciale dans le processus de développement du logiciel. Des mesures ont prouvé que les erreurs de spécification sont les plus coûteuses à corriger et sont malheureusement fréquentes [Boe82].

La spécification formelle est une bonne réponse à ce problème [Cho87, CG88, JS90]. La spécification formelle met en valeur un certain nombre de propriétés que doit respecter le logiciel. Bien qu'une spécification soit plutôt destinée à être lue, il se peut que le langage utilisé soit exécutable.

Enfin, les spécifications formelles peuvent jouer un rôle fondamental dans la réutilisation d'une part en donnant précisément les fonctionnalités d'un module réutilisable et d'autre part en facilitant l'intégration d'autres modules lors du raffinement.

Ainsi donc, les avantages des méthodes formelles sont dus aux aspects formels, à la précision et à l'abstraction [Cas et al. 93] [BH94].

a) Aspects formels :

- Des propriétés peuvent être établies par raisonnement formel, ce qui n'est pas le cas des autres formes de spécification. Les intuitions sont démontrables par une argumentation stricte.
- Les conséquences d'une spécification peuvent être mises en évidence. Ainsi, les contradictions sont détectées avant la réalisation.
- Les outils de preuve calculent et vérifient automatiquement et uniquement ce qui a été décrit.

b) Précision :
-Le domaine du problème est mieux perçu. A force de réfléchir sur le sujet, il est mieux compris, le vocabulaire est mieux défini, les imprécisions sont levées, les contradictions sont mises en valeur.
-Un langage commun lève les ambiguïtés et facilite la communication entre les acteurs. Si le langage est quelque peu hermétique, une conversion de la spécification en langage naturel garantit la qualité de la description.
-La spécification formelle est un avant-projet de la réalisation, qui permet au réalisateur de savoir exactement les modules qu'il doit programmer.

c) Abstraction :
-Seules les caractéristiques essentielles sont retenues, la spécification est plus concise,
-Une description abstraite est plus évolutive et plus perméable aux changements des besoins; elle améliore la maintenabilité du logiciel et facilite l'accès au logiciel de nouveaux acteurs du développement.
-Une distinction plus nette est établie entre étude et réalisation, dont les choix et les décisions sont différents. Ainsi, l'informaticien n'a plus un rôle aussi prépondérant dans les décisions de spécification.
-Les composants logiciels sont plus ables et plus généraux donc plus réutilisables.

Tous ces avantages se traduisent concrètement par des réductions de coûts en aval de la spécification et une rentabilité accrue des investissements en amont de la réalisation. La vérification est un point clé de la certification et de l'assurance qualité, surtout pour les systèmes critiques.

II.1.2.3 Classification des méthodes formelles

La spécification formelle de systèmes modulaires comprend une description abstraite des modules (données et traitements) et leurs interactions Nous proposons deux types de classement dans ce contexte : un classement selon les modèles utilises, qui résume les principaux courants de la spécification formelle actuels, et un classement selon les processus de développement.

L'approche par modèle abstrait (Z [Hay92], VDM [Jon93]) définit le module par une structure de données (informatique ou mathématique) et un ensemble d'opérations. Les opérations sont des abstractions procédurales axiomatiques (pré- et post-conditions) ou opérationnelles (algorithmes). La concurrence entre modules est généralement implicite. Les structures de données mathématiques sont souvent abstraites et donc plus simples à spécifier, mais les preuves sont plus difficiles a réaliser et a automatiser [San90]. L'approche algébrique [San90], [Bid89], [Gau90] est une approche plus déclarative et abstraite que celle de la théorie des types. Le module définit un type de données mais aucune structure particulière n'est exhibée. Les propriétés du système sont décrites par des axiomes et sous certaines conditions, les preuves peuvent être automatisées (par exemple, par les systèmes de réécriture [JL86] ou les langages fonctionnels [San90]).

L'approche dynamique modélise un module par un processus. Dans une telle spécification l'accent est mis sur les interactions entre modules plutôt que sur la structure. Ce type de spécification a été étudié dans les algèbres de processus comme CSP [Hoa85]

ou CCS [Mil89] ou encore dans les systèmes de transitions (automates, les réseaux de Petri) ou dans les logiques temporelles [Arn92].

1. Spécifications par modèle abstrait

Dans cette approche, un modèle particulier du type à spécifier est construit. Ce modèle possède un état, défini en termes d'autres types de données, et ce récursivement jusqu'a obtenir des combinaisons de types de base. Les constructeurs sont issus des mathématiques (ensembles, produit cartésien, séquences), des langages de programmation ou même des spécifications algébriques (listes, arbres).

Une spécification par modèle abstrait correspond à un système formel dont la syntaxe est donnée par les types et opérations du modèle abstrait et dont la sémantique est donnée par la théorie sous-jacente au modèle abstrait (ex: théorie des ensembles, logique du premier ordre, théorie des types, etc.).

2. Spécifications algébriques

Une spécification algébrique de type abstrait de donnée, ou type abstrait algébrique, est la donnée d'une signature et d'un ensemble d'axiomes. La signature comprend des noms de types (les sortes) et des opérations définies par leur profil. La signature permet de construire toutes les valeurs des types de données par application des opérations. Les axiomes sont des formules logiques.

Une spécification algébrique est un système formel dont le langage est donné par la signature et le système d'inférence est basé sur les axiomes et la déduction équationnelle et dont l'interprétation est donnée en termes d'algèbres [Gal87].

3. Spécifications hybrides

Les approches hybrides modélisent les systèmes par des processus ou des systèmes de transitions. Les données manipulées sont définies par des spécifications algébriques. Les aspects fonctionnels sont soit explicites dans les processus séquentiels soit implicites par transformation des données dans les systèmes de transition.

LOTOS [Bol89] est un langage basé sur l'algèbre de processus CCS [Mil89], enrichie par certains mécanismes de CSP [Hoa85]. Les données sont décrites par le langage de spécification algébrique ACT-ONE [EM85]. LOTOS présente l'intérêt d'être une norme ISO et d'être outille (simulation, exécution, vérification).

Les réseaux algébriques hiérarchiques [Gue94] définissent des réseaux de Petri dont les jetons sont des valeurs de types de données. Les places sont des stocks de valeurs et les transitions des transformations de données.

Conclusion

Les mérites des spécifications formelles sont désormais largement reconnus. Entre autre, les spécifications formelles induisent une compréhension plus poussée du problème à résoudre et une meilleure utilisation de l'abstraction, facteur important de la conception du logiciel. Cependant, les spécifications formelles ne résolvent pas tous les problèmes soulèves par la conception et le développement du logiciel.

En particulier, un problème crucial à résoudre est la mise en évidence des propriétés intéressantes de la spécification et leurs preuves. La catégorie de théorèmes prouvables (équationnel, inductif, par l'absurde, etc.) et les propriétés globales de la spécification (cohérence, complétude) dépendent à la fois de la puissance de la logique acceptée et des outils présents dans l'environnement.

Nous pensons toutefois que le langage de spécification et le langage de conception doivent être distincts pour favoriser l'abstraction. En ce sens, les descriptions algébriques semblent plus adaptées à la spécification que les modèles abstraits.

Néanmoins, la spécification formelle présente de nombreux intérêts pour la conception des systèmes d'information. Nous pensons que la définition de méthodes simples et claires de conception de ces spécifications permettra de développer plus encore leur usage.

Enfin une des dernières motivations concerne la structuration d'un ensemble de spécifications dans des bibliothèques. Ce point rejoint le problème du découpage de la spécification, celui de la construction progressive de spécifications à partir de spécifications existantes, celui de l'identification de composants adéquats dans la bibliothèque. La réutilisabilité des composants est un facteur clé de la rentabilité des méthodes formelles. Dans le prochain chapitre, nous allons présenter un état de l'art des différentes approches à base de composants. Cette étude nous permet l'identification des différents types de composants susceptibles d'être utilisés lors du processus de développement de systèmes d'information. Nous abordons aussi, les différentes formes de réutilisation en ingénierie du logiciel, et comment les composants peuvent être réutilisés dans le développement des systèmes d'information.

CHAPITRE 2

Les approches à composants et réutilisation

Introduction

Dans un contexte de réutilisation, un système d'information est un assemblage de composants réutilisables. Généralement, un composant réutilisable est défini comme une unité de conception (de n'importe quel niveau d'abstraction) identifiée par un nom, avec une structure spécifique et des directives de conception sous la forme de documentation pour supporter sa réutilisation. La documentation du composant illustre le contexte dans lequel le composant peut être utilisé en spécifiant les contraintes et les autres composants dont il a besoin pour offrir sa solution.

Face à l'émergence de collections de composants réutilisables de différents types (conceptuels, logiciels, etc.), certains environnements professionnels de développement d'applications ont évolué vers une gestion sommaire de composants. Or, si ces outils permettent effectivement de gérer des collections de composants, ils n'en facilitent pas pour autant leur sélection et leur utilisation par une recherche adaptée. Les bases de composants sont des éléments clés dans les environnements de développement à base de composants et de réutilisation de composants. Ce chapitre est constitué de deux sous-chapitres

Dans le premier sous-chapitre, nous présentons un état de l'art des différentes approches à base de composants. Nous définissons les notions de composant, de modèle de composants et de type de composant. Ensuite, nous proposons un ensemble de caractéristiques permettant de décrire les composants réutilisables. Enfin nous citerons les différents cycles de vie dans le développement par composants.

Dans le deuxième sous-chapitre, nous allons d'abord définir ce qu'est la réutilisabilité puis citer les différentes formes de réutilisation en ingénierie du logiciel. Nous faisons un état de l'art portant sur les techniques de recherche de composants, la réutilisation et le principe de découplage entre composants qui constituent deux des paradigmes principaux sur lesquels s'appuient nos travaux.

I Notions générales sur les composants

L'approche à base de composants est considérée comme un nouveau paradigme de développement des systèmes d'information [Bar et al.02]. Les activités de programmation ont été les premières concernées par ce nouveau paradigme.

Le développement par composants est perçu comme un atout majeur pour le développement des systèmes complexes. C'est une avancée considérable dans le domaine de la méthodologie du développement logiciel. Cette avancée est comparable à la transition entre la programmation procédurale et les langages à objets.

La technologie des composants permet la structuration des systèmes complexes dans le processus de développement. Un système composite, c'est-a-dire composé de composants, permet une structuration claire et donc une maintenance facilitée. Grace à la spécification des composants et à la définition d'architectures, les composants peuvent être réutilisés, même s'ils sont développes par des organisations complètement différentes avec des technologies elles aussi différentes.

I.1 Composants

I.1.1 Les limites de tout-objet

Même si la technologie objet a apporté de grands bouleversements dans le domaine informatique, celle-ci n'est pas suffisante pour répondre aux besoins devenus de plus en plus complexes. La réutilisabilité des objets est limitée à cause de certains problèmes liés au paradigme objet, comme l'anomalie de l'héritage dans les langages concurrents [MY93] ou encore le problème de l'encapsulation des aspects transversaux [Kiz et al. 97] comme la gestion de la répartition, la persistance, la tolérance aux défaillances, etc. De plus, il est difficile d'intégrer des objets hétérogènes conçus pour différents contextes applicatifs, en particulier s'ils sont dépendants de leur contexte d'exécution. Ceci est du aux interactions qui sont insérées dans les différentes parties du code fonctionnel [Ber02].

Il fallait donc trouver un mécanisme d'appel de méthodes qui ne connaisse pas explicitement l'objet appelé. Ce mécanisme est proposé dans le paradigme à composants qui va permettre une meilleure réutilisation que ne le permet le modèle à objets. En effet, ce paradigme définit des composants totalement indépendants de l'environnement logiciel dans lequel ils vont être utilises. Un composant définit une interface de communication qui contient des points d'entrée et de sortie appelés *ports*. Un point de sortie va être connecte à un ou plusieurs points d'entrée. Un composant va donc exporter l'ensemble des services qu'il propose mais aussi l'ensemble des services qu'il requiert.

I.1.2 Notion de composant

A l'heure actuelle, il n'existe pas de normalisation de la notion de composant. Généralement un composant réutilisable est défini comme « Une unité de conception (de n'importe quel niveau d'abstraction) identifiée par un nom, avec une structure définie et des directives de conception sous la forme de documentation pour supporter sa réutilisation ».

La documentation d'un composant illustre le contexte dans lequel il peut être utilisé en spécifiant les contraintes et les autres composants dont il a besoin pour offrir sa solution [Per et al. 00]. Nous adoptons cette définition générale du concept de composant car elle admet la possibilité d'utiliser le concept de composant dans des niveaux d'abstraction autres que le niveau d'implantation. En effet, le concept de composant a été largement

utilisé dans les langages de programmation pour palier aux trois principales limites rencontrées dans la technologie orientée objet [Mey97] [Fin et al. 98] [Vil03]:
−un mécanisme d'assemblage assez limité : il s'agit en effet d'une simple relation entre deux classes : une classe hérite ou invoque les méthodes d'une autre classe. Cette construction, réalisée à l'étape d'implantation, est exprimée à travers le langage de programmation.

Désormais, les classes sont étroitement liées, ce qui empêche de séparer le processus de développement en deux étapes : la construction (codage) des éléments composables et l'assemblage de ces éléments.
−une réutilisation à grande échelle limitée qui se heurte aux difficultés d'une réutilisation de type « boîte blanche » fondée sur l'héritage où les implantations des entités réutilisées (i.e. des classes) sont visibles et modifiables ;
−une absence des caractéristiques non fonctionnelles dans le modèle à objet original.

L'objectif premier du développement par composants est de structurer un système logiciel en composants vus comme des unités de déploiement.
Clemens Szyperski définit trois principales propriétés pour un composant [Szy02]:
-un composant est une unité de déploiement indépendante ;
-un composant est une unité de composition ;
-un composant n'a pas d'état observable au niveau de l'environnement qui l'entoure.

Ces propriétés vont donc avoir plusieurs implications. Premièrement, le fait qu'un composant soit une unité de déploiement indépendante implique que celui-ci doit être entièrement sépare de l'environnement et des autres composants. Un composant pouvant être composable avec d'autres composants, il doit avoir une spécification précise de ses besoins mais aussi de ce qu'il peut fournir. En d'autres termes, un composant encapsule son implantation et interagit avec l'environnement qui l'entoure par des interfaces clairement définies. Finalement, le fait qu'un composant n'ait pas d'état observable au niveau de l'environnement qui l'entoure implique qu'on ne pourra pas le distinguer des autres composants qui offrent des services identiques.
Le contrat du composant spécifie ses dépendances, ses interfaces, ses modes de déploiement et d'instantiation ainsi que le comportement de ces instances à travers les interfaces offertes.

I.1.3 Les interfaces du composant

L'interface d'un composant résume les propriétés qui sont visibles depuis son environnement extérieur. Elle va lister les différentes signatures des opérations fournies par le composant. Cette liste va permettre d'éviter les erreurs de typage au niveau des connexions du composant puisque les composants se connectent *via* leurs interfaces (*via* leurs ports). Les informations contenues dans les interfaces d'un composant facilitent donc la vérification de l'interopérabilité entre composants, et permettent à certaines propriétés d'être vérifiées plus tôt dans le cycle de conception (par exemple la vérification statique des erreurs de typage).

Techniquement, une interface est un ensemble d'opérations nommées qui vont être invoquées par le client. La sémantique de chaque opération est spécifiée. Cette spécification joue un double rôle car elle permet :

-au fournisseur d'implanter le composant ;
- au client de pouvoir utiliser l'opération.

Le fournisseur et le client ne se connaissent pas, la spécification de l'interface devient donc le médiateur qui permet aux deux parties de travailler ensemble. Un composant peut :
– soit fournir directement des interfaces qui correspondent aux interfaces procédurales des librairies classiques ;
– soit implanter des objets qui, s'ils sont accessibles par les clients, fournissent des interfaces. Ces interfaces implantées indirectement correspondent aux interfaces des objets contenus dans le composant.

Une spécification de composant nomme toutes les interfaces auxquelles pourraient adhérer un composant et ajoute toutes les propriétés spécifiques au composant.

I.1.4 L'implantation du composant

L'implantation du composant est la réalisation exécutable du composant, obéissant aux règles du modèle de composant. Il n'est pas nécessaire qu'un composant contienne seulement des classes, ou même contienne de classes du tout. Il peut :
- contenir simplement des procédures classiques et des variables globales,
- être écrit entièrement dans une approche de programmation par fonctions,
- utiliser un langage assembleur,
- ...

I.2 Trois types de composants

Trois types de composants existent : les composants conceptuels, les composants logiciels et les composants métier. Pour chacun de ces types, la définition générale donnée dans la section précédente est complétée avec des définitions spécifiques à chaque type de composants.

I.2.1 Composants conceptuels

Un composant conceptuel est une solution à un problème conceptuel sous la forme d'un modèle (ou une partie d'un modèle) destinée à être réutilisée. Suivant l'approche de modélisation utilisée, la solution peut être spécifiée avec le langage UML (Unified Modeling Language) ou tout autre langage de modélisation. Les composants conceptuels peuvent être de deux types :

-Les composants produit : un composant produit est une partie cohérente d'un modèle qui peut être réutilisée avec d'autres composants produit pour assembler un modèle complet. Un produit correspond au but à atteindre lorsqu'on utilise la solution offerte par le composant produit. Par exemple, le patron « composite » de Gamma [Gam et al. 95] peut être considéré comme un composant conceptuel produit.

-Les composants processus : un composant processus est une partie cohérente d'un processus qui peut être réutilisée avec d'autres composants processus pour assembler un processus complet. Un processus correspond au chemin à parcourir pour atteindre

la solution offerte par le composant processus. Par exemple, le patron « revue technique» d'Ambler [Amb98] est un composant conceptuel processus.

I.2.2 Composants logiciels

Un composant logiciel est un paquetage cohérent d'implantations logicielles qui peut être indépendamment développé et délivré. Il possède des interfaces explicites spécifiant les services offerts et les services requis par le composant. Il peut être composé avec d'autres composants et être éventuellement paramétrable sans pour autant modifier son implantation [DW99].

D'autres définitions s'intéressent à des aspects spécifiques des composants logiciels : «Un composant est une brique logicielle préfabriquée conçue pour être composée et assemblée avec d'autres composants» [MN97].

La définition de Meijler insiste sur la séparation du processus de production et de réutilisation du composant. Cette propriété de séparation est fondamentale dans les approches de développement à base de composants. Elle permet l'industrialisation du processus de production des systèmes d'information divisé en deux sous-processus : le processus de développement pour la réutilisation dans lequel les composants sont développés avec une optique de réutilisation et le processus de développement par réutilisation de composants dans lequel les systèmes d'information sont construits en utilisant des assemblages de composants.

La définition de composant que nous retiendrons dans ce mémoire est celle donnée par Clemens Szyperski dans la deuxième édition de son ouvrage *Component Software Beyond Object-Oriented Programming* [Szy02]:

Définition : *Un composant logiciel est une unité de composition ayant des interfaces contractualisées ainsi que des dépendances contextuelles. Ces dépendances explicitent les interfaces exigées ainsi que les plateformes d'exécutions possibles. Un composant peut être déployé indépendamment, et être sujet `a la composition. De ce point de vue, le composant est une unité exécutable.*

I.2.3 Composants métiers

A l'heure actuelle, il n'existe pas de normalisation de la notion de composant métier. Le concept de composant métier résulte de celui d'objet métier ou « business object » [Szy98]. Ainsi, l'OMG (Object Management Group) définit la notion d'objet métier comme suit :

« Business Objects are representations of the nature and behaviour of real world things or concepts in terms that are meaningful to the enterprise. Customers, products, orders, employees, trades, financial instruments, shipping containers and vehicles are all examples of real-world concepts or things that could be represented by Business Objects ».

D'autres spécialistes définissent un composant métier comme une unité de réutilisation de connaissances de domaines, d'un point de vue conceptuel uniquement, par exemple, Casanave donne la définition suivante :

«Un composant métier est vu comme une représentation de la nature et du comportement d'entités du monde réel dans des termes issus du vocabulaire d'une entreprise » [Cas96].

I.2.4 Modèle de composants

Un modèle de composants consiste en un ensemble de conventions à respecter dans la construction et l'utilisation des composants. L'objectif de ces conventions est de permettre de définir et de gérer d'une manière uniforme les composants. Elles couvrent toutes les phases du cycle de vie d'un système d'information à base de composants : la conception, l'implantation, l'assemblage, le déploiement et l'exécution. Concrètement, un modèle de composants décrit certains aspects des composants comme la définition de composants à partir d'objets (classes, modules, etc.), les relations entre les composants, les propriétés fonctionnelles et non fonctionnelles de chaque composant, les techniques d'assemblage des composants, le déploiement et l'exécution d'un système d'information à base de composants, etc.

Dans la pratique, un modèle de composants donné est spécifique à une phase du cycle de vie du composant et du système d'information. On trouve ainsi des modèles de composants pour la phase de conception (patrons de conception), et d'autres pour les phases d'implantation et de déploiement (EJB, CCM, etc.).

I.3 Les différents cycles de vie dans le développement par composants

Le CBSE (*Component Based Software Engineering)* [HC01] utilise les méthodes, outils et principes de l'ingénierie logicielle classique. Cependant, la CBSE distingue deux cycles de vie : un pour le développement du composant (*Design for reuse*) et un autre pour le développement d'un système à base de composants (*Design by reuse*).

On va avoir une approche différente pour chacun des cas.

- Dans la phase de développement du composant, la réutilisabilité est l'enjeu principal: les composants sont crées pour être utilisés mais surtout réutilises dans différentes applications, dans la plupart des cas, celles-ci n'existant pas encore.

Un composant doit être spécifie précisément et formellement, facile à comprendre, assez générique, facile à adapter, à délivrer, à déployer et à replacer. La spécification d'un composant est donnée par ses interfaces sous forme de méthodes externes, séparées de l'implantation du composant.

Dans la plupart des cas, une interface spécifie uniquement les propriétés syntaxiques. Une interface qui spécifie aussi les propriétés comportementales ou non fonctionnelles est appelée une *interface enrichie*.

-Le développement d'un système à base de composants se focalise sur l'identification des entités réutilisables et des relations qui les relient entre-elles. L'implantation du système n'est pas la partie la plus longue. Le plus laborieux se situe au niveau du choix des composants : les localiser, sélectionner le plus approprie, les tester, les vérifier etc...

On peut donc distinguer deux cycles de vie : le cycle de vie du composant logiciel en lui même et celui du système à base de composants.

I.3.1 Cycle de vie d'un système à base de composants

Le développement d'un système à base de composants diffère d'un système traditionnel. Il se focalise sur l'identification des entités réutilisables et leurs relations.
L'identification initiale des besoins est déterminée comme pour un développement traditionnel avec une attention particulière pour la cohérence entre le système et les exigences des composants. La réutilisabilité étant l'enjeu principal dans le développement de système à composants, lors de la phase d'identification des besoins du système, les besoins des composants devront être identifiés.
Le cycle de vie d'un système à base de composants comporte cinq phases :

La phase de conception de l'application qui comporte deux étapes essentielles :
-La vue logicielle du système qui spécifie les architectures du système en termes de composants et leurs interactions. Dans cette vue, les composants sont représentes par la spécification de leurs interfaces, on pourra inclure la spécification des propriétés non fonctionnelles. Dans les systèmes temps réel, on pourra inclure les propriétés temporelles.
-La vue structurelle spécifie les architectures du système constituées des implantations des différents composants. L'implantation d'un composant doit être conforme à un modèle de composant particulier décrit pour une plate-forme d'exécution particulière, à un *Framework* de composant et aux différents services qui sont spécifiques à la technologie employée. Le modèle de composant et le *Framework* ont un impact significatif dans la solution de conception : ils doivent donc être pris en compte au plus tôt dans la phase de conception.

Le processus de spécification de l'architecture est combine par la recherche, l'´evaluation, la sélection et l'adaptation des composants qui correspondent aux rôles définis par l'architecture du système. Les besoins du système vont être reformulés afin de faciliter l'adaptation des composants disponibles pour ce système : la conception du système doit tenir compte des besoins des différents composants et du système.

La phase d'implantation inclut l'adaptation, la composition et le déploiement des composants grâce à un *Framework* de composant.

La phase de vérification ou test vérifie le système en le testant. Un modèle de composant enrichi permet à une partie significative du système d'être vérifiée dés la phase de conception, ce qui permet une économie considérable dans la phase de test.

La phase de maintenance permet le remplacement ou la mise à jour des composants du système.

I.3.2 Le cycle de vie d'un composant

Le processus de développement d'un composant est, par plusieurs aspects, similaire au développement d'un système ; les besoins doivent être capturés, analysés et définis, le composant doit être conçu, implanté, vérifie, valide et livre. Lors de la construction d'un nouveau composant, les développeurs peuvent réutiliser d'autres composants et utiliser des procédures d'évaluation du composant semblables à celles utilisées pour le développement d'un système. Ils vont être réutilisés dans différents produits. Ceci a pour conséquence:
-Plus de difficultés dans la gestion des besoins, causées par l'interaction entre le composant et le système.

-Plus de précisions dans la spécification du composant.
-De plus gros efforts sont nécessaires pour créer des unités réutilisables.
-Plus de rigueur et de documentation dans la vérification de la spécification des composants surtout lors des transferts de composants entre organisations.

Une fois que le composant a été testé, spécifié, stocké dans une librairie de composants, la prochaine étape dans le cycle de vie du composant est la phase de déploiement dans le système. Le déploiement du composant consiste à son enregistrement dans le système ainsi qu'à établir la communication avec le reste du système. Cette communication est obtenue par lien dynamique entre les interfaces du composant et le système.

Le déploiement doit se faire de manière automatique et sans provoquer de changement dans le reste du système. Un modèle de composant fournit le support pour lier le composant dans le système par un ensemble de fonctions ou de composants qui sont spécifiques à la plate-forme.

II Réutilisation

La réutilisation s'inscrit dans l'ensemble des solutions en cours de recherche et de développement pour faire face à ce que l'on appelle aujourd'hui la crise du logiciel. Elle se définit comme une nouvelle approche d'ingénierie de systèmes selon laquelle il est possible de construire un système à partir d'éléments existants. Elle s'oppose aux approches usuelles dans lesquelles la construction d'un nouveau système part de rien (« from scratch ») et nécessite de tout réinventer à chaque fois.
Pour notre part, nous nous intéressons à l'ingénierie de systèmes d'information basée sur la réutilisation. Nous nous inscrivons dans une approche reposant sur le paradigme «*composant conceptuel*».

Deux types de problématiques de recherche existent dans le domaine de la réutilisation. Celles qui relèvent de l'ingénierie pour la réutilisation (« design for reuse ») visent à développer des méthodes et des outils pour supporter le processus de production de composants. Celles qui relèvent de l'ingénierie par réutilisation (« design by reuse ») ont pour objectif de proposer des méthodes et des outils pour exploiter des composants réutilisables.

Nous proposons dans ce qui suit un rapide panorama des différentes activités liées à la réutilisation. La mise en œuvre de la réutilisation nécessite des méthodes et des techniques pour *l'ingénierie de composants réutilisables* (design for reuse) et *l'ingénierie de systèmes par réutilisation de composants* (design by reuse).

II.1 Réutilisabilité

La réutilisabilité est une des plus grandes promesses de la technologie orientée objet, elle consiste à réutiliser des composants, des parties de codes déjà existants, évitant de réécrire des procédures et des composants qui fonctionnent.

II.1.1 Définitions:

Plusieurs définitions ont été proposées. Nous retenons celles qui sont les plus proches à

nos perspectifs.

-La réutilisabilité est l'aptitude d'un logiciel à être réutilisé en tout ou en partie pour de nouvelles applications [Bai87].

-La réutilisabilité consiste à se servir d'un composant pour en créer un nouveau [Mor95].

II.2.1. Techniques de réutilisation

Il existe plusieurs techniques de réutilisation. Parmi ces techniques nous distinguons:

1. **Duplication:** Est d'après [Cou96] la plus ancienne, la plus connue et la plus pratiquée des techniques de réutilisation. Cette technique de base consiste à recopier du code (souvent sous forme de procédure) d'applications existantes vers des applications en cours de développement. Le code sera ensuite modifié pour qu'il s'adapte exactement au nouveau besoin.

2. **Bibliothèques:** Une bibliothèque est un ensemble de sous programmes, lors de l'appel d'un sous programme, il est possible de lui passer des paramètres qui vont influer sur le traitement effectué. Pour améliorer le paramétrage d'un sous-programme sans augmenter son nombre de paramètres, on utilise souvent des variables globales. Cependant le seul risque qu'il y ait est celui de l'interférence entre les variables globales utilisées.

3. **Packages:** Un package est un ensemble constitué de sous-programmes et des variables globales nécessaires à leur fonctionnement. Leurs idées viennent des problèmes rencontrés dans les bibliothèques avec les variables globales. En effet l'approche par packages consiste à maîtriser l'usage de ces variables globales en ne les rendant visibles que par un sous-ensemble parfaitement identifié de sous-programmes qui sont les seuls à les manipuler. Cependant elle laisse à désirer en ce qui concerne l'adaptabilité.

4. **Orienté Objet:** Le principe de base est l'utilisation d'objets et la possibilité de mettre en œuvre les techniques d'héritage, de polymorphisme et de liaison dynamique et bénéficier de leur grand potentiel à faire valoir la réutilisation et cela en utilisant un des nombreux langages de programmations OO.

II.2 Différentes formes de réutilisation.

On peut distinguer trois grandes approches permettant la réutilisation des logiciels: les bibliothèques de composants (ou «toolkits»), les canevas (ou "frameworks") et les patrons (ou "patterns") :

-L'approche bibliothèque [Pou95] est la plus ancienne et la plus rudimentaire. Elle offre des ensembles de composants logiciels pour lesquels la recherche, la composition et l'adaptation restent à la charge du développeur. La granularité de ces composants est directement liée aux constructeurs d'un langage de programmation (la classe, la procédure,

la fonction....) et leur niveau d'abstraction est peu élevé puisque ces composants ne fournissent ni le contexte dans lequel on peut les utiliser ni les adaptations que l'on peut leur apporter.

-Les canevas (ou « frameworks ») [Wil90], [Fuk93] sont le plus souvent dédiés à un domaine d'application, ils proposent des architectures-types globales pour un domaine.
Des canevas ont été proposés pour la conception d'interfaces graphiques [Wil90] et le développement des systèmes d'exploitation [Mad et al .89]. Ils ont un réel avantage par rapport aux bibliothèques : ils dispensent le développeur de choisir les classes, de fournir les interconnexions, de découvrir quelles méthodes sont disponibles, de trouver celles qui doivent être appelées et dans quel ordre. Les canevas cachent toute cette complexité en offrant un niveau d'abstraction plus élevé. Cependant dans certaines situations, ils peuvent devenir trop rigides compte tenu de leur taille.

-Les patrons ont été introduits par Alexander [Ale77] dans le domaine de l'architecture.
Cette approche connaît aujourd'hui un réel succès à travers les modèles de conception («design patterns ») d'E. Gamma [Gam94]. Les modèles de conception sont des descriptions d'objets et de classes communicants qui sont personnalisés pour résoudre un problème général de conception, dans un contexte particulier. Cette forme de composant présente plusieurs avantages sur les approches précédentes. D'abord, la granularité d'un patron fournit une unité de raisonnement très modulaire, en effet, chaque patron existe pour répondre à un problème type. Par ailleurs, l'intégration dans un même patron d'un problème type et d'une solution constitue une aide à la recherche et à l'intégration de composants. Bien sûr des problèmes restent à résoudre quant à la composition de patrons et à leur organisation pour permettre une réutilisation efficace.

II.3 Etat de l'art sur les techniques de recherche de composants.

Les techniques de recherche de composants présentées dans cet état de l'art, sont organisées selon quatre catégories : classification externe, appariement structurel, recherche comportementale et navigation. Ces catégories reflètent l'usage de ces techniques par les concepteurs ou ingénieurs d'applications selon leur degré de connaissances des bibliothèques de composants et leurs expériences de réutilisation ainsi que le degré de définition du problème à résoudre.

II.3.1 Classification externe

Les techniques de recherche par classification externe indexent une représentation externe des composants souvent préparée manuellement. Par exemple, l'approche de classification par facettes [PW95] [Zha et al. 2000] pour la recherche de composants consiste à caractériser le composant par un ensemble de facettes. Chaque facette représente une information permettant d'identifier et de sélectionner un composant. Une facette est définie par son nom et son espace de termes appelé vocabulaire. Par exemple, une facette décrivant la technologie d'implantation d'un composant, porte le nom « technologie d'implantation » et a comme vocabulaire les termes Java, C, etc.

II.3.2 Appariement structurel

Cette catégorie regroupe les techniques qui s'intéressent aux propriétés structurelles des composants. Par exemple, les techniques à base de signatures [Rit92] [ZW93] exploitent les techniques d'appariement et de transformation de types pour retrouver les composants.

Dans un cadre standard de développement par objets, un composant est construit par composition de classes. Une classe déclare un ensemble de méthodes, de types et d'attributs. Un composant peut donc être représenté par l'ensemble des signatures des entités qu'il contient. Les signatures peuvent être basées sur des types simples, des types complexes voire des fonctions. Cette catégorie est intéressante si l'ingénieur d'applications connaît à priori la forme ou le nom d'une méthode appartenant au composant qu'il cherche. Elle peut de plus améliorer sensiblement la production d'un acteur qui intervient dans une phase tardive du cycle de développement des SI et qui manipule des centaines de composants logiciels.

II.3.3 Recherche comportementale

Les techniques de recherche de composants dites comportementales [Chou et al.96], [Park97] exploitent la trace d'exécution des composants pour les retrouver. La trace d'exécution d'un composant représente son comportement dynamique. Pour rechercher un composant possédant un comportement spécifique ou qui s'en approche le plus, il faut définir une relation d'ordre sur les comportements de composants. La requête de l'ingénieur d'applications se présente sous la forme d'un programme qui appelle systématiquement un sous-ensemble des opérations de chaque composant de la bibliothèque et qui récupère leurs comportements pour les comparer au comportement recherché. Seuls les composants qui vérifient le comportement indiqué dans la requête sont fournis à l'ingénieur d'applications.

II.3.4 Recherche par navigation

Les techniques de recherche de composants par navigation exploitent les relations implicites ou explicites qui peuvent exister entre les composants appartenant à une bibliothèque de composants. Selon ses besoins, l'utilisateur navigue à l'intérieur de graphes guidés par la sémantique associée à ces relations. Certaines bases de composants ont adopté l'approche de recherche par navigation comme l'environnement Smalltalk-80 System Browser [Gol84] et la technologie de l'hypertexte [Fre94]. Le point faible de cette approche est qu'elle exige à l'utilisateur de piloter le processus de navigation, ce qui risque de devenir une tâche difficile si la taille de la base de composants est importante. De plus, le résultat final du processus de recherche dépend fortement du nœud de départ dans le graphe de recherche.

II.4 Caractéristiques des composants réutilisables

Nous présentons dans cette section un ensemble de critères permettant de caractériser les composants réutilisables. Il est possible de caractériser les composants selon un certain nombre de critères, à savoir le type de connaissance, la couverture, la portée, la nature de la solution, la technique de réutilisation, l'ouverture, la granularité et l'architecture.

1. Type de connaissance : Le type de connaissance précise la nature (produit ou processus) d'un composant conceptuel. Une connaissance de type produit correspond à un but à atteindre tandis qu'une connaissance de type processus correspond au chemin à suivre pour atteindre un résultat.

2. Couverture : Le degré de couverture d'un composant est défini par rapport à un domaine d'application. Un composant réutilisable peut être un composant général (resp. domaine, entreprise) si le problème traité est fréquent dans de nombreux domaines d'applications (resp. dans un domaine d'applications, dans une entreprise particulière).
3. Portée : La portée d'un composant est évaluée en fonction de l'étape d'ingénierie (analyse, conception, implantation) à laquelle le composant s'adresse.
4. Technique de réutilisation : Un composant peut être réutilisé par adaptation, par spécialisation, par composition etc.
5. Ouverture : L'ouverture caractérise le niveau de transparence du composant :
-Boîte noire : seule l'interface du composant est visible, la structure interne n'est ni visible ni modifiable.
-Boîte blanche : le composant est modifiable.
-Boîte en verre : la structure interne est visible mais non modifiable.
5. Granularité : La granularité des composants orientés objets est souvent mesurée en nombre de classes.
6. Architecture : L'architecture d'un composant peut être distribuée ou locale.

II.5 Ingénierie de composants réutilisables

L'ingénierie de composants réutilisables a pour finalité la production d'une infrastructure de réutilisation. Elle recouvre l'identification et la spécification des composants, leur organisation et leur implantation.

L'activité d'identification et de spécification de composants réutilisables est essentielle dans le processus puisqu'elle a pour objectif de produire les ressources réutilisables. Il existe des méthodes et des techniques que l'on peut classer en deux catégories.

L'approche par rétro-ingénierie est considérée comme ascendante. Elle se caractérise par l'exploitation des produits de développement existants pour produire des composants. Cette approche peut se décliner sous différentes formes comme les modèles de réutilisation

L'approche par analyse du domaine est la deuxième tendance. C'est une approche descendante qui peut être définie simplement comme l'activité d'identification des objets et des opérations d'un champ d'application [Pri90] [Ara94]. Le résultat essentiel de l'analyse de domaine est un référentiel de domaine. Il existe plusieurs méthodes d'analyse de domaine [Cam et al .90] [Kan et al.90] [Bai92] [WP92] [Sem98] présentant de fortes différences quant aux outils utilisés, aux processus mis en œuvre et aux résultats produits.

L'organisation et l'implantation des composants réutilisables concernent la conception et la réalisation d'infrastructures de réutilisation. Deux approches sont aujourd'hui mises en avant dans la conception d'architectures de réutilisation. L'approche statique définit une organisation statique des composants. Cette organisation prescrit entièrement le processus de recherche et / ou de composition des composants. Elle est largement utilisée dans les bibliothèques de composants dans lesquelles le lien d'héritage définit statiquement un chemin entre les classes. Cette approche est aussi utilisée dans les « Framework » qui prédéfinissent la structure du produit final.

L'architecture de réutilisation constitue le dispositif de couplage entre les deux activités essentielles de la réutilisation, l'ingénierie de composants réutilisables que nous venons de décrire, et l'ingénierie de systèmes par réutilisation de composants à laquelle nous allons maintenant nous intéresser.

II.6 Ingénierie de systèmes par réutilisation de composants

L'ingénierie de systèmes par réutilisation de composants consiste à exploiter une architecture de réutilisation pour produire un système. Cette ingénierie peut être très différente en fonction de l'architecture de réutilisation disponible.

A ce propos, il faut noter que l'ingénierie de composants réutilisables et l'ingénierie de système par réutilisation de composants sont encore considérées comme deux activités indépendantes dans lesquelles les sorties de l'une constituent tout au plus les entrées de l'autre ou inversement. Le couplage entre les deux activités est donc faible et il n'existe pas de modèle les intégrant véritablement.

Dans la pratique, les formes d'organisation les plus utilisées sont l'organisation producteur/consommateur et l'organisation entrelacée. Dans l'organisation producteur/ consommateur, l'activité de développement du système utilise des composants réutilisables fournis par l'activité d'ingénierie de composants.

Dans l'organisation entrelacée, l'activité de développement du système, non seulement utilise des composants mais contribue aussi au développement de ces composants.

La première activité est la recherche et la sélection de composants. Elle consiste à rechercher dans la bibliothèque un composant qui répond à un problème particulier de développement.

Dans la plupart des situations il n'y a pas un seul composant candidat qui répond aux besoins mais plusieurs. Il est alors nécessaire de comparer les composants candidats en vue de sélectionner celui qui est le plus proche du besoin exprimé.

L'adaptation et l'intégration de composants constituent la deuxième activité. Les techniques d'adaptation sont nombreuses, depuis la modification du composant jusqu'au mécanisme d'instanciation en passant par la paramétrisation et la spécialisation. Dans certaines approches, l'adaptation peut être guidée si le système de réutilisation propose des alternatives avec des arguments qui aident aux choix. La réutilisation devient effective si le composant est intégré au produit en cours de développement. Ce problème d'intégration peut être simple si le langage de développement et le langage de spécification des composants sont les mêmes; l'intégration se fait à travers des mécanismes de communication intégrés au langage commun. Il devient plus difficile si les langages de développement et de spécification des composants sont différents.

Conclusion

Dans ce chapitre, nous avons présenté les besoins face aux limites du paradigme objet: le paradigme à composant répond à ces besoins. Nous avons vu qu'un composant est : *une unité de composition ayant des interfaces contractualisées et des dépendances contextuelles. Un composant peut être déployé indépendamment et être sujet à la composition [Szy02]*.

Nous avons présenté un état de l'art sur les composants. Trois catégories de composants sont identifiées et décrites : les composants conceptuels, les composants logiciels et les composants métier.

Différents types de composants susceptibles d'être utilisés lors du processus de développement de systèmes d'information existent. Pendant chacune des phases de

développement (analyse, conception, implantation), Les acteurs d'une équipe de développement de systèmes d'information sélectionnent et réutilisent des composants qui répondent partiellement ou totalement à leurs besoins.

Puis nous avons étudie les deux cycles de vie dans le développement par composants :
-le cycle de vie du système à composants,
-le cycle de vie du composant.

Nous avons aussi présenté une synthèse sur la réutilisation. Cette présentation synthétique de la réutilisation a permis de mettre en évidence l'étendue du champ de la recherche dans le domaine. Elle a également montré la grande variété des approches proposées pour répondre parfois à une même préoccupation. Elle permet aussi de mettre en évidence les problématiques nouvelles à étudier : la définition de *modèles de composants* respectant les principes d'abstraction et de variabilité, la proposition de *démarches de développement à base de composants* et la proposition de démarches pour développer des composants, et le développement d'outils qui intègrent de manière systématique un développement par réutilisation.

Il est maintenant possible d'envisager une approche de développement basée sur la réutilisation de composants existants et éprouvés. Une telle approche doit permettre de réduire le temps de conception des systèmes d'information, d'en améliorer la qualité et d'en faciliter la maintenance.

La réutilisation de patrons nous semble la forme de réutilisation la plus adaptée à l'ingénierie des SI. En effet, elle peut être utilisée dans toutes les étapes du cycle de développement d'un SI (expression des besoins, conception, implantation).
La réutilisation de patrons peut être mise en œuvre dans toutes les étapes du cycle de développement des SI. Cette approche s'avère possible et intéressante pour les raisons suivantes :
-**La diversité des intervenants.** Dans l'entreprise industrielle, l'information technique est destinée à une multitude d'acteurs dont les métiers, les connaissances et les rôles sont bien différenciés : chef de projet, responsable du bureau d'étude, responsable de l'industrialisation, projeteur, dessinateur, préparateur code numérique**,** ingénieur de calculs etc. Une approche à base de patrons permet de capitaliser sous une forme personnalisée, l'ensemble des problèmes et des solutions relatives à chaque métier.
-**L'homogénéité des composants réutilisables**. La notion de patron peut être vue comme un concept générique (indépendant d'un domaine, indépendant d'un langage) que l'on peut utiliser pour décrire de manière homogène et modulaire des formes de composants très variées. La notion de patron peut être utilisée, par exemple pour décrire à la fois des problèmes (et les solutions associées) de conception logicielle et des problèmes (et les solutions associées) de SI.

Dans ces deux exemples, seule la nature du problème est différente. Il semble possible de proposer un cadre méthodologique pour la conception de SI dans lequel le concept de patron peut être utilisé aux différents niveaux du développement (patrons métier, patrons de conception logicielle, patrons de dérivation ....).

Dans le prochain chapitre, nous allons nous intéressé aux langages de description d'architectures (ADLs).

CHAPITRE 3

Etat de l’art sur les Langages de Description d’Architecture (ADLs)

Introduction

Depuis l'émergence du domaine des architectures logicielles, de nombreux travaux ont été entrepris pour l'utilisation formelle des architectures Parmi ces travaux, une variété de langages pour la conception architecturale a été créée pour fournir les architectes logiciels avec des notations pour spécifier des modèles architecturaux et pour pouvoir raisonner dessus [Mon et al. 97]. Les Langages de Description d’Architecture (ADL) et leurs outils soutiennent le développement centré architecture [MT 97].

Dans ce chapitre, nous présentons un état de l'art sur les outils formels utilisés pour la conception architecturale en correspondance avec la problématique que nous avons définie. Ainsi, nous nous plaçons dans un contexte de conception de systèmes d’information et d'utilisation des styles architecturaux.

Nous étudierons la notion d’architecture logicielle. Puis nous définissons la notion de style architectural. Nous étudierons également les langages de description d'architecture. Nous établirons ensuite un bilan des systèmes présentés selon un ensemble de critères.

I.1 La notion d’architecture logicielle

L’architecture logicielle est une discipline récente qui s’intéresse aux structures d’un logiciel. Elle s’adresse plus particulièrement à la conception de systèmes logiciels de grandes tailles ou de familles de produits logiciels. Ainsi les coûts doivent être toujours plus faibles, le temps de développement du logiciel toujours plus court, tout en améliorant la qualité du produit [Rev02]. Plusieurs définitions du terme «architectures logicielles» sont proposées dans la littérature [Bas et al. 99] [Gar et al. 92] [PW92]. Nous en proposons ici celle donnée par M.Shaw et D.Garlan:

L'architecture logicielle [est un niveau de conception qui] implique la description des éléments à partir desquels les systèmes sont construits, les interactions entre ces éléments, les patterns (patrons) qui guident leur composition et les contraintes sur ces patterns *[SG96]*.

Par conséquent, la description architecturale d'un système spécifie:

-sa **structure** : composants et interactions,
-son **comportement** : fonctionnalités et protocoles de communication, dynamisme, évolution,
-ses **propriétés globales** : propriétés fonctionnelles ou non fonctionnelles.

Les architectures de la plupart des systèmes logiciels ont longtemps été décrites de façon informelle (diagrammes où les composants logiciels sont des « boîtes » et les interactions des «lignes»). Cette description induit d'une part, des difficultés au niveau de leur interprétation et, d'autre part, des limitations [Abo et al. 95].

Aussi, des langages de description d'architectures (ADL) ont été définis. Ils offrent de nombreux avantages par rapport aux approches, semi-formel, comme par exemple la précision, la capacité à prouver des propriétés, et la possibilité d'analyser la structure architecturale. Ainsi spécifiées, les architectures logicielles jouent un rôle important dans au moins six aspects du développement logiciel [Gar00]:

1. **Compréhension** : les architectures logicielles rendent plus facile la compréhension du fonctionnement de systèmes complexes en les représentant à un haut niveau d'abstraction,

2. **Réutilisation** : les descriptions architecturales supportent la réutilisation à de multiples niveaux. Les travaux actuels, dans le domaine de la réutilisation, se concentrent généralement sur l'utilisation de bibliothèques de composants. La conception orientée architecture supporte, en plus, la réutilisation de composants complexes et des structures dans lesquelles ces composants peuvent être intégrés.
Ceci est prouvé par de nombreux travaux existants dans les domaines des *Domain Specific Software Architectures*, des architectures de référence, ou des patrons de conceptions [Bus et al. 96],

3. **Construction** : une description architecturale fournit un plan de développement en indiquant les composants principaux et les dépendances entre eux,

4. **Evolution** : l'architecture d'un système peut décrire la manière dont ce système est censé évoluer. La définition explicite des limites d'évolution d'un système permet de faciliter sa maintenance et d'estimer plus précisément les coûts des modifications. De plus, les descriptions architecturales distinguent les aspects fonctionnels des composants de la façon dont ces composants interagissent entre eux. Cette séparation permet de modifier facilement les mécanismes de connexion, ce qui favorise l'évolution en termes de performance, d'interopérabilité et de réutilisation,

5. **Analyse** : les descriptions architecturales fournissent des moyens d'analyse, tels que la vérification de la cohérence d'un système [AG94] [Luc et al. 95], la vérification de la conformité aux contraintes imposées par un style architectural [Abo et al. 93], la vérification de la conformité à des attributs qualité [Cle et al. 95], l'analyse de dépendance [Stafford et al. 93], ainsi que des analyses spécifiques au style à partir des architectures construites [CS93] [Mag et al. 95] [Gar et al. 94],

I.2 Notion de style architectural

La définition d'un style architectural selon [Abo et al. 93] est la suivante :

*Un **style architectural** permet de caractériser une famille de systèmes qui ont les mêmes propriétés structurelles et sémantiques.*

De ce fait, un style architectural précise les propriétés et les contraintes qui fixent les règles et les limites de construction de l'architecture.
L'objectif des styles architecturaux est de simplifier la conception des logiciels et la réutilisation, en capturant et en exploitant la connaissance utilisée pour concevoir un système [Mon et al. 97].

Un style architectural est moins contraignant et moins complet qu'une architecture spécifique. Il spécifie uniquement les contraintes les plus importantes, au niveau par exemple de la structure, du comportement, de l'utilisation des ressources des composants et des connecteurs [Rev02].

D'une façon générale, les styles architecturaux permettent à un développeur de réutiliser l'expérience concentrée de tous les concepteurs qui ont précédemment fait face à des problèmes similaires [KK99]. En outre, l'utilisation des styles architecturaux comporte des intérêts précis :
-elle favorise la réutilisation dès la conception du système [MG96],
-elle favorise la normalisation des familles d'architecture, ce qui facilite la compréhension de l'organisation d'un système,
-elle autorise l'utilisation d'analyses spécifiques au style concerné [CM96].

Les styles architecturaux ont longtemps été définis et utilisés comme des guides de conception informels, n'ayant pas de langages dédiés à leur exploitation. Les travaux menés, depuis lors, ont souligné l'intérêt de formaliser les styles [Abo et al. 95] et de nombreux langages dédiés à cette tâche ont été développés.

I.3 Les Langages de Description d'Architecture (ADLs)

Pour répondre à ces besoins, des langages de description d'architectures ont émergé ces dernières années. Ils permettent de décrire, de manière formelle, la structure des systèmes et fournissent différents outils permettant leur analyse, leur simulation, leur réalisation... Selon le formalisme utilisé par ces langages, ils aident à la construction d'architectures correctement structurées. S'il s'agit de langages strictement formels, des propriétés sur la structure et le comportement du système peuvent être prouvées.

Ainsi les langages de description d'architectures ont des objectifs différents, et possèdent un vocabulaire qui leur est propre. Un certain nombre de concepts sont pourtant communs à l'ensemble de ces langages. Ainsi les architectures décrites par ces langages sont fondées sur :
-un modèle de composant,
-un modèle de connecteur,
-un modèle de configuration.

Un ***composant*** est une unité de calcul ou de stockage de données dans une architecture. Ces éléments sont les constituants de base. Ils sont indépendants de la structure dans laquelle ils sont placés, et possèdent une interface permettant de communiquer avec cet environnement.

Un ***connecteur*** représente le protocole de communication entre différents composants. Il modélise leurs interactions et spécifie les règles qu'ils doivent respecter durant la communication. Les connecteurs ne sont pas toujours directement référencés dans les langages de description d'architectures.

Une ***configuration*** permet de connecter les composants par leur interface de communication. Selon les langages les composants peuvent, soit être connectés directement entre eux, soit passés explicitement par un connecteur. L'ensemble des composants d'un système et de leur interaction forme l'architecture du logiciel.

Ayant présenté les éléments communs aux architectures, nous pouvons donner la définition d'un langage de description d'architectures que nous avons retenu dans la littérature. Selon Medvidovic [Med00]:

Un langage de description d'architectures doit explicitement modéliser des composants, des connecteurs et leurs configurations ; de plus, pour être vraiment utilisable et utile, il doit fournir un support d'outils pour le développement et l'évolution des architectures.

I.3.1 Les concepts de base des Langages de Description d'Architecture (ADLs)

En 2000, N. Medvidovic et R Taylor proposent dans un article [MT00] un schéma de classification pour l'étude comparative d'une dizaine d'ADL. Leur approche et surtout leur classification donne une bonne idée des procédés de modélisation et du potentiel d'utilisation offert par les ADL. Précisons que l'article fait depuis référence en la matière.

Voici, résumées, les définitions de ces caractéristiques d'après l'article :

1. La modélisation des composants

Un composant, au sens architectural, est un lieu de traitement ou de stockage de données plus ou moins complexe. Ils vont de la simple procédure à l'application. Par définition tous les ADL décrivent les composants.

Le composant est décrit au minimum à travers son interface, sa description contient en général des éléments de modélisation additionnels:

L'interface : L'interface du composant constitue l'ensemble des points d'interactions entre le composant et son environnement. Elle spécifie précisément les services (message, variables, opérations) que le composant fournit. Globalement, l'interface décrit les fonctionnalités et les contraintes d'utilisation du composant.

Le type : Les types de composants sont des abstractions qui encapsulent leurs caractéristiques dans des blocs réutilisables. L'avantage d'un type est double : il peut être instancié de multiples fois, et il peut être étendu par un autre type. Les mécanismes de typage facilitent surtout la phase de conception.

La sémantique : La notion de sémantique définie dans l'article concerne la modélisation du comportement. Cette partie de la description fournit les données d'entrée aux procédures d'analyse.

Les contraintes : Une contrainte est une assertion ou une propriété du système (ou une de ses parties) qui ne doit pas être violée. La spécification des contraintes permet de respecter certains comportements voulus, de fixer des limites d'utilisation ou d'établir des relations de dépendance parmi ses éléments internes.

L'évolution : L'évolution d'un composant se traduit par des modifications de ses caractéristiques (interface, comportement, implémentation, etc.). Certains ADL offrent des mécanismes d'évolution par le recours au sous typage ou au raffinement.

Les propriétés non fonctionnelles : Les propriétés non fonctionnelles d'un composant (sécurité, sûreté, performance et portabilité) ne peuvent être déduites de la seule description du comportement. La spécification des propriétés non fonctionnelles des composants d'un système est nécessaire pour simuler son comportement à l'exécution, pour analyser et vérifier le respect de ses contraintes d'exécution, ou encore facilite la gestion du projet.

2. La modélisation des connecteurs

Le connecteur est un concept architectural dont la fonction est la modélisation des interactions entre les composants et des lois qui régissent ces interactions. A la différence des composants, les connecteurs n'ont pas forcément d'implémentations correspondantes dans le système. Les connecteurs sont de natures variées : service de routage, service de transport, variable partagée, table d'entrées, structure de données dynamique, séquence d'appels de procédure intégrée dans le code, paramètres d'initialisation, protocoles client/serveur, pipes, etc. La description du connecteur peut être enrichie par les mêmes éléments de modélisation que les composants :

L'interface : La description de l'interface des connecteurs assure une bonne connectivité entre les composants. Le connecteur n'ayant aucune fonction de traitement, les interfaces ne font qu'exporter tels quels les services attendus des composants vers les autres composants.

Le type : Les types de connecteurs sont des abstractions qui encapsulent les caractéristiques de communication, de coordination et de médiation des composants. Au niveau architectural, les interactions sont caractérisées par des protocoles complexes qu'il est avantageux de généraliser à des fins de réutilisation ou de normalisation.

La sémantique : La sémantique d'un connecteur est, à l'instar des composants, la modélisation de haut niveau de son comportement.

Les contraintes : Les contraintes permettent d'assurer le respect du comportement des connecteurs aux protocoles d'interactions, elles servent à établir les dépendances entre les connecteurs et imposent des limites d'usage, par exemple en limitant le nombre de composants qui interagissent.

L'évolution : L'évolution des connecteurs est à l'image de l'évolution des composants. Elle repose sur des mécanismes de sous typage, de raffinement ou de filtrage.

Les propriétés non fonctionnelles : De même que pour les composants, la spécification des propriétés non fonctionnelles des connecteurs peut s'avérer nécessaire pour la simulation ou la vérification du comportement du système à l'exécution, pour l'évaluation de la qualité de service, ou le dimensionnement des ressources.

3. La modélisation de la configuration

La configuration (topologie) est le graphe de connexion qui permet de décrire la structure de l'architecture. La description de la configuration par un ADL peut être qualifiée par trois types de caractéristiques : Les qualités propres à la description (compréhensibilité, compositionnabilité, raffinement et traçabilité, hétérogénéité du formalisme), les qualités du système (hétérogénéité des éléments, extensibilité, évolutivité, dynamisme) et les propriétés du système (dynamisme, contraintes, propriétés non-fonctionnelles).

La compréhensibilité : Pour permettre une bonne compréhension par les différents acteurs qui coopèrent et communiquent, un système complexe doit être décrit à un niveau d'abstraction élevé et au moyen d'une syntaxe simple et claire pour tous. En général, les ADL décrivent la topologie indépendamment de la description détaillée des composants et des connecteurs. La plupart permettent un affichage graphique facilement lisible

Le mécanisme de composition-décomposition : La capacité de composition permet de hiérarchiser la description suivant des niveaux de granularité. Une structure complexe peut ainsi être explicitement décrite à un niveau de détail donné ou être représentée par abstraction par un simple composant ou un simple connecteur de niveau supérieur.

Le raffinement et la traçabilité : De la spécification à la réalisation du système exécutable, l'architecture se raffine en passant par différents niveaux d'abstraction. Pour maintenir l'exactitude et la consistance de l'architecture à travers ces niveaux, les ADL doivent pouvoir spécifier et tracer ces évolutions.

La qualité d'hétérogénéité : Le développement de systèmes de grande taille s'appuie sur l'utilisation et la réutilisation d'éléments hétérogènes de par leurs niveaux de granularité, les formats de spécification, les langages de programmation utilisés, les plates formes d'exécution ou les protocoles d'interaction. A ce niveau, il est souhaitable que les langages soient "ouverts", et facilitent la spécification et le développement avec des composants et des connecteurs hétérogènes.

La capacité de croissance : L'architecture sert à gérer la complexité mais aussi la taille des grands systèmes. Les ADL qui le permettent, doivent supporter la spécification et le développement des systèmes de grande taille susceptibles de s'agrandir.

La capacité d'évolution : La conception de nouveaux systèmes s'effectue souvent à partir de versions antérieures dont on adapte ou améliore les fonctionnalités. L'évolution d'un système complexe peut représenter un coût important et plutôt que de reconcevoir le système en entier, il est préférable d'avoir des mécanismes qui simplifient l'évolution en permettant directement l'ajout, le remplacement, la suppression et la reconnexion des composants et des connecteurs.

Le dynamisme architectural : Certains systèmes peuvent nécessiter des modifications de leur configuration ou le changement d'éléments en cours d'exécution. C'est le cas de systèmes critiques ou des systèmes que l'on ne peut arrêter. Les ADL peuvent fournir des moyens de modélisation et d'évaluation des changements dynamiques de l'architecture et des moyens techniques pour les rendre effectifs.

Les contraintes de configuration : Les contraintes de configuration viennent compléter les contraintes définies au niveau des composants et des connecteurs. En général les contraintes globales dérivent ou dépendent des contraintes locales. C'est le cas par exemple de la performance globale qui dépend de la performance des éléments constituants.

Les propriétés non fonctionnelles : La spécification des propriétés non fonctionnelles de ce niveau peut être nécessaire pour la sélection de composants et de connecteurs appropriés, pour l'analyse et l'évaluation de la qualité de service, ou pour la gestion du projet.

4. Les caractéristiques d'exploitation

Les ADL ont été développés dans le but de modéliser l'architecture mais aussi de fournir des moyens pratiques de manipulation et de calcul. L'efficacité d'un langage est directement liée aux fonctionnalités offertes par les outils qui l'accompagnent. Medvidovic et Taylor ont sélectionnés six critères qui permettent de caractériser ce potentiel : L'aide active à la spécification, la représentation multiple, l'analyse, le raffinement formel, la génération de code et l'assemblage, la gestion du dynamisme architectural [Bol et al. 00] [CO01] [Cha02].

I.3.2 Description des principaux ADLs

Dans cette partie nous allons présenter différents ADLs. Pour chacun d'entre eux nous présenterons son objectif, ses éléments de base, et un exemple de client-serveur permettant d'illustrer ces éléments. Puis pour chaque langage nous présenterons ses aspects dynamiques. Enfin nous citerons pour chaque langage ces avantages inconvénients.

Les langages de description d'architecture sont des langages dits déclaratifs. Ils peuvent être classés en deux grandes familles. La première correspond aux langages qui privilégient la description des éléments de l'architecture et leur assemblage structurel, la seconde définit les langages qui se centrent sur la description de la configuration d'une architecture et sur la dynamique du système.

Nous allons dans cette section présenter les ADLs les plus connus, c'est-à-dire ACME, C2SADL, Darwin, Rapide, et Wright. Nous donnerons leurs avantages et leurs inconvénients.

1. ACME [Gar et al. 97]

i. Objectif du langage

ACME est un langage d'échanges d'architectures, permettant de supporter le «mapping» des spécifications d'un ADL à un autre. c'est-à-dire qu'il propose un format intermédiaire capable de spécifier une architecture dans un langage d'architectures donné, et de le

transcrire dans un langage d'architectures ciblé. L'objectif de ce langage est de permettre l'utilisation des différentes caractéristiques ainsi que les outils des ADLs existants.

Pour ce faire, il permet d'une part de rassembler le plus petit dénominateur commun des langages incluant les différents aspects que doivent comporter un ADL et, d'autre part d'incorporer les caractéristiques de l'ensemble des langages.

ii. Les éléments de base du langage

Même si ACME n'est pas un ADL, il en comporte certaines caractéristiques. Ainsi le langage pivot comporte les modèles de base d'un ADL défini dans l'introduction :
-le modèle de composant est référencé par ACME comme *composant*. Ces éléments possèdent des interfaces nommées des *ports*. Ces interfaces sont typées,
-le modèle de connecteur est référencé par ACME comme *connecteur*. Ces éléments possèdent des interfaces qui sont nommées *rôles*,
-la connexion entre les ports d'un composant et les rôles d'un connecteur est explicitée dans la configuration de l'architecture nommée *système*. Les attachements sont explicites.

iii. Illustration du langage par l'exemple du client-serveur

Bien qu'il soit essentiellement utilisé comme format intermédiaire, le langage ACME permet de décrire une architecture. Prenons l'exemple simple du client-serveur illustré par la figure ci-dessous.

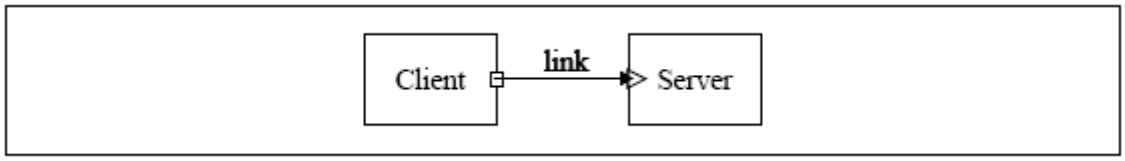


Figure1 - Diagramme représentant un système Client-Serveur en ACME

La spécification en ACME de ce système est le suivant :

```
System ClientServer = {
  Component client = {Port request}
 Component server = {Port reply}
 Connector link = {Rôle {reply, request}}
Attachment: {
 client.request to link.reply ;
 Server.reply to link.request }
  }
```

Le système *ClientServer* représente la configuration composée des éléments de l'architecture. Les composants du système sont le client *Client* et le serveur *Server*. Leurs interfaces sont respectivement *request* et *reply*. Il est à noter que les composants sont définis par un ensemble de ports. Chaque port identifie un point d'interaction entre le composant et son environnement. Un composant peut fournir plusieurs interfaces en utilisant différents types de ports. Le connecteur du système est *Link*, possédant un ensemble de rôles étant *reply* et *request*. L'assemblage des composants et du connecteur décrit la nature de leurs interactions.

Pour la translation d'un ADL à un autre, ACME utilise trois étapes :
- le langage de description source est traduit en ACME, tout en conservant ses spécifications sous forme d'annotations dans son propre langage,
- les annotations sont traduites vers le langage de destination,
- le langage ACME obtenu est transcrit vers le langage cible enrichi de propriétés.

iv. Avantages

ACME se rapproche plus d'un outil que d'un langage. En effet, il ne propose pas de nouveau formalisme ou de nouveau concept. Cependant, il sert d'intégrateur et de pivot à d'autres ADLs. Il fournit ainsi un moyen simple et efficace :
- d'unifier les concepts proposés par les ADLs existants ; ces concepts sont essentiellement liés à la spécification structurelle; il s'agit des éléments suivants : le composant, le connecteur, le système, le port le rôle, la représentation et la carte de représentation ;
- d'intégrer des notions d'ADLs existants plus spécifiques à un domaine ou à la spécification du comportement d'un composant,
- de réutiliser ou de stocker des éléments définis antérieurement comme par exemple les gabarits de conception ou les styles d'architecture.

ACME supporte aussi la définition des styles et permet d'assurer le respect des contraintes de conception à l'aide de ses outils. ARMANI [Mon98] est une extension d'ACME conçu pour modéliser les contraintes architecturales. Dynamic ACME [Wil01] est une extension permettant de décrire la dynamicité des architectures.

v. Inconvénients

ACME ne fournit aucun moyen de spécifier de manière simple et claire la dynamique d'un système; en effet, le système est décrit comme un ensemble d'instances de composants liées entre eux par des connecteurs. Cependant cette description de l'assemblage reste statique. En effet il est utilisé pour décrire des systèmes fermés, c'est-à-dire des systèmes où tous les éléments ont été complètement spécifiés.

De plus, même si ACME est plus proche d'un outil que d'un langage, il ne fournit pas de moyen automatique pour raffiner la spécification d'application et n'encourage pas une description de l'application par une approche modulaire. Ainsi, ACME s'appuie sur la dynamicité des autres ADL mais ne propose rien [Gar et al. 97].

2. C2SADEL [Med 96] [Med et al. 99]

i. Objectif du langage

C2SADEL (Software Architecture Description and Evolution Language) est un ADL dont l'objectif est de décrire des architectures de systèmes hautement distribués, évolutifs, et dynamiques. Les systèmes qu'il souhaite spécifier sont complexes, multi-langages, multi-plates-formes et ayant une longue exécution. Pour pouvoir être économiquement viables ces systèmes, dont la maintenance est estimée à soixante pour cent du coût total de développement, doivent être évolutifs. C'est pourquoi C2SADEL propose un langage permettant le changement de topologie d'une architecture.

ii. Eléments de base du langage

Comme tout ADL, C2SADEL comporte les modèles de base définis dans l'introduction:

-le modèle de composant est référencé par C2SADEL comme *composant*. L'interface *interface* est définie par un *top_port* ou/et un *bottom_port*. Par la sortie d'un *top_port* le composant émet des *messages* et les réceptionne par son entrée. Ces messages sont typés.

Or pour que les interfaces puissent manipuler n'importe quels messages, ces derniers sont placés dans des *enveloppes* cachant leur type. Un composant possède un comportement qui lui est associé. Ce comportement possède un invariant (spécifiant des propriétés qui doivent être vraies durant l'ensemble des états du composant) et un ensemble d'opérations pouvant être fourni ou requis.
- le modèle de connecteur est référencé par C2SADEL comme *connecteur*. Par contre, les interfaces des connecteurs sont différentes de celles des composants : elles sont génériques.

En effet les connecteurs sont indifférents des types de données qu'ils manipulent puisque celles-ci sont placées dans des enveloppes masquant leur type. Leur principale tâche est donc de coordonner la communication entre les composants. De plus un connecteur C2SADEL peut supporter un nombre arbitraire de composants. Une des caractéristiques de C2SADEL est qu'une interface d'un connecteur est déterminée par les interfaces (potentiellement dynamiques) des composants qui communiquent à travers elle. C2SADEL base les types de connecteurs sur les protocoles d'interactions.
- la connexion des composants, via les connecteurs, est explicitée dans la configuration de l'architecture nommée *architecture*. Ces attachements sont explicites.

iii. Illustration du langage par l'exemple du client-serveur

C2SADEL possède une notation graphique, l'exemple du client-serveur ci-dessous utilise cette notation.

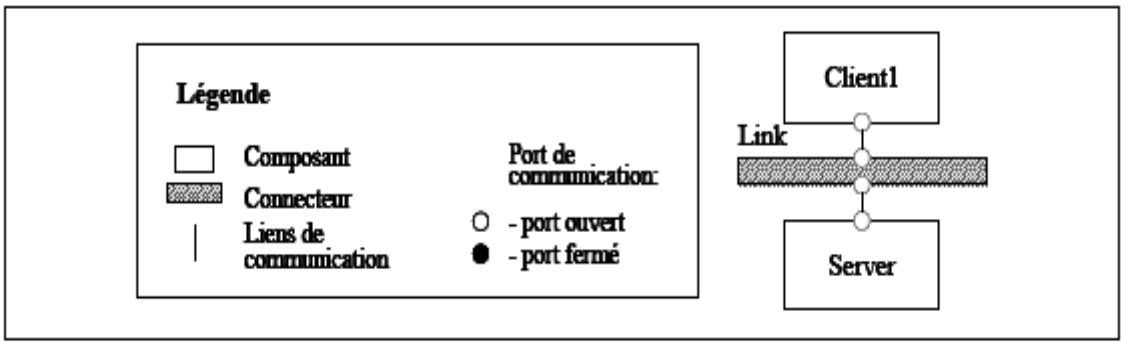


Figure 2 - Diagramme d'architecture client-server dans le style C2

La spécification en C2SADEL de ce système est le suivant :

```
architecture ClientServer is {
components {
Client;
Server;        }
connectors {
  Link;        }
topology {
 connector Link connections {
   top_port {
```

```
      Client filter no_filtering;
}
  bottom_port {
     Server filter no_filtering;
} } } }
```

L'architecture possède deux composants *Client* et *Server*, et un connecteur *Link*. La topologie de cette architecture est définie par la connexion au connecteur *Link* des deux composants. Le composant *Client* est connecté au connecteur par son *top_port* et le *server* par son *bottom_port*.

C2SADEL permet de spécifier les attachements et les composants dynamiques. Par contre ces changements doivent être planifiés avant la définition de l'architecture. De plus, dans la littérature, ni l'origine de la création ou de la suppression de ces composants, ni les moyens de gestion de ces composants une fois créés ne sont spécifiés.

iv. Avantages

L'un des avantages de C2 est de favoriser l'intégration de composants existants et hétérogènes:
- en rendant les composants partiellement indépendants de l'architecture ; en effet, un composant n'est dépendant que du « haut » de l'architecture qui lui fournit les services requis ; il est totalement indépendant du « bas » de l'architecture et de son évolution ;
- grâce à l'architecture interne d'un composant qui permet d'intégrer un composant existant en tant qu'objet interne ;

La description de la configuration présente l'avantage de séparer la description statique et dynamique du système et celle de l'architecture logicielle..
Le langage C2-SADL fournit également un moyen d'exprimer la dynamique d'une application, c'est-à-dire
- l'ajout ou la suppression d'un composant dans l'architecture logicielle,
- la création d'un lien d'un composant avec son implantation ou la substitution d'une implantation avec une autre.

Il fournit en outre une base formelle pour le raffinement architectural ;

v. Inconvénients

Parmi les inconvénients de C2 :

La notion de connecteur, bien que celle-ci soit définie de manière explicite, s'apparente plus à un filtre d'événements asynchrones et donc à une notion d'implantation (médiateur ou bus) qu'à un patron de connexion réutilisable et donc qu'à une notion de conception. La sémantique d'un connecteur n'est pas définie de manière précise ; en effet on ne peut pas spécifier des rôles spécifiques joués par les composants connectés. Il n'y a que deux rôles possibles : celui qui envoie des messages de manière asynchrone et celui qui en reçoit de la même manière. C2 ne propose pas de moyen précis de définir un style d'architecture. Ainsi, il est difficile avec C2 de créer et de réutiliser des patrons de conception récurrents à des applications de même style.

C2SADEL permet de spécifier les attachements et les composants dynamiques. Par contre ces changements doivent être planifiés avant la définition de l'architecture. De plus,

dans la littérature, ni l'origine de la création ou de la suppression de ces composants, ni les moyens de gestion de ces composants une fois créés ne sont spécifiés.

3. Darwin [MK 96]

i. Objectif du langage

Darwin est un langage de configuration défini pour des systèmes parallèles et distribués. Il fournit une structure de composants avec des correspondances (binding) dynamiques. Cet aspect dynamique est soutenu par la base formelle du langage fondé sur un langage de processus : le PI-Calcul. Ce dernier est conçu pour modéliser les calculs concurrents constituant les processus qui interagissent et dont la configuration est changeante. Ainsi Darwin permet de modéliser des architectures de systèmes hautement distribués, dont le dynamisme est guidé par des soutiens strictement formels.

ii. Eléments de base du langage

Les principales abstractions gérées par Darwin sont les composants. Darwin les décrit en termes de services qu'ils fournissent et qu'ils requièrent pour leur permettre de communiquer avec d'autres composants. Deux types de composants existent : les primitifs et les composites.

Ainsi Darwin comporte les modèles de base définis dans l'introduction :
-le modèle de composant est référencé par Darwin comme *composant* (primitif). Ces éléments possèdent des interfaces permettant de fournir *provide* ou de requérir *require* des *canaux de communication* par lesquels sont respectivement reçues ou émises des valeurs,

-le modèle de connecteur n'est pas référencé par Darwin. Par contre Darwin permet leur abstraction à travers des comportements de connexions complexes dans des *composants connecteur*,
-les connexions des composants s'effectuent en liant leurs ports. Elles sont explicitées dans un *composant (*composite*)* par des attachements explicites. Les modèles d'interaction et les propriétés de composition des composants d'un composite sont exprimés en Pi-Calcul.

iii. Illustration du langage par l'exemple du client-serveur

Darwin possède une notation graphique, l'exemple du client-serveur ci-dessous utilise cette notation.

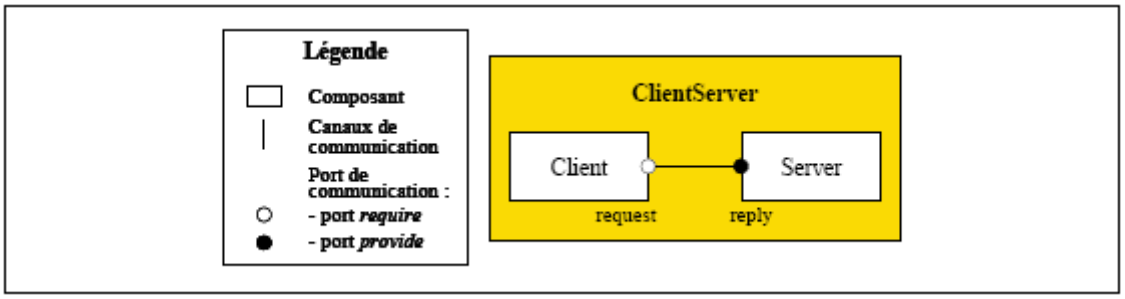


Figure 3 - Diagramme représentant un système Client-Serveur en Darwin

La spécification en Darwin de ce système est le suivant :

```
component ClientServer{
inst client : Client ;
inst server : Server ;
bind client.request --server.reply ;
}
```

Le composite englobant le système du client-serveur est un composant ne possédant pas d'interface.
L'opérateur *inst* permet de déclarer les instances *client* et *server* des composants *Client* et *Server*.
L'opérateur *bind* relie, à l'aide du constructeur --, un port requis en partie gauche avec un port fourni en partie droite.
Il ne propose pas de moyen de les gérer une fois créés. En effet pour les composants d'un autre type, un composant dynamique n'est pas accessible, ce qui limite leur utilisation. De plus Darwin ne permet pas leur suppression.

iv. Avantages

L'originalité de Darwin est de fournir un moyen unique pour décrire de manière explicite les communications entre composants et les schémas d'instanciations dynamiques des composants (grâce à la réplication de composants).

Ce langage considère un élément de l'architecture (le composant) comme une entité pouvant être instanciée. Ceci a pour conséquence de spécifier de manière plus précise les interactions entre composants. En effet, il est, par exemple, utile de décrire qu'une instance de composant est connectée à plusieurs instances d'un autre composant pendant l'exécution ou de définir un paramètre dont la valeur sera déterminée pendant l'exécution et qui exprime la cardinalité des liens entre composants.

Le langage de Darwin fournit une syntaxe riche permettant de décrire plus finement les interactions entre composants en détaillant la collaboration.

v. Inconvénients

Parmi les inconvénients de Darwin, on peut citer le fait qu'un composant n'est associé qu'à une seule sémantique qui est le processus. Ainsi, un composant ne permet pas d'exprimer un autre élément comme un fichier ou une mémoire partagée.
Enfin, la description de la dynamique de l'application est limitée. Par exemple, il n'est pas possible de supprimer des composants dynamiquement ni de permettre à un composant primitif de communiquer avec des composants dynamiquement créés.

4. Rapide [Luk et al. 95] [Tea97]

i. Objectif du langage

Rapide est un langage orienté objet basé sur des événements concurrents. Il est spécifiquement conçu pour prototyper des architectures dans des systèmes distribués. Ainsi il autorise la simulation et l'analyse du comportement des architectures d'un système, très tôt, dans sa phase de développement.

ii. Eléments de base du langage

Rapide comporte certains modèles de base définis dans l'introduction :
-le modèle de composant est référencé par Rapide comme *composant*. Leur interface possède des *constituants*. Ceux ci sont créés à partir de constructeurs permettant de définir des modèles de communication entre les composants. Le constructeur *action* spécifie une communication asynchrone et le constructeur *functions* une communication synchrone.

Chacun de ces constructeurs peut être utilisé avec les mots clés *extern* ou *public* qui permettent de définir respectivement un port de sortie ou un port d'entrée,
-le modèle de connecteur n'est pas référencé par Rapide. Par contre Rapide permet leur abstraction à travers des comportements de connexions complexes dans des *composants connecteur*. Les connexions des composants s'effectuent en liant les *constituents extern* et *public* de leurs interfaces. Elles sont explicitées dans une architecture *architecture*. Elle contient, en plus de l'ensemble de règles de connexions, un ensemble de contraintes.

iii. Illustration du langage par l'exemple du client-serveur

La spécification en Rapide de l'exemple du client-serveur est donnée ci-dessous.

```
architecture ClientServer is
...
//Déclaration des instances des composants de l'application susceptible d'exister
c : Client ; //création d'une instance de Client
s : Server ; // création d'une instance de Server
service : Service ; //création d'un paramètre de type Service
// Règle d'interconnexion
connect
c.request (service) to s.reply (service) ; // Si un client transmet un paramètre de type Service
// alors l'événement est transmis au serveur avec
// ce paramètre.
end ClientServer
```

L'architecture *Client-serveur* contient la déclaration des instances *c* et *s* des composants *Client* et *Server*, ainsi que la donnée *service* transmise durant leur communication. Toutes les instances sont représentées par des variables. L'architecture contient également les règles d'interconnexion entre les composants. Une règle est composée d'une partie droite et d'une partie gauche. La partie gauche contient l'expression d'événement qui doit être vérifiée avant que les événements contenus dans la partie droite soient déclenchés. Dans l'exemple quand *c* génère un événement *request* alors *s* observe un événement *reply* avec la même donnée *service*.

iv. Avantages

Il permet de simuler les architectures et propose des outils pour analyser les résultats de ses simulations. Il ne présente pas de mécanisme propre à la formalisation des styles

L'avantage de Rapide est de permettre une expression forte de la dynamicité d'une application avec par exemple la création, la suppression, la description plus fine du schéma d'instanciation du composant.

v. Inconvénients

Même si les interactions entre composants peuvent être spécifiées de manière statique et dynamique, le langage ne fournit pas un moyen de typer un connecteur. En effet, un composant est décrit par une interface ; il est donc typé et son interface peut être réutilisée. Par contre, un connecteur n'est pas décrit de manière explicite. Ceci ne favorise pas la réutilisation des connecteurs.
Il ne présente pas de mécanisme propre à la formalisation des styles

5. Dynamic Wright [All 97] [All et al. 97]

i. Objectif du langage

Dynamic Wright permet de spécifier et d'analyser le comportement dynamique des systèmes concurrents. Il s'intéresse plus particulièrement aux interactions entre les composants architecturaux (plus précisément analyse de deadlock). Ces analyses sont possibles grâce à la formalisation des interactions dans le langage de processus CSP. Ce dernier ne permet de modéliser que des configurations statiques de processus, ce qui est un frein à la dynamicité de dynamique Wright.

ii. Les éléments de base du langage

Comme tout ADL, Dynamic Wright comporte les modèles de base définis dans l'introduction:
-le modèle de composant est référencé par Dynamic Wright comme *composant*. Les interfaces de ces composants sont référencées comme *port*. Leur sémantique d'interaction est spécifiée en CSP,
-le modèle de connecteur est référencé par Dynamic Wright comme *connector*. Ces éléments possèdent des interfaces nommées *rôles*,
-la connexion entre les ports d'un composant et les rôles d'un connecteur est explicitée dans la configuration de l'architecture nommée *configuration*. Les attachements entre les ports d'un composant et les rôles d'un connecteur sont explicites.

iii. Illustration du langage par l'exemple du client-serveur

Dynamic Wright ne propose pas de notation graphique de description d'architectures. La spécification en Dynamic Wright de l'exemple du client-serveur est donnée ci-dessous.

Configuration ClientServer
Instances
C : Client ;
L: Link;
S: Server
Attachments
Client.request as link.reply;
Server.reply as link.request
End Configuration

Dans la configuration sont déclarées les instances *C* et *S* des composants de type *Client* et *Server*, ainsi que l'instance *L* du connecteur de type *Link*. Puis dans la partie *attachment* sont déclarés les attachements entre les instances de composants et les instances de connecteurs. Dans l'exemple, le port *request* de l'instance *C* est attaché au rôle *reply* de l'instance *L* et le port *reply* de l'instance *S* est attaché au rôle *request* de l'instance *L*.

iv. Avantages

Le premier avantage de Wright est de fournir un langage formel (CSP) pour la spécification des composants et des connecteurs. Ainsi, les différents acteurs (concepteur, architecte, développeur) d'un projet peuvent communiquer sans ambiguïté leurs points de vue sur l'architecture d'une application.

Un autre point essentiel de cet ADL est qu'il sépare la notion de composant et de connecteur en proposant un modèle de type de composant et de type de connecteur. Ainsi, un composant peut être spécifié de manière indépendante des connecteurs, ce qui le rend nécessairement plus indépendant par rapport à son contexte d'exécution. Le type de connecteur est considéré comme un patron d'interconnexion et peut alors être réutilisé plusieurs fois dans une même architecture ou dans des architectures différentes.

Wright permet de spécifier de manière abstraite et formelle le comportement (sémantique liée à la dynamique) des composants, des connecteurs et de l'architecture. Pour les composants, il définit le comportement de chaque port d'une part et, d'autre part, le comportement global du composant avec ses ports (*computation*). Il permet également la spécification des connecteurs, c'est-à-dire le comportement de chaque rôle du connecteur d'une part et, d'autre part, le comportement global entre tous les rôles du connecteur (*glue*).

Pour l'architecture, le comportement des instances de composants et de connecteurs est défini ainsi que leur assemblage de manière dynamique; en effet, Wright offre la possibilité de créer et détruire de nouvelles instances.

v. Inconvénients

Le premier inconvénient de Wright est qu'il est difficile à assimiler. En effet, l'outil de spécification (CSP couplé à Wright) n'est pas facile à comprendre pour un programmeur débutant. Il peut donc s'avérer inefficace lorsque les délais d'un projet sont courts ou lorsque les compétences font défaut.

Finalement, le langage Wright ne permet pas de spécifier les contraintes non fonctionnelles séparément de la spécification fonctionnelle de l'architecture. Ainsi, les contraintes fonctionnelles et non fonctionnelles sont exprimées de la même manière et sans distinction.

Conclusion

Dans ce chapitre, nous avons présenté différents langages.

Parmi les points forts de ces ADLs, nous citons :

-La possibilité de description hiérarchique des composants. En effet, un ensemble de composants peut être vu comme étant lui-même un composant, ce qui permet une description récursive et facilite la réutilisation.
-Les interactions sont décrites de façon explicite. Tous les ADLs proposent au moins une définition d'interconnexion syntaxique. Les connecteurs sont vus comme des entités de première classe.
-Certains ADLs travaillent sur la notion de type, classe et instance. Il nous semble important de travailler à ce niveau là, puisque nous souhaitons détailler le plus possible l'assemblage et faciliter la réutilisation des éléments.
-Certains ADLs proposent la définition de style d'architecture. Ces styles facilitent la définition de modèle d'architecture.

Les points faibles rencontrés dans les ADLs sont de plusieurs ordres. On peut citer les points suivants :
-L'approche proposée par la plupart des ADLs est plutôt structurale, c'est-à-dire statique. La dynamicité, la description du comportement individuel des composants, ou encore du comportement global de l'application ne sont pas ou peu exprimées.
- Aucun ADL ne propose de démarche associée complète. Le langage et son utilisation reste bien souvent à la discrétion de l'architecte et des concepteurs.
- Finalement, plusieurs langages sont proches de la réalisation. Ils manquent d'abstraction. En effet, plusieurs langages sont qualifiés comme étant des langages de configuration. Il est difficile sur ces langages de proposer des outils de vérification de propriétés.

En conclusion, les Langages de Description d'Architecture (ADL) que nous avons présentés, offrent un cadre formel mais qui ne convient pas pour le problème que nous voulons résoudre: les liens dynamiques ne sont pas pris en compte, et, surtout, le comportement du composant est nécessaire pour assurer que la configuration de composants ne provoquera pas d'erreurs.

Le pouvoir d'expression d'un langage est une des choses primordiales à étudier pour notre cas. Le fait de pouvoir représenter les composants d'un point de vue structurel et comportemental est une chose importante. Mais le fait de pouvoir les réutiliser, les vérifier que cette représentation est correcte par rapport au modèle choisi est aussi important.
Aussi, il faux que de nombreuses règles structurelles devaient être respectées. Il faut donc que le langage nous offre la possibilité de spécifier l'ensemble de ces règles de construction.

Pour pouvoir répondre pleinement à cette problématique, l'ADL qui correspondra le mieux à doit vérifier les points incontournables:
-le pouvoir d'expression du langage,
-la notion de style architectural,
-la vérification de propriété,
-la réutilisation,
-les environnements de développement.

Ceci constitue, un des objectifs de notre travail. Nous allons essayer de proposer un langage de description d'architecture, que nous nommons LOTOS-ADL (chapitre 8).

Dans le prochain chapitre nous nous intéressons au langage LOTOS. Nous présentons ce langage et expliquons ses mécanismes.

CHAPITRE 4

Présentation du langage LOTOS

Introduction

Ce chapitre présente les notions syntaxiques et sémantiques du langage LOTOS nécessaires à la compréhension de la suite de ce document. Ces définitions sont suffisantes, de sorte que la connaissance de la norme LOTOS [ISO87] ne constitue pas un pré-requis obligatoire. Une présentation illustrée avec des exemples de LOTOS est fournie par l'annexe A.

De la définition formelle du langage LOTOS [ISO87] ne sont repris que les aspects strictement indispensables à notre thèse. C'est pourquoi les notions de sémantique des types abstraits algébriques ne sont pas traitées ici. Dans ce chapitre, nous présentons l'aspect contrôle (défini par Basic LOTOS).

LOTOS [Bol89] (abréviation de Language Of Temporal Ordering Specification) est une technique de description formelle, normalisé ISO. Il s'appuie sur langage CCS de Milner [Mil89] étendu par un mécanisme de synchronisation multiple hérité de CSP [Hoa85] pour la partie comportementale ; la partie description des structures de données est inspirée d'ACT-ONE [EM85], un formalisme de description de types de données abstraits algébriques.

Le concept sous-jacent à LOTOS est que tous système peut être spécifié en exprimant les relations qui existent entre les interactions constituant le comportement observable des composants du système. En LOTOS, un système est vu comme un processus, qui peut être constitué de sous-processus, un sous-processus étant un processus lui-même. Une spécification LOTOS décrit ainsi un système en hiérarchie de processus. Un processus représente un une entité capable de réaliser des actions internes (non observable) et d'interagir avec d'autre processus qui forment son environnement.

Les définitions de processus sont exprimées à partir d'un ensemble réduit d'opérateurs donnant la possibilité d'exprimer des comportements aussi complexes que l'on désire.

I.1 Présentation du Basic LOTOS:

Nous appelons Basic LOTOS, le sous-ensemble de LOTOS où les processus interagissent entre eux par synchronisation pur. En basic LOTOS les actions sont identiques aux portes de synchronisation des processus. Les principaux opérateurs sont listés dans la figure 1.

Opérateur		Notation	Description informelle
Inaction		`stop`	Processus de base n'interagissant pas avec son environnement
Terminaison avec succès		`exit`	Processus qui se termine (action δ) et se transforme en `stop`.
Préfixage par une action	non observable	$i;P$	Processus qui réalise l'action i ou g, puis se transforme en P.
	observable	$g;P$	
Choix non-déterministe		$P_1 \Box P_2$	Processus qui se transforme en P_1 ou en P_2 suivant l'environnement.
Composition parallèle	cas général	$P_1 \mid [g_1,\ldots g_n] \mid P_2$	P_1 et P_2 s'exécutent en parallèle et se synchronisent sur les portes $g_1,\ldots g_n$ et δ
	asynchrone	$P_1 \mid\mid\mid P_2$	P_1 et P_2 s'exécutent en parallèle sans se synchroniser (sauf sur δ)
	synchrone	$P_1 \mid\mid P_2$	P_1 et P_2 s'exécutent en parallèle et se synchronisent sur chaque porte visible
Intériorisation		`hide` $g_1,\ldots g_n$ `in` P	Les actions $g_1,\ldots g_n$ sont cachées à l'environnement de P et deviennent des actions internes.
Composition séquentielle		$P_1 >> P_2$	P_2 est activé dès que P_1 se termine.
Préemption		$P_1 [> P_2$	P_2 peut interrompre P_1 tant que P_1 ne s'est pas terminé.

Figure 1- Principaux opérateurs de Basic LOTOS

La syntaxe formelle de Basic LOTOS est donnée par :

Processus X [g1, g2, gn]:= P **endprocess**

P::= **stop** / **exit** / **X(L)** / ***i;*** P / ***g;*** P / P**[ ]**P
/ *P* **[**(L)**]** / **hide L in P** / P>>P / P**[**>P

I.2 Les spécifications LOTOS

Une spécification LOTOS est un texte ASCII qui regroupe un ensemble de définitions de processus (encadrées par les mots-clés « **process** » et « **endproc** »).

LOTOS possède une structure de blocs imbriqués : chaque définition de processus peut contenir des définitions de processus ou de types qui lui sont locales ; en revanche une définition de type ne peut pas englober d'autres définitions de types ni de processus. Au plus haut niveau, une spécification LOTOS se comporte comme une définition de processus. Pour rendre cette introduction à LOTOS plus accessible, les exemples ont été délibérément très simples.

I.3 La partie contrôle de LOTOS

Les structures de contrôle dont dispose LOTOS sont issues des recherches sur les algèbres de processus, inspirées notamment par les travaux de Hoare sur CSP [Hoa78] et Milner sur CCS [Mil80].

Syntaxiquement parlant, le contrôle est décrit en LOTOS par les termes d'une algèbre : ce sont des expressions mathématiques, appelées comportements, construites à partir des opérateurs de contrôle qui sont fournis par LOTOS.

Sémantiquement parlant, chaque comportement représente un automate d'états finis ou infinis, les règles de traduction des comportements en automates constituant la sémantique dynamique de LOTOS.

I.3.1 Portes et signaux

Plus concrètement, LOTOS permet de décrire des comportements qui s'exécutent en parallèle et qui correspondent par rendez-vous. En LOTOS, le rendez-vous est le moyen unique pour exprimer la synchronisation et la communication.
On appelle signal ou action la proposition effectuée par un comportement qui désire participer à un rendez-vous. Un signal se compose d'une porte et une liste d'offres pour l'émission ou la réception de valeurs typées (dans la terminologie ISO pour LOTOS, le type d'une valeur ou d'une variable porte le nom de sorte). Par exemple, le signal :

OUTPUT !2 !X1 !X2

signifie que l'on cherche à émettre simultanément, sur la porte OUTPUT, les trois valeurs 2, X1 et X2. De même, le signal :

INPUT ?A:REAL ?B:REAL ?C:REAL

indique que l'on s'attend à recevoir simultanément, sur la porte INPUT, trois valeurs réelles qui seront rangées dans les variables A, B et C. Enfin on peut combiner émissions et réceptions au cours d'un même rendez-vous et spécifier des conditions sur les valeurs émises ou reçues. C'est ainsi que le signal :

EXCHANGE !2 ?X1:REAL ?X2:REAL [X1 < X2]

exprime que, sur la porte EXCHANGE, on désire émettre la valeur 2 tout en recevant deux valeurs réelles X1 et X2. En outre le rendez-vous est conditionné par une garde booléenne : il ne sera accepté que si la valeur X1 est plus petite que la valeur X2.

I.3.2 Opérateurs séquentiels

Tout programme LOTOS décrit un automate dont chaque transition correspond à un signal ; cet automate peut aussi être considéré comme un graphe. L'opérateur « **;** » permet l'exécution séquentielle d'un comportement.

Voici un exemple assez simple d'automate qui, après avoir reçu une valeur réelle X1sur la porte INPUT, émet la valeur X2 sur la porte OUTPUT:

INPUT ?X1:REAL **;** OUTPUT !X2

I.3.3 Opérateurs sur les valeurs

Il existe plusieurs opérateurs LOTOS permettant de manipuler les valeurs et de créer des variables.
L'opérateur « **[….] ->** … » permet de conditionner l'exécution d'un comportement par une garde booléenne ; il s'apparente une instruction « if…then …».
L'opérateur « **choice [ ]** » permet de choisir, de manière non-déterministe, une valeur dans un domaine.
L'opérateur « **let …in…** » permet de donner un nom à une expression en définissant une variable.

L'exemple suivant, consacré au calcul des racines d'un polynôme du 2nd degré, illustre l'emploi de ces opérateurs:

```
INPUT ?A:REAL ?B:REAL ?C:REAL;
   (
   let DELTA:REAL = (B ^ 2) - (4 * A * C) in
      (
      [DELTA > 0] ->
         (
         let X1:REAL = (-B - sqrt (DELTA)) / (2 * A) in
            let X2:REAL = (-B + sqrt (DELTA)) / (2 * A) in
              OUTPUT !2 !X1 !X2;
                 stop
         )
      []
      [DELTA = 0] ->
         (
         let X0:REAL = -B / (2 * A) in
            OUTPUT !1 !X0 !X0;
              stop
         )
      []
      [DELTA < 0] ->
         (
         choice X1, X2:REAL []
           OUTPUT !0 !X1 !X2;
              stop
         )
      )
   )
```

I.3.4 Opérateurs parallèles

Ce qu'on a vu jusqu'à présent de LOTOS permet de décrire des programmes séquentiels, dans un style fonctionnel certes, mais relativement proche de la programmation algorithmique classique. Mais cela seul ne suffirait pas à faire de LOTOS un langage adapté aux systèmes répartis.

Comme les aspects séquentiels, les aspects parallèles s'expriment par des opérateurs.

En LOTOS l'opérateur « ||| » exprime l'exécution simultanée de deux comportements sans aucune synchronisation entre eux (sauf en ce qui concerne la terminaison par « exit » (le processus qui se termine le premier attend l'autre).

Pour permettre aux comportements concurrents de se synchroniser et de communiquer, LOTOS possède un opérateur noté « | **[G1,…,Gn]** | », qui indique que les deux comportements auxquels il s'applique doivent fonctionner en parallèle, tout en se synchronisant par rendez-vous sur les portes G1,….,Gn et sur ces portes-ci exclusivement (en outre la terminaison par « exit » est toujours synchrone). De même LOTOS admet l'opérateur « >> » , qui exprime la composition séquentielle.

Les opérateurs parallèles de LOTOS offrent un moyen naturel pour décrire l'architecture d'un ensemble de processus communicants. On peut facilement exprimer le schéma client-serveur, où plusieurs processus clients sont en concurrence pour l'accès à une ressource fournie par un processus serveur:

```
  (
CLIENT [G]
    |||
CLIENT [G]
  ||| ... |||
CLIENT [G]
  )
 | [G] |
SERVER [G]
```

I.4 L'évolution de la théorie du LOTOS.

Nous allons donner un bref aperçu de la situation de la recherche sur le langage LOTOS, en ce qui concerne les travaux théoriques, le développement d'outils logiciels, et les applications. Il est difficile de fournir un compte rendu complet des activités concernant le LOTOS, car ce langage est utilisé par un grand nombre d'équipes sur plusieurs continents.

Nous nous limitons à citer les travaux les plus intéressants desquels nous sommes au courant, surtout travaux publiés. De plus, nous ne parlons que des travaux qui parlent du langage dans sa forme courante.

I.4.1 Origines théoriques.

Robin Milner est le premier à avoir utilisé une méthode algébrique pour décrire les comportements asynchrones des systèmes communicants comme termes d'un calcul.

Différentes notions d'équivalences entre comportements ont été définies dans ce calcul. Ce travail a mené Milner à la découverte de CCS [Mil89], un langage algébrique de processus.

La sémantique de ce langage est décrite en fonction de règles d'inférences. Ces règles rendent possible la définition d'un ensemble de relations d'équivalences entre comportements et un calcul qui permet de raisonner sur les processus en se basant sur un ensemble d'équations. D'où le lien établi, pour la première fois, entre une théorie

équationnelle et une relation d'équivalence entre comportements, notamment l'équivalence observationnelle, en se basant sur une sémantique opérationnelle.

I.4.2 Modèles sémantiques.

Les opérateurs de LOTOS ont été choisis de telle façon qu'il est possible de démontrer un ensemble de règles algébriques sur ces derniers, comme celles de CCS.
Cependant, il faut noter que ni CCS ni LOTOS n'offre une théorie algébrique complète qui permet le traitement et l'interprétation de tous les comportements possibles.

Le fait que LOTOS utilise l'action interne i, équivalente de « tau » dans CCS, pour représenter les comportements non-déterministes et que le modèle sémantique de LOTOS a hérité du caractère opérationnel de CCS, a permis l'utilisation du concept d'équivalence de bisimulation pour dériver ces règles algébriques et avoir une définition équationnelle des comportements LOTOS.

I.4.3 Méthodes de preuve.

En général, on est intéressé à prouver deux genres de propriétés des systèmes distribués: sureté (safety) et vivacité (liveness). Les propriétés de solidité décrivent les comportements désirés des systèmes et les propriétés de vivacité décrivent des comportements non-désirés. D'où le besoin d.une ou plusieurs méthodes qui permettent de démontrer qu'un processus LOTOS donné satisfait une ou plusieurs propriétés de ce genre. De plus, la conception des systèmes distribués se fait sur plusieurs niveaux d'abstractions. D'où le besoin de démontrer qu'un processus LOTOS est équivalent à un autre processus LOTOS plus élaboré afin de valider sa conception.

La méthode de preuve la plus utilisée pour prouver qu'un processus LOTOS satisfait certaines propriétés est basée sur l'équivalence de bisimulation. Un rapport sur les algorithmes et outils associés à cette méthode peut être trouvé dans [Bol89]. En utilisant la bisimulation, on peut prouver qu'un processus LOTOS est équivalent à un autre processus LOTOS.

Une approche similaire est proposée dans [Fan90]. Elle utilise la logique temporelle pour exprimer les propriétés des processus et se base sur une sémantique temporelle des comportements LOTOS. L'application de cette méthode est encore limitée, vu sa complexité.

I.4.4 Les outils de LOTOS.

Depuis le début des travaux sur le langage LOTOS, un accent a été mis sur l'élaboration d'outils. Bien que la plupart de ceux-ci ont été produits sous forme de prototypes, on peut y constater une certaine qualité et la rigueur (surtout sémantique) de leur analyse et de leur mise en œuvre. On peut classifier les outils disponibles en quatre catégories: - les outils de simulation et d'exécution, - les outils de transformation, - les outils de validation, - les outils de représentation graphique.

1. Outils de simulation et d'exécution.

Ces outils font en général l'analyse syntaxique, la vérification de la sémantique statique et la génération du code intermédiaire en vue d.une exécution. Les méthodes d'implantation de ces outils reposent soit sur des fonctions du système d'exploitation sur lequel s.est effectuée cette implantation (généralement Unix) soit sur des modèles d'exécution pouvant être implantés sur toutes sortes de machines (modèles abstraits, langages spécialisés, ...). Parmi ces outils, nous citons les plus connus dont:

LOTOS/OTTAWA: Développé à l'Université d'Ottawa [Log90]. C'est le précurseur des interpréteurs du langage LOTOS.
Ce système prend comme entrée une spécification en LOTOS et l'analyse syntaxiquement et sémantiquement pour en produire un code qui sera interprété. Le système couvre aussi la partie des données abstraites de LOTOS ainsi que la librairie standard des types de données. L'exécution se fait soit en mode pas à pas, soit en mode de génération d'arbre symbolique dans lequel les valeurs des variables sont traitées symboliquement.

JAPON/LOTOS: Un compilateur pour LOTOS est décrit dans [Nom90]. Ce système gère la communication et l'ordonnancement entre processus parallèles. Cet ordonnanceur est implanté en C et prend en charge la gestion des événements et de leurs files d'attente. Un modèle abstrait sous forme d'arbre de synchronisation est utilisé. Il définit la hiérarchie entre processus ainsi que leurs points de synchronisation.

HIPPO: C'est un système développé dans le cadre du projet ESPRIT/ SEDOS [Van89] [Tre89] en Europe. Il permet la simulation après avoir effectué l'analyse syntaxique et sémantique. Il exécute les spécifications de comportements et effectue l'évaluation des expressions de valeurs. Il permet de construire des arbres de communication de la spécification qui contient l'historique de l'exécution jusqu.au point courant.

2. Outils de vérification.

Les outils de vérification sont les moins nombreux. Ces outils sont très souvent basés sur les notions d'équivalence observationnelle et bisimulation.

SQUIGGLES [Bol89]: C'est un outil de vérification basé sur la notion d'équivalence observationnelle. Il a été implanté en C et en Prolog. Il comprend plusieurs composantes, dont un analyseur syntaxique et un traducteur vers une représentation interne. D'autres modules vérifient la bisimulation forte, la bisimulation faible et l'équivalence de traces.

CAESAR-ALDEBARAN [Gar90]: Cet outil traduit une spécification LOTOS dans un réseau de Petri (partie contrôle) et dans des procédures C (partie données).
La spécification traduite est exécutée avec le but d'obtenir un graphe d'états fini. Si un tel graphe peut être obtenu, il peut être vérifié en utilisant ou bien la logique temporelle (*model checking*), ou bien des relations d'équivalences d'automates.

Conclusion

Nous avons présenté dans ce chapitre, les notions syntaxiques et sémantiques du langage LOTOS nécessaires à la compréhension de la suite notre travail (Une présentation plus accessible de LOTOS est fournie par l'annexe A).

LOTOS est un langage permettant la spécification formelle de comportements concurrents Il semble bien correspondre aux besoins ayant donné lieu à sa création et ce même si certains comportements parallèles sont difficilement ou pas du tout modélisables en LOTOS.

De nombreux exemples sont disponibles et montrent l'intérêt que LOTOS suscite. LOTOS est utilise par un grand nombre d'équipes sur plusieurs continents. Cela met l'accent sur la nécessite de définir un style, voir une méthode de conception des spécifications LOTOS.

De plus il parait intéressant sinon nécessaire d'étudier la réutilisation des composants spécifiés en LOTOS et ce peut être en relation avec l'intégration des concepts de certains paradigmes existants comme la concurrence et les composants conceptuels.

Dans les chapitres 6 et 7, nous allons utiliser le langage LOTOS pour enrichir et spécifier les composants conceptuels.

CHAPITRE 5

Les Design Patterns (Patrons de Conception)

Introduction

Les patterns ont été introduits par l'architecte Christopher Alexander [Ale et al.77]. Son idée de base est la suivante : *"chaque pattern décrit un problème qui se manifeste constamment dans notre environnement et décrit le cœur de la solution à ce problème, de telle façon qu'elle puisse être réutilisée des millions de fois et jamais deux fois de la même manière"*. Il a défini un catalogue de descriptions de problèmes et de solutions associées.

Un pattern (patron) est une solution à un problème récurrent dans un contexte donné. La solution est exprimée sous la forme de schéma statique et/ou dynamique, de pseudo code, etc. Une solution pourra être **réutilisée** et **adaptée** indéfiniment.

Contrairement aux composants logiciels, les patterns (patrons) sont considérés comme des **composants non logiciels**. Ils proposent des entités de réutilisation plus abstraites que celles proposées par les langages de programmation (sous-programme, module et classe) et les middlewares (composant CORBA, composant EJB, etc). Ils permettent de renforcer et d'étendre la réutilisation en couvrant plus largement toute l'ingénierie des systèmes d'information dès les étapes d'analyse et de conception.

Il n'existe pas aujourd'hui de consensus sur un formalisme de description de patterns. Toute définition
de pattern comprend un certain nombre de rubriques de base :
– *Nom :* nom du pattern avec éventuellement d'autres noms reconnus pour le même pattern.
– *Problème :* description de la situation que doit résoudre le pattern.
– *Contexte :* cas qui justifient l'utilisation du pattern.
– *Solution :* description de la structure, du comportement et de la résolution du problème dans le contexte donné.
– *Conséquences :* évolution du contexte après utilisation du pattern, problèmes résolus et ceux qui ne le sont pas.
– *Exemples de code :* extraits de programmes qui illustrent l'emploi du pattern.
– *Modèles apparentés :* patterns en relation étroite avec le pattern défini et patterns avec lesquels il peut être utilisé.
– *Utilisations remarquables :* exemples de systèmes définis à l'aide du pattern.

Les patterns peuvent être classés relativement à leur niveau d'abstraction. On peut distinguer trois catégories:
– *Patterns ou styles d'architecture :* un style d'architecture décrit le schéma structurel d'un système. Il décompose le système en composants, spécifie leurs fonctionnalités et fournit des guides pour décrire les relations entre les différents composants. Il est utilisé à l'initialisation du développement.

– *Patterns de conception :* un pattern de conception fournit un schéma de raffinement d'un composant.
– *Patterns d'implémentation :* un pattern d'implémentation est un pattern de très bas niveau dépendant du langage de programmation. Il décrit une façon de programmer un aspect particulier du système.

Par la suite, nous étudierons essentiellement les patterns de conception. Plus particulièrement, le catalogue [Gam et al.95]. Les représentations communément utilisées pour les patterns consistent à décrire ceux-ci en rassemblant des descriptions informelles, des diagrammes d'UML et des exemples d'implantation dans un langage orienté objets particulier. Ces représentations soulignent la souplesse d'utilisation d'un pattern. Bien que diverses informations sur un pattern soient fournies, leur manque de précision peut aboutir à des interprétations ambiguës et incorrectes. L'application et la réutilisation des patterns se voient souvent limitées par leur manque de formalisation, empêchant d'avoir des critères précis pour sélectionner les patterns les mieux adaptés à un problème particulier.

I.1 Les patrons de conception de GoF

Les patrons de conception (design patterns) sont reconnus comme des bonnes techniques du génie logiciel à objets. Cette technique améliore le cycle de vie du logiciel en facilitant la conception, la documentation, la maintenance et la rétro-conception. Les patrons de conception et notamment ceux de Gof [Gam et al. 85] sont de plus en plus utilisés. En effet, des environnements de développement intégrés - qui incitent l'utilisateur à appliquer un catalogue de patrons et l'aident à améliorer ses programmes en y extrayant automatiquement des micro-architectures correspondant à celle de patrons-commencent à apparaître [Alb et al.02].

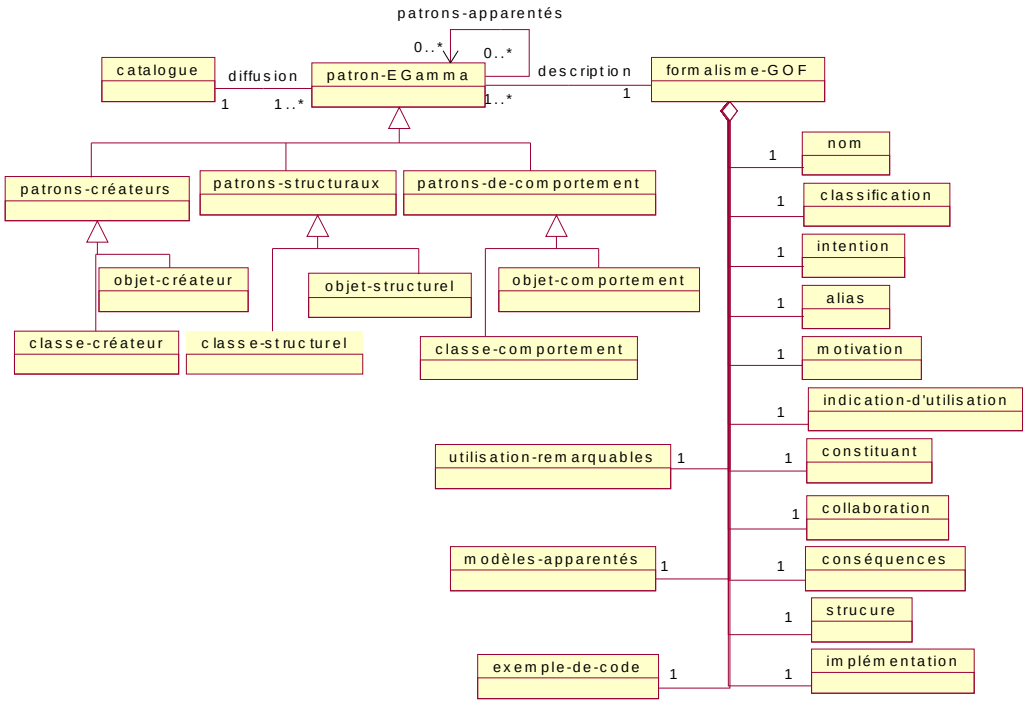


Figure 1- Un modèle de patrons de GoF [Gam et al. 85]

La figure1 propose un modèle décrivant les patrons d'E.Gamma. La structure de ce modèle. L'association <<patrons-apparentés>> référence les patrons utilisant ou utilisés par ce patron.

Les deux critères de classification (**rôle** et **domaine**) utilisés pour organiser le catalogue E.Gamma sont modélisés par une arborescence à deux niveaux. Le premier niveau comporte les abstractions <<patrons-créateurs>>, <<patrons-structuraux >> et <<patrons-de-comportement>> dérivant de <<patrons-E.Gamma>>. Le second niveau comporte deux abstractions <<classe>> et <<objet>> pour chaque abstraction citée au niveau précédent.

Les patrons de conception les plus répandus sont au nombre de 23. Ils sont couramment appelés "patrons GoF" (de "Gang of Four", d'après les quatre créateurs du concept).

On distingue trois familles de patrons de conception selon leur utilisation (Table1):

- **Créateurs** : ils définissent comment faire l'instanciation et la configuration des classes et des objets.
- **Structuraux** : ils définissent comment organiser les classes d'un programme dans une structure plus large (séparant l'interface de l'implémentation).
- **Comportementaux** : ils définissent comment organiser les objets pour que ceux-ci collaborent (distribution des responsabilités) et expliquent le fonctionnement des algorithmes impliqués.

	Créateurs	**Structuraux**	**Comportementaux**
Classe	Factory Method	Adaptateur (classe)	Interpreter Template Method
Objet	AbstractFactory Builder Prototype Singleton	Adapter(objet) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsability Command Iterator Mediator Memento Observer State Strategy Visitor

Table1- Le catalogue des patterns de GOF

I.2 Description des patrons de conception de GoF

I.2.1 Patterns Créateurs (Creationnel Patterns): Les modèles de conception créateurs permettent:

-D'expriment le principe de processus d'instanciation.

-De rendre le système indépendant de la façon dont les objets sont créés, composés, assemblés, représentés. De plus un modèle créateur de classe utilise l'héritage pour instancier des variantes de la classe, alors qu'un modèle créateur d'objets délègue l'instanciation à un autre objet. Nous distinguons les patterns:

1. Abstract factory (Fabrication abstraite): Il permet de fournir une interface pour la création de famille d'objets sans spécifier les classes concrètes.
On utilise l'AbstractFactory lorsque :
• un système doit être indépendant de la façon dont ses produits sont créés, assemblés, représentés
• un système repose sur un produit d'une famille de produits
• on veut définir une interface unique à une famille de produits concrets

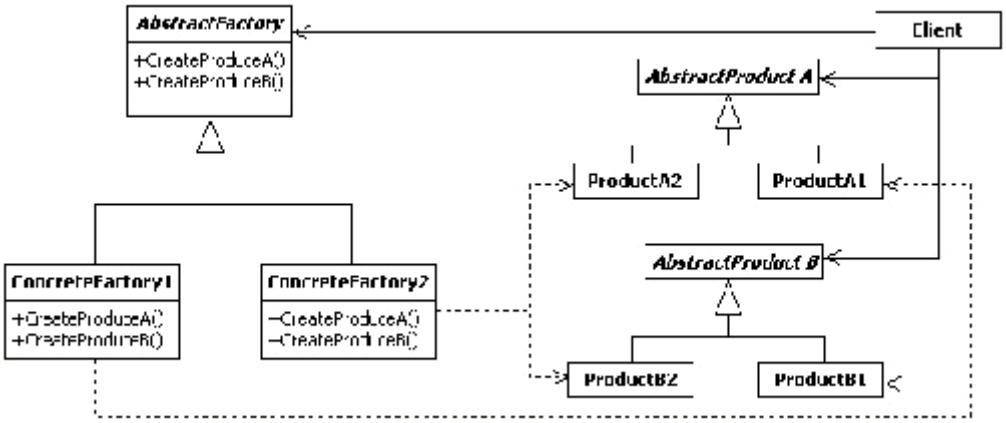


Figure2- Diagramme de classes : Fabrique abstraite

2. Builder (Monteur): Il permet de séparer la construction d'un objet complexe de sa représentation. Le même procédé de construction peut donc créer différentes représentations. On utilise le Builder lorsque :
• l'algorithme pour créer un objet doit être indépendant des parties qui le compose et de la façon de les assembler
• le processus de construction permet différentes représentations de l'objet construit

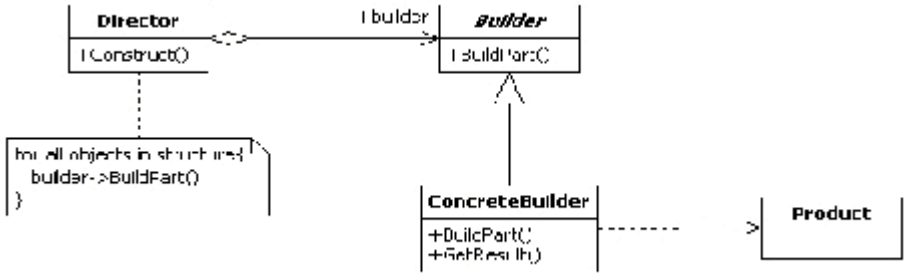


Figure 3- Diagramme de classes: Menteur

3. Factory Method (Fabrication): Définir une interface pour la création d'objet, en laissant les sous classes décider quelle classe instancier. (Laisser le travail d'instanciation aux sous classes). On utilise le FactoryMethod lorsque :
• une classe ne peut anticiper la classe de l'objet qu'elle doit construire
• une classe délègue la responsabilité de la création à ses sous classes, tout en concentrant l'interface dans une classe unique

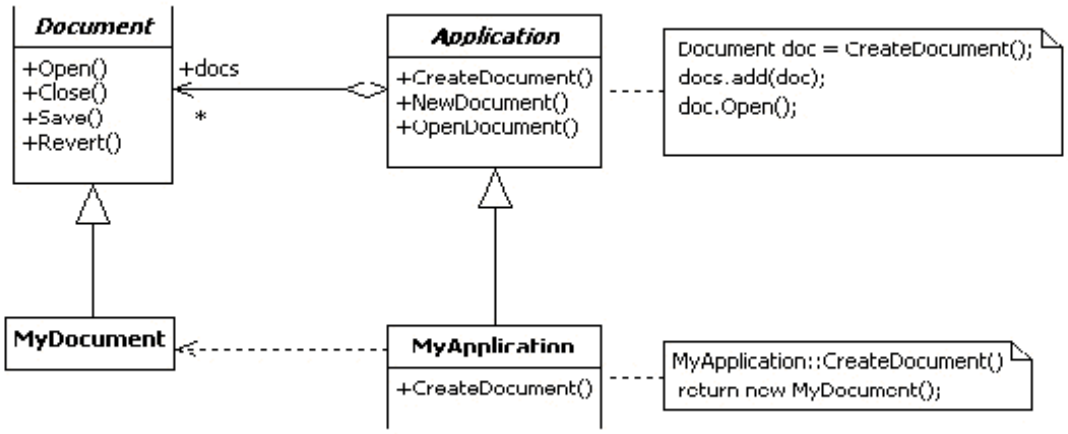


Figure4- Diagramme de classes : Fabrication

4. Prototype (Prototype): Spécifier le type d'objet à créer en utilisant une instance d'un prototype, et créer de nouveaux objets en copiant ce prototype. On utilise le Prototype lorsque :

• un système doit être indépendant de la façon dont ses produits sont créés, assemblés, représentés

• quand la classe n'est connue qu'à l'exécution

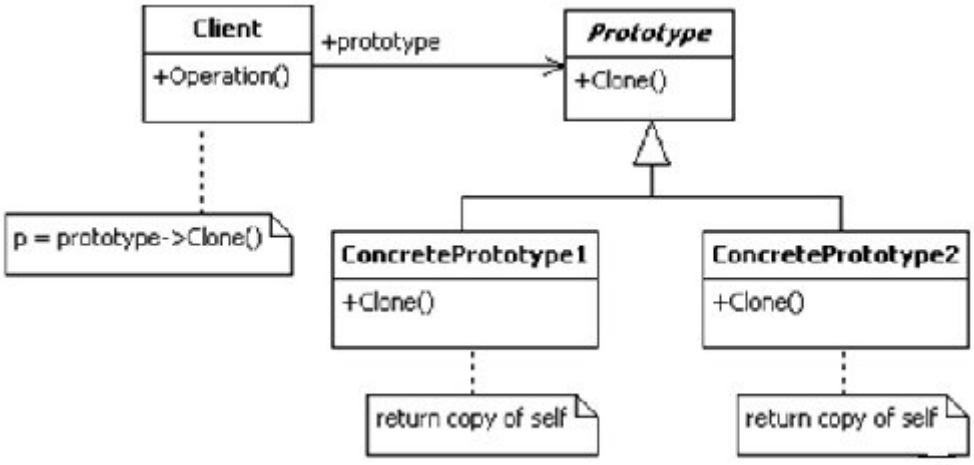


Figure 5- Diagramme de classes : Prototype

5. Singleton (Singleton): S'assurer qu'une classe n'ait qu'une instance et lui fournir un point d'accès global. On utilise le Singleton lorsque :

• il n'y a qu'une unique instance d'une classe et qu'elle doit être accessible de manière connue

• une instance unique peut être sous-classée et que les clients peuvent référencer cette extension sans avoir à modifier leur code.

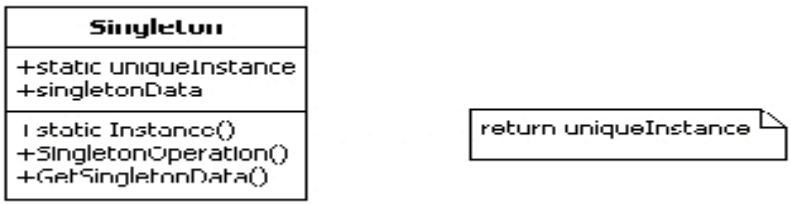


Figure 6- Diagramme de classes : Singleton

I.2.2 Patterns structuraux (Structural Patterns): Les modèles de conception structurels étudient la façon:
• Comment les objets et les classes sont assemblés pour réaliser des structures plus importantes
• Les patterns sont complémentaires les uns des autres. Nous distinguons les patterns:

1. Adapter (Adaptateur): Convertir l'interface d'une classe dans une autre interface compatible avec d'autres clients. On utilise l'Adapter lorsque on veut utiliser :
• une classe existante dont l'interface ne convient pas
• plusieurs sous-classes mais il est coûteux de redéfinir l'interface de chaque sous-classe en les sous-classant. Un adapter peut adapter l'interface au niveau du parent.

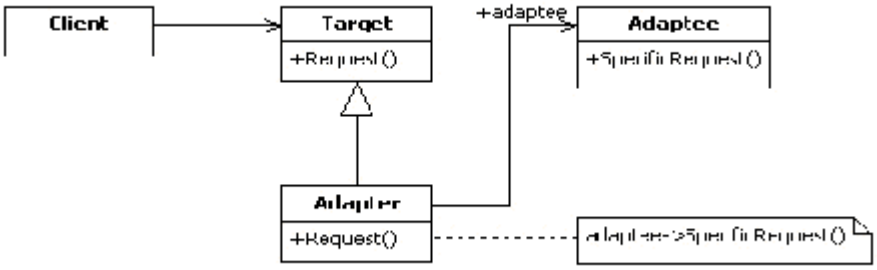


Figure7- Diagramme de classes : Adaptateur

2. Bridge (Pont): Séparer une abstraction de son implémentation dans le but que chacune puisse être modifiée indépendamment. On utilise Bridge lorsque :
• on veut éviter un lien permanent entre l'abstraction et l'implantation (ex: l'implantation est choisie à l'exécution)
• l'abstraction et l'implantation sont toutes les deux susceptibles d'être raffinées
• les modifications subies par l'implantation ou l'abstraction ne doivent pas avoir d'impacts sur le client.

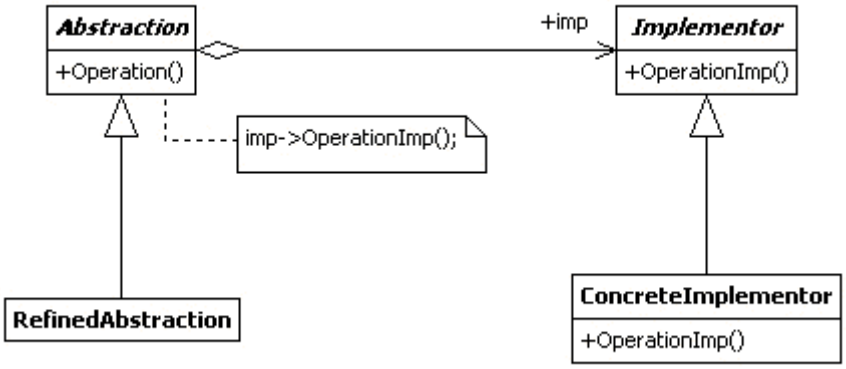


Figure8- Diagramme de classes : Pont

3. Composite (Composite): Composer des objets en structure d'arbre hiérarchisé. On utilise Composite lorsque on veut :
• représenter une hiérarchie d'objets
• ignorer la différence entre un composant simple et un composant en contenant d'autres.

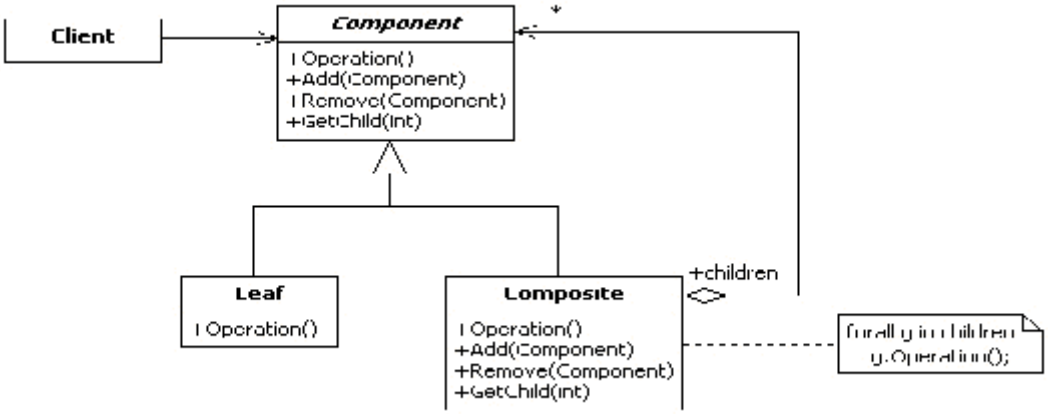


Figure9- Diagramme de classes : Composite

4. Decorator (Décorateur): Ajouter dynamiquement des responsabilités à un objet. On utilise Decorator lorsque :
• il faut ajouter des responsabilités dynamiquement et de manière transparente
• il existe des responsabilités dont on peut se passer

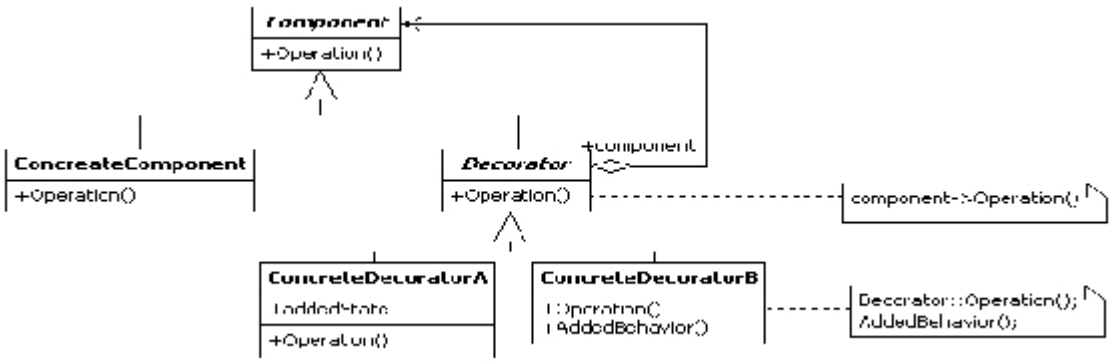


Figure 10-Diagramme de classes : Décorateur

5. Facade (Façade): Proposer une interface commune à un ensemble d'interfaces d'un sous système (améliorer l'utilisation d'un sous système). On utilise Façade lorsqu'on veut :
• fournir une interface simple à un système complexe
• introduire une interface pour découpler les relations entre deux systèmes complexes
• construire le système en couche.

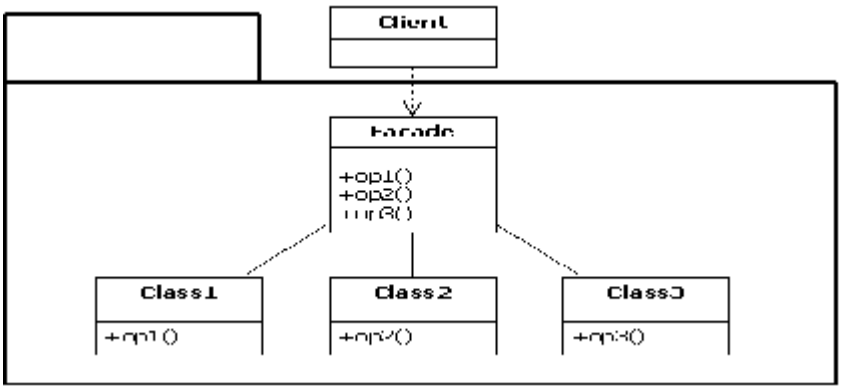


Figure11- Diagramme de classes : Façade

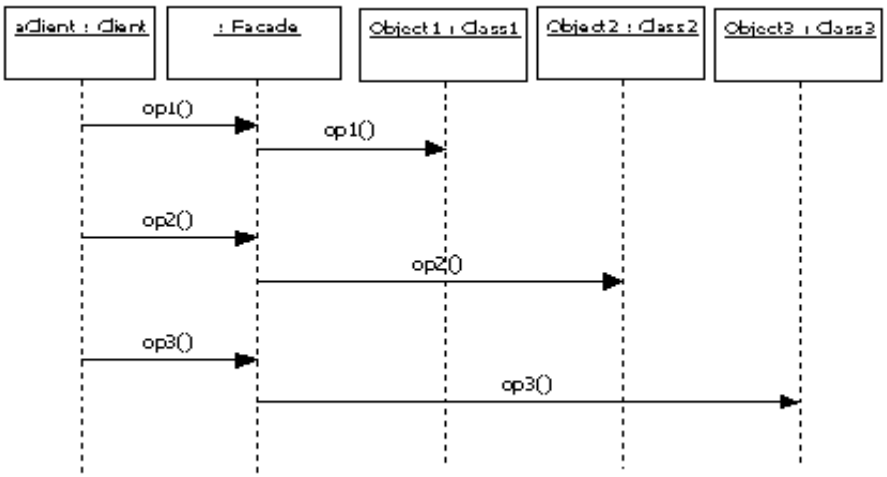


Figure 11.1-Diagramme de séquences : Façade

6. Flyweight (Poids Mouche) : Utiliser le partage afin de supporter un large panel d'objets bas niveau. On utilise Flyweight lorsque :
• on utilise beaucoup d'objets, et
• les coûts de sauvegarde sont élevés, et
• de nombreux groupes d'objets peuvent être remplacés par quelques objets partagés, et
• l'application ne dépend pas de l'identité des objets

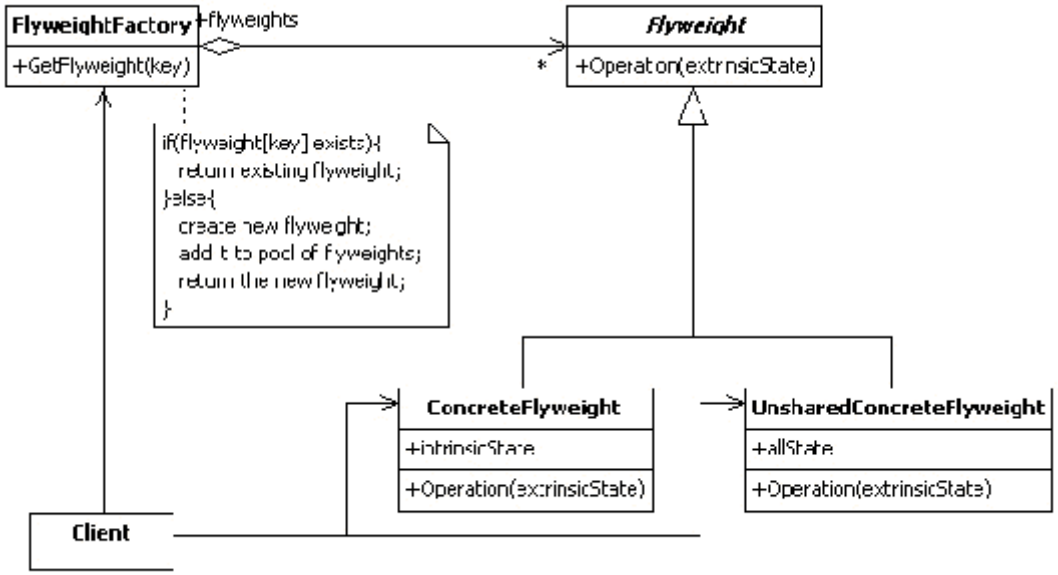


Figure 12- Diagramme de classes : Poids mouche

7. Proxy (Procuration): Proposer un clone ou un point d'entrée à un objet pour contrôler son accès. On utilise le Proxy lorsqu'on veut référencer un objet par un moyen plus complexe qu'un pointeur...

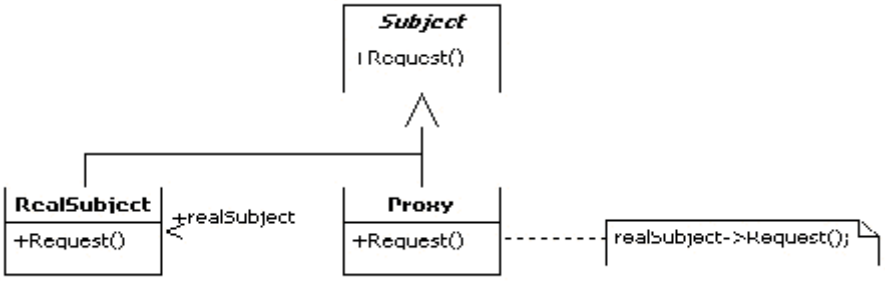


Figure 13-Diagramme de classes : Procuration

I.2.3 Patterns comportemntaux (Behavioural Patterns) : Forme des comportements pour décrire :
• des algorithmes
• des comportements entre objets
• des formes de communication entre objet. Nous distinguons les patterns

1. Chain of responsability (Chaîne de responsabilité): Permettre d'associer l'émetteur et le récepteur d'une requête en donnant à plus d'un objet la chance de manipuler la requête (enchaîner des objets et parcourir cette chaîne avec la requête jusqu'à trouver quelqu'un capable de la manipuler) On utilise Chain of Responsibility lorsque :
• plus d'un objet peut traiter une requête, et il n'est pas connu a priori
• l'ensemble des objets pouvant traiter une requête est construit dynamiquement

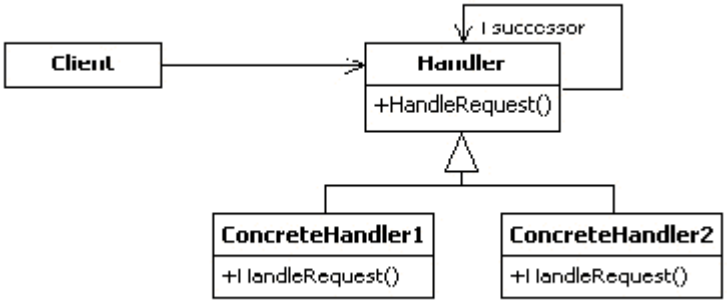


Figure 14-Diagramme de classes : Chain of responsabilité

2. Command (Commande): Encapsuler une requête dans un objet. On utilise Command lorsque:
• spécifier, stocker et exécuter des actions à des moments différents.
• Les commandes exécutées peuvent être stockées ainsi que les états des objets affectés...
• on veut implanter des transactions.

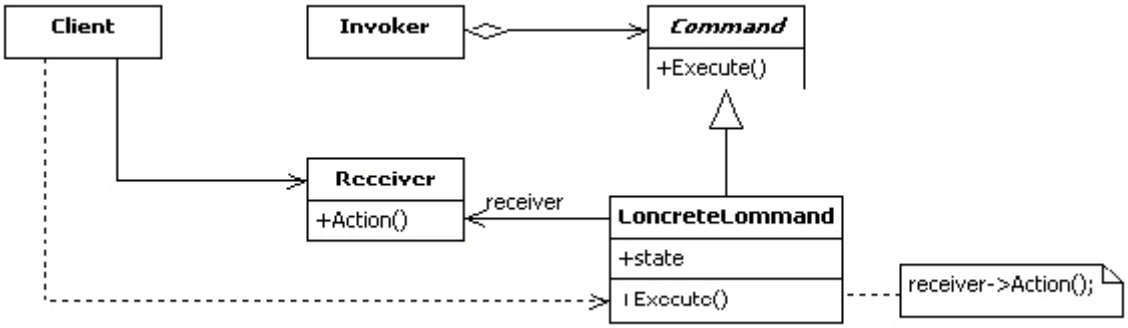


Figure15- Diagramme de classes : Commande

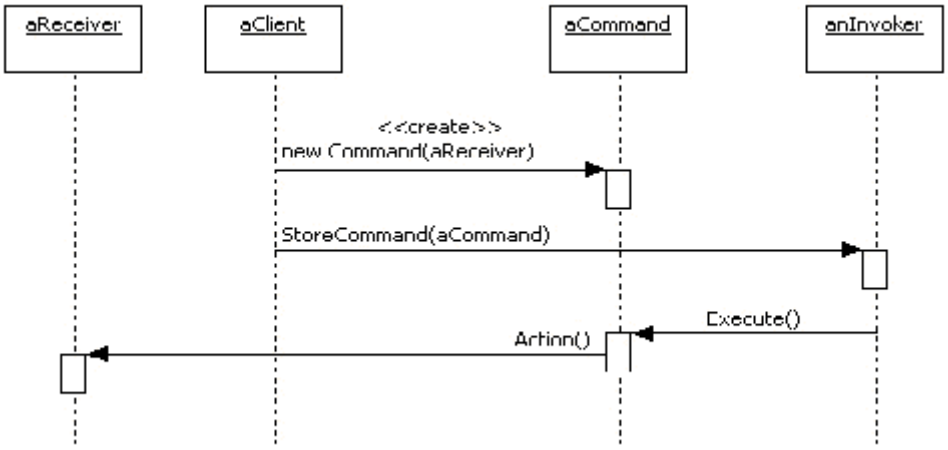


Figure 15.1- Diagramme de séquences : Commande

3. Interpreter (Interprète): Proposer un langage, définir une représentation pour sa grammaire et fournir un interprète capable de manipuler cette grammaire.
On utilise Interpreter lorsqu'il faut interpréter un langage et que :
• la grammaire est simple
• l'efficacité n'est pas un paramètre critique

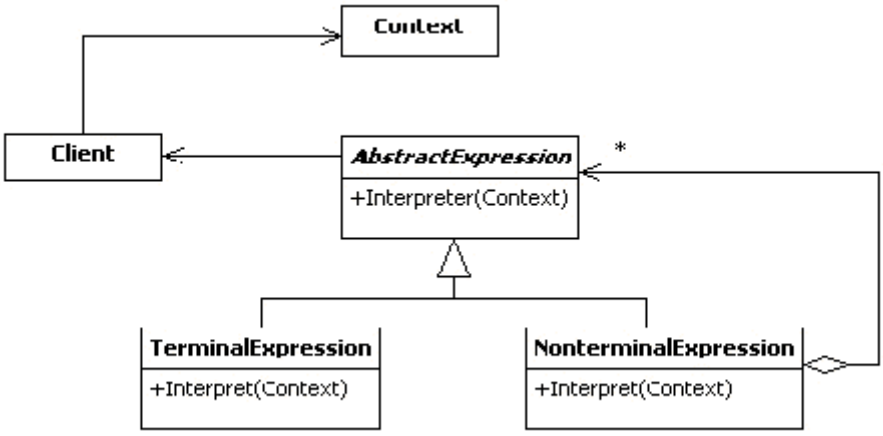


Figure 16- Diagramme de classes : Interpréteur

4. Iterator (Itérateur): Proposer séquentiellement une façon d'accéder aux éléments d'un objet. On utilise Iterator lorsque :
• pour accéder à un objet composé dont on ne veut pas exposer la structure interne
• pour offrir plusieurs manières de parcourir une structure composée
• pour offrir une interface uniforme pour parcourir différentes structures.

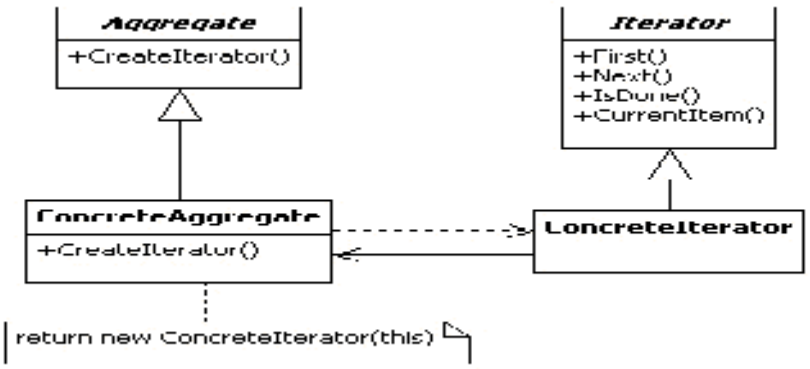


Figure 17- Diagramme de classes : Itérateur

5. Mediator (Médiateur): Définir un objet qui encapsule la façon dont interagit un autre ensemble d'objets. On utilise Mediator lorsque :
• quand de nombreux objets doivent communiquer ensemble
• la réutilisation d'un objet est délicate car il référence et communique avec de nombreux autres objets.

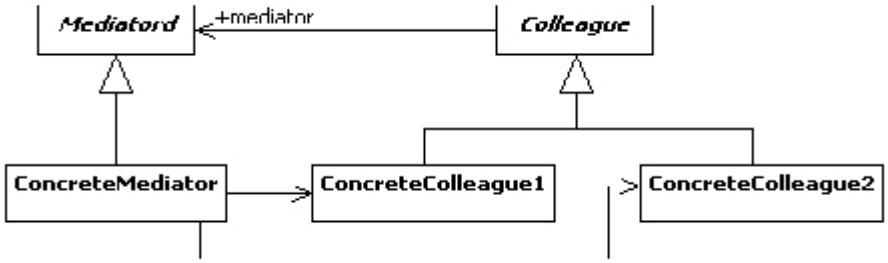


Figure 18- Diagramme de classes : Médiateur

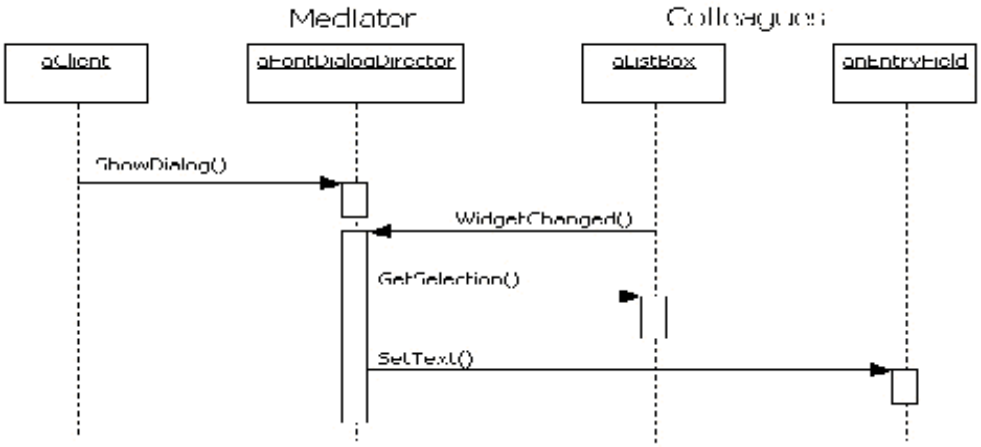


Figure 18.1- Diagramme de séquences : médiateur

6. Memento (Mémento): Sans toucher à l'encapsulation, capturer et extraire l'état d'un objet de manière à ce que cet objet puisse retrouver son état initial. On utilise Memento lorsque :

• on veut sauvegarder tout ou partie de l'état d'un objet pour éventuellement pouvoir le restaurer, et

• une interface directe pour obtenir l'état de l'objet briserait l'encapsulation.

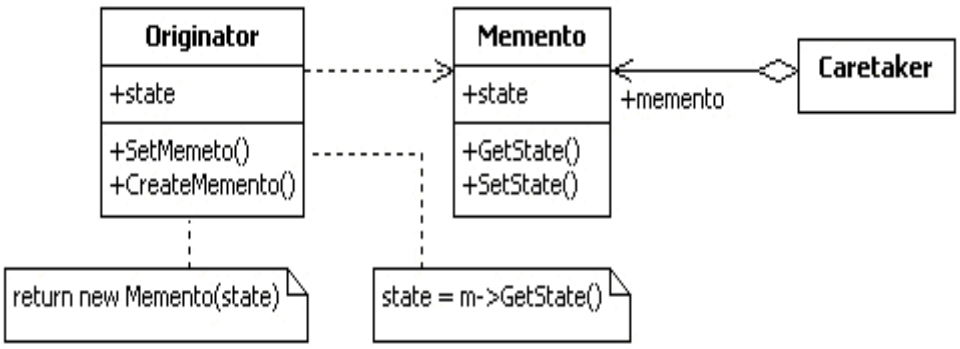


Figure 19- Diagramme de classes : Mémonto

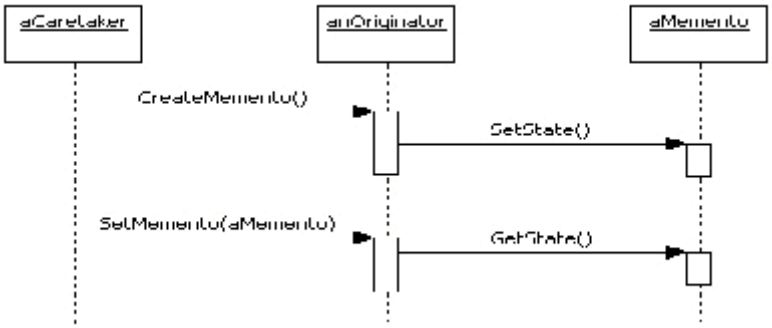


Figure19.1- Diagramme de séquences : Mémonto

7. Observer (Observateur): Proposer une dépendance entre objets de manière à mettre à jour automatiquement toutes ces dépendances lors du changement d'état d'un objet. On utilise Observer lorsque :

• Une abstraction a plusieurs aspects, dépendant l'un de l'autre.
• Quand le changement d'un objet se répercute vers d'autres.
• Quand un objet doit prévenir d'autres objets sans pour autant les connaitre.

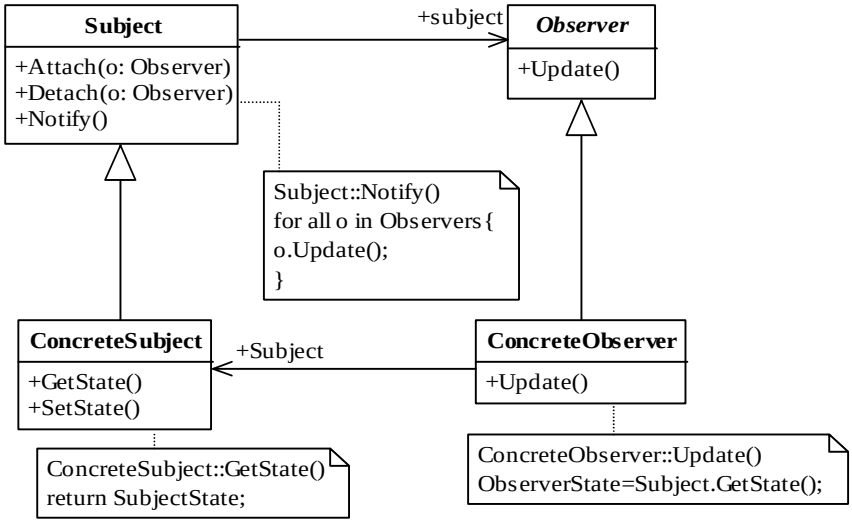


Figure 20- Diagramme de classes : Observateur

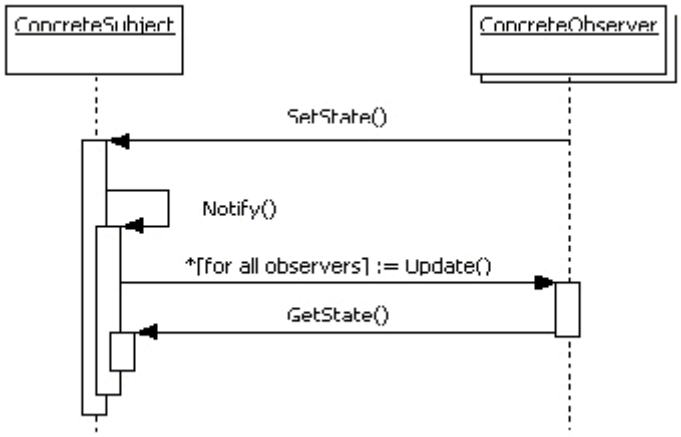


Figure 20.1- Diagramme de séquence : Observateur

8. State (Etat): Permettre à un objet de modifier son comportement lorsque son état interne change également (changement de classe). On utilise State lorsque :
• Le comportement d'un objet dépend de son état, qui change à l'exécution
• Les opérations sont constituées de parties conditionnelles de grande taille (case)

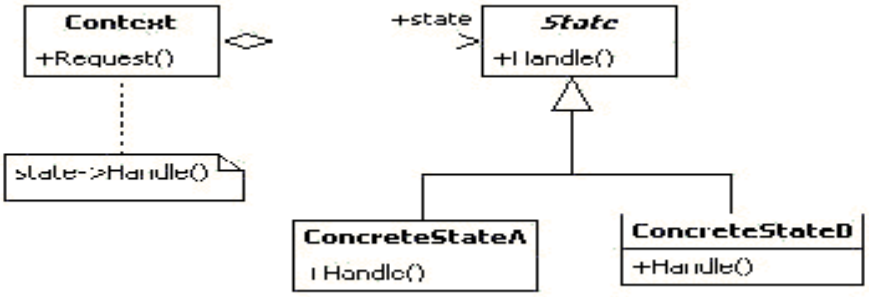


Figure 21- Diagramme de classes : Etat

9. Strategy (Stratégie): Définir une famille d'algorithmes, l'encapsuler, la rendre interchangeable, et permettre à chaque algorithme de varier indépendamment du client qui l'utilise. On utilise Strategy lorsque :
• de nombreuses classes associées ne diffèrent que par leur comportement. (Stratégie offre un moyen de configurer une classe avec un comportement parmi plusieurs).
• on a besoin de plusieurs variantes d'algorithme.
• un algorithme utilise des données que les clients ne doivent pas connaitre.

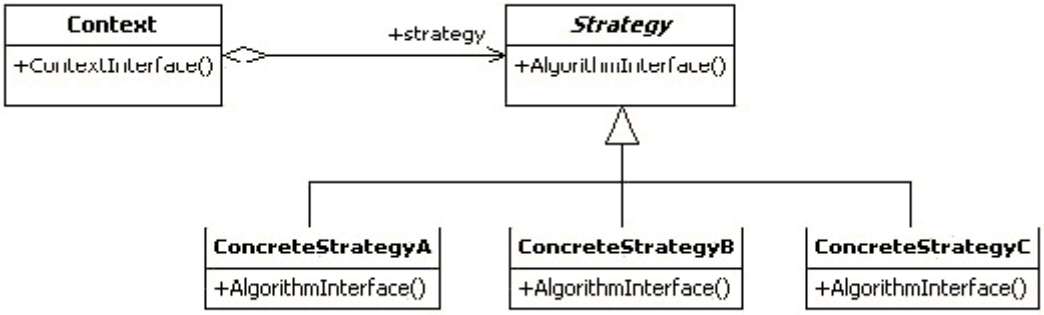


Figure 21.1 Diagramme de classes : Stratégie

10. Template method (Patron de Méthode): Définir le squelette d'un algorithme en une opération et allouer certaines parties aux sous classes (les sous classes redéfinissent certaines parties de l'algorithme sans changer la structure principale). On utilise TemplateMethod :
• pour implanter une partie invariante d'un algorithme.
• pour partager des comportements communs d'une hiérarchie de classes.
• pour contrôler des extensions de sous-classe.

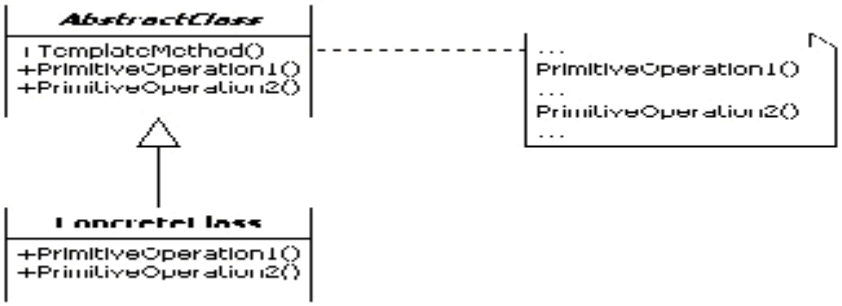


Figure 22- Diagramme de classes : Patron de méthode

11. Visitor (Visiteur): Permettre de définir une nouvelle opération sans changement de classe des éléments utilisés. On utilise Visitor lorsque :
• une structure d'objets contient de nombreuses classes avec des interfaces différentes et on veut appliquer des opérations diverses sur ces objets.
• les structures sont assez stables, et l'opération sur leurs objets évolutives.

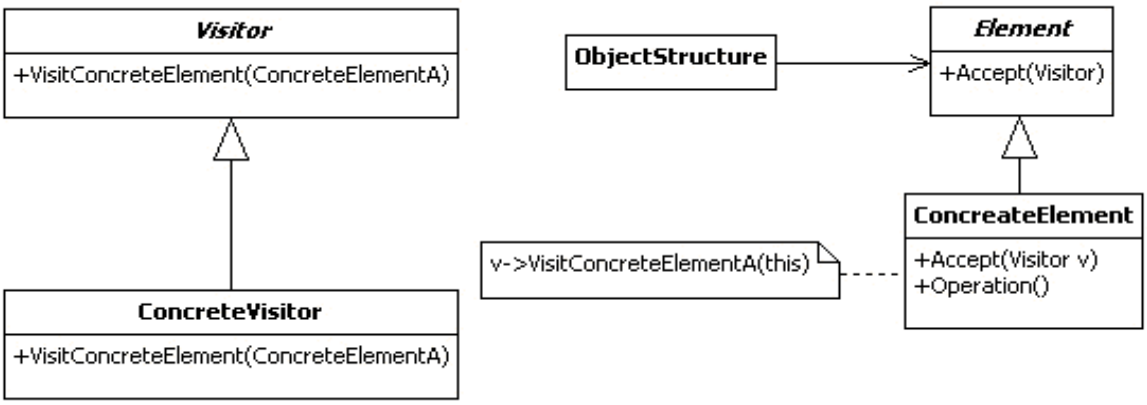


Figure 23- Diagramme de classes : Visiteur

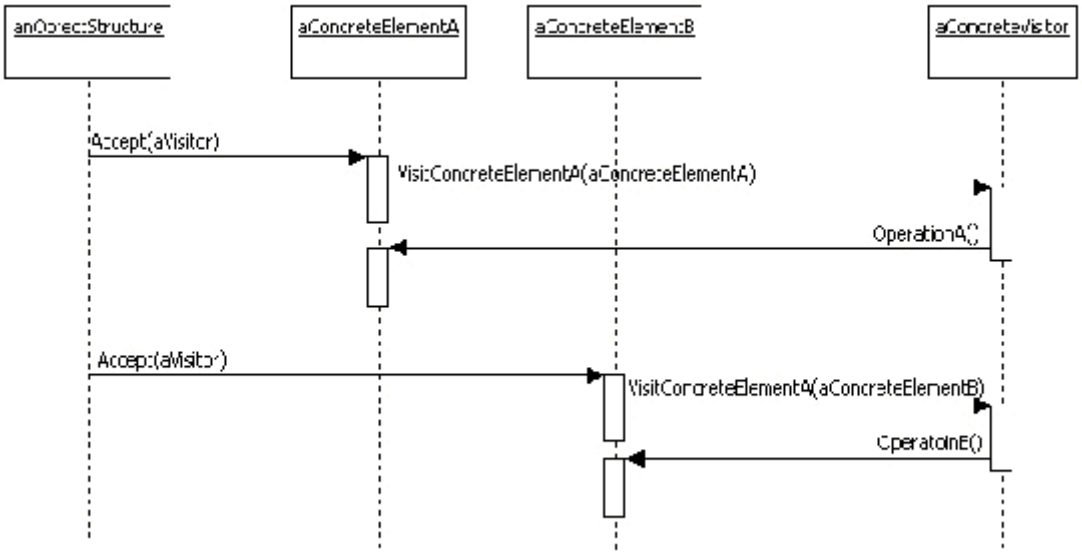


Figure 23.1- Diagramme de séquences : Visiteur

I.3 Synthèse de quelques travaux sur les Design Patterns

Depuis l'émergence des « design patterns » au début des années 90, et le livre référence de Gamma et al. [Gam et al.95], de nombreux travaux ont été menés autour de:
La formalisation des patterns de conception afin d'en faciliter l'utilisation, l'échange entre personnes, voire d'en permettre l'« instanciation et la vérification de pattern de conception» [Flo et al .97].

Les différentes notations graphiques standards (comme UML ou OMT) ne suffisent pas à expliciter toutes les informations relatives aux patterns de conception. Par exemple, le nombre minimal et maximal de classes autorisées dans une représentation concrète du pattern est difficile à modéliser. Aussi différents travaux [Ede et al.98],[LK98] proposent des extensions de notations graphiques, tel que les stéréotypes….

D'autres approches visent à définir des langages spécialisés pour la représentation de patterns. [Bos96] propose LayOM, un langage support pour les patterns de conception, basé sur la notion de couche permettant d'encapsuler les objets. [Ede et al.97] présente une méthode de spécification formelle couplée à une notation graphique.

D'un autre point de vue, les variantes d'implémentation constituent, le principal intérêt et la principale difficulté de l'utilisation des patterns. Elles sont nécessaires car les patterns représentent les connaissances de conception et l'importance de celle-ci et d'être indépendante de l'implémentation. Cependant, elles rendent difficile l'implémentation et sa réutilisation. Certains travaux de recherche proposent des techniques d'implémentation et des outils facilitant la réutilisation des patterns.

L'objectif de ces outils et la gestion des patterns, la représentation des patterns, la reconnaissance des patterns, et enfin la documentation de l'implémentation d'instance des patterns. Parmi ces outils nous distinguons [Sun99]:

Le prototype d'outil PatternGen de l'équipe de recherche MetaFor, appartenant au thème OASIS, du laboratoire d'informatique de Paris VI (LIP6), son objectif est d'aider le concepteur à développer son application [Sun99].

Pattern Tool [MEI96]: un outil qui permet l'intégration des patterns à du code Smalltalk.

FACE [ME96]: un environnement de développement, dont l'objectif principal est de simplifier l'utilisation des Framework.

LayOM [Bos96]: permet d'intégrer les patterns de conception à un modèle à objet étendue (Layered Object Model) grâce à la notion des couches.

La plupart de ces outils utilisent un méta modèle pour représenter les patterns et représenter explicitement les instances de pattern dans un modèle de conception [Sun99]

Conclusion

Nous avons pu constater dans ce chapitre, la multitude et la diversité des patterns proposés, à la lumière des nombreux articles, et ouvrages. Une chose apparaît certaine, les patterns permettent d'unifier la discussion entre concepteurs. A ce titre, il est opportun de ne pas les négliger.

Les design patterns sont le type de patterns le plus important du fait de leur portée d'application qui est totalement isolée des spécificités des langages de programmation ou de la méthode de modélisation orientée objet utilisée.

En effet, les patterns de conception, en caractérisant des problèmes récurrents de conception et en spécifiant des solutions claires et élégantes à ces problèmes, représentent le vocabulaire commun de l'expertise des concepteurs de logiciels.

Ce vocabulaire leur fournit un niveau de description adéquat pour discuter les choix de conception ou de restructuration d'un système, au delà des détails connus de conception, l'application du pattern de conception simplifie la compréhension de sa structure.

CHAPITRE 6

Proposition d’un Framework pour l'utilisation des design Patterns par intégration du langage de spécification LOTOS

Introduction

Il existe aujourd’hui deux grandes familles d’approches pour la modélisation de systèmes d’information: (i) les approches semi-formelles orientées objets telles que UML [B98] et (ii) les approches formelles fondées sur des sémantiques algébriques ou ensemblistes telles que LOTOS et la logique de description [Abr96]. Ces deux familles sont complémentaires dans le processus de développement. Les descriptions semi-formelles offrent des mécanismes de structuration très riches tout en proposant une représentation intuitive et synthétique du système à construire. Les notations graphiques utilisées sont maîtrisées par les clients et facilitent les échanges entre clients et développeurs. Elles ne sont pas ambiguës dans le sens où elles peuvent être traduites automatiquement en des squelettes de programmes dans des langages orientés objets. Cependant, ces approches ne permettent pas d’exprimer les propriétés des objets manipulés. Il n’est pas possible de préciser des invariants de classes ou le comportement des opérations. Il est nécessaire pour cela d’utiliser une description formelle. Le langage OCL a été proposé [WK99] comme faisant partie de UML, pour spécifier des invariants de classes, des contraintes, des pré-conditions d’opérations et des pré-conditions d’activation d’événements. Mais il n’existe pas aujourd’hui d’outils associés permettant d’analyser, de valider, de vérifier ou de simuler de telles spécifications.

Les approches formelles permettent de faire des preuves de propriétés, de détecter des incohérences et d’animer des spécifications. Pour modéliser des systèmes, une technique a été proposée, indépendamment des différentes approches ou langages existants : les *patterns* [Ale et al.77]. Jusqu’à présent, les *patterns* ont été essentiellement définis pour modéliser des systèmes à objets [Gam et al. 95]. Ils sont un moyen de documenter les architectures et permettent dans le cas d’un développement en équipe de disposer d’un vocabulaire commun.

Notre objectif est d’utiliser des patterns existants pour spécifier des systèmes de manière à bénéficier aussi bien des avantages des approches semi-formelles que de ceux des approches formelles. Notre démarche s’effectue en deux temps : compléter par une spécification formelle LOTOS (Langage Of Temporal Ordering Specification) les patterns existants, puis les utiliser dans le développement des systèmes d’information.

Ainsi donc, notre première contribution dans cette thèse est une proposition d’un Framework pour l'utilisation des design patterns par intégration du langage de spécification LOTOS [Zit05] [Zit07].

L’objectif est de définir la description de patrons de conception par intégration des

approches formelle et semi-formelle. Nous décrivons une méthode de spécification de patrons, intégrant deux paradigmes, les méthodes UML (semi- formelle) et le langage de spécification formelle LOTOS. La description de modèles UML est enrichie à l'aide d'une description formelle LOTOS (figure1).

Nous donnons une interprétation précise sur l'aspect structure et communication entre patterns. La démarche consiste à enrichir formellement les patterns décrits en UML, en vue de leur vérification.

I.1 Approche globale "spécification de patterns en UML et LOTOS".

Actuellement, l'expertise en matière de conception de systèmes distribués a été recueillie sous forme d'outils et de techniques qui ont été proposés. Ces techniques permettent aux développeurs d'avoir des logiciels, qui devront répondre aux exigences majeurs de ces systèmes d'information répartis. Parmi ces techniques, on distingue les "Patrons de conceptions connus sous le nom de "design pattern".

La faiblesse sémantique des représentations actuelles des patterns entraîne des interprétations ambiguës et limite leur application. La spécification formelle s'avère être un mécanisme très utile permettant l'adaptation de solutions à un problème d'architecture ou de conception d'un système.

Nous proposons une approche consistant à définir deux niveaux de représentation complémentaires pour les patrons: d'une part, une description semi-formelle visant à en donner une définition intuitive; d'autre part, une spécification formelle visant à donner une description précise du comportement des objets. Les deux descriptions permettent d'appliquer des patterns de façon méthodique et systématique lors de la spécification d'un système. Cette approche [Zit04] est définie comme suit: tout patron choisit est décrit par les diagrammes (UML) de classes et de séquences **(1)**, son comportement dynamique sera enrichit par une spécification formelle **(2)**.

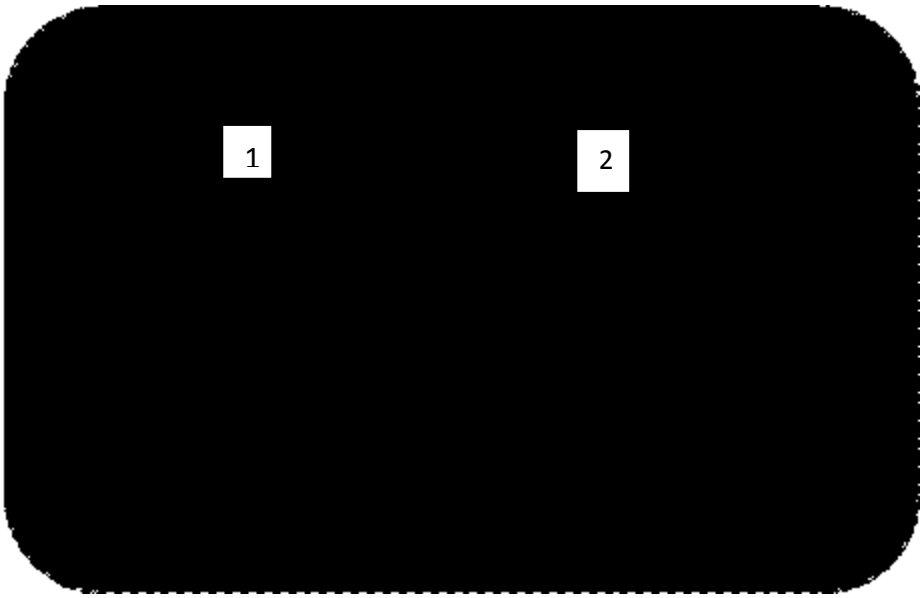


Figure 1- Approche poursuivie

I.2 Motivation de l'approche proposée.

La description semi-formelle d'un pattern est construite en utilisant le langage de modélisation orienté-objets UML. Elle est composée de diagrammes d'objets décrivant les aspects statiques relatifs à la structure du pattern : les objets participants, leurs attributs, leurs opérations et leurs associations. Des diagrammes d'interactions permettent de modéliser certains aspects dynamiques du patron.

Cependant ces diagrammes peuvent être construits en désaccord avec le modèle objet et ne permettent ni de valider ni de vérifier les comportements des patterns. Les erreurs sont en conséquence rarement détectées. C'est en spécifiant formellement un pattern que les contraintes ainsi que les propriétés associées aux objets et leurs relations peuvent être exprimées et vérifiées. Nous proposons d'utiliser le langage de spécification formel LOTOS pour les raisons suivantes :

-La spécification LOTOS permet de modéliser de manière plus détaillée le comportement et l'interaction entre les différentes entités d'un modèle.
- LOTOS est une méthode est basé sur la logique, qui permet de formaliser la sémantique comportementale des systèmes concurrents
- LOTOS permet de modéliser de manière plus détaillée le comportement et l'interaction entre les différentes entités d'un modèle,
- La spécification LOTOS permet une compréhension approfondie de l'application à développer. Elle met en évidence la plupart des ambiguïtés laissées par les spécifications informelles
- Les spécifications LOTOS sont basées sur les représentations mathématiques, ce qui permet la validation de leurs propriétés.

I.3 Transformation de la représentation du Pattern en UML vers la spécification LOTOS.

La transcription du pattern architectural en une spécification LOTOS est décrite par la relation de correspondance que nous allons définir.

Définition: Soient $\Re c$ une relation de correspondance, Ci l'ensemble des classes d'un pattern, et Πi l'ensemble des opérations d'une classe donnée alors:

Ci $\Re c$ Pi ⇒ (Toute classe se transforme en un processus LOTOS)

Πi $\Re c$ Ei ⇒(l'ensemble des opérations (méthodes) d'une classe se transforme en une suite d'actions décrivant le comportement de cette classe).

I.4 Etude de cas.

Spécifier en UML et en LOTOS, permet de bénéficier des avantages des approches semi formelles et des approches formelles. Pour cela nous illustrons notre approche à l'aide du patron de conception Client-Serveur fournit dans [Ber92] qui est simple, académique et relativement bien connu.

I.4.1 Spécification UML du patron Client-Serveur :

Ce patron est définit comme un style d'architecture, permettant la construction de systèmes distribués. Il apparaît comme un patron générique, définissant deux types de correspondants : les clients, demandeurs de services, et les serveurs, fournisseurs de services. La relation entre clients et serveurs consiste en la demande d'un service par un client à un serveur. Dans une première définition du patron, aucun détail sur la communication entre les clients et les serveurs n'est considéré. Le modèle Client-Serveur (figure2) est un style permettant de construire une première version d'un système, dont seul le choix d'architecture est déterminé. De ce fait, il peut être considéré comme la donnée d'un problème consistant à décrire des clients et des serveurs qui communiquent.

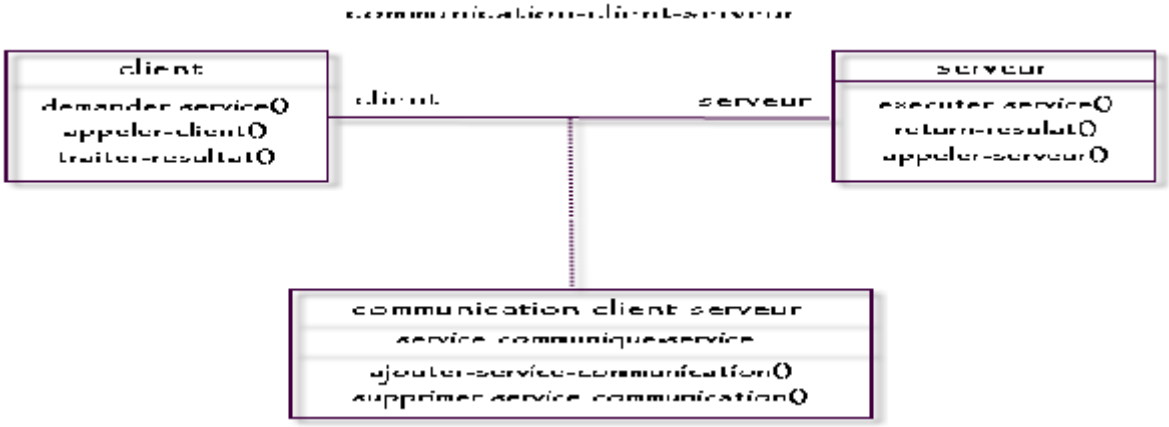


Figure 2- Modèle objet du patron client-serveur.

Le diagramme d'interaction de la (Figure3) illustre les échanges de messages entre les objets du pattern Client-Serveur.

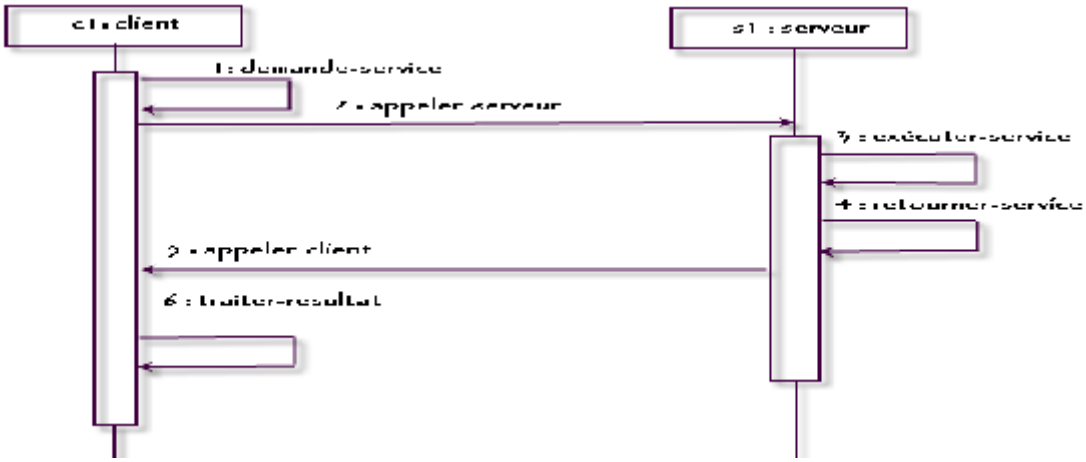


Figure 3- Diagrammes d'interactions du patron Client-Serveur.

Par la suite nous complétons la description du patron en lui associant une spécification formelle LOTOS.

I.3.2 Spécification LOTOS du pattern Client-Serveur :

Dans le but de formaliser et valider la spécification, nous allons par la suite montrer une partie de la spécification LOTOS associée au patron décrit précédemment.
Ceci peut ce faire, par l'application de la relation de correspondance que nous avons définit précédemment, ainsi que la syntaxe de Basic LOTOS.

Soient Pi la liste des processus, et Ei la liste de tous les évènements possibles.

Pi:= ({Pattern-client-serveur} et {Communication-client-serveur; Client; Serveur})
Ei:= ({ajouter-service-communication; supprimer-service-communication, demander-b service**;** Appeler-serveur**;** traiter-résultat; exécuter-service**;** retourner service**;** appeler-client}).
La spécification du pattern architectural Client- Serveur est comme suit:

/Exécution en parallèle des trois processus/

Process Pattern-Client-Serveur **:=**

Client |CCS|
Communication-client-serveur|CCS|
Serveur
End process

/Exécution indéterministe des processus client et serveur/

Process Communication-client-serveur [CCS]**:= noexit**
Ajouter-service-communication; Client
[CCS]
Serveur; supprimer-service-communication;
End process

/Spécification de l'exécution du processus client/

Process Client [CCS]**:= noexit**

Demander-service**;**
Appeler-serveur**;**
>> Serveur**;**
Traiter-résultat
End process

/Spécification de l'exécution du processus serveur/

Process Serveur [CCS]**:= noexit**

Exécuter-service**;**
Retourner-service**;**
Appeler-client**;**
>> Client
End process

Nous remarquons que dans cette spécification, le comportement du processus Pattern-Client-Serveur est donné par "l'exécution en parallèle" des trois processus (objets) du patron. Ces processus se synchronisent à travers la porte de synchronisation **(CCS).** Le comportement des processus est donné par la liste des interactions des classes du patron.

Enfin le comportement du processus communication-client-serveur est donné par le comportement du processus Client, ou le comportement du processus serveur (ce choix est indéterministe).

I.4.3 Validation de la spécification LOTOS du pattern Client-Serveur:

Concernant la vérification de notre proposition, nous utilisons environnement de validation FOCOVE (Formal Concurrency Verification Environment) (www.focove.new.fr) (Figure. 4).

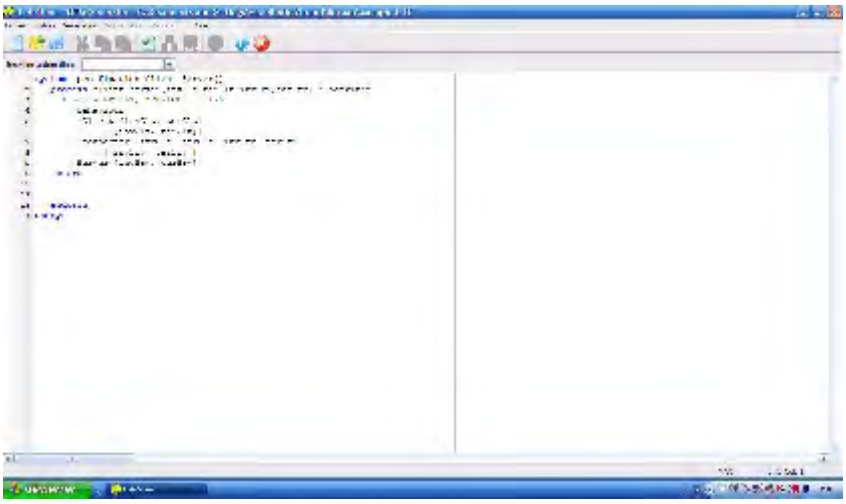


Figure 4- Environnement de validation FOCOVE

N.B. Des détails de l'utilisation de l'environnement, ainsi que la validation seront donnés dans le chapitre 8.

I.5 Proposition d'un Méta-pattern Client-Serveur enrichi par une spécification LOTOS.

Nous parlerons de notre représentation d'un patron de conception comme d'un méta-patron. Le terme de méta-pattern a été emprunté à [Pre94], et qui propose une réification partielle de l'aspect structurel des patrons dits de compositions, et a été étendu afin de prendre en compte toutes les informations d'un patron, mais également d'y effectuer des manipulations (instanciation, spécification).
Chaque entité d'un méta-pattern est un patron, Chaque patron porte à la fois des informations structurelles et comportementales qui lui sont propres.
UML ne fournit pas de notation spécifique pour modéliser les designs patterns ou même les méta-patterns. Nous avons dû étendre le méta modèle UML afin d'y intégrer notre notation. Cette extension consiste en l'introduction d'un nouveau stéréotype: <<pattern>>.

Le pattern CliSerEL (**Cli**ent-**Ser**ver **E**nrichi par **L**OTOS) a pour but d'expliciter et de formaliser les interactions et les relations entre les différentes classes du pattern (client,

serveur et communication-client-serveur) (Figure5.1).

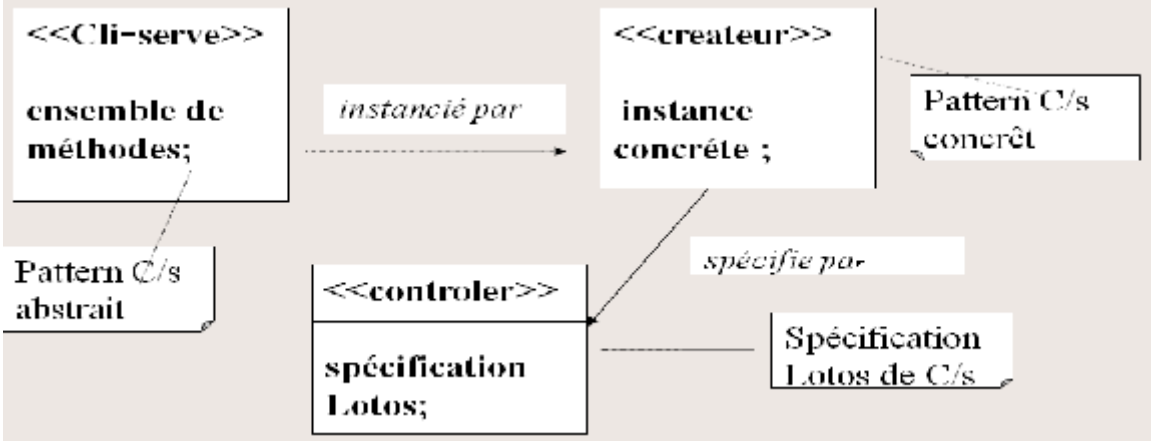


Figure 5.1- Pattern CliSerEL

Notre modèle est constitué, en plus du pattern client-serveur de deux patrons

-Créateur: ce patron est lui aussi un modèle de conception relativement intuitif. Il consiste en la détermination de la responsabilité de la création d'une instance d'un patron.
-Contrôleur: ce patron représente le scénario issu d'un cas d'utilisation. Et est chargé de traiter tous les événements systèmes contenus dans un scénario de cas d'utilisation.

Une application immédiate de ce travail consiste pour nous à regarder l'impact d'une telle méta-modélisation sur les schémas d'applications (framework). En effet, un des avantages de cette modélisation est la représentation des dépendances entre les éléments du patron afin d'expliciter leur instanciation commune. Or, c'est justement à notre sens un des problèmes lié à l'instanciation d'un framework que de comprendre comment les différents éléments qui le composent s'organisent.

Notre framework propose à un utilisateur une bibliothèque de design patterns représentés en UML. (Dans les applications SI, les patterns les plus utilisés et qui sont nécessaires. Citons les patrons: Decorator, Factory method, Proxy, Abstract-factory, Interceptor, Observer [Gam et al. 95]).

Il s'agit pour un utilisateur de choisir un design pattern pour son application et de l'introduire dans son modèle de conception. (Figure 5.2).

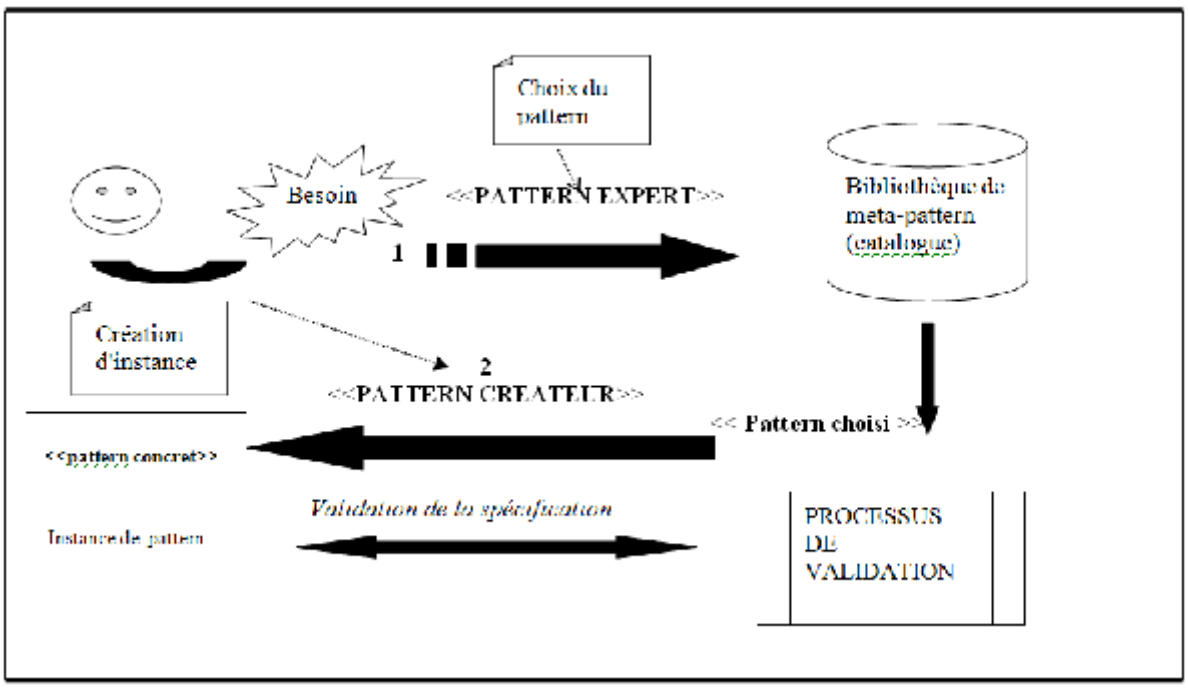


Figure5.2- Framework de développement

I.6 Etude de cas : le système de réunions virtuelles [Pol05]

Il s'agit d'un système serveur permettant d'organiser des réunions virtuelles, dont le fonctionnement est similaire dans le principe aux serveurs de "chat". C'est un exemple tiré de la thèse de damien Pollet [Pol05]. Le cahier des charges de ce système est le suivant :

Il s'agit de réaliser la partie serveur d'une application client-serveur permettant de faire des réunions virtuelles multimédia sur Internet. L'objectif de cette application est de permettre d'imiter le plus possible le déroulement de réunions de travail classiques.

Le serveur devra permettre de planifier et de gérer le déroulement de plusieurs réunions simultanées. Nous supposerons l'existence de programmes clients permettant à des personnes (identifiées par leur adresse électronique) désirant organiser des réunions virtuelles ou y participer de dialoguer avec le serveur.

Les communications se font à travers un réseau local ou Internet selon une architecture *clients-serveur* classique, et nous nous intéresserons à la partie serveur. Via leur logiciel client, les utilisateurs doivent pouvoir :

– planifier des réunions virtuelles (définition du sujet, ordre du jour, date de début et durée prévue),

– consulter les détails d'organisation d'une réunion,

– entrer et sortir virtuellement d'une réunion précédemment ouverte,

– participer aux réunions ; et pour les réunions dont ils sont organisateurs :

– modifier les détails d'organisation,

– ouvrir et clôturer la réunion.

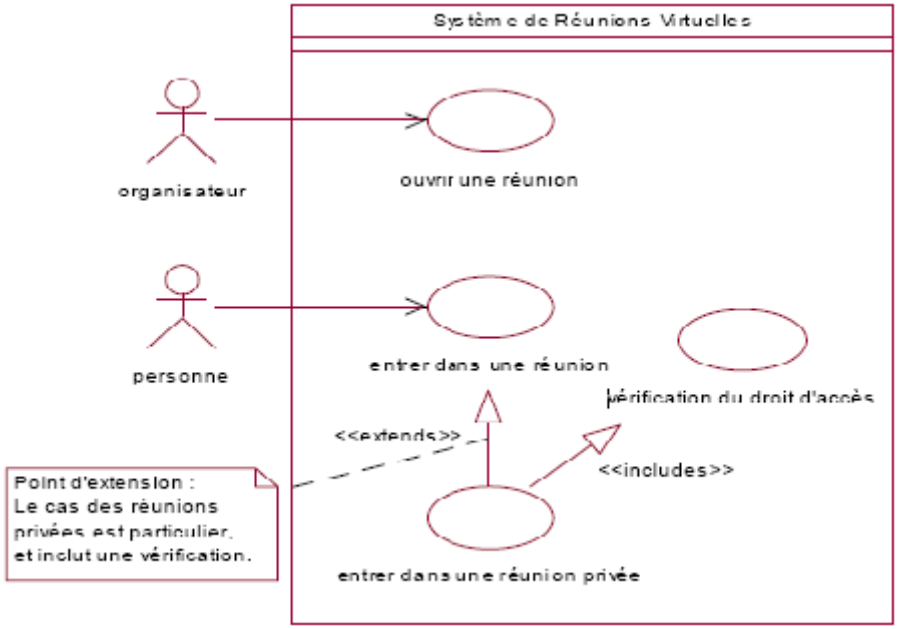
En cours de réunion, un participant peut demander à prendre la parole. Quand elle lui est accordée, il peut entrer le texte d'une intervention qui sera transmise en "temps réel" par le serveur à tous les participants de la réunion.

Plusieurs réunions doivent pouvoir être organisables :

Réunions standards, avec un organisateur qui se charge de la planification de la réunion et désigne un animateur chargé de choisir les intervenants successifs parmi ceux qui demandent la parole.
Réunions privées, qui sont des réunions standards dont l'accès est réservé à un groupe de personnes défini par l'organisateur.
Réunions démocratiques, qui sont planifiées comme des réunions standards, mais où les intervenants successifs sont choisis automatiquement par le serveur sur la base d'une politique premier demandeur, premier servi.

I.6.1 Analyse avec UML

Cas d'utilisation : La première étape de la modélisation consiste à déterminer les besoins des utilisateurs du système, et ceci plus précisément qu'en langage naturel comme dans la plupart des cahiers des charges. Un diagramme de cas d'utilisation ou *use-case* met en relation le système modélisé sous forme d'un rectangle avec des acteurs représentant l'environnement extérieur ; les utilisations possibles du système sont représentées par des ellipses portant le nom du cas d'utilisation et connectée aux acteurs concernés. L'ensemble des cas d'utilisations peut être structuré par des relations de spécialisation, inclusion, ou extension.



Figue6.1 - Une partie des cas d'utilisations des réunions virtuelles

Les *use-case* servent à identifier la frontière entre le système à concevoir et son environnement, ainsi que les fonctionnalités à travers lesquelles les acteurs extérieurs peuvent interagir avec le système.

Pour entrer dans une réunion, un client (préalablement authentifié) doit envoyer le message ENTER réunion (où réunion représente le nom de la réunion). En retour, chacune des personnes qui participent déjà à la réunion en question reçoit le message ENTERING réunion nouveau_participant, et le nouveau participant reçoit quant à lui la liste des autres participants grâce au message PRESENT réunion participant1... participantN.

UML permet de présenter cette même interaction de deux manières différentes :
– Par un diagramme de collaboration, donné par la figure 6.2.

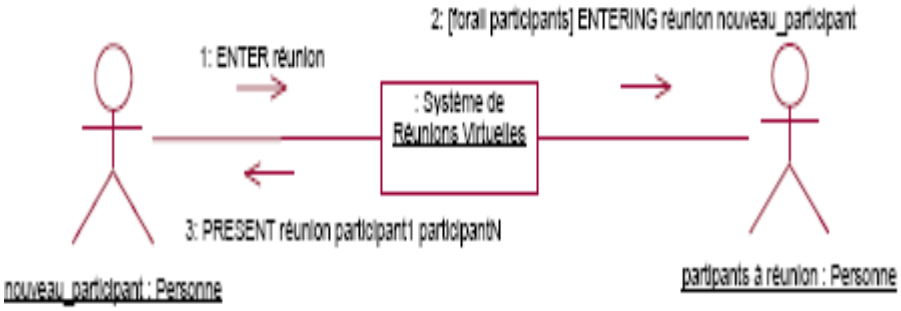


Figure 6.2- Entrée : Diagramme de collaboration

– Par un diagramme de séquences, donné par la figure 6.3

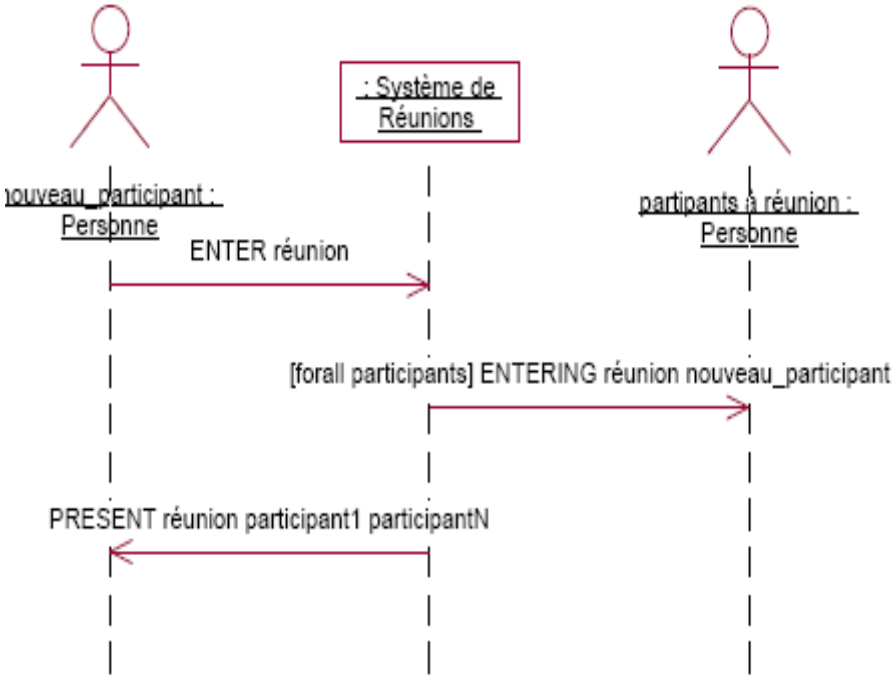


Figure 6.3 - Entrée : Diagramme de séquences

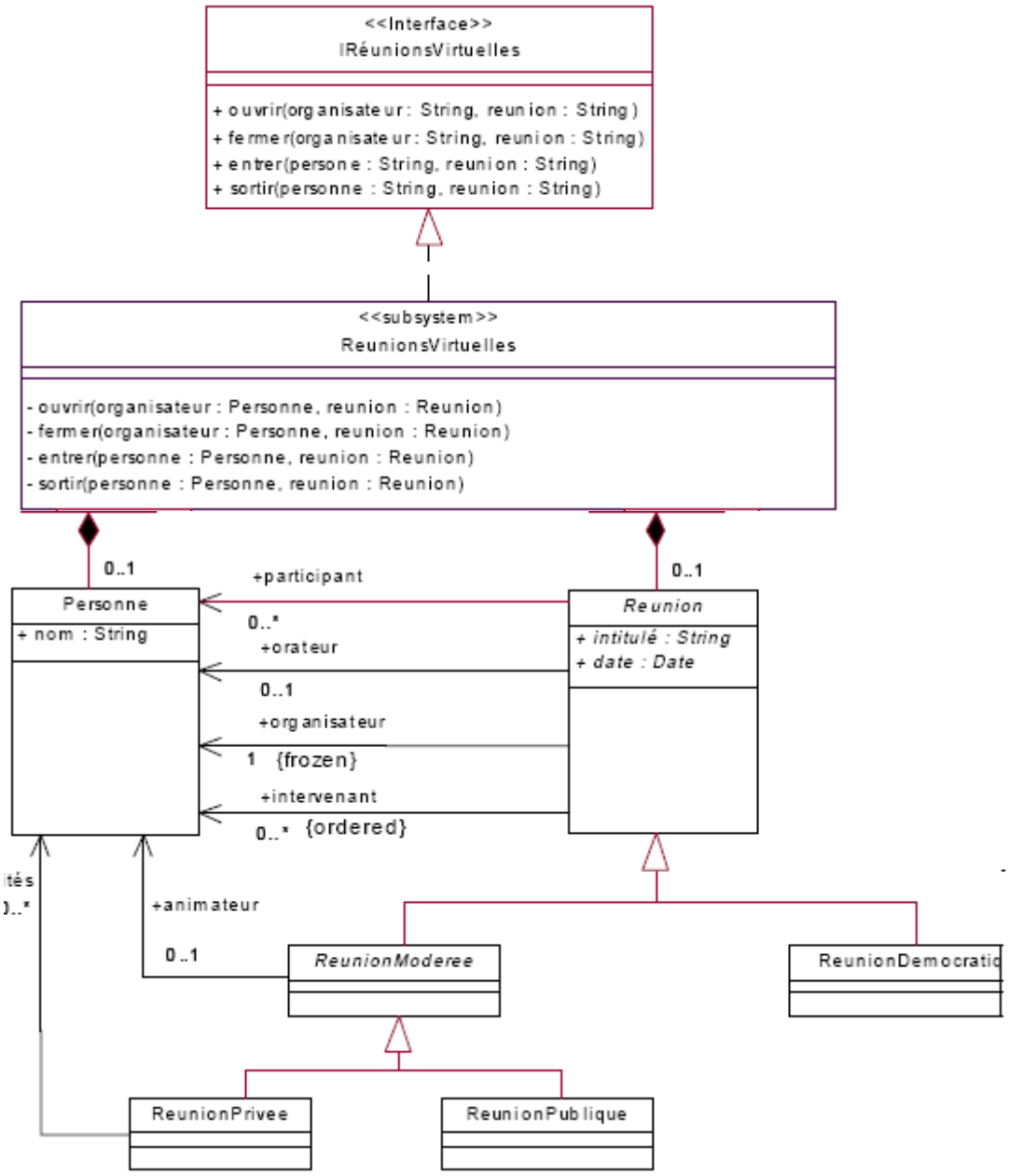


Figure 6.4 - Diagramme de structure des classes à l'analyse des réunions virtuelles

I.6.2 Utilisation des Patrons de conception

La phase initiale de la conception est souvent difficile car il faut passer d'un modèle très abstrait à l'architecture d'un logiciel. Juger des différentes possibilités pour aboutir à une architecture non seulement fonctionnelle, mais aussi évolutive sans être trop coûteuse, requiert expérience et intuition ; en insérant des design patterns dans le modèle d'analyse, on réutilise des solutions « clés en main » et on peut se concentrer sur les choix importants. Par exemple, pour obtenir un modèle de conception du système de réunions virtuelles, il faut s'attaquer à plusieurs problèmes :

– traiter les messages provenant des clients, en utilisant par exemple le design pattern Commande,

– gérer le parallélisme à la réception des messages puis pour les traiter (utilisation du pattern Réacteur),

– créer et maintenir l'état des objets métier (Fabrique et État),
– prendre en compte les changements dynamiques des caractéristiques d'une réunion (Décorateur),
– implémenter la logique d'adaptation des médias en fonction des destinataires d'un message (Stratégie).

Prenons l'exemple du traitement des messages: le serveur va recevoir des messages par l'intermédiaire d'une couche de communication — qui ne sera pas détaillée mais qu'on pourrait implémenter par un composant soap ou Corba par exemple — et il faudra décoder ces messages, vérifier leur validité et leur provenance, puis y réagir. Au vu des capacités de l'application, on peut déjà dire que les types de messages possibles seront assez nombreux et constituent des groupes relatifs aux différents cas d'utilisation. Le patron Commande (figure6.5) consiste à encapsuler un message sous la forme d'un objet, pour pouvoir le manipuler ensuite de manière abstraite; chaque commande concrète est responsable d'aller chercher les informations nécessaires. En découplant ainsi le traitement de chaque message du reste de l'application, on réduit l'impact sur le système de l'ajout d'une commande; les cas d'utilisation et les messages correspondants pourront donc être gérés progressivement tout en gardant un système exécutable.

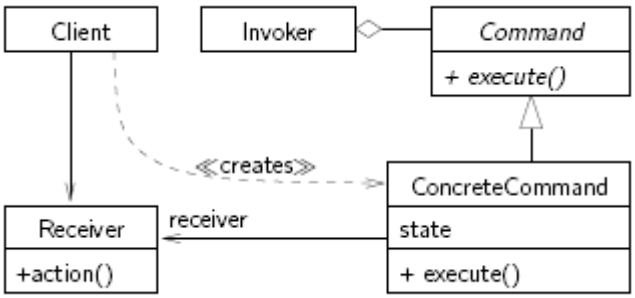


Figure 6.5 -Structure du design pattern Commande. Parmi les classes du système, il faut identifier celles qui joueront les rôles définis ici.

Pour appliquer le pattern au système de réunions virtuelles, il faut distribuer les rôles en créant au besoin de nouvelles classes (figure. 6.6). En particulier pour chaque type de message, on aura une commande concrète implémentant la réaction au message ; ces commandes concrètes se conforment à une interface spécifiée par une classe abstraite Command. Le rôle du *récepteur* est joué par différentes classes du système suivant chaque commande concrète : Person, Meeting. . . Le *client* et l'*invocateur* peuvent être joués par la même classe, VirtualMeetingServer en l'occurrence.
D'une part, en utilisant une construction syntaxique identifiant dans un modèle les classes participant à un pattern, le concepteur peut s'abstraire de détails d'implémentation connus et se concentrer sur des tâches plus importantes. D'autre part, on a besoin d'une telle représentation abstraite des design patterns pour pouvoir automatiser leur insertion ou leur identification dans des programmes existants. Un outil supportant les patterns peut vérifier que les contraintes sont respectées — par exemple qu'un sujet notifie bien tous ses observateurs quand son état change — ou libérer le programmeur de tâches rébarbatives comme l'ajout de méthodes de redirection dans un visiteur ou un décorateur.

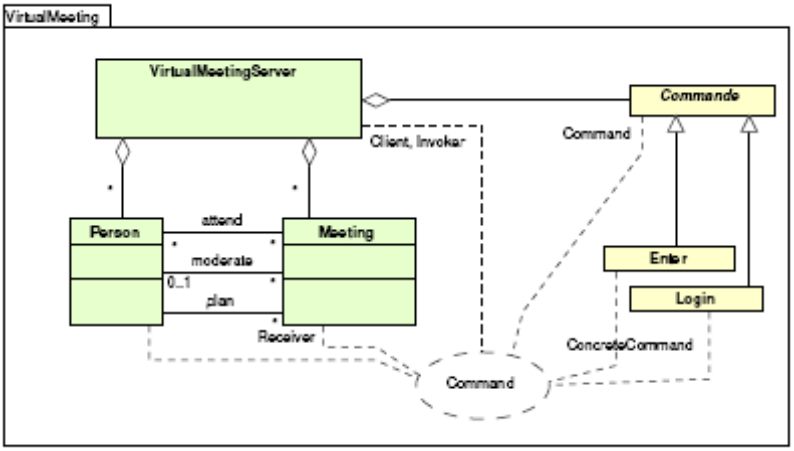


Figure 6.6 - Application de Commande au serveur de réunions virtuelles.

NB. La présence du pattern est mise en évidence en syntaxe UML par l'ellipse en pointillés, indiquant les rôles que jouent chacune des classes.

I.6.3 Exemple : insertion du design pattern Observer

Les patrons de conception ou design patterns sont des solutions génériques à des problèmes de conception récurrents. Si le choix de tel ou tel patron pour résoudre un problème donné dépend de facteurs difficilement quantifiables, l'insertion du schéma de solution dans le modèle de conception est une affaire relativement systématique, et qui gagnerait donc à être automatisée.

Dans le système de réunions virtuelles, quand un participant entre, sort, ou prend la parole dans une réunion, tous les participants de la réunion doivent être notifiés. Durant la phase de conception il faudra donc introduire le design pattern Observer; en passant, l'utilisation du pattern est ici préférable à une solution ad-hoc parce que les participants peuvent arriver ou repartir à tout moment de la réunion, mais aussi parce qu'ils peuvent n'être intéressés que par certains types d'événements. À l'analyse, la présence d'Observer est donc spécifiée dans le modèle du système de réunions virtuelles par le précurseur en figure 6.7.

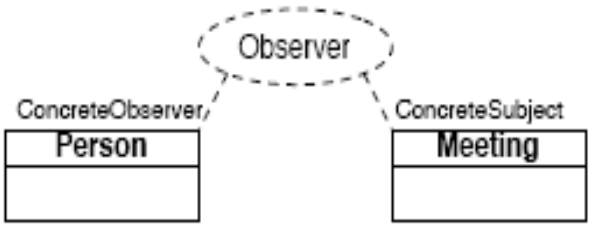


Figure 6.7- Design pattern Observer dans la structure du serveur de réunions virtuelles

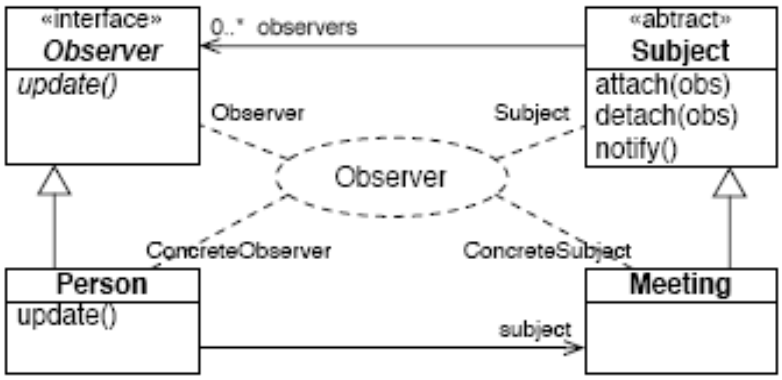


Figure 6.8 - Structure du serveur de réunions virtuelles après ajout du design pattern Observer.

Pour expliciter la structure de ce design pattern et obtenir la figure 6.8, il faut ajouter la superclasse abstraite Subject à Meeting et l'interface Observer à Person.

Dans cet exemple on se contente de déclarer les méthodes, mais dans les cas où le langage d'implémentation est connu il serait tout à fait possible de fournir une implémentation raisonnable pour attach, detach et notify. On pourrait également insérer des appels à notify dans tous les accesseurs en écriture de Meeting, mais cela dépend de la sémantique de chaque application et n'est donc pas souhaitable dans le cas général. De même, l'implémentation de update restera à la charge du développeur.

I.6.4 Exemple : insertion du design pattern Command

Dans le système de réunions virtuelles, la classe VirtualMeetingServer a pour rôle de traiter les messages en provenance des logiciels clients. Ces messages sont acheminés *via* la couche de gestion des communications implémentée dans le package Networking. Il faut en particulier reconnaître le type de chaque message pour pouvoir traiter les données spécifiques à chaque type.

Dans un premier temps ces traitements peuvent être mis sous la responsabilité de VirtualMeetingServer, par exemple sous la forme des opérations spécifiées en figure 6.9. Évidemment, au fur et à mesure qu'on prend en compte de nouveaux aspects du cahier des charges, le nombre de types de messages augmente et avec lui le nombre de méthodes dans VirtualMeetingServer.

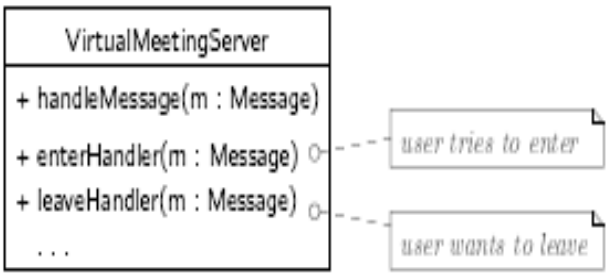


Figure 6.9 - Version préliminaire de la gestion des messages

Une meilleure façon de faire est d'utiliser le design pattern Commande, en déléguant la gestion des messages à une classe spécialisée pour chaque type, plutôt qu'à une méthode de VirtualMeetingServer. Modifier le modèle actuel pour faire apparaître Commande ne changera pas le comportement de VirtualMeetingServer du point de vue d'un appelant de handleMessage() ; par contre, on pourra ensuite ajouter et tester indépendamment les

Conclusion

Dans ce chapitre, nous avons présenté une approche pour spécifier des patterns pour des architectures de systèmes en UML et en LOTOS, permettant ainsi de bénéficier aussi bien des avantages des approches semi-formelles que de ceux des approches formelles. Nous n'avons pas abordé, les différentes discussions sur l'intérêt de tel ou tel pattern [AC98], ni sur la multiple représentation d'un pattern en fonction du langage visé [Sun99].

Néanmoins, à l'instar de plusieurs autres travaux [EGY97, ACL96], nous sommes convaincus que c'est en apportant une approche plus formelle à la représentation des patrons que ces problèmes seront résolus.

Bien que diverses informations sur un patron soient fournies, leur manque de précision peut aboutir à des interprétations ambiguës et incorrectes. En conséquence, la sélection, la composition, l'évolution, l'instanciation et la réutilisation se voient souvent limitées, par le manque de critères précis pour le choix du pattern le mieux adapté à un problème particulier.

L'objectif principal est de proposer aux architectes de Systèmes d'Information un cadre d'analyse et de conception adapté à l'approche de conception de systèmes par composants conceptuels. Ce cadre inclut le concept de contrat pour résoudre les problèmes d'intégration.

Pour cela, le cadre architectural que nous allons définir, propose différentes catégories de contrats apportant des informations complémentaires pour résoudre le problème d'intégration. Ce cadre s'inscrit dans les approches de spécification d'architecture. Dans ces approches, les langages de description d'architecture (ADL : Architecture Description Language) permettent une meilleure compréhension des concepts tels que les composants, les connecteurs et les configurations [Med00]. Cependant, les propositions actuelles ne fournissent pas une vue suffisamment significative et cohérente d'un assemblage fondé sur les contrats.

Une autre perspective de ce travail, est d'exploiter l'enrichissement des design patterns par le langage de spécification LOTOS en tant qu'ADL (Architecture Description Language). C'est dans cette optique, que se situent nos prochaines contributions.

CHAPITRE 7

Une approche basée contrat pour analyser les composants conceptuels

Introduction

L'idée de cette partie s'articule essentiellement sur la proposition d'un cadre formel favorisant la réutilisation et la composition des composants à base de contrats pour la construction des applications. Ceci peut se faire par l'exploitation de l'enrichissement des designs patterns par le langage de spécification LOTOS en tant qu'ADL, pour spécifier et formaliser les comportements structurels et dynamiques des composants. Cette idée à pour finalité de simplifier le travail des concepteurs (architectes) et des développeurs.

La motivation de cette idée est que, dans la construction des systèmes d'information en général, on cherche un certain nombre de critères qualitatifs, tels que la réutilisation, la facilité maintenance et d'utilisation.

Pour résoudre ces critères un des éléments primordiaux est le cycle de vie d'une application. Ce cycle de vie va de l'analyse des besoins à la réforme d'une application, en passant par différentes phases qui peuvent s'organiser de manières différentes en fonction du processus de développement utilisé.

En effet, notre but est de fournir une démarche formelle (sous forme d'un Framework), permettant au développeur de logiciels de modéliser et intégrer des composants pendant la face de conception.

Cette démarche inclus les processus de sélection, d'instanciation, d'intégration et de validation, ainsi que l'analyse de leur composition à base de contrats. Ceci permet aux composants conceptuels d'être réutilisable.

Ainsi, cette démarche permet au concepteur (non seulement) de modéliser les composants (qu'il sélectionne) d'une manière claire et précise, et aussi de détecter les erreurs d'interaction entre les composants et pouvoir les corriger avant leur mise en œuvre.

I.1 Description de la démarche

Nous nous intéressons dans notre démarche à l'étape de construction et de modélisation des applications à base de composants réutilisables. Cette étape consiste à construire l'architecture de l'application, c'est-à-dire la modélisation des composants participant à la réalisation d'un service et leurs interactions. Pour cela nous définissons notre propre démarche de construction d'application basée sur un modèle de composant abstrait permettant de spécifier, construire, déployer et superviser une architecture logicielle typée.

Dans ce chapitre, nous présentons une vue d'ensemble de notre approche. Nous séparons la spécification abstraite des composants de leur mise en œuvre (Figure 1).

Notre but principal est de fournir une approche systémique pour un développeur de logiciel et faciliter le choix des patterns selon les besoins de l'utilisateur, afin de modéliser et analyser l'intégration des composants pendant la phase de modélisation.

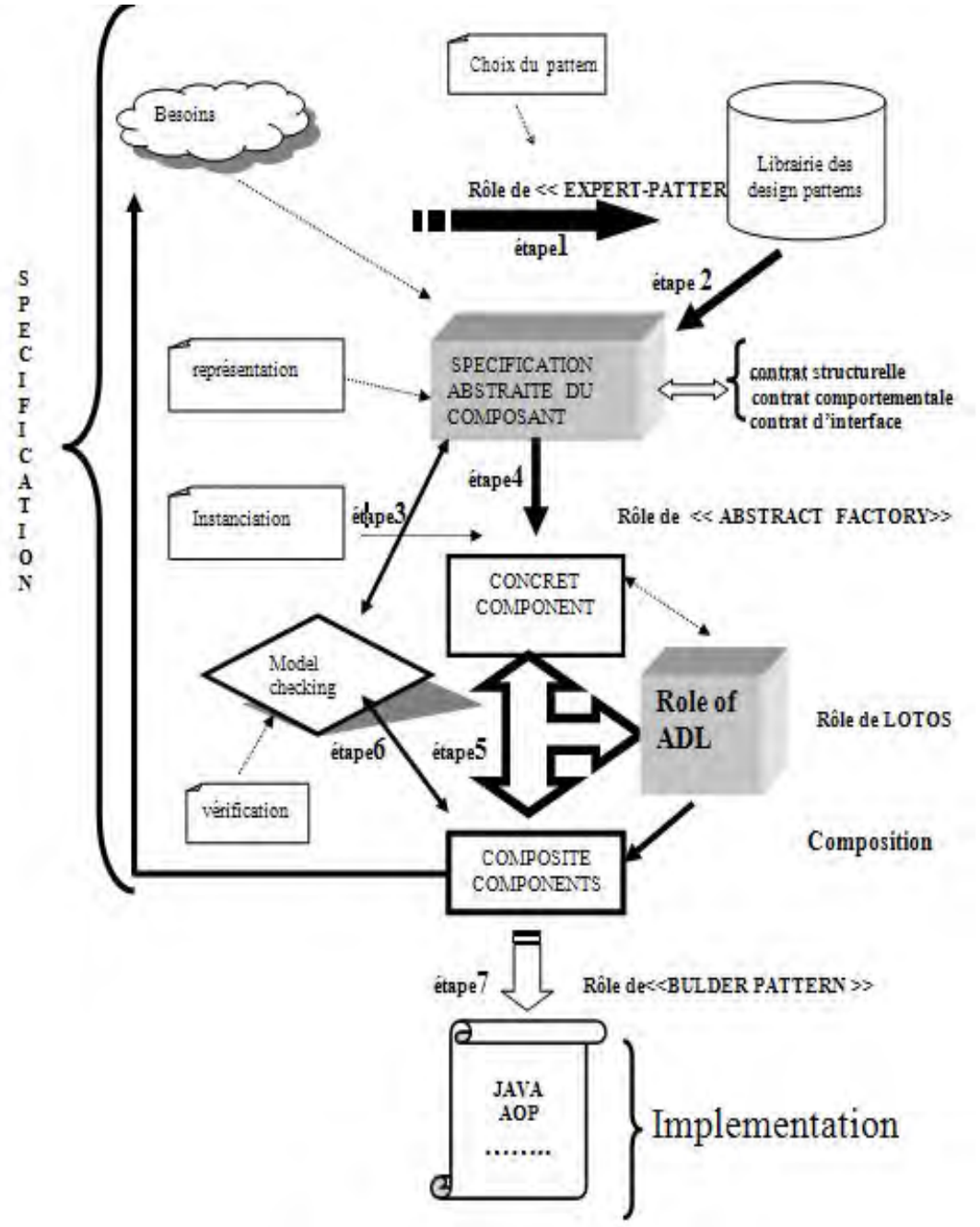


Figure 1- Approche proposée

La démarche suit les étapes suivantes:

Etape 1 : Choix du design pattern (selon les besoins de l'utilisateur) disponible dans une bibliothèque de patrons (Patterns créateurs, Patterns structuraux, et Patterns comportementaux) (Par utilisation des « use cases d'UML » par exemple). Dans cette étape, nous analysons les responsabilités et les fonctionnalités de chaque composant, et nous identifions le pattern candidat. Pour ce faire, nous avons développé un environnement permettant à un utilisateur de choisir le pattern nécessaire pour son application.

Etape 2 : Spécification abstraite du design pattern : contient un modèle formel de design pattern, appelé ***contrat pour design pattern (ASC)***, et une catégorie de propriétés structurelles et comportementales et d'interface.

Etapes 3, 4, 5 ,6 : Validation, Evolution, Composition et Instanciation, des composants, nous proposons l'adoption du langage LOTOS pour spécifier le comportement de ces derniers (LOTOS est utilisé en tant qu'un langage de description d'architectures).

Etape 7: Génération du code JAVA : Dans cette partie nous proposons des règles de transformation des spécifications LOTOS vers le langage JAVA.

I.2 Description des étapes.

I.2.1 Choix du design pattern (composant conceptuel)

Notre objectif est d'outiller un environnement de développement intégré pour lui ajouter les assistants logiciels permettant (figure2):

i) de représenter un catalogue de patterns ;
ii) d'instancier les patterns de ce catalogue pendant le développement ;
iii) de reconnaître ces patterns dans du code source existant.

Pour cela, nous avons développé un environnement graphique permettant de représenter les différents partons de conception qui existent. Pour chaque patron nous avons donné sa spécification LOTOS, ainsi que son implémentation JAVA.

Dans un premier temps, nous avons crée une liste des candidats potentiels. Une fois cette liste complète, nous analysons chaque composant et nous déciderons si on l'inclut ou non dans notre système.

Nous allons présenter brièvement notre environnement de développement, ainsi que la façon de l'utiliser. Il est composé d'un ensemble de fenêtres (interfaces) permettant à un utilisateur de consulter une bibliothèque de patrons, de faciliter le choix du patron qui répond à ses besoins et comment l'utiliser.

La première fenêtre représente le menu principal. Elle se divise en quatre parties qui sont (voir Annexe B2 pour plus de détails) :

1. le menu patron : est une représentation des différents patrons existants. Il donne la possibilité à l'utilisateur de choisir un patron parmi ceux existants. Il se décompose en trois sous menu (structuraux, créateurs, et comportementaux) qui donnent un accès direct à un patron. Chaque patron est présenté avec son diagramme de classe et de séquence, ainsi qu'une brève description.

Nous proposons aussi, pour chaque patron, une ***spécification LOTOS,*** cette dernière n'apparaît que si le diagramme de séquence de ce patron existe, sinon un message d'erreur s'affichera. L'utilisateur peut aussi consulter l'implémentation ***JAVA*** du patron sélectionné, passer au suivant ou retourner à la page précédente, ou encore annuler toutes les opérations et quitter l'application on cliquant sur le bouton : « **annuler**»

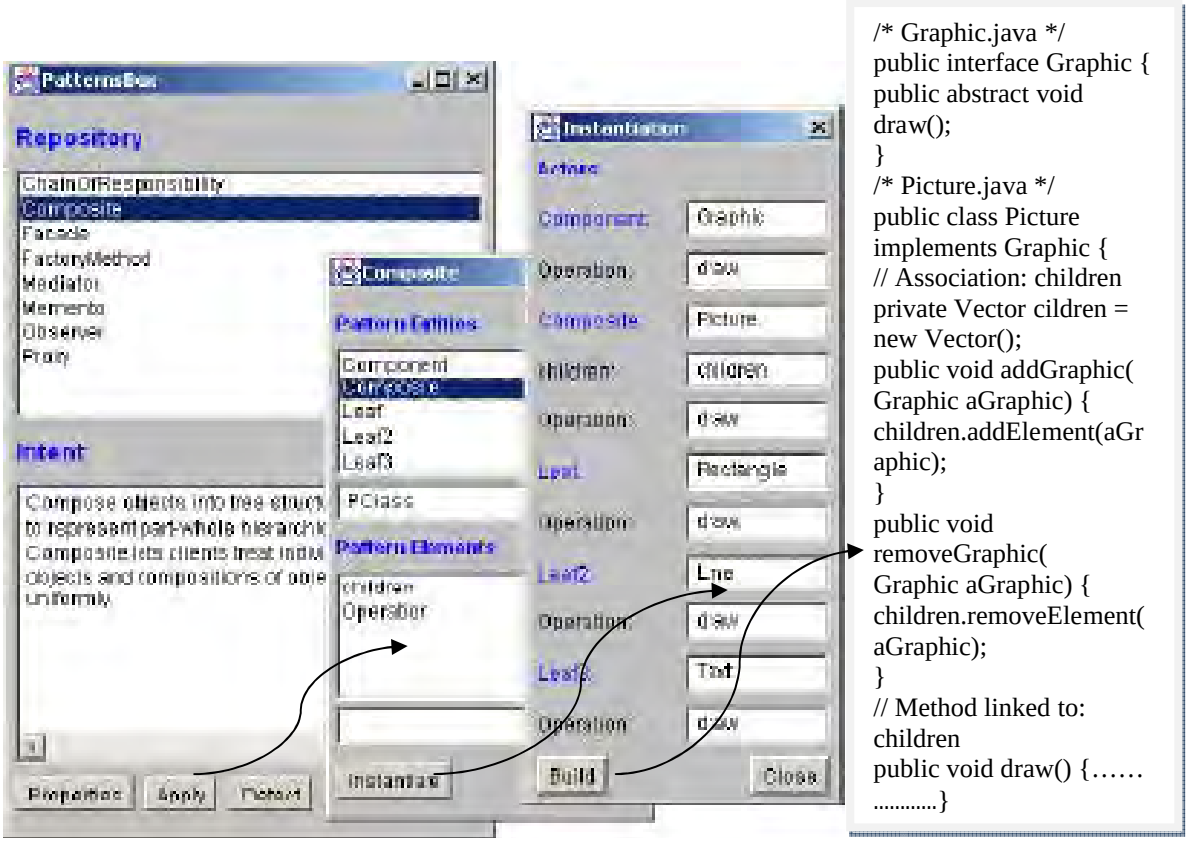


Figure 2- Environnement de développement

I.2.2 Spécification abstraite du composant choisi

Un patron décrit un problème fréquemment rencontre dans un contexte ainsi que la solution consensuelle qui le résout. Dans le domaine des systèmes informatiques et pour ce qui nous intéresse dans celui de la conception de systèmes d'information, on peut bien évidemment citer les patrons de conception et parmi les plus connus ceux du Gang of Four (GoF) (Gamma et al. 1995), qui nous serviront à la fois de référence et d'exemple.

Bien que complétée par de nombreuses informations, la solution d'un patron est souvent limitée a un diagramme de classes UML, qui est pour nous une spécification incomplète de cette solution. Nous proposons de spécifier les patrons sous la forme d'un mini-système à deux vues: dynamique (diagrammes de séquences) et statique (diagrammes de classes).

Dans de nombreux cas, et particulièrement pour le GoF, la description du patron est clairsemée d'informations exprimant des variantes possibles de cette solution, et ceci selon tous les aspects : fonctionnels, dynamiques ou statiques. Nous intégrons cette variabilité au sein de la spécification des patrons afin d'instrumenter la sélection de variantes dans le

processus de réutilisation. Ce processus, permet d'extraire la solution du patron et de l'appliquer dans un système en construction.

De plus, et dans l'optique de réutilisation et d'intégration d'un composant logiciel, il est indispensable de comprendre très précisément ce que fait ce composant. Ceci implique d'associer au composant des spécifications précises. Ces spécifications décrivent les services offerts par le composant, mais aussi ceux dont il a besoin pour fonctionner. Elles représentent l'ensemble des contraintes que le composant impose à ses utilisateurs, ainsi que celles venant de son environnement qu'il s'impose de respecter.

Pour ce faire, nous allons donner au design pattern une spécification abstraite. Les spécifications abstraites décrivent de manière indépendante de toute implantation les opérations visibles (offertes ou requises) par un composant. C'est à ce niveau que figurent les principales informations permettant de choisir un composant pour les fonctions qu'il réalise (figure3).

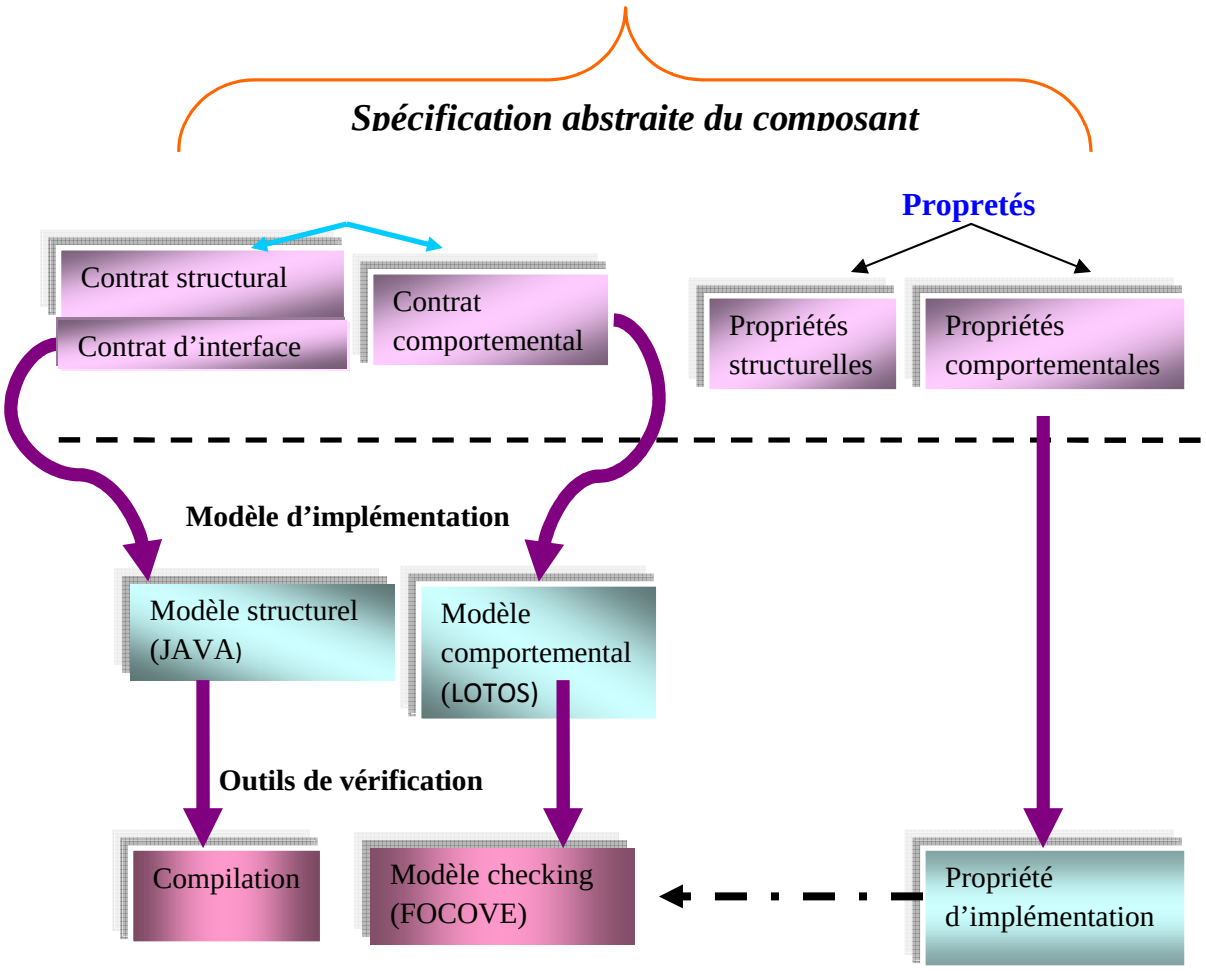


Figure3- Vue générale d'une spécification abstraite du composant

Nous allons isoler trois aspects principaux de modélisation d'un contrat. Selon la nature de l'entité (type, classe ou instance), et la nature des propriétés exprimées (syntaxique, sémantique).

La spécification abstraite contient un modèle formel de design pattern, appelé **contrat pour design pattern (ASC : Abstract Specification Contract)**, et une catégorie de propriétés structurelles et comportementales et d'interface.

Le contrat pour design pattern (ASC) est décrit par des contrats structuraux (SC), comportementaux (BC) et d'interface(IC) (figure4).

Les contrats structuraux décrivent l'aspect structurel d'un composant sous forme de classes, d'attributs, de méthodes, et relations entre classes telles que : les relations d'héritage, d'agrégation, d'association etc.….

Les contrats d'interface définissent l'ensemble des ports d'entrée et de sortie des composants, ainsi que l'ensemble des messages émis et reçus par les composants.

Par contre les contrats comportementaux décrivent l'aspect dynamique du composant, tels que la collaboration entre les objets du composant qui participent dans la création et la suppression de nouveaux objets. Pour cela nous avons adopté l'utilisation du langage LOTOS pour définir un modèle formel de contrat comportementaux, car il représente une approche puissante pour modéliser les comportements.

Définition: Nous définissons ainsi, la spécification abstraite d'un composant basé sur la notion de contrat comme suit :
Chaque contrat du composant possède un nom unique, et est décrit par des contrats structurel, comportemental et d'interface et un ensemble d'opérations. Nous distinguons trois types d'opérations:

1. **Instanciation :** comme chaque contrat est un méta modèle du design pattern. L'application du composant exige l'instanciation de ses définitions génériques.
2. **Evolution** : chaque composant est modélisé suivant les changements de l'environnement dans lequel il décrit le chemin de l'évolution attendu, en prenant en compte l'ajout et/ou la suppression de certaines parties du design pattern.
3. **Intégration :** la composition est définie par différents aspects de chaque composant individuel. Sa composition structurelle est accomplie par l'union, alors que celle du contrat comportemental est accomplie par composition parallèle.

Ces contrats représentent l'ensemble des contraintes que le composant impose à ses utilisateurs, ainsi que celles venant de son environnement qu'il s'impose de respecter.

Le terme de *contrat* est utilisé pour décrire les propriétés nécessaires au bon fonctionnement du composant.

Syntaxiquement, nous définissons « ***contrat pour design pattern (ASC)*** » du design pattern comme suit :

ASC ::=< Component-Name> Where <assertion>and <SC>and<IC>and <BC> End

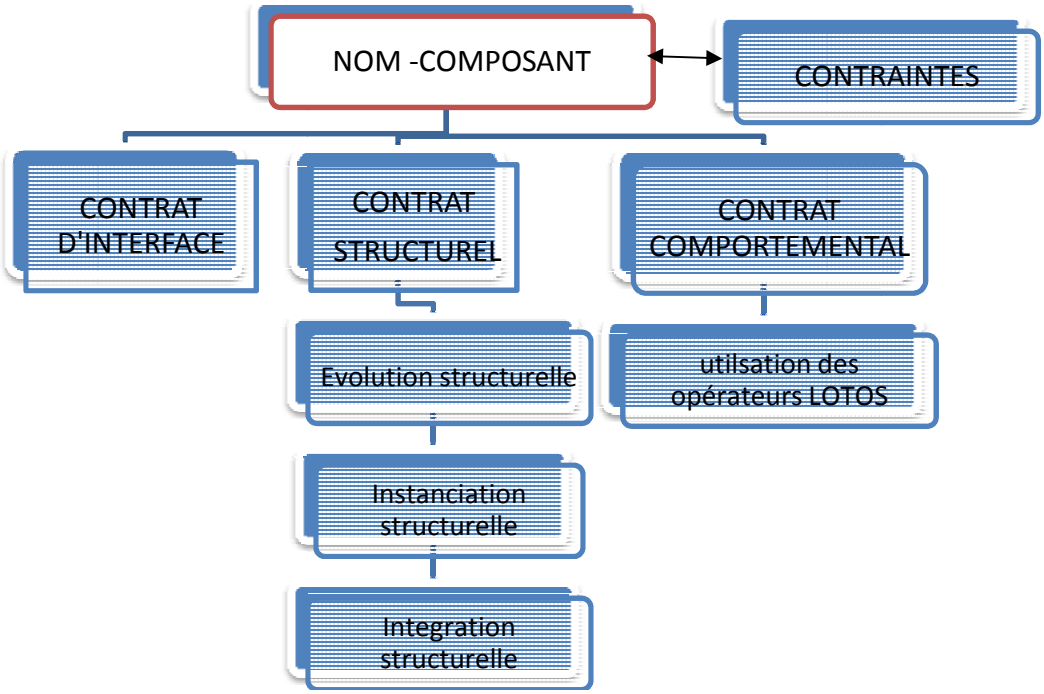


Figure 4- Contrats pour design pattern.

1. Contrats structurels:

Nous définissons l'aspect structurel d'un contrat du design pattern comme suit [Zit et al. 08b]:

SC = (C, A, M, T, Ar, Pc, Pa) tels que:

- **C** est un ensemble de classes du composant qui définit les participants dans chaque design pattern.
- **A** est un ensemble d'attributs défini dans les classes C.
- **M** est un ensemble de méthodes défini dans les classes C**.**
- **T** est un ensemble de types qui sont utilisés pour définir les attributs et méthodes dans les classes C.
- **Ar** est un ensemble de droits d'accès que les attributs et méthodes peuvent avoir dans une classe de C.

- Par exemple : Ar = {public, protégé, privé}.

- **Pc** est un ensemble de symboles d'attributs de connexion qui capturent les relations types entre le rôle de chaque design pattern.

- Par exemple (Héritage, association, agrégation,..).

- **Pa** est un ensemble de symboles d'attributs d'actions qui peuvent s'exécuter.

- Par exemple dans un design pattern (invoquer, nouveau, retour…).

Pour la formalisation de l'aspect structurel du design pattern, nous proposons l'utilisation de la logique du premier ordre (FOL) [Zit et al.08b]. En effet les relations entre les participants du design pattern, peuvent être facilement exprimées avec le formalisme des logiques de descriptions en tant que prédicats [TN01]. Dans le formalisme des logiques

de descriptions, un concept permet de représenter un ensemble d'individus, tandis qu'un rôle représente une relation binaire entre individus.

Les logiques de descriptions permettent de représenter les connaissances d'un domaine à l'aide <<descriptions>> qui peuvent être des concepts (classes d'individus), des rôles (relations entre classes) et d'individus. Les concepts et les rôles sont organisés en hiérarchies sur lesquelles opèrent les processus de classification et d'instanciation.

Pour décrire l'aspect « contrat structurel du composant », nous proposons l'utilisation des prédicats suivants : « connectivité ('∧'), quantification ('∃'), et élément (ϵ) », ainsi qu'un ensemble de noms de concepts.

Nous définissons deux groupes de prédicats :

* **prédicats des entités:** définissent si un composant conceptuel admet une classe spécifique (abstraite ou concrète), comment une méthode est elle définie dans une classe (Table1).

* **prédicats des relations :** définissent les relations entre classes, attributs, opérations et les actions que leurs rôles peuvent assumer dans un composant (Table2).

Table1- Prédicats des entités

Prédicats	Description
Abstract-Class (C)	C joue le rôle d'une classe abstraite dans un composant
Class (C)	C joue le rôle d'une classe concrète dans un composant
x ϵ X	x est un élément de l'ensemble X

Notons que, La classe abstraite (*abstract class*) est une implémentation abstraite d'un type de données. Cette description possède les mêmes propriétés qu'une classe mais elle peut décrire des méthodes abstraites (sans corps) et ne peut pas posséder d'instance propre.

Table 2- Prédicats des relations

Prédicats	Description
Inherit (A, B)	B est une sous classe de A
Associate (A, B)	A, B sont connectés avec la relation d'association
Aggregate(A,B)	A contient une référence à B
Invoke (A,m1,B, m2)	La méthode m1 définie dans la classe A appelle la méthode m2 définie dans la classe B
New(A,m,O)	La méthode m de la classe A crée un nouvel objet de type A
Return (A,m,O)	La méthode m de la classe A retourne un objet O de type A
Reference (C1,C2, n,m)	La multiplicité des relations d'association ou d'agrégation de C1 à C2

Nous allons illustrer notre proposition à l'aide de l'exemple du patron« observer ». Considérons la structure (diagrammes de classes, d'interaction) du design pattern « **observe**r» (figures4.1et 4.2):

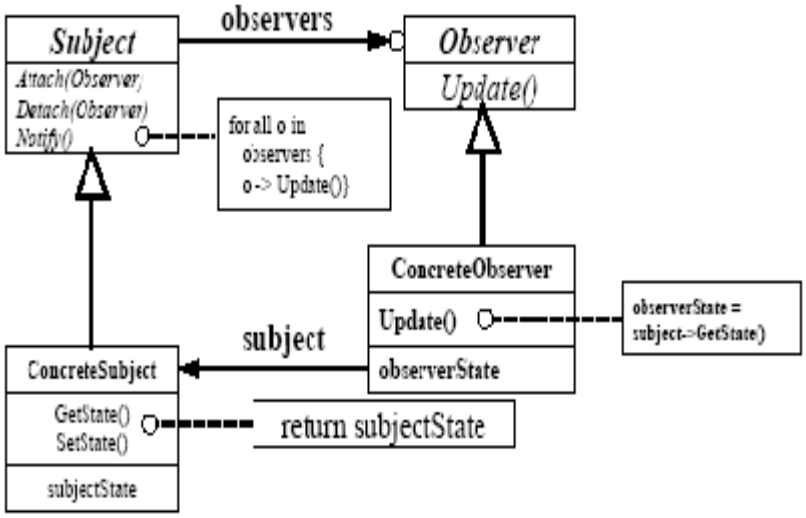


Figure 4.1- Diagramme de classes du patron Observer

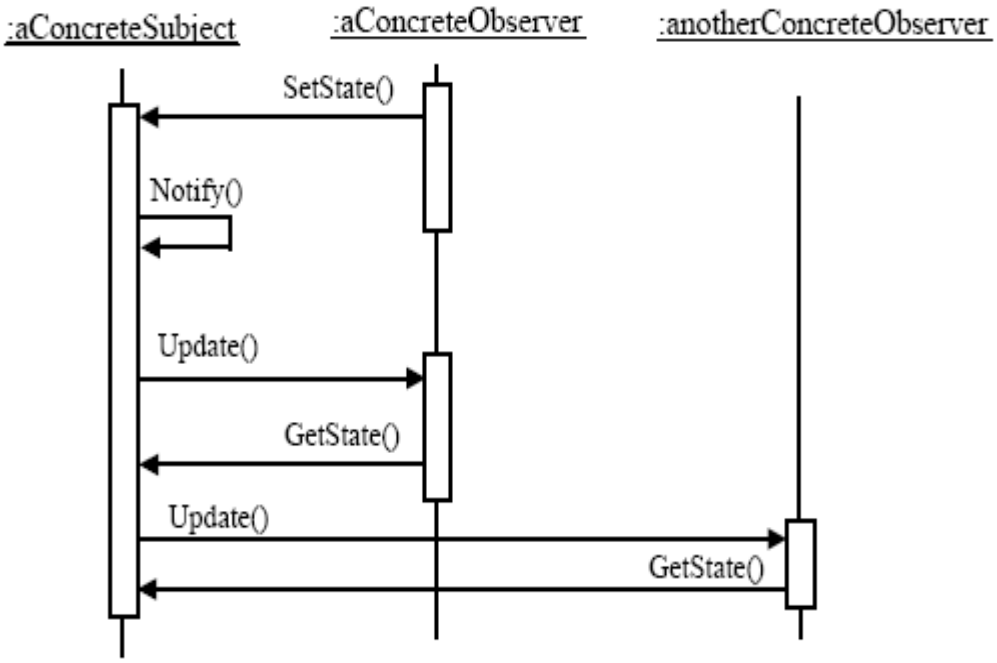


Figure 4.2- Diagramme de séquences du patron Observer

La spécification du contrat structurel du design pattern Observer est donné par [Zit08b] :

ASC= <observer> and <SC> = (C, A, M, T, Ar, Pc, Pa) where.

(0) Component-name is Observer ***where:***
(1) ∃ ***abstract-class*** *(Subject, Observer) ∈ C;*
(2) ∧ ∃ ***class*** *(ConcreteObserver, ConcreteSubject)} ∈ C;*
(3) ∧∃ *(attach, detach, getstate, update, notify) ∈ M;*
(4) ∧ ∃ *(void, datatyp)} ∈ T;*
(5) ∧ ∃ ***Inherit*** *{(Observer, ConcreteObserver)*
∧ (Subject, ConcreteSubject)};
(6) ∧∃ ***Invoke*** *{(Invoke (Subject, attach, observer, append)*
∧ (Subject, detach, observer, remouve)
∧ (Subject, notify, observer, update)};
(7) ∧∃ ***Return*** *(concreteSubject, getstate, subjectstate)*
(8) ***Where*** ∃ *Method {(attach, detach, notify) ∈ Subject*
∧ (updtate) ∈Observer ∧ (getstate,notify) ∈ConcreteSubject
∧ (updtate) ∈ConcreteObserver}

Cette description formelle de l'aspect structurelle du patron observer est facilement compréhensible et peut être traduit sans difficulté majeure dans un pseudo code proche à l'implémentation JAVA [Zit08b].

```
Public interface Observer {
        Public void Update (subject s) ;}
Public interface Subject {
        Public void attach (Observer o) ;
        Public void detach (Observer o);
        Public void notify ();     }
Public Class Concrete Subject implements Subject {
        Public void attach (Observer o) {……};
        Public void detach (Observer o) {……};
        Public void notify () {……};            }
Public Class Concrete Observer implements Observer {
        Public void Update (subject s) {……} ;}
```

1.1 Instanciation structurelle

Le contrat structurel du design pattern est instancié, évolué et intégré avec d'autres contrats dans une application particulière. Pour qu'il soit intégré dans une application spécifique, le design pattern doit être instancié.

Nous définissons l'opération d'instanciation du contrat structurel du composant notée <<instance- du- composant>> comme suit :

<<instance-du-Composant-name>> := <SC, SC', > avec :

-SC = (C, A, M, T, Ar, Pc, Pa) désigne le contrat structurel du composant Conceptuel <composant-name>,
-SC' = (C', A', M', T', Ar', P'c, P'a) désigne l'instance du contrat structurel du composant conceptuel <composant-name>

- $: C \times A \times M \times T \times A \times Pc \times Pa \rightarrow C' \times A' \times M' \times T' \times A' \times P'c \times P'a,$

Tels que :

C' = (C) est l'ensemble de classes concrètes du design pattern,
A' = (A) est l'ensemble d'attributs défini dans les classes C',
M' = (M)= M'
T' = (T) = est l'ensemble des nouveaux types pour définir les attributs et les méthodes dans les instances de classes C'.
Ar' = (Ar) = Ar
Pc' = (Pc) = Pc
Pa' = (Pa) = Pa

Voici un exemple : Considérons l'instance (figure5) du design pattern "Observer", définie dans la figure4.1. Cette instance décrit « l'application de visualisation » des éléments de données (a, b, c).

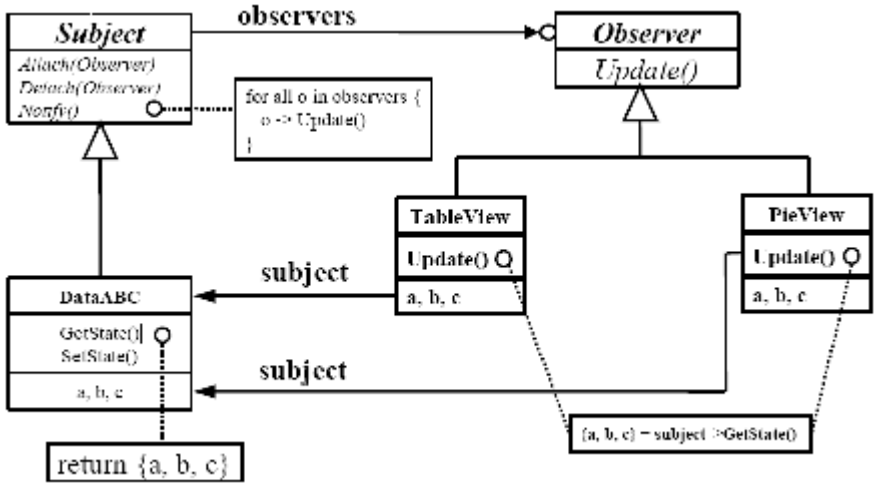


Figure 5- Une instance du Pattern "Observer"

Le contrat structurel SC de l'instance du pattern (observer) sera définie comme suit:

ASC : = <<instance-de-observer>> and <SC', > where
SC' = (***C', A', M', T', Ar', P'c, P'a)*** avec l'opération d'instanciation **,** avec :

- **C'= (C))** est l'ensemble des classes de l'instance du design pattern observer.
C'= {Subject, Observer, TableView, PieView, DataABC}

- ***A' =*** ***(A)*** est l'ensemble des attribues définis dans les classes C'
A' = {subject, observers a, b, c}

- ***M' =*** ***(M)= M'*** est l'ensemble des méthodes définies dans les classes C'

M' = {attach, detach, GetState, update, notify, append, remove},

- ***T' =*** ***(T)*** est l'un ensemble de types qui sont utilisés pour définir les attributs et méthodes dans les classes C'

T'= {void, float},

- ***Ar' =*** ***(Ar) = Ar*** est l'ensemble de droits d'accès que les attributs et méthodes peuvent avoir dans une classe de C

Ar = {public, protégé, privé},

- ***Pa' =*** ***(Pc) = Pc*** est l'ensemble de symboles d'attributs de connexion qui capturent les relations types entre le rôle de chaque design pattern

Pc'= {inherit} avec:

∃**Inherit** {(Observer, TableView)
∧ (Observer, PicView)
∧ (Subject, DataABC)}

- ***Pa' =*** ***(Pa) = Pa*** est l'ensemble de symboles d'attributs d'actions qui peuvent s'exécuter Pa {invoke, return} avec:

∃ **Invoke** {(Subject, attach, observer, append)
∧ (Subject, detach, observer, remouve)
∧ (Subject, notify, observer, update)
∧ (TablView, update,Subject, Getstate)
∧ (Pieview, update, subject, GetState) }

∃ **Return** {(DataABC, getstate, a)
∧ (DataABC, getstate, b)
∧ (DataABC, getstate, c)}

1.2 Evolution structurelle

Afin de suivre des besoins utilisateurs en constante évolution, un logiciel doit être régulièrement modifié. Actuellement, même si une phase d'analyse sérieuse a été effectuée et qu'une spécification de l'architecture en résulte, celle-ci est souvent absente de la phase d'évolution. Le problème est alors double:
- premièrement il apparaît un décalage entre la spécification et l'application,
- deuxièmement l'analyse sur la qualité de la spécification est perdue,

Le but principal des designs patterns est leurs capacités d'évolution.
Après avoir choisi et introduit le design pattern, le concepteur peut changer la conception de son application. Ce changement peut être guidé par cette capacité d'évolution des designs patterns. L'information d'évolution de chaque pattern permet au concepteur de changer le système avec un minimum d'impact sur les autres systèmes

Pour cela nous proposons une solution qui est basée sur l'extension de l'aspect contrat structurel du composant. En effet nous définissons de nouvelles primitives et contraintes.
Les primitives (table3):

Table3- Prédicats d'évolution

Primitives	Description
+(R,C1,C2)	**ajout de la relation R entre les classes C1 and C2**
- (R,C1,C2)	**Suppression de la relation R entre les classes C1 and C2**
C1* C2 (r)	**C1 et C2 sont associés par r**
C1 # C2	**C1 et C2 sont identiques**
C1 as R(C2)	**C1 peut jouer le rôle R**
Tr (C1, r_i)	**Définis tous les relations de C1**

Ainsi donc, l'aspect du contrat structurel du composant est définit comme suit :

SC = (SC–, SC~, preSC, postSC), tel que:

- SC– définit l'aspect structurel « statique » du composant (permanent)
- SC~ définit l'aspect structurel « dynamique » du composant (qui peut changer)
- preSC et preSC sont des contrainte (invariant, post-condition and pre-condition).

Nous définissons l'aspect évolutif du composant comme suit :

SC~ = (C~, R~, TR(C, r1,….,r_i)) tel que

- C~ est la liste des classes du design pattern, qu'on peut instancier, ajouter et supprimer dans le même composant,
- R~ sont les listes de toutes les relations de C~, et
- TR= U_i(Tr_i, i=1,..) sont les traces de chaque C~, avec les contraintes:

SC =SC~ ∪ SC– and C~ ∩ C– = {∅}

Par exemple l'aspect structurel du contrat sera définit comme suit :

(0) Component-name is Observer **Where**:
(1) ∃ **abstract-class** (Subject, Observer) ∈ SC–;
(2) ∧∃ **class** (ConcreteObserver, ConcreteSubject)} ∈ SC~;
(3) ∧ …….
∧ (ConcreteObserver,ConcreteSubject)} ∈ C~.
(..) ∧ (Subject, ConcreteSubject) }∈ TR~ ;
(..) **Where** TR~= {TR~ (ConcreteObserver, Inherit),
TR~ (ConcreteSubject, Inherit) }

Les preSC, postSC sont des contraintes sur les primitives (+(R,C1,C2) and -(R,C1,C2)). Avec ces contraintes on peut affirmer que, la pre-condition de l'évolution est vérifiée, et que la post-condition sera vérifiée après évolution. Autrement dit, l'ajout ou la suppression des classe d'un composant, n'influx pas la structure du composant.

L'ajout d'une classe dans le design pattern est possible, si la nouvelle classe admet une copie (joue le même rôle) dans le design pattern, et que la classe ancienne appartient (membre de) à SC~ (c'est à dire: Cardinalité de TR > 1).

La primitive d'évolution est définie comme suit:

+(R, C1, C2):= **new** C1
Pre: for all $C_i \in SC \sim \wedge$ card $(TR(C, r_1, r_2, \ldots, r_i) \geq 1$
$\wedge$ C1 # C_i $\wedge$ C1 as R (C2)
Post: for all $r_i \in TR(C, r_1, r_2, \ldots, r_i)$ then C1* C2 (r_i)

Par exemple l'ajout d'une nouvelle ConreteObserver (Figure6) sera exprimé par la primitive d'évolution:

(**new**(ConcreteObserve2) $\wedge$ +**Inherit** (Observer,Concrete Observer2)).

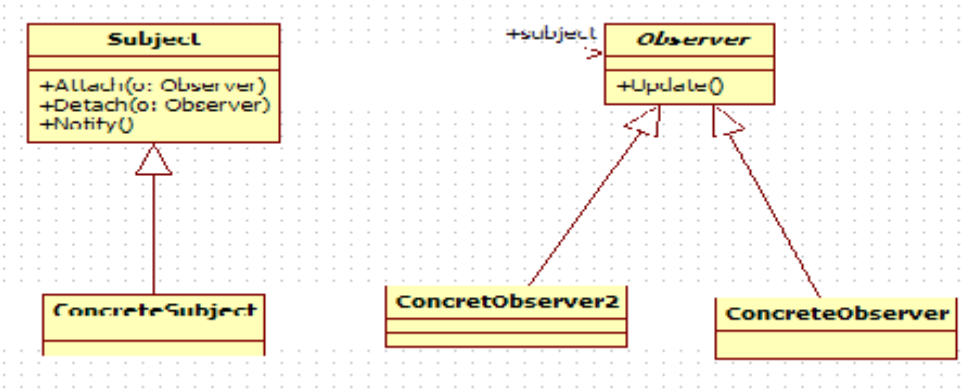


Figure 6- Le pattern Observer (avec deux ConcretObserver)

1.3 Composition structurelle :

Une fois que tous les patterns concernés par notre application seront choisis et validés, l'étape suivante serait de les composer afin de modéliser l'application complète.
Chaque intégration peut ce faire avant ou après le processus d'instanciation.

Nous définissons l'intégration de deux composants comme suit [Zit et al.08c]:

Soient SC_1= (C_1, A_1, M_1, T_1, Ar_1, Pc_1,Pa_1) et SC_2= (C_2, A_2, M_2, T_2, Ar_2, Pc_2,Pa_2) deux contrats structuraux, alors l'intégration de SC_1 et SC_2 dénoté par SC =(C, A, M, T, Ar, Pc,Pa) par la fonction de «mapping » que nous définissons : ∇ tel que :
∇: $C_1 \cup C_2 \rightarrow C$, $A_1 \cup A_2 \rightarrow A$, $T_1 \cup T_2 \rightarrow T$, $Ar_1 \cup Ar_2 \rightarrow Ar$, $Pc_1 \cup Pc_2 \rightarrow Pc$, $Pa_1 \cup Pa_2 \rightarrow P$

Avec comme contrainte: $C = \nabla C_1 \cup \nabla C_2$ / $\forall c \in C_1 \cup C_2$ et $\nabla c \in C_1 \cup C_2$

Considérons le contrat structurel SC(mediator) du design pattern (Mediator),
L'intégration de l'instance Mediator ($SC_{mediator}$) avec l'instance du pattern Observer ($SC_{observer}$) (voir Figure7), est définie par:

SC=(C, A, M, T, Ar, Pc,Pa), et les fonctions de « mapping »:

∇_1: $C_1 \rightarrow C$, et ∇_2: $C_2 \rightarrow C$, tels que:

-L'ensemble des classes du design pattern est C= {Subject, Observer, TableView, PieView, DataABC, Mediator, ConcreteMediator}

-L'ensemble des attribues definis dans les classes C est AV = {subject,observers, mediator, a, b,c}

-L'ensemble des methodes définies dans les classes C est M = {attach, detach, GetState, update, notify, append, remove},

-L'ensemble de types qui sont utilisés pour définir les attributs et méthodes dans les classes C est T={void, float},

-L'ensemble de droits d'accès que les attributs et méthodes peuvent avoir dans une classe de C est Ar = {public, protégé, privé},

- L'ensemble de symboles d'attributs de connexion qui capturent les relations types entre le rôle de chaque design pattern Pc= {inherit},

∃**Inherit** {(Observer, TableView)
∧ (Observer, PicView)
∧ (Subject, ConcreteSubject)
∧ (Subject, DataABC)}

- L'ensemble de symboles d'attributs d'actions qui peuvent s'exécuter Pa {invoke, return}.

∃ **Invoke** {(Subject, attach, observer, append)
∧ (Subject, detach, observer, remouve)
∧ (Subject, notify,Mediator, update)
∧ (Mediator, notify, observer, update)}
∧ (TablView, update,Mediator, update)
∧ (PieView, update,Mediator, update)}
∧ (Mediator, update, subject, GetState)

∃ **Return**{(DataABC, getstate, a)
∧(DataABC, getstate, b)
∧ (DataABC, getstate, c)}

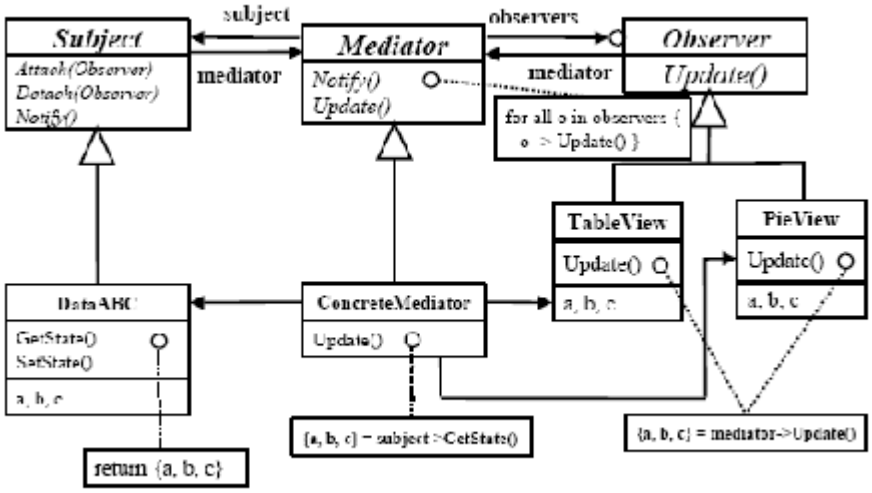


Figure7- Intégration de deux Patterns (Mediator et Observer) [3]

2. Contrats d'interface :

Un service est un comportement défini par contrat, qui peut être réalisé et fourni par tout composant pour être utilisé par tout composant, sur la base unique du contrat».

Pour fournir ses services, un composant repose généralement sur des services qu'il demande à d'autres composants. Le service offert par un composant est défini au travers de plusieurs interfaces. Ces interfaces externes sont associées aux différentes facettes de l'activité du composant.

Par ailleurs, un composant s'appuie sur d'autres composants pour fonctionner. À cette fin, un composant définit des interfaces requises (les interfaces dont il a besoin dans le cadre de son exécution). Ces interfaces doivent être connectées à des interfaces offertes par d'autres composants.

Nous définissons le contrat d'interface (IC) comme suit [Zit08]:

IC = (P, IP, OP, IM, OM, IMI).

- **P** est un ensemble fini de noms de processus.
- **IP** est un ensemble fini de ports d'entrée attaché à un processus.
- **OP** est un ensemble fini de ports de la production attaché à un processus.
- **IM** est un ensemble fini de messages de l'entrée envoyé à un processus.
- **OM** est un ensemble fini de messages de la production envoyé par un processus.
- **IMI** est l'ensemble fini de messages de l'entrée envoyé depuis l'extérieur du design pattern à un processus.

Le contrat d'interface du design pattern observer est :

IC = (P, IP, OP, IM, OM, IMI)

(0) Component-name is Observer **where**: =

(1) ∃ (aConcreteSubject, aConcreteObserver, anotherConcreteObserver), ϵ C

(2) ^ ∃ (inOS, inSO, self, input) ϵ IP

(3) ^ ∃ (outOS, outSO, output), ϵ OP

(4) ^ ∃ (attach, detach, getstate, setstate, update, notify, change) ϵ IM

(5) ^ ∃ (attach, detach, getstate, setstate, update, notify), ϵ OM

(6) ^ ∃ (change) ϵ IMI

Pour permettre la composition, et l'évolution dynamique, nous introduisons d'autres contraintes sur les portes des composants, nous enrichissons la sémantique des interfaces en suivant la notion d'interface fournies/requises. Pour ce faire nous définissons de nouvelles contraintes et des assertions sur les interfaces des composants.

Ces assertions apparaissent sous forme de pre-conditions et post-conditions.

Le contrat d'interface du design pattern sera :

Soit IC1 = (IC, Contraintes) tel que:

IC = (P, IP, OP, IM, OM, IMI) et
p ∈ IP(p) = {i ϵ IP \ gate_Ini = p} ∧
p ∈ OP(p) = {i ϵ OP \ gate_Outi = p} ∧
m ∈ IM(p) = {i ϵ IP, m ϵ IM \ gate_Ini ?m} ∧
m ∈ OM(p) = {i ∈ OP, m ∈ OM \ gate_Outi !m} ∧
all-gateIN= { ∀ IP (p)/ p ϵ Component } ∧
all-gateout= { ∀ OP (p)/ p∈ Component} ∧

Avec comme contraintes sur les portes **<Constraint>:=**

/*Les noms tous différents de ses portes d'entrées gate_In et de sorties gate_list_Out* /
$\forall i, j \in 1, n \rightarrow$ gate_Ini ≠gate_Inj ∧
$\forall i, j \in 1, n \rightarrow$ gate_Outi ≠gate_Outj ∧

/*chaque porte d'entrée d'un composant A lui correspond une sortie d'un autre composant B, qui lui est attaché*/
$\forall j \in 1, n \rightarrow \exists ! \in 1, n/$Agate_Inj ?mi ϵ Bgate_Outi !m i ∧
$\forall j \in 1, n \rightarrow \exists ! \in 1, n/$ Agate_Outj!mi ϵ Bgate_Outi ?m

3. Contrats comportementaux:

Le contrat comportemental décrit l'information dynamique, tel que la collaboration entre les objets qui participent à la composition et la création de nouveaux objets. Le contrat comportemental est modélisé par la collaboration entre objets qui jouent des rôles différents. Le contrat comportemental est essentiel parce que le contrat structurel capture seulement l'information statique, mais les modèles sont aussi caractérisés par les interactions entre les objets et leurs opérations.

Nous avons adopté Basic-LOTOS pour définir un modèle formel de contrats comportementaux car il représente une approche puissante pour modéliser le comportement et la concurrence entre les objets des composants conceptuels [Zit08a].

L'aspect comportemental du contrat du design pattern est exprimé par la spécification **LOTOS**, qui décrit le déroulement des événements observables : {Attach, detach, getstate, setstat, Updat, notify, change} échangés par les éléments du composant: {Concret Observer, Concret Subjet} par le biais d'un ensemble restreint d'opérateurs. Comme il est proposé dans **[Zit08b]**:

La spécification LOTOS du comportement du design pattern « observer » est donnée comme suit.

```
Specification Observer [input, output]: noexit: =
                    /*…. Signature……*/
    Behaviour
        aConcreteSubject [input, output]
                |[input, output]|
        aConcreteObserver [input, output]
                        [ ]
        AnotherConcreteObserver [input, output]
     Where

    Process aConcreteSubject [inCS, outCS]:= noexit
      ?setstate; !notify; !Update; ?getsate;
          AConcreteSubject [inCS, outCS]
    Endprocess

    Process aConcreteObserver [inaCO, outaCO]:= noexit
          I; !setstate; ?update; !getstate
         aConcreteObserver [inaCO, outaCO]
    Endprocess

    Process bConcreteObserver [inbCO, outbCO] := noexit
          I; !setstate; ?update; !getstate
         aConcreteObserver [inbCO, outbCO]
    Endprocess

Endspec
```

Conclusion

L'une des originalités de nos travaux est de ne pas systématiquement composer des patrons pour proposer de nouveaux patrons (design for reuse), mais de spécifier des schémas conceptuels par intégrations d'imitations de patrons (design by reuse). Toute opération d'imitation et/ou d'intégration sur des patrons décrit partiellement ou totalement le schéma conceptuel du système à concevoir.

Notre démarche vise à construire un cadre conceptuel pour l'ingénierie des SIs par composants. Ce cadre, vise la construction de modélisations informationnelles pertinentes et évolutives, et est basé sur les concepts de composant de SI.

Un modèle de SI prend la forme de plusieurs diagrammes représentant la structure du SI (aspects statiques) et son comportement (aspects dynamiques et règles d'intégrité).

Pour la modélisation des aspects statiques du composant conceptuel, nous avons proposé l'utilisation de la logique du premier ordre (FOL). Elle a la particularité de n'autoriser que des liens binaires existentiels et de spécialisation-généralisation entre ses classes. Pour la modélisation des aspects dynamiques, nous avons adopté LOTOS.

Dans l'espace statique d'un modèle de SI, l'utilisation des classes s'avère souvent insuffisante pour prendre en compte tous les concepts du domaine. Le concept de composant est une généralisation du concept de classe. Construit à partir d'un sous-ensemble de classes, un composant décrit le champ informationnel d'une zone de responsabilité du SI. Au concept composant, sont associes les concepts d'objet, attribut et de méthode, respectivement équivalents aux concepts d'objet, d'attribut et de méthode pour une classe.

Dans ce chapitre, nous nous somme focalisé essentiellement sur l'étape de la spécification abstraite basée contrat des composants conceptuels. En ce qui concerne, les étapes de validation, la composition dynamique des composants (qui est définie par le contrat de composition externe), et l'étape d'implémentation, nous allons les étudier dans le chapitre suivant.

CHAPITRE 8

Proposition d'un Langage de Description d'Architectures (ADL-LOTOS)

Introduction

Un aspect important dans la conception de n'importe quel système est son architecture. Une description d'architecture, devrait fournir des spécifications formelles de l'architecture en termes de composants et connecteurs et comment ils se composent.

Ce chapitre définit notre proposition de langage de description d'architecture ADL-LOTOS. Nous définissons notre méta-modèle afin d'obtenir un langage de description aussi concis que possible dont chacun des constituants puisse être aisément traduit dans un langage de programmation Nous illustrerons ces concepts dans une application.

Deux points de vue sont fréquemment employés dans l'architecture de logiciel [IEEE00] : le point de vue structurel et le point de vue comportemental.

Du point de vue structurel, une description d'architecture devrait fournir des spécifications formelles de l'architecture en termes de composants et connecteurs ainsi que leur composition. De plus, dans le cas d'une architecture dynamique, elle doit fournir des spécifications de la façon dont les structures de ces composants et connecteurs peuvent changer.

Le point de vue comportemental peut être indiqué en termes d'un système s'exécutant ou participant dans des actions. Parmi ces actions, des actions pour indiquer des comportements, relations entre composants et connecteurs, et comment ils se comportent et agissent les uns sur les autres.

Un grand défi pour un langage de description d'architecture (ADL) est la capacité de décrire la charge statique mais également les architectures dynamiques de logiciel d'un point de vue structural et comportemental.

En effet, pour décrire des architectures dynamiques, un ADL doit pouvoir décrire le changement des structures et des comportements de composants et connecteurs (suppression, reconfiguration et création…) au cours d'exécution.

Notre choix est porté sur le langage de spécification LOTOS, pour diverses raisons :

Ce langage possède les mécanismes appropriés pour décrire efficacement les styles architecturaux. En ce qui concerne le pouvoir d'expression du langage, LOTOS est un des plus puissants; puisqu'en plus du fait de pouvoir représenter la structure d'une architecture, il est aussi capable d'en décrire le comportement. Ceci est dû au fait que LOTOS est basé sur l'algèbre de processus. LOTOS est qui, en plus d'être efficace, permet la spécification de contraintes comportementales.

En plus il possède un environnement de développement complet. Enfin, un élément non négligeable qui joue en sa faveur, est que le laboratoire LIRE possède de nombreuses personnes ayant elles même travaillées à l'élaboration des outils supportant ce langage. Ceci permet donc d'avoir toutes les informations nécessaires rapidement.

I.1 Description de ADL-LOTOS

Syntaxiquement, l'ADL que nous proposons est décrit comme suit:

<LOTOS-ADL>:= < structural viewpoint, behavioural viewpoint>;
< Structural viewpoint>:= <component, connector, configuration>/
component:= <cp1, cp2, ….., cpn> n ≥ 2 and
connector := <ct_1, ct_2, ….., ct_m> m ≥ 1
With **constraints:**
$\forall cp1, cp2 \in$ component / name.cp1≠name.cp2
$\forall ct1, ct2 \in$ connector / name.ct1≠ name.ct2
configuration: = < /* LOTOS operators construct*/>
<behavioural viewpoint>:= < LOTOS behavior expression >
<end-spec>

Les idées de base de notre formalisme sont les suivantes :

-Les composants et les connecteurs d'une architecture sont modélisés par des processus Lotos et caractérisés par des schémas des variables à instancier;
-La configuration globale est définie par un schéma de communication qui décrit la composition des composants et des connecteurs.

En effet une spécification Lotos décrit le comportement de processus interagissant les uns avec les autres. Un processus peut être paramétré par des types de données, et il peut appeler des fonctions pour manipuler des données. Une expression du comportement décrit la suite des actions observables qui peuvent se produire. Un comportement peut être une instanciation d'un processus, une action de communication entre comportements ou être la composition de plusieurs comportements reliés par des operateurs.

D'un point de vue plus théorique, de nombreux formalismes existent pour décrire les comportements des composants et effectuer des vérifications sur ceux-ci.
Dans ce qui suit, nous voulons réconcilier les mécanismes formels de LOTOS avec les architectures construites à base de composants. Pour aboutir à cela, nous définissons notre méta-modèle de composants représentant la définition des concepts manipulés dans notre ADL (figure1).

Dans notre méta-modèle, un composant fournit des services et peut requérir des services d'autres composants. Les services sont accessibles par des ports uniquement.
Un port est un point de connexion sur un composant définissant deux ensembles d'interfaces: fournies (outgate) et requises (ingate).

Notre méta-modèle distingue deux types de composants : primitif (component) et composite (composite). Les composants primitifs sont décrits par des contrats structuraux, comportementaux et d'interface (voir chapitre précédant)

Les services fournis et requis par un composant primitif sont accessibles grâce à des ports primitifs qui sont les seuls points d'entrée d'un composant primitif. Les composites sont utilisés comme mécanisme pour gérer un groupe de composants comme un tout, en cachant certaines fonctionnalités des composants de ce groupe.

Notre méta-mode n'impose pas de limite dans le nombre de niveaux de composition. Il existe deux façons de définir l'architecture d'une application : utiliser les connecteurs entre les ports de composants ou utiliser un composite pour encapsuler un groupe de composants.

Un connecteur associe le port d'un composant avec un port situé sur un autre composant. Deux ports peuvent être liés ensemble seulement si les interfaces requises par l'un sont fournies par l'autre et vice-versa. Les services fournis et requis par un composant fils d'un composant composite sont accessibles grâce à des ports délégués qui sont aussi les seuls points d'entrée d'un composant composite. Un port délégué d'un composite est connecté à un et un seul port de composant fils.

Avec les définitions d'interface et de service, le composant déclare des éléments structurels sur les services fournis et requis. La spécification de comportement définit l'interaction du composant avec son environnement. Le comportement est décrit par les operateurs de LOTOS.

Dans notre méta-modèle, nous avons aussi représenté l'aspect statique et dynamique du contrat comportemental. L'intérêt de cette séparation est de pouvoir raisonner d'une manière indépendante de n'importe quelle situation.

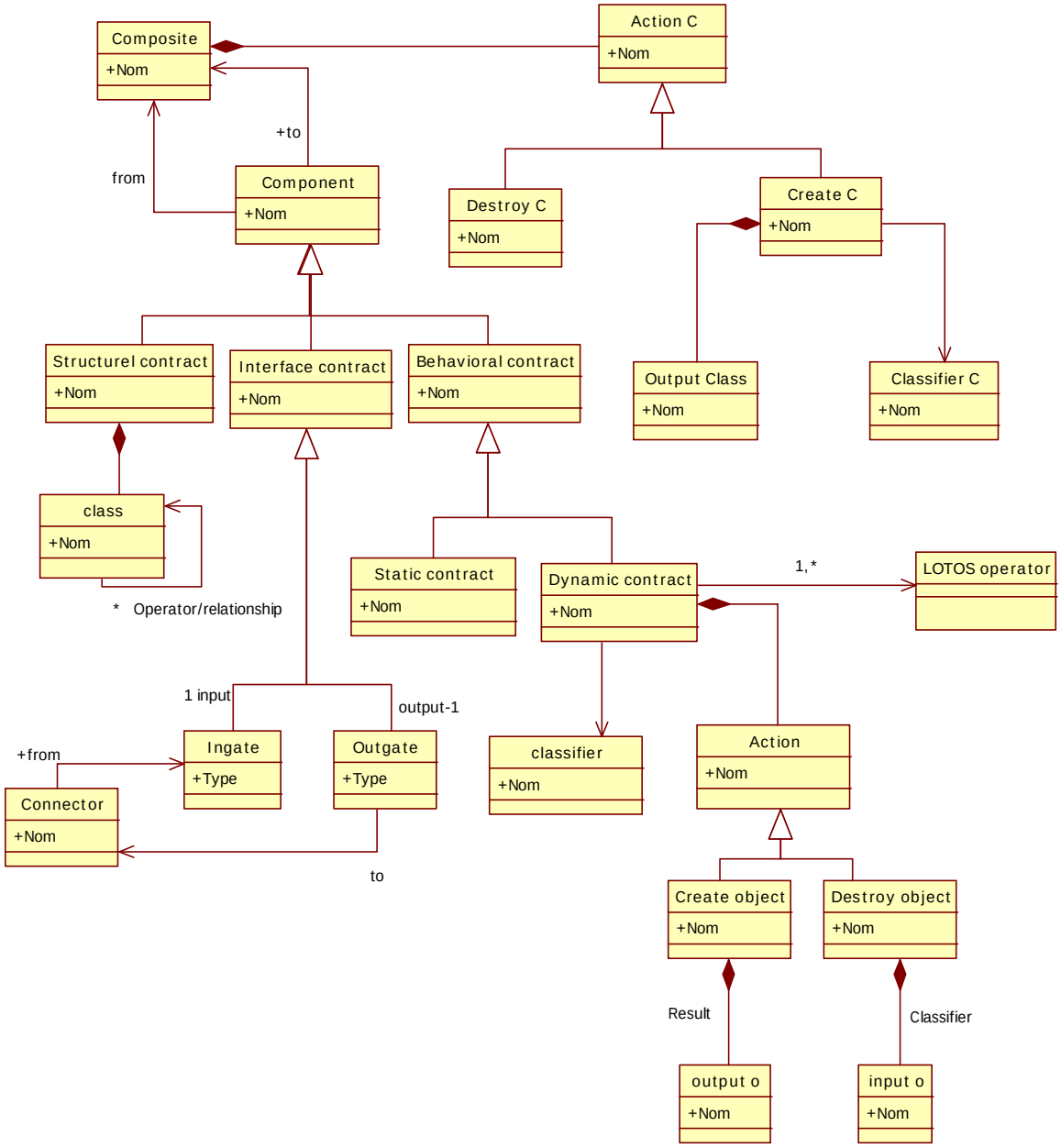


Figure 1- Meta-modèle de LOTOS-ADL

I.2 Etude de cas

Nous allons illustrer notre approche à l'aide du patron de conception Client-Serveur qui est simple, académique et relativement bien connue.

Ce patron est définit comme un style d'architecture, permettant la construction de systèmes distribués. Il apparaît comme un patron générique, définissant deux types de correspondants : les clients, demandeurs de services, et les serveurs, fournisseurs de

services. La relation entre clients et serveurs consiste en la demande d'un service par un client à un serveur.

I.2.1 Spécification

Dans une première définition du patron, aucun détail sur la communication entre les clients et les serveurs n'est considéré. Le modèle Client-Serveur est un style permettant de construire une première version d'un système, dont seul le choix d'architecture est déterminé. De ce fait, il peut être considéré comme la donnée d'un problème consistant à décrire des clients et des serveurs qui communiquent.

La configuration globale d'une architecture, décrit le comportement général d'une spécification, la manière dont interagissent les composants et le connecteur:

Elle spécifie la composition parallèle des processus Client (component client), Server (component server) avec le processus connector (connector) (Figure 2).

```
specification Client-Server [invClt,terClt,invSrv,terSrv] : noexit:=
library RESULT, SERVICES endlib
  behaviour
   Client [invClt, terClt]
       |[invClt, terClt]|
   connector [invClt, terClt, invSrv, terSrv]
      |[invSrv, terSrv]|
  Server [invSrv, terSrv]
 where
   ……
   ……
Endprocess
```

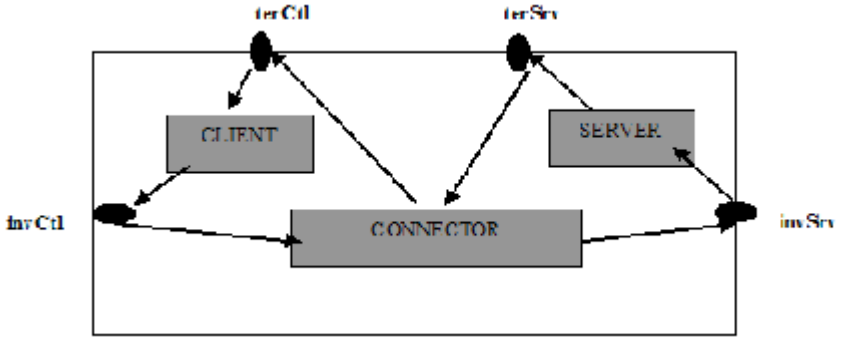


Figure2- Illustration de la spécification LOTOS du Client-Serveur

Le comportement du connecteur est défini par l'ordonnancement des opérations d'invocation au niveau de son interface.

L'interface du connecteur (connector) est constituée de quatre portes: invCtl, terCtl, invSrv, terSrv.

```
process Connector [invClt,terClt,invSrv,terSrv] : noexit: =
  invClt ? s : SERVICE ? op: OPER     /* le client passe la  requête au  connecteur* /
  invSrv ! s ! op;                    /* le  connector passe la  requête au server*/
```

```
   terSrv ! s ? r : RESULT;              /*le serveur passe  la réponse au connecteur*/
   terClt ! s ! r;                       /*le connecteur passe  la réponse  au client*/
        Connector [invClt, terClt, invSrv,terSrv]
 Endproc
```

Un connecteur a deux portes: une d'entrée ou il reçoit des données (sous forme d'événements) et une porte de sortie ou il renvoi, sans aucune transformation et dans le même ordre. Un connecteur se caractérise par ses portes d'entrée et de sortie.

L'architecture du connecteur peut être définie comme étant constitué de plusieurs services (comme un middleware). Afin de spécifié le comportement du connecteur, nous supposons qu'il est constitué de trois services (Figure 3) (service1, service2, service3) et un simple connecteur (CommunicationService).

La spécification LOTOS de cette architecture sera donnée comme suit :

```
/ composition parallèle des  services et  du  processus CommunicationService/
 process Connector_Abstract [invClt,terClt,invSrv,terSrv] : noexit : =
  hide inv, ter in
   ((Service1 [inv, ter ] |||Service2 [inv, ter ] ||| Service3 [inv, ter ])
                          ||
     ServiceOrdering [inv, ter ])
                  |[inv, ter ]|
    CommunicationService  [inv, ter, invClt,terClt,invSrv,terSrv]
 Where
  ……
  ……
 Endproc
```

Suivant les contraintes imposées par ServiceOrdering, la specification LOTOS de ServiceOrdering est:

```
 Process ServiceOrdering [inv,ter] : noexit : =
  inv ! Service1 ? op: OPER
  ter ! Service1 ? r : RESULT
  inv ! Service2 ? op: OPER
  ter ! Service2 ? r : RESULT
  inv ! Service3 ? op: OPER
  ter ! Service3 ? r : RESULT
    ServiceOrdering [invClt, terClt, invSrv,terSrv]
 Endproc
```

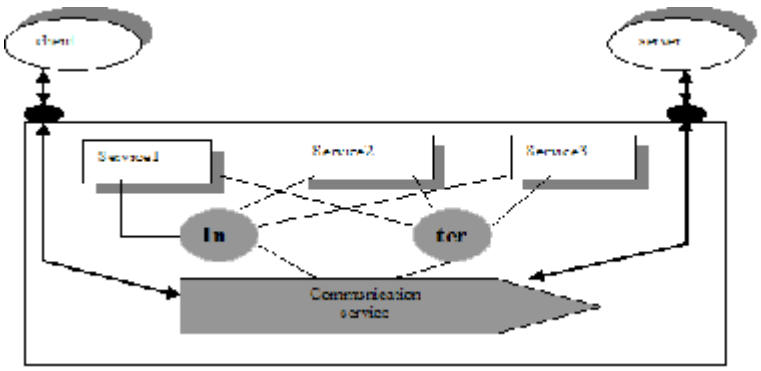


Figure 3 - Illustration de la spécification LOTOS du client-serveur avec trois services

I.2.2 Vérification de la spécification

Pour la vérification on utilise l'environnement de validation FOCOVE (Formal Concurrency Verification Environment) (www.focove.new.fr) (Fig. 7)

L'environnement FOCOVE est une boite à outils pour la vérification formelle de programmes LOTOS. Elle intègre un ensemble cohérent d'outils permettant de traiter des spécifications comportementales et logiques, en utilisant les méthodes basées sur les modèles.

Les composants regroupés dans cette boite à outils peuvent être classes en deux catégories :

-le compilateur : utilisé en amont, son rôle est de traduire le programme à vérifier vers un modèle, appelé ici graphe (ou graphe d'états, ou système de transitions étiquetées, ou automate d'états finis). La boite à outils contient un compilateur, qui traite le contrôle des programmes Lotos.

-le vérificateur : utilisé en aval, il effectue des vérifications sur les graphes engendrés par les compilateurs. La boite à outils comprend un vérificateur de spécifications comportementales, qui compare deux graphes modulo diverses relations, ainsi qu'un vérificateur de spécifications logiques, qui évalue des formules de la logique temporelle arborescente sur un graphe. Ces outils sont capables, lorsque le programme a vérifié est incorrect, de fournir a l'utilisateur des diagnostics en termes de séquences dans le graphe.

-Etape1: Saisie de la spécification LOTOS.

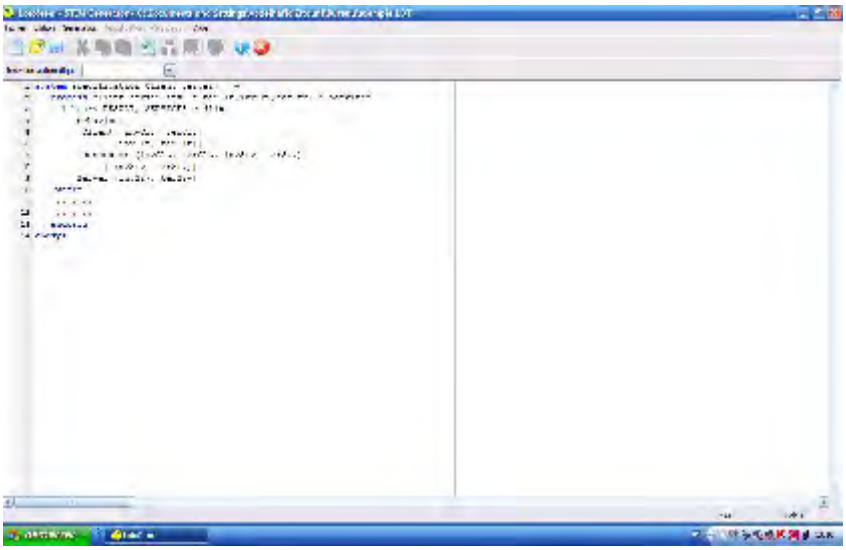


Figure 4- Environnement de FOCOVE

-Etape 2: Compilation de la spécification écrite en basic LOTOS

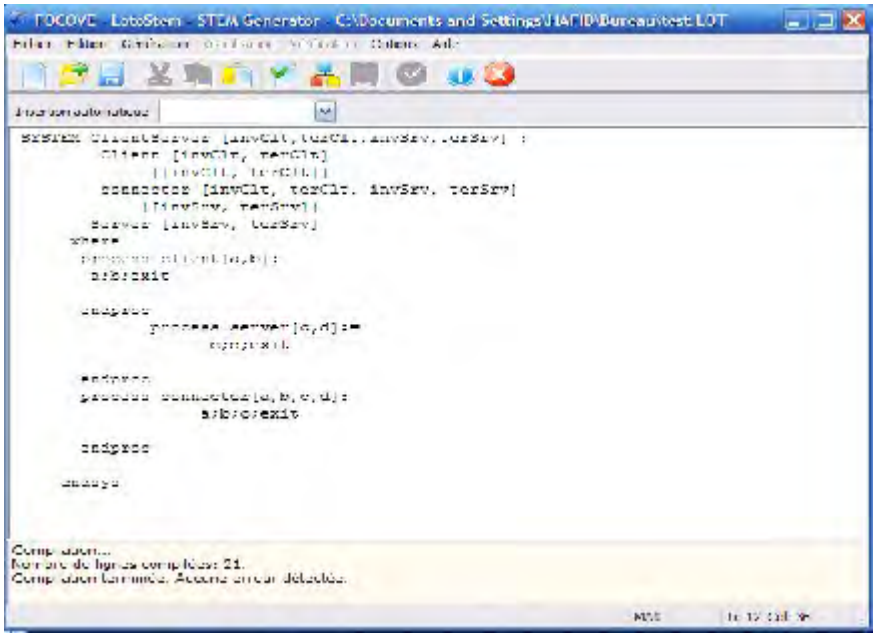


Figure5- Compilation de la spécification

Etape 3:Génération des systèmes de transitions étiquetées (LTS), ou des STEMs

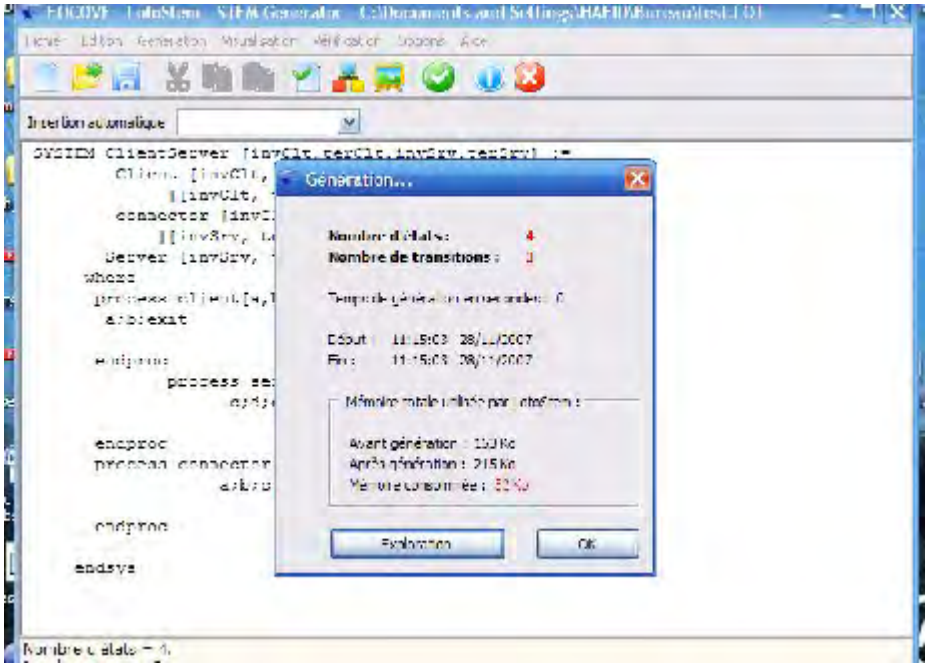


Figure 6- Génération des LTS

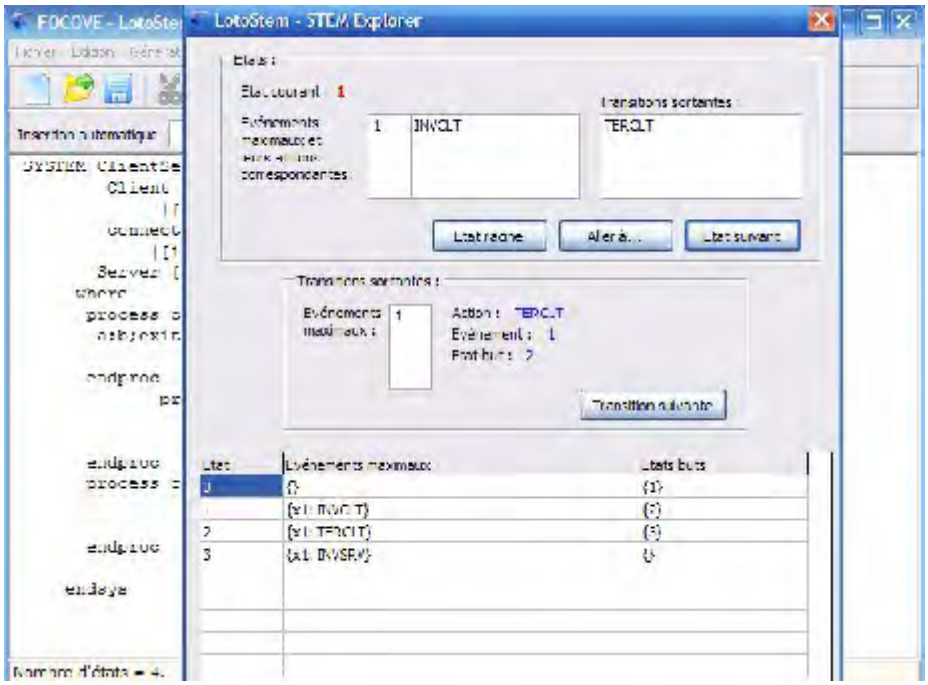


Figure 7- Génération des STEM

Etape 4: Génération graphique des LTS

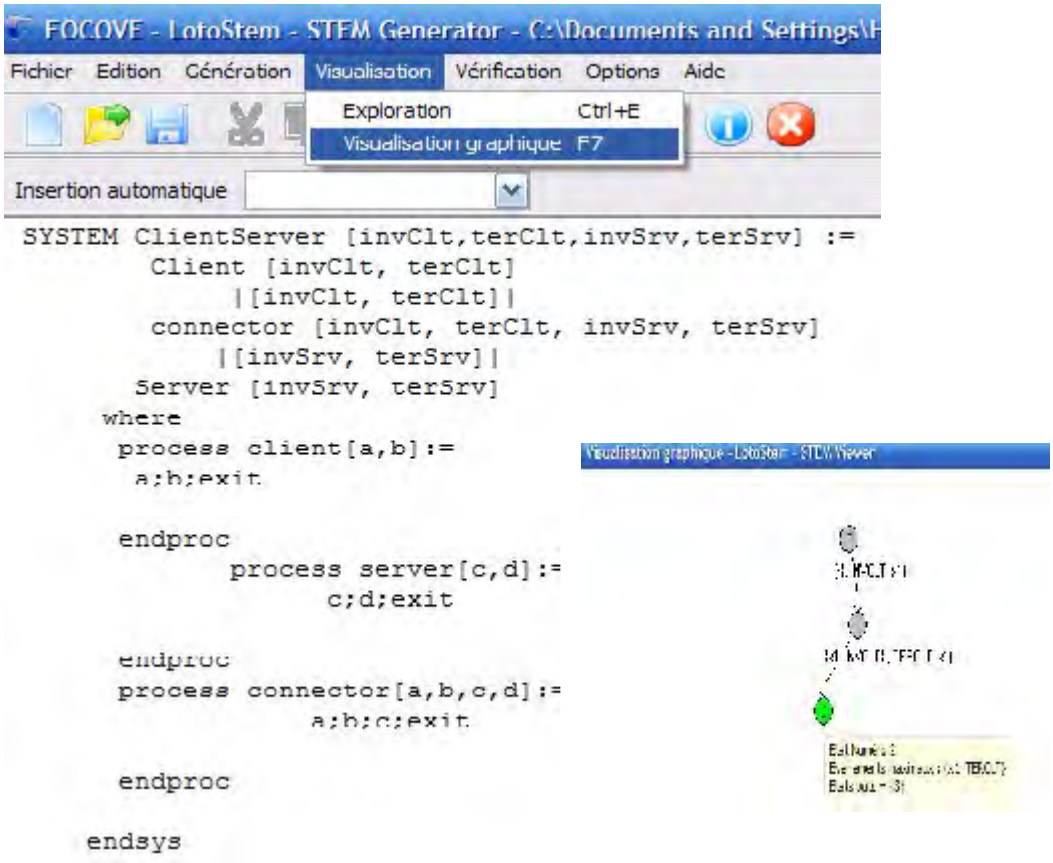


Figure 8- Génération graphique des LTS

Etape5: La vérification des propriétés de sureté et de vivacité écrites en CTL sur les systèmes de transitions qui ont été généré (LTS).

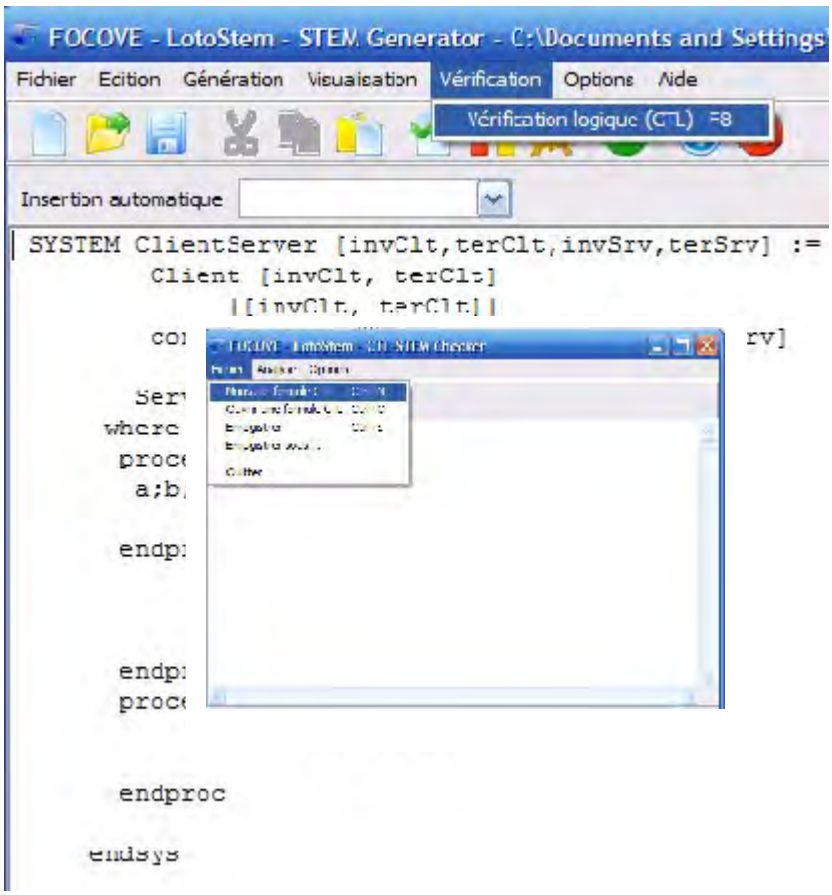


Figure9- Vérification de la spécification avec CTL

Le model-checking [Mer77] est une technique permettant la vérification automatique de propriétés (exprimée le plus souvent par une formule de logique temporelle comme CTL[CE81]) sur des systèmes de transitions [Arn et al.94]. La formule de logique temporelle est alors traduite en un système de transitions qui est « composé » avec le modèle du système. L'exploration exhaustive de l'espace d'états du système de transitions global permet de répondre formellement à la question : le modèle vérifie-t'il la propriété?
L'un des principaux problèmes d'utilisation des techniques de model-checking est lié à la taille de l'espace d'états à explorer.

Conclusion

Dans ce chapitre, nous avons défini un langage de description d'architecture, que nous nommons ADL- LOTOS. Ce dernier est un langage formel permettant la description d'architectures logicielles abstraite.

Nous avons proposé un modèle simple, formel et outillé de description d'architecture logicielle dans lequel on puisse à la fois concevoir simplement les architectures et disposer de conditions d'assemblage riches et flexibles. Dans ce modèle qui est nommé ADL-LOTOS, les éléments architecturaux sont des composants dans lesquels les services sont des entités de première classe. Cette vision nous permet, contrairement à la plupart des autres approches de type CBSE, de nous rapprocher des architectures orientées services, notamment pour la composition. Nous considérons le développement de composants indépendants de toute plateforme d'implantation (*composants abstraits*), qui interagissent via des services. L'assemblage des composants dépend ainsi directement des liaisons entre services.

Enfin pour l'étape d'implémentation, en plus de l'environnement SLD que nous développé, et qui permet à un utilisateur de choisir le pattern désiré. Nous avons proposé des règles de transformation des spécifications LOTOS vers du code JAVA (cette proposition est donné dans l'annexe B1).

Conclusion et Perspectives

Cette thèse à consisté à définir une approche permettant d'utiliser les patrons de conception (composants conceptuels) et les méthodes formelles existants et à les intégrer dans le but *de spécifier, modéliser et développer les systèmes d'informations*.

La complexité croissante des SI et leur incessante évolution rend leur développement plus laborieux, plus coûteux et moins fiable. Dès lors, il émerge des entreprises des besoins très forts en matière de *réutilisation*. En effet, la réutilisation s'inscrit dans l'ensemble des solutions en cours de recherche et de développement pour faire face à ce que l'on appelle aujourd'hui communément la crise du logiciel. Elle se définit comme une nouvelle approche d'ingénierie de systèmes selon laquelle il est possible de construire un système à partir d'éléments existants.

De plus, les spécifications formelles s'imposent progressivement comme une condition essentielle a un bon développement logiciel. Leurs avantages principaux sont la rigueur dans la spécification du problème, la preuve de propriétés et une conception méthodique, automatisable en partie.

La réutilisation de composants pour développer un logiciel a d'abord été employée lors de la phase d'implémentation du logiciel. Elle a également été adaptée à la phase de conception. Ainsi, il est aujourd'hui courant de réutiliser des patrons de Conception exprimés en UML pour concevoir un nouveau système d'information. De nombreux patrons de conception applicables à divers domaines ont été définis. Bien que ces patrons soient connus de la plupart des concepteurs, ils n'ont pas de définition formelle précise.

D'un autre côté, les spécifications formelles sont de plus en plus utilisées dans l'industrie et il devient intéressant de réutiliser en partie ces spécifications dans de nouveaux projets. Réutiliser une spécification formelle signifie d'abord définir une notion de composant de spécification formelle (que nous appelons patron de spécification formelle) et aussi la façon de combiner ces composants lors de la construction d'une nouvelle application.

Le premier objectif était de définir la description de patterns par intégration des approches formelles et semi-formelles.

Nous avons proposé de compléter cette notation graphique par des bases formelles et des outils complémentaires, afin de rendre enfin accessibles aux développeurs de logiciels des techniques formelles permettant d'améliorer la qualité de leurs réalisations. Plutôt que d'attendre que les développeurs viennent aux techniques formelles, il nous a semblé plus réaliste et efficace d'amener doucement les techniques formelles aux développeurs, sans brusquer ces derniers et surtout sans révolutionner leurs habitudes et leurs environnements de développement.

Nous avons décrit une méthode de spécification de patterns, intégrant deux paradigmes, les méthodes UML (semi- formelle) et le langage de spécification formelle Lotos. Pour

cela nous avons donné une interprétation précise sur l'aspect structure et communication entre patterns. La démarche a consisté à enrichir par une spécification formelle en LOTOS les patterns décrits en UML, en vue de leur vérification.

L'ajout de formalisme aux méthodes d'analyse et conception à objets a permis une meilleure structuration et une organisation plus naturelle des spécifications. D'un autre coté, les objets seront plus sûrs et la conception plus rigoureuse.

Nous avons utilisé le langage LOTOS et la logique de description et le principe de contrat pour spécifier formellement la notion de patron de conception, que nous avons nommé spécification abstraite pour composant conceptuel. Cette spécification permet diverses façons de réutiliser les patrons de conception. Réutiliser un patron de spécification consiste soit à l'instancier, soit à le composer avec d'autres patrons de spécification, soit à l'étendre.

Nous avons aussi fourni, un canevas pour la spécification d'architectures logicielles. Une solution aux limitations des langages de description des architectures a été proposée. Celle-ci a consisté de définir un ADL structurés et sur mesure afin de construire des applications à base de composants.

Un Langages de Description d'Architecture (ADL) a été proposé afin de supporter la description des architectures logicielles. Il permet de décrire (à différents degrés de formalité) la structure des systèmes et fournis différents outils permettant leur réalisation, leurs analyses, leurs simulations,.... Il s'agit de langages strictement formels, des propriétés sur la structure et le comportement du système peuvent être prouvées.

Notre langage présente donc la particularité non seulement d'imposer des contraintes sur le comportement du composant, mais aussi d'imposer des contraintes sur l'environnement du composant. Cela a un impact assez fort sur les règles de compatibilité.

La formalisation des architectures promeut la réutilisation (la réutilisation des éléments architecturaux est facilitée) et la compréhension (la description formelle lève toute ambiguïté). Pour aller plus loin et capturer l'expertise de conception pour un domaine particulier, la notion de style architectural est mise en avant.

Pour atteindre cet objectif ambitieux, il faut donner aux représentations des composants conceptuels (Les design patterns) un sens, c'est-à-dire une sémantique.

C'est donc par son biais que les techniques formelles doivent être mises à la disposition des concepteurs. Ces techniques formelles doivent s'intégrer dans l'environnement de développement.

Nous avons considéré dans cette thèse, qu'une bibliothèque de composants est un cas particulier de système d'information. Elle est au centre du processus de réutilisation et doit être capable de gérer toutes les informations qui se rapportent aux composants et aux processus de réutilisation et de capitalisation des connaissances et du savoir-faire de l'équipe.

Nous avons adopté une approche descriptive car nous avons jugé, qu'il est aussi important de gérer une représentation des composants plutôt que les composants eux mêmes. L'approche descriptive garantit à notre bibliothèque de composants la possibilité de s'intégrer dans un environnement de développement déjà existant avec le minimum de modifications dans la façon de travailler de l'équipe de développement.

L'approche proposée est une approche orientée problème et centrée utilisateur (un utilisateur est un concepteur de systèmes d'information utilisant un processus de recherche de composants). Cette approche propose aux concepteurs de formuler des problèmes de conception et de décrire la situation dans laquelle ils souhaitent les résoudre.

Des tentatives de génération du code JAVA à partir des spécifications LOTOS ont été entreprises. Pour cela nous avons proposé des règles de transformation des spécifications LOTOS vers JAVA.

Pour la validation, nous avons utilisé l'environnement de validation FOCOVE. Parmi les perspectives de ce travail, plusieurs pistes peuvent être poursuivies :

Pour générer le code à partir d'une spécification en un langage source, il existe deux méthodes principales. La première est la méthode directe, que nous avons utilisé de ce travail de thèse : il s'agit de décrire le code source directement à partir du modèle. Nous avons remarqué, que cette méthode, n'est pas réutilisable. La seconde méthode, est une passerelle, entre la spécification et l'implémentation, complètement indépendante du modèle traité.

La transformation de modèle utilise le méta-modèle du langage source et celui du langage cible Il s'agit ensuite de définir les règles de transformation du premier vers le second. Cet axe de recherche parait donc bien adapté pour réaliser la seconde méthode de génération de code. Nous proposons, ainsi de s'investir dans un nouveau domaine de recherche : MDA (Model Driven Architecture). C'est un cadre d'architecture qui vise à faciliter le développement de systèmes en donnant une place central au modèle du système

Un autre axe de recherche, et qui est d'actualité, c'est de généraliser l'application de notre contribution aux nouvelles générations des systèmes d'information. Pour cela nous envisagerons, soit d'enrichir et d'étoffer les composants conceptuels, soit de proposer de nouveau composant conceptuel, permettant de solutionner les problèmes posés par cette nouvelle génération des systèmes d'information

Dans une perspective de mobilité des composants, au niveau des composants eux-mêmes la possibilité « d'agentifier » ces composants devraient être étudiés. Il pourrait être intéressant de voir si les composants pourraient être complétés par des « couches » spécifiques aux agents. Il en résulterait un système multi agents où chaque composant serait un agent qui communiquerait avec les autres agents du système ainsi qu'avec l'utilisateur à l'aide de protocoles de communication évolués.

Concernant le langage de description d'architectures LOTOS-ADL, nous pourrions envisager de l'étendre pour spécifier des architectures dont les composants sont mobiles.

Cette thèse n'est pas la définition complète d'un langage de spécification ou d'une méthode de développement. Elle est plutôt une définition précise d'un cadre de travail, dans le domaine vaste du génie logiciel. Pour être pleinement utilisables cette démarche doit être étendue, validé sur des applications industrielles, simplifiés et outillés.

Références Bibliographiques

[Abo et al. 93] G. Abowd, R. Allen, D. Garlan., -Using Style to Give Meaning to Software Architecture-, SIGSOFT'93:Foundation Software Eng., ACM, New York, 1993.
[Abo et al. 95] G. Abowd, R. Allen, D. Garlan, -Formalizing Style to Understand-
[Abr96] Abrial (J.R.). – The B Book - Assigning Programs to Meanings. – Cambridge University Press, 1996. ISBN 0-521-4961-5.
[Ale et al.77] Alexander (C.), Ishikawa (S.), Silverstein (M.), Jacobson (M.), FiskdahlKing (I.) et Angel (S.). -A Pattern Language-, Oxford University Press, 1977. ISBN 0-195-01919-9.
[Alb et al. 02] H.Albin-Amoit,P.Comite,Y-G Guéhémenc,-Un méta-modèle pour coupler application et détection des designs patterns-, LMO2002, Revue L'objet- 8/2002.
[All et al. 97] R. Allen, R. Douence, D. Garlan, -Specifying Dynamism in Software Architectures-, Proceedings of the workshop on foundations of component-based systems, pp. 11-22, Zurich, Switzerland, September 1997.
[All 97] R. Allen, -A formal approach to software architecture-, Ph. D. Thesis, Carnegie Mellon University, CMU Technical Report CMU-CS-97-144, May 1997.
[Arn et al. 94] A. Arnold, D. Bégay, et P. Crubillé. -Construction and analysis of transition systems with MEC-, volume 3 de AMAST Series in Computing. World Scientific, 1994.
[Arn92] André Arnold, -Systèmes de transitions finis et sémantique des processus communiquant-, Collection Etudes et recherches en informatique. Masson, 1992.
[Amb98] Ambler S.W, -An Introduction to Process Pattern-, AmbySof White paper, 1998. http://www.Ambysoft.com.
[AG94] R. Allen et D. Garlan. -Formalizing Architectural Connection-. Proceedings of 16th International Conference Software Engineering, IEEE Computer Soc. Press, Los Alamitos, Californie, pp. 71-80, 1994.
[Ara94] Arango G. -Domain analysis methods in software reusability-, Eds by W. Schäfer, R. Prieto-Diaz & M. Matsumoto, Ellis Horwood, 1994.
[ASL02] Aprille L, Saqui-sannes P, Lohr C. -A new UML profile for reel-time system formal design and validation-, in LNCS 2185, 2001

[Bai87] C. Bailly, J.F. Challine, P.Y. Gloess, H.C. Ferri, B. Marchesin, -Les languages orientés objet: Concepts, Languages et Applications-, Cepadues editions, 1987.
[Bai92] Bailin S. -Domain analysis with Kaptur Courses notes-, CTA Inc. Rockville, MD20852, 1992
[Bar et al. 02] Barbier F., Cauvet C., Oussalah M., Rieu D., Bennasri S.,Souveyet C.,-Composants dans l'Ingénierie des Systèmes d'Information : Concepts clés et techniques de réutilisation-, Assises du GDR I3, Nancy, Décembre 2002.
[Bas et al. 99] L. Bass, P. Clements, R. Kazman, -Software Architecture in Practice-. Addison Wesley, ISBN 0-201-19930-0, 1999.
[Ber92] Berson (A.). -Client Server Architecture-, McGraw Hill, 1992.
[Ber02] L. Berger. -Interactions et modèles de programmation : Support des interactions par les modèles à objets et à composants-. L'Objet, 8(3):9–38, 2002.
[BH94] Jonathan P. Bowen and Michael G. Hinchley. -Seven More Myths of Formal Methods-. In LNCS Springer Verlag, editor, Proceedings FME'94 Symposium, Wolfson Building, Parks Road, Oxford OX1 3QD, 1994.

[Bid89] Michel Bidoit. -Pluss, un langage pour le développement de spécifications algébriques modulaires-, Thèse d'état, Université de Paris Sud, Orsay, May 1989.
[Boch et al. 98] Booch (G.), Rumbaugh (J.) et Jacobson (I.). -The Unified Modeling Language User Guide.– Addison-Wesley, 1998. ISBN 0-201-57168-4.
[Boe82]B.W. Boehm. -Les facteurs du cout logiciel-, Technique et Science Informatique, 1(1):5{24, 1982.
[Bol et al. 00] T. Bolusset, C. Chaudet, F. Oquendo, -Description d'architectures logicielles dynamiques : langage formel et applications industrielles-, Génie Logiciel, Juin 2000.
[Bol98] T.Bolognesi, E.Brinksma. -Introduction to the ISO specification language LOTOS-. In Van EIJK, 1989 pp 23-73.
[Bos96]: Bosch J. -Language Support for Design Patterns-. In : Proceedings TOOLS Europe'96 Paris (F), February 1996.
[Bra88] Gilles Bracon. -La spécification de logiciels dans l'industrie. BIGRE + GLOBULE-, 58:12, 19 January 1988. IRISA-AFCET.
[Bus et al. 96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad et M. Stal, -Pattern Oriented Software Architecture: A System of Patterns -. John Wiley & Sons, 1996.
[BC02] Bieber, G. and Carpenter, J. (2002). -Introduction to Service-Orient Programming-. http://www.openwings.org/.

[Cal90] Jean-Paul Calvez. -Spécification et conception des systèmes : une méthodologie-Masson, May 1990.
[Cam et al. 90] Campbell G., Faulk S., Weiss D. -Introduction to synthesis- Technical report intro-synthesis-90019-N, Software productivity Consortium, June 1990
[Cas et al. 93] Eduardo Casais, Claus Lewerentz, Thomas Lindner, and Weber Franz. -Formal Methods and Object-Orientation-. In Boris Magnusson, Bertrand Meyer, editors, TOOLS EUROPE '93, Versailles, March 1993. International Conference & Exibition, Prentice Hal.l.
[Cas96] Casanave C., -Business Object Architectures and Standards-, Data Access Corporation, Miami USA, 1996.
[CE81] E.M. Clarke et E.A. Emerson. -Synthesis of synchronization skeletons for branching time temporal logic-. In Workshop on Logic of Programs, volume 131 de LNCS, 1981.
[CG88] Christine Choppy and Marie-Claude Gaudel. -Impact des spécifications formelles sur le développement de logiciel-. BIGRE + GLOBULE, 58:3{11, January 1988. IRISA-AFCET.
[Cha02] C. Chaudet, -π-Space : langage et outils pour la description d'architectures évolutives à composants dynamiques. Formalisation d'architectures logicielles et industrielles-. Thèse de Doctorat, université de Savoie, 2002.
[Cho et al. 96]Chou S.C., Chen J.Y., Chung C.G, -A behavior-based classification and retrieval technique for object oriented specification reuse-, Software – Practice and Experience, 26(7):815-832, Juillet 1996.
[Cho87] Christine Choppy. -Formal specifications, prototyping and integration tests-. In Proceedings of the 1st European Software Engineering Conference, pages 185{192,
[Cle et al. 95] P. Clements, L. Bass, R. Kazman et G. Abowd, -Predicting Software Quality by Architecture-Level Evaluation-. Proceedings of the Fifth International Conference on Software Quality, Austin, Texas, octobre 1995.
[CM96] P. Ciancarini, C. Mascolo -Analysing and Refining an Architectural Style-. Proceedings of 10th International Conference of Z Users (ZUM'97), 1996.
[CO01] C. Chaudet, and F. Oquendo, -π-Space: Modeling Evolvable Distributed Software

Architectures-, Proceedings of the 2001 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001), Monte Carlo Resort, Las Vegas, Nevada, USA, 25-28 Juin 2001.
[Cou96] B. Coulange, -Réutilisabilité du logiciel-, MIPS, Masson, 1996.
[CS93] L. Coglianese, R. Szymanski, -DSSA-ADAGE: An Environment for Architecture-based Avionics Development-. Proceedings of AGARD'93, mai 1993.
[Cal91] Calderia G., Basili R.V. -Identifying and qualifying reusable software components-, IEEE computer, February 1991
[Cas92] Castano S., De Antonellis V.-A model for reusable requirements-, Esprit project, report pdm 2-1-3-r1, 1992
[Con01] Conte A., Giraudin J.P., Hassine I., Rieu D.,-Un environnement et un formalisme pour la définition, la gestion et l'application de patrons-, Revue ISI "Ingénierie des Systèmes d'Information", numéro spécial "Formalismes et Modèles pour les Systèmes d'Informations", volume 6, numéro 2, Hermès, 2001.

[Don02] Jing Dong.: -Design component contracts-, Phd thesis. Computer Science department, University of Waterloo, June 2002.
[Don05] Dong J, Paulo S C Alencar, Donald D Cowan.: -Automating the analyse of design component contracts-, In software Practice and Experience, 2005.
[Don05a] Dong, J., Yang,S., Huynh, D:. -Evolving Design Patterns Based on Mode Transformation-, Proceedings of the Ninth IASTED I C S and Applications (SEA), pp 344-350, USA 2005.
[DW99] D'Souza DF., Wills AC., -Objects, Components and Frameworks with UML: The Catalysis Approach-, Addison Wesley, Reading, MA, 1999.

[Ede97] A. H. Eden, J. Gil, A. Yehudai, -Automating the Application of Design Patterns-, Journal of Object Oriented Programming,, May 1997.
[Ede et al.98] A. H. Eden, J. Gil, A. Yehudai,, -LePUS : A Declarative Pattern Specification Language-, Technical report 326/98, 1998.
[EM85] Ehrig,H., Mahr,B., 1985. -Fundamentals of Algebraic specification-, volume1, Springer-verlag, Berlin.

[Fin et al. 98] Finch L., So Much OO, -So Little Reuse-. Dr. Dobb's Journal, mai 1998, http://www.ddj.com/articles/1998/9875/.
[Flo et al.97]: G. Florijn, M. Meijers, P. V Winsen, -Tool Support for Object Oriented Patterns -, ECOOP 97 p472-495 – Springer-Verlag LNCS1241.
[Fre94] Freitag B.,-A Hypertext-Based Tool for Large Scale Software Reuse-, proceedings of the 6th Conference on Advanced Information Systems Engineering CAISE'94, pages. 283-296. Ultrecht, The Netherlands, 6-10 Juin 1994.
[Fuk93] A. Fukanaga, W. Pree, T. Kimura. -Functions as data objects in a data flow based visual language-, ACM Computer Science Conference, Indianapolis, 1993.

[Gam et al. 95] Gamma E., Helm R., Johnson R., Vlissides J.,- Design Patterns- Elements of Reusable Oriented Software-. Addison-Wesley, 1995.
[Gal87] Jean. H. Gallier. -Logic for Computer Science : Fondation of Automatic Theorem Proving-. J. Wiley & Sons, 1987.
[Gar90] H. Garavel, J. Sifakis. -Compilation and Verification of LOTOS Specifications-. Dans [Logrippo 90] 379-394.

[Gar et al. 92] D. Garlan, M. Shaw, C. Okasaki, C. Scott, and R. Swonger. -Experiences with a Course on Architectures for Software Systems-, Proceedings of the 6th SEI Conference on Software Engineering Education, 1992.
[Gar et al. 94] D. Garlan, R. Allen et J. Ockerbloom. -Exploiting Style in Architectural Design Environments-, SIGSOFT'94, ACM Press, New York, décembre 1994.
[Gar et al. 97] D. Garlan, R. Monroe, D. Wile, -ACME: an architecture description interchange language-, In Proceedings of CASCON'97, November 1997.
[Gar et al. 00] Darlan Garlan, Robert T. Monroe, and David Wile. -Acme: Architectural Description of Component-Based Systems-. Foundations of Component-Based Systems, Leavens G. and Sitaraman M., eds. Cambridge University Press, pages 47–67, 2000.
[Gar00] D. Garlan. -Software Architecture: a Road Map-. Proc. of the conference on The future of Software engineering, mai 2000.
[Gau90] Marie-Claude Gaudel. -Algebraic Specifications-, chapter 22, pages {Software Engineers Reference Book. Butterworths, Mc Dermid, J. (Ed.), March 1990.
[Gol84] Goldberg A., -SMALLTALK-80: the interactive programming environment-, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1984.
[Gue94] Nicolas Guel, -Les réseaux algébriques hiérarchiques : un formalisme de spécifications structurées pour le développement de systèmes concurrents-. PhD thesis, Université Paris XI Orsay, September 1994.

[Hay92] Ian Hayes, -Specification Case Studies-. C.A.R Hoare Series. Prentice-Hall International, 2e edition, 1992.
[HC01] Heineman G., Councill W. T, -Component-Based Software Engineering: Putting the Pieces Together-, Addison-Wesley, 2001.
[Hoa85] T. Hoare, -Communicating Sequential Processes-. Prentice Hall International, 1985.
[ISO87] ISO. –LOTOS: A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. Draft International Standard 8807, International Organization for Standardization- Information Processing Systems—Open Systems Interconnection, Genève, juillet 1987.

[JL86] Jean-Pierre Jouannaud and Pierre Lescanne. -La réécriture-. Technique et Science Informatique, 5(6):433{452, 1986.
[Jon93] Cliff B Jones. -VDM, une méthode rigoureuse pour le développement du logiciel-. Masson, Paris, 2e edition, 1993.
[JS90] Cliff B. Jones and Roger C. Shaw. -Case Studies Systematic Software Development-. C.A.R Hoare Series. Prentice-Hall International, 1990.

[Kan et al. 90] Kang K., Cohen S., Hess J., Novak W, -Feature-Oriented Domain Analysis (FODA) Feasability study-, CMU/SEI-90-TR-21, software engineering institute, Carnegie-Mellon university, Pittsburgh, Pennsylvania, 1990.
[Kiz et al. 97] G. Kizales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. -Aspect-oriented programming-. In Springer Verlag, editor, Proceeding of ECOOP'97, number 1241 in LNCS, pages 220–242, Jyvaskyla (Finlande), 1997.
[KK99] M. Klein et R. Kazman, -Attribute-Based Architectural Styles-, Technical Report CMU/SEI-99-TR-022. ESC-TR-99-022, octobre 1999.

[Lau86] Jean.-L. Lauriere. -Résolution de problèmes par l'Homme et la machine-. Eyrolles, 1986.

[LK98]: A. Lauder, S. Kent, -Precise Visual Specification of Design Patterns -, ECOOP'98, Springer-Verlag, 1998.
[Log90] L. Logrippo, T. Melanchuck, R.J. DuWors.-The Algebraic Specification Language LOTOS: An Industrial Experience-. Dans: M. Moriconi (Ed.) Proceedings of the ACM SIGSOFT International Workshop on Formal Methods in Software Development (Napa, CA., 1990), 59-66.
[Luc et al. 95] D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan and W. Mann, -Specification and analysis of system architecture using Rapide-, IEEE Transaction on Software Engineering, vol. 21, n°4, pp. 336-355, April 1995.

[Nom90] S. Nomura, T. Hasegawa, T. Takizuka. A LOTOS Compiler and Process Synchronization Manager. Dans [Logrippo 90], 165-184

[Mag et al. 95] J. Magee, N. Dulay, S. Eisenbach, J. Kramer., -Specifying Distributed Software Architectures-, Proceedings of the Fifth European Engineering Conference (ESEC'95), Barcelona, septembre 1995.
[Med et al. 97] N. Medvidovic. -A Classification and Comparison Framework for Software Architecture Description Languages-. Technical Report UCI-ICS-97-02, Department of Information and Computer Science, University of California, Irvine, février 1997.
[Med et al. 99] N. Medvidovic, D.S. Rosenblum and R.N. Taylor, -A language and environment for architecture-based software development and evolution-, Proceedings of the 21eme International Conference on Software Engineering (ICSE'99), Los Angeles, CA, May 1999.
[ME96]: T.D.Meijler,R. Enjel, -Making Design Patterns Explicit in FACE, a Framwork Adaptive Composition Environnement-, EuroPLop Conference proceeding, 1996
[Med96] N. Medvidovic, -ADLs and dynamic architecture changes-, In A.L. Wolf, ed., Proceedings of the second international software architecture workshop (ISAW-2), pp. 24-27, San Fransisco, CA, October 1996.
[Mei96]: M. Meijers, -Tool Support for Object-Oriented Design Pattern-, Ultrecht University, Ultrecht 1996
[Mer77] S. Merz. -Model checking-. In F. Cassez, C. Jard, B. Rozoy, et M. Ryan, éditeurs, Proceedings of the summer school on MOdelling and VErification of Parallel processes - MOVEP'2k, pages 51–70, 2000.
[Mey97] Meyer B, -Object-Oriented Software Construction-, Prentice-Hall, second edition, 1997.
[Mey89] Meyer B.- Applying design by contract-. IEEE Computer1992; (October):40-51.
[MG96] R. Monroe, D. Garlan. -Style-Based Reuse for Software Architectures-. Proceedings of the 1996 International Conference on Software Reuse, avril 1996.
[MK96] J. Magee, J. Kramer, -Dynamic structure in software architectures-, Proceedings of ACM SIGSOFT'96: Fourth symposium on the foundations of software engineering (FSE4), pp. 3-14, San Francisco, CA, October 1996.
[Mil89] Robin Milner. -Communication and Concurrency-, C.A.R Hoare Series. Prentice-Hall International, 1989.
[MN97] Meijler T., Nierstrasz O.,-Beyond objects: Components-,. Cooperative Information.
[Mon et al. 97] R.T. Monroe, A. Kompanek, R. Melton et D. Garlan. Architectural Styles, Design Patterns, and Objects. Proceedings of the IEEE Transactions on Software Enginering, 1997.
[Mon98] R.T. Monroe, -Capturing Software Architecture Design Expertise With Armani-, Technical Report CMU-CS-98-163, 1998.

[MT00] N. Medvidovic, R. Taylor, -A Framework for Classifying and Comparing Architecture Description Languages-, IEEE Transactions on Software Engineering, 2000.
[Mor95]: R. Moreau, L'approche objets: Concepts et techniques, Masson, 1995.
[MY93] S.Matsuoka and A. Yonezawa. -Analysis of inheritance anomaly in object-oriented concurrent programming language-. In G. Agha, P. Wegner, and A. Yonezawa, editors, Research Directions in Concurrent Object-Oriented Programming, pages 107–150. MIT Press, 1993
[Mad et al. 89] P.W. Madany, R.H. Campbell, V.F. Russo,. -A Class Hierarchy for building Stream-oriented file systems-, In Proc. Of the ECOOP'89 Conference, Nottingham, UK, 1989.

[Par97] Park, P., -Generating samples for component retrieval by execution-, Technical report, University of Windsor, Windsor, Ontario, Canada, 1997.
[Per et al. 00] Pernici B., Mecella M., Batini C., -Conceptual Modeling and Software Components Reuse: Towards the Unification-, In Solvberg A., Brinkkemper S., Lindencrona E. (eds.): Information Systems Engineering: State of the Art and Research Themes. Springer Verlag, 2000.
[Pol05] Damien Pollet,-Une architecture pour les transformations de modèles et la restructuration de modèles uml-, Thèse de Doctorat, université de Rennes1, juin 2005.
[Pou95] J.S. POULIN. -Populating Software Repositories: Incentives and Domain Specific Software-, Journal of Systems Software, 30, pp 187-199, 1995.
[Pre94] Pree, W. -Meta Patterns, A means for capturing the essential of reusable object oriented design-, ECOOP'94, p150 – 162. Springer-Verlag, 1994.
[Pri90] Prieto-Diaz R. -Domain analysis, an introduction-, ACM SIGSOFT, software engineering notes vol.15 – n°2, April 1990.
[PW92] D. Perry, A. Wolf, -Foundations for the Study of Software Architecture-. ACM SIGSOFT, Software Engineering Notes, Vol. 17, no 4, pp. 40-52, octobre 1992.
[PW95] Poulin J., Werkman K.J.,-Modeling structured abstracts and the world wide webfor retrieval of reusable components-. In Proceedings, Symposium on software Reuse, Seattle, Washington, Avril 1995.

[Rev02] J. Revillard, -Approche centrée architecture pour la conception logicielle des instruments intelligents-, Thèse de Doctorat, université de Savoie, 2005.
[Rit92] Ritti M., -Retrieving library identifiers via equational matching of types-, Technical Report 65, Programming Methodology Group, Dept of Computer Science, Chalmers University of Technology and University of Goteborg, Goteborg, Sweden, 1992.
[RC04] N.S. Rosa, Paulo R Cunha, -A software architecture-based approach for formalising middleware behavior-, in ENTCS 2004.

[San90] Donald Sanella. -A survey of formal software development methods-. Technical Report ECS-LFCS-88-56, University of Edinburgh, Laboratory for Foundations of Computer Science, July 1990.
[Sem98] Semmak F. -Réutilisation de composants de domaine dans la conception des systèmes d'information-, Thèse de doctorat de l'Université Paris I – Février 1998.
[SG 96] M. Shaw et D. Garlan. -Software Architecture: Perspectives on an Emerging Discipline-. Prentice-Hall, Englewood Cliffs, N.J., 1996.
[SUN99] G. Sunyé, -Génération de code à l'aide de patrons de conception-, LMO'99 Hermes science.
[SUN99a] G. Sunyé, -Mise en ouvre de patterns de conception: un outil-, thèse de doctorat, université Paris 6, 1999

[Szy98] Szyperski C., -Component Software - Beyond Object-Oriented Programming-, Addison-Wesley, 1998.
[Szy02] C. Szyperski. -Component Software: Beyond Object-Oriented Programming-, Second Edition. Addison-Wesley, 2002.

[Ta04] Taibi T. and Ngo D.C.L (2001). -Modeling of distributed objects computing design patterns combination-. Journal AMCS vol 13 N° 2 pp 239-253, 2004.
[Tea97] Rapide Design Team, -Rapide 1.0. Pattern Language-, Reference Manual, July 1997.
[Tre89] J. Tretmans. HIPPO: -A LOTOS Simulator-. Dans [van Eijk 89] 391-396.
[Vil03] Villalobos J., -Fédération de composants: une architecture logicielle pour la composition par coordination-. Thèse en Informatique à l'Université Joseph Fourier (Grenoble), juillet 2003
[Van89] P.H.J. van Eijk. The Design of a Simulator Tool. Dans [van Eijk 90] 351-390.

[Wil90] D.A. Wilson, -Programming with MacApp-, Reading, Massachusetts, Addison-Wesley.
[Wil01] D. Wile, -Using Dynamic ACME-, In Proceedings of a Working Conference on Complex and Dynamic Systems Architecture, Australie, décembre 2001.
[WK99] Warmer (J.) et Kleppe (A.). – The Object Constraint Language, precise modeling with UML. – Addison-Wesley, Object Technology Series, 1999.
[WP92] Wartik S., Prieto-Diaz R. -Criteria for comparing domain analysis approaches-, International journal of software engineering and knowledge engineering 2(3), p.403-431, 1992.

[Zha et al. 00] Zhang Z., Svensson L., Snis U., Srensen C., Fgerlind H., Lindroth T., Magnusson M., Stlund C., -Enhancing Component Reuse Using Search Techniques-, Proceedings of IRIS 23. Laboratorium for Interaction Technology, University of Trollhttan Uddevalla, 2000.
[Zit04] A. Zitouni, -Un Framework pour l'utilisation des design patterns par intégration du langage de spécification Lotos-, Séminaire national en informatique, SNIB'04, pp 299-307, Biskra, Algérie, Mai 2004,
[Zit05] A. Zitouni -Un Framework pour l'utilisation des design patterns dans le développement des systèmes d'information-, Conférence Internationale d'informatique appliquée CIIA'05, N°ISBN 9947-0-1042-2, pp 39-44, BBA, Algérie, Novembre 2005.
[Zit07] A. Zitouni, -Vers un enrichissement des comportements des design patterns-, revue TSI, volume N°26, N°ISNN-1111-5041, pp 61-69, décembre 2007.
[Zit et al. 07] A. Zitouni, M ; Boufaida, L. Senturier, -Compositional pattern for component-based software architecture -, soumis a ECSA, Barcelone, Espagne 2007
[Zit et al. 08a] A. Zitouni, M ; Boufaida, L. Senturier, -Specifying components, with compositional patterns, LOTOS and design by contract- , soumis à ICSOFT, Portugal 2008
[Zit et al. 08b] Zitouni, Abdelhafid., Seinturier, Lionel., Boufaida., Mahmoud,. -Contract-based approach to analyze software components-, International Conference on Engineering of Complex Computer Systems (ICECCS2008/UML&AADL), IEEE, Belfast April, pp 237, 242.
[Zit et al. 08c] Zitouni, A., Seinturier, L., Boufaida, M., Pautet,L. Perseil ,I, Canal, A, Gérard, S, -UML&AADl grand challenges-, in revue international ACM SIGBED to appear.

[ZW93] Zaremski A. M., Wing J. M., -Signature matching: A key to reuse-. Technical Report CMU-CS-93-151, Carnegie Mellon University, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, Mai 1993.

Liens Internet

-- composant

CETUS http://www.objenv.com/cetus/top_distributed_objects_components.html

Component-Based Software Development / COTS Integration
http://www.sei.cmu.edu/str/descriptions/cbsd.html

Component Software Ressources
http://www.geocities.com/march/components.html

Modèles http://iamwww.unibe.ch/~scg/Research/ComponentModels/

Contrat
http://www.eiffel.com/doc/manuals/technology/contract/page.html

TrustedCmp http://trusted-components.org/

-- langages de description d'architecture

ACME http://www.cs.cmu.edu/~acme/

Aesop http://www.cs.cmu.edu/afs/cs/project/able/www/aesop/

C2 http://www.ics.uci.edu/pub/arch/

Darwin http://www.cs.cmu.edu/~darwin/

MetaH http://www.htc.honeywell.com/projects/dssa/dssa_tools.html

Rapide http://pavg.stanford.edu/rapide/

SADL http://www.sdl.sri.com/programs/dsa/sadl-main.html

UniCon http://www.cs.cmu.edu/afs/cs.cmu.edu/Web/People/UniCon/

Wright http://www.cs.cmu.edu/~able/wright/

Annexe A

Eléments lexicographiques et syntaxiques de LOTOS

- Expressions de comportement :

On appelle opérateurs de comportement les structures de contrôle utilisées dans le langage : composition séquentielle, composition parallèle.
Les expressions de comportement sont dénotées par le non terminal B.

-Expressions de valeur

On appelle expression de valeur (value expression) – ou plus simplement valeur – un terme algébrique construit à partir de variables et d'opérateurs, ces expressions apparaissaient dans les équations algébriques, elles sont aussi utilisées dans les expressions de comportement. Les expressions de valeur dénotées par le non terminal V.

LOTOS est fortement typé : chaque valeur ne peut avoir qu'une seule sorte (domaine de valeur), qu'il est possible de déterminer statiquement.
Les sortes et les opérations qui sont utilisées dans les comportements LOTOS ne doivent pas être formelles ; elles doivent appartenir à des types complètement instanciés.
Si S est une sorte, on note domaine (S) l'ensemble quotient de touts les termes de sorte S par la relation de congruence définie par les équations associées à S. On fait, l'hypothèse que le domaine de chaque sorte n'est pas vide.
Si V_1 et V_2 sont deux valeurs de sorte S on note $V_1 = V_2$ le fait que V_1 et V_2 soient congrus modulo la relation de congruence définie par les équations associées à S.

-Variables

Une variable est un nom donné à une valeur. LOTOS est un langage fonctionnel : chaque variable est initialisée dès sa déclaration et sa valeur ne peut pas être modifiée.

-Portes

En LOTOS, on appelle porte (gate) un canal de communication permettant la synchronisation par rendez-vous et l'échange de valeurs entre plusieurs tâches qui se déroulent en parallèle.

On note T l'ensemble de tous les identificateurs de portes, définis par l'utilisateur, qui figurent dans une spécification LOTOS. Deux portes spéciales sont prédéfinies, qui n'appartiennent pas à T :

1. La porte invisible, notée ''i''. cette porte peut apparaître dans les programmes LOTOS, mais uniquement dans le contexte d'un opérateur '' ;''.

2. La porte de terminaison, notée ‘’δ’’. Cette porte ne peut jamais être employée explicitement dans un programme LOTOS mais elle est utilisée dans la définition sémantique du langage.

-Identificateurs

Chaque classe d’identificateurs est dénotée par un symbole non terminal défini comme suit :

- G : portes
- P : processus
- X : variables
- T : types
- S : sortes
- F : opérations

On emploie en outre les abréviations suivantes :

- $\vec{G}$: liste non vide de portes G_0,… G_n.
- $\vec{X}$: liste non vide de variables X_0,… X_n.

On introduit à présent tous les opérateurs de contrôle du langage LOTOS.

Description des opérateurs

1. Opérateur **« stop »** : Dénote un comportement inactif, qui ne propose aucun rendez-vous avec l’environnement ni aucune transition ‘’i’’ interne.

2. Opérateur « **;** » : permet de spécifier le rendez-vous. Si G est une porte et B_0 un comportement, la construction suivante :

G ; B_0

Dénote le comportement qui propose un rendez-vous sur la porte G et, une fois qu’il a eu lieu, exécute B_0. La notation ‘’ ;’’ a une signification séquentielle : on dit que le comportement B est préfixé par la porte G. les termes événement et interaction seront utilisés comme synonyme de rendez-vous.

Exemple : Le comportement suivant effectue une interaction ARGENT (acquisition de pièces de monnaie) puis une interaction BILLET (distribution d’un billet), après quoi il s’arrête.

```
ARGENT ;
    BILLET_ALGER ;
        Stop
```

En faite cette exemple d’écrit également le comportement d’un utilisateur qui, après avoir payé, reçoit un billet.
Cette forme simple de rendez-vous, qui ne comporte pas d’émission ni de réception de valeurs, ne permet que la synchronisation pure. Il existe une construction plus générale qui prend en compte l’échange de valeurs :

$$G\ O_0, \ldots O_n\ ;\ B_0$$

Ou $O_0,\ldots O_n$ est des offres, définies comme suit :

$$O \equiv ! V$$
$$| ?\ X0,\ldots X_n : S$$

Une offre de la forme ‘’!V’’ correspond à l’émission sur la porte G de la valeur de l’expression V. Une offre de la forme ‘’ ? X0,… X_n : S’’ correspond à la réception sur la porte G de n+1 valeurs v_0, . . . vn de sorte S ; chacune de ces valeurs v_i est ensuite affectée à la variable X_i correspondante.

Le rendez-vous est bloquant aussi bien pour l’émission que la réception : l’exécution d’un comportement qui attend un rendez-vous est suspendue et ne repend qu’après que le rendez-vous a eu lieu. Le rendez-vous LOTOS est absolument symétrique ; aucune distinction n’est faite entre émetteur et récepteur.
Un seul et même rendez-vous peut comporter plusieurs émissions et réceptions qui se déroulent simultanément. De plus, une même porte peut être successivement utilisée dans plusieurs rendez-vous, tantôt avec des émissions, tantôt avec des réceptions.
LOTOS permet de conditionner le rendez-vous par une garde qui est une expression booléenne ‘’ [V_0] ’’, soit une équation simple ‘’ [V_1=V_2] ’’. Le rendez-vous n’a pas lieu si la condition définie par la garde n’est pas satisfaite.

Exemple : Le comportement suivant modélise un distributeur qui effectue successivement trois interactions :
1. Acquisition d’une somme d’argent (interaction ARGENT) en dinar et en centime ; au moyen d’une garde on interdit le rendez vous si cette somme est inférieure au prix attendu
2. Distribution d’un billet (interaction BILLET)
3. Restitution de la monnaie (interaction MONNAIE)

```
ARGENT ? DINAR : NAT ? CENTIME : NAT [TOTAL (DINAR, CENTIME) ge  PRIX] ;
      BILLET ;
            MONNAIE !  MON_DINAR  (DINAR,  CENTIME) !  MON_CENTIME  (DINAR
CENTIME) ;
                        STOP
```

Les opérations comptables sont décrites à d’aide d’un type abstrait :

```
Type MONNAIE is NATURELNUMBER, BOOLEAN
  Opns      PRIX : → NAT
            TOTAL : NAT, NAT → NAT
            MON_DINAR : NAT, NAT → NAT
            MON_CENTIME : NAT, NAT → NAT
  Oqns      forall DINAR, CENTIME : NAT ofsort NAT
            PRIX =10 ; (*prix du billet, exprime en centime*)
            TOTAL (DINAR, CENTIME) = (100*DINAR) +CENTIME ;
            MON_DINAR (DINAR, CENTIME)= (TOTAL (DINAR, CENTIME) -  PRIX) DIV
100 ;(*la monnaie en dinar*)
            MON_CENTIME (DINAR, CENTIME)= (TOTAL (DINAR, CENTIME) -  PRIX) MOD
100 ;(*la monnaie en centime*)
 fin type
```

On peut faire figurer la porte ‘’i’’ à gauche de l’opérateur ‘’ ;’’, mais elle ne doit comporter ni offre ni garde. Le préfixage par la porte ‘’i’’ spécifie une évolution interne qui n’est jamais bloquante.
En résumé la syntaxe générale de l’opération de préfixage est donc :

$$I; B_0$$
$$| G [O_0 \ldots On [[V_0]]]; B_0$$
$$| G [O_0 \ldots On [[V_1=V_2]]]; B_0$$

Les variables éventuellement définies dans les offres O_0,…On n’est visibles que dans V_0, V_1, V_2 et B_0

3. Opérateur ‘’ **[ ]**’’

L’opérateur ‘’ [ ]’’ permet de spécifier le choix non déterministe, si B_1 et B_2 sont deux comportements, la construction suivante : B_1 [ ] B2. Dénote le comportement qui peut exécuter soit B_1 soit B_2.

Example:

```
ARGENT;
        (
          BILLET_ALGER;
                    Stop
          []
          BILLET _ORAN;
                    Stop
            []
BILLET_ANNABA ;
                    Stop
          )
```

4. Opérateur ‘’**choice**’’ sur les portes

Soit B_0 un comportement qui contient des occurrences d’utilisation d’une porte G. Soient G_1 ….. G_n des portes et soient B_1 ….. B_n le comportement définis de la manière suivante : B_i est obtenu à partir de B en remplaçant G par G_i. Pour exprimer le comportement :

$$B_1 [] \ldots B_n$$

LOTOS permet d’utiliser une notation abrégée :

$$\text{Choice G in } [G_1 \ldots G_n] [] B0$$

La porte G sert d’indice à cette itération. Il n’est pas indispensable que les portes G_1, … G_n soient deux à deux distinctes.

Exemple:

```
ARGENT ;                    (
Choice BILLET in [BILLET_ANNABA, BILLET_ALGER, BILLET_ORAN]  [ ]
        PRENDRE ;
                    Stop
                            )
```

L'opérateur ''choice'' possède une forme plus générale permettant d'itérer sur plusieurs portes : Choice $\dot{G}_0$ in $[\dot{G'}_0]$… $\dot{G}_n$ in $[\dot{G}_n]$ [] B_0
Les portes définies dans les listes $\dot{G}_0$, …$\dot{G}_n$ ne sont visibles que dans B_0.

5. Opérateurs ''||'', ''|||'' et ''| […]|''

Les opérateurs qui ont été présentés jusqu'ici sont strictement séquentiels ; LOTOS comprend aussi des opérateurs parallèles. Si B_1 et B_2 sont deux comportements et G_0, …. G_n une liste de porte, la construction suivante :

$$B_1 \mid [G1 \ldots G_n] \mid B_2$$

Dénote le comportement qui exécute B_1 et B_2 en parallèle. La synchronisation et la communication entre opérandes B_1 et B_2 s'effectuent uniquement par rendez-vous sur les portes de l'ensemble $\{G_0 \ldots G_n , \delta\}$.

Lorsqu'un des opérandes veut effectuent une transition étiquetée par une porte G de $\{G_0 \ldots G_n , \delta\}$, il doit attendre que l'autre opérande puisse en faire autant. Lorsque le rendez-vous et possible, les deux opérandes effectuent simultanément une même transition synchrone étiquetée G ; puis ils reprennent chacun leur exécution.
En revanche si l'un des opérandes veut effectuer une transition étiquetée par une porte G qui n'appartient pas à $\{G_0 \ldots G_n, \delta\}$, il le fait indépendamment de l'autre opérande, de manière asynchrone.

Exemple:
Pour composer en parallèle le distributeur a billets (exemple 5) et un demandeur d'un billet pour Alger (exemple 1) il faut les synchroniser sur les quatre interactions ARGENT, BILLET_ALGER, BILLET_ORAN et BILLET_ANNABA. Comme le client n'effectue pas les interactions BILLET_ORAN et BILLET_ANNABA. Le distributeur, qui doit se synchroniser avec lui, ne peut pas non plus.

```
ARGENT ;
    (
    Choice   BILLET in [BILLET_ANNABA, BILLET_ALGER, BILLET_ORAN]  [ ]
                                PRENDRE ;
                    Stop
    )
  |[ARGENT, BILLET_ANNABA , BILLET_ALGER, BILLET_ORAN]|
                    ARGENT ;
                            BILLET_ALGER ;
                  Stop
```

LOTOS possède deux autres opérateurs de composition parallèle :

- Le premier opérateur exprime la synchronisation sur aucune porte. Sauf '' δ'' (interleaving). Sa syntaxe est :

B_1 ||| B_2.

Les deux comportements B_1 et B_2 sont exécutés de manière totalement indépendante (terminaison sur '' δ'' exceptée) : ils ne se synchronisent ni ne communiquent l'un avec l'autre. En revanchent ils sont capables d'interagir avec leur environnement commun : '' B_1 ||| B_2 '' peut participer à un rendez-vous si et seulement si B_1 ou B_2 le peut

- Le second opérateur exprime la synchronisation sur toutes les portes, y compris '' δ'' (full synchronisation). Sa syntaxe est :

B_1 || B_2

Les deux comportements B_1 et B_2 sont exécutés en parallèle de manière entièrement synchrone : ils doivent se synchroniser sur toutes leurs interactions. Ils peuvent interagir avec leur environnement commun : ''B_1 || B_2 '' peut participer à un rendez-vous si et seulement si B_1 et B_2 le peuvent.

Exemple:

Si l'on veut modéliser le comportement simultané de trois utilisateurs du distributeur de billets décrit dans l'exemple 5, il faut employer l'opérateur ''|||''. En effet ces consommateurs (billet_annaba :1, billet_alger :2, billet_oran :3) sont en concurrence pour l'accès à la ressource commune constituée par la machine. En revanche, comme la machine ne peut servir qu'un seul client à la fois, le groupe des trois consommateurs doit être synchronisé avec le distributeur sur toutes les portes (ARGENT, BILLET_ANNABA, BILLET_ALGER et BILLET_ORAN) ; on peut donc employer l'opérateur ''||'' :

```
ARGENT;
    (
   Choice BILLET in [BILLET_ANNABA, BILLET_ALGER, BILLET_ORAN]  [ ]
                PRENDRE;
                    Stop
  )
||
  (
   ARGENT;
                BILLET_ALGER;
                    Stop
|||
                BILLET _ORAN;
                    Stop
|||
                 BILLET_ANNABA;
                    Stop
   )
```

Lorsqu'il y a confrontation entre une émission (offre '' !''), et une réception (offre '' ?''), la valeur émise est affectée à la variable de réception (value passing).

Exemple:

C'est le cas lorsqu'on compose en parallèle un distributeur qui rend la monnaie et un acheteur de billet pour Alger qui fournit 1 Da à la machine et reprend sa monnaie.

```
ARGENT ? DINAR : NAT ? CENTIME : NAT [TOTAL (DINAR, CENTIME) > PRIX] ;
      (
  Choice BILLET in [BILLET_ANNABA, BILLET_ALGER, BILLET_ORAN]   [ ]
                PRENDRE ;
                  MONNAIE ! MON_DINAR (DINAR, CENTIME) ! MON_CENTIME (DINAR, CENTIME) ;
                Stop
)
|[ARGENT, BILLET_ANNABA , BILLET_ALGER, BILLET_ORAN, MONNAIE]|
ARGENT !1 !0 ;
       BILLET_ALGER ;
         MONNAIE ? DINAR : NAT ? CENTIMES : NAT ;
                Stop
```

LOTOS autorise également les confrontations entre deux émissions (‘’ !’’ Et ‘’ !’’) Ou deux réceptions (‘’ ?’’ Et ‘’ ?’’). Dans le premier cas (value matching), le rendez-vous n'a lieu que si les deux valeurs émises sont égales. Dans le second cas (value génération), les deux variables de réception reçoivent une valeur identique, choisie de manière non déterministe.
LOTOS permet le rendez-vous n-aire, c'est-à-dire la synchronisation, sur un même événement, de n comportements concurrents.

Exemple:

On peut imaginer un distributeur qui délivre simultanément deux billets de transport après avoir accepté successivement deux pièces de monnaie. Il est utilisé par deux acheteurs qui se synchronisent sur la porte BILLET_ANNABA (puisqu'ils doivent recevoir simultanément leur Billet) et ne se synchronisent pas sur la porte ARGENT (puisqu'ils paient à tour de rôle). On a ainsi un rendez-vous à trois sur la porte BILLET_ANNABA entre le distributeur et les deux clients :

```
ARGENT ;
        ARGENT ;
                BILLET_ANNABA ;
                        stop
        |[ARGENT , BILLET_ANNABA]|
                        (
ARGENT ;
        BILLET_ANNABA ;
                        stop
  |[BILLET_ANNABA]|
ARGENT;
        BILLET_ANNABA;
                        stop
                        )
```

Au cours d'un rendez-vous n – aire, n offres O_1 … O_n peuvent s'unifier si et seulement si l'intersection des ensembles de valeurs permis par les offres O_1 … O_n est non vide (s'il y a

des gardes, il faut se restreindre aux valeurs pour lesquelles les conditions des gardes sont vérifiées), la valeur échangée est choisie de manière non déterministe dans cette intersections. En particulier ce

Opérateur ‘’hide’’

LOTOS possède un opérateur qui permet de cacher certaines portes d'un comportement. Si $G_0,\ldots G_n$ sont des portes et B_0 un comportement, la construction suivante :

hide $G_0,\ldots G_n$ in B_0

Dénote le comportement B_0 avec lequel elles n'interférent plus. Vu de l'extérieur ces interactions sont invisibles (puisque étiquetées ‘’i") et ont lieu spontanément sans aucune participation de l'environnement de B_0 : le rendez-vous sur une porte cachée n'est jamais bloquant.

Exemple:

Bien souvent un comportement est décrit comme la mise en parallèle de plusieurs sous comportements qui se synchronisent sur un ensemble de portes qu'il convient de dissimuler vis-à-vis de l'environnement. Dans cet esprit, on peut décomposer le distributeur décrit dans précédement en deux sous-systèmes :

- Le premier reçoit une somme d'argent, s'assure que le montant est suant, calcule la monnaie à rendre et envoie une autorisation a l'autre sous-système via une porte GRANT.
- le second, lorsque l'autorisation est accordée, délivre un billet (BILLET_ANNABA, BILLET_ALGER, BILLET_ORAN) et rend la monnaie (la somme qu'il faut restituer lui a été communiquée via la porte GRANT)

On cache la porte GRANT au moyen de l'opérateur ‘’hide" car il s'agit d'un détail d'implémentation qui n'est pas pertinent pour un observateur extérieur.

Le distributeur est compose en parallèle avec un demandeur de billet qui fournit $1.00 pour payer. Noter que l'opérateur ‘’||" impose la synchronisation sur les portes BILLET_ANNABA, BILLET_ALGER, BILLET_ORAN mais pas GRANT, qui est cachée.

```
Hide GRANT in
        (
        MONEY? DOLLARS:NAT ?CENTS:NAT [TOTAL (DOLLARS, CENTS) ge COST] ;
          GRANT !CHG_DOLLARS (DOLLARS, CENTS) !CHG_CENTS (DOLLARS, CENTS) ;
                    stop
        |[GRANT]|
          GRANT ?DOLLARS:NAT ?CENTS:NAT;
          (
            choice BILLET in [BILLET_ANNABA, BILLET_ALGER, BILLET_ORAN] []
                    PRENDRE;
                        CHANGE !DOLLARS !CENTS;
                    stop
          )
        )
    ||
        ARGENT !1 !0;
            PRENDRE;
                GRANT ? DOLLARS:NAT ? CENTS:NAT ;
                  stop
```

Le rendez-vous ‘’unaire" est correct et non bloquant ; par exemple :
Hide G **in** G ; stop est équivalent, vu de l'extérieur, à : **i ; stop**.

Annexe B

B1. Proposition des règles de transformation des spécifications LOTOS vers du code JAVA

Structure de données proposée :

prédicats	***Argument types***	***Proposition***

Classe	Ar, C, Pc	<Modificateurs Classe> classe <Nom Classe>
		[**Hérite** Classe Mère]
		{
		// insérer ici les champs et les méthodes
		}

méthode	Ar, T, M	<modificateurs Méthode> <type retourné> <nom méthode> (arg1, ...) {...}
		// définition des variables locales et du //bloc d'instructions
		}

attributs	Ar, T, A	<modificateurs Attributs> <type> Nom Attribut ;
Héritage	C, Pc	class Fille **Hérite** Classe Mère {...}
Création	Pa	Ma Classe m = **Créer** Ma Classe ();

LOTOS est un langage parallèle, le contrôle des programmes est décrit par des expressions algébrique appelées *comportements*. La synchronisation et la communication s'effectuent exclusivement par rendez-vous via un canal de communication appelé *porte* (gate).

Expression du parallélisme en JAVA.

JAVA dispose d'un moyen puissant pour exprimer le parallélisme appelé multithreading. Deux implémentations sont possibles :

1- Classe Thread : l'héritage à partir de la classe Thread qui implémente l'interface Runnable et qui permet l'utilisation de méthodes de manipulation d'un thread (mise en attente, reprise,...).
2- Interface Runnable : l'implémentation de l'interface Runnable, On effectue ce choix habituellement lorsque l'on veut utiliser une classe ClassB héritant d'une classe ClassA et que cette ClassB fonctionne comme un thread.

Sachant que JAVA n'admet pas l'héritage multiple. L'adoption de la première implémentation est impossible car nous serons obligés à composer des classes utilisant la notion d'héritage.

Expression des portes et de la synchronisation en JAVA.

Les canaux de communication (portes) sont représentés par des objets (un vecteur, une liste, un fichier…). Quant à la synchronisation, JAVA offre le modificateur **synchronized** qui permet de contrôler l'accès aux objets partagés dans un environnement « multithreading ».

La description du comportement et des différentes interactions entre processus est exprimée par des spécifications LOTOS.

Le comportement d'un processus (ou plusieurs) constitue le corps d'une spécification. Un processus est une suite d'action décrivant un comportement.

Implémentation de l'aspect comportemental en JAVA.

L'implémentation est vue en deux niveaux :

1- Au niveau d'une classe : implémenter le comportement de chaque classe.
2- Au niveau des composants: implémenter les interactions entre classes.

Transformation d'une spécification LOTOS vers du code JAVA.

- **Aspect structurel**

o **Règles de transition structurelles :**

Proposition	***EN JAVA***

<Modificateurs Classe> classe <Nom Classe> [**Hérite** Classe Mère]
{
// insérer ici les champs et les méthodes
}
	<Modificateurs Class> class <NomClass> [extends SuperClass] [implemenents Interfaces]
	{
	// insérer ici les champs et les méthodes
	}

<modificateurs Méthode> <type retourné> <nom méthode> (arg1, ...) {...}
// définition des variables locales et du //bloc d'instructions
}
	<modificateurs Methode> <type_retourné> <nom_méthode> (arg1, ...) {...}
	// définition des variables locales et du //bloc d'instructions
	}

<modificateurs Attributs> <type> Nom Attribut ;	<modificateursAttributs> <type> NomAttribut ;
class Fille **Hérite** Classe Mère {...}	class Fille extends Mère {...}
Ma Classe m = **Créer** Ma Classe ();	MaClasse m = new MaClasse ();

- Vu que notre syntaxe est proche de celle de JAVA, Il serait inutile d'imposer à l'utilisateur une syntaxe différente. De se fait, On adoptera comme convention :

- classe = ''class''
- Hérité = ''extends''
- créer = ''new''

- **Aspect comportemental**

- **Règles de transition comportementales :**

Operators	***Description***	***JAVA***
! G	Emission d'un message sur la porte G.	Emission d'un message via l'objet G.
? G	Réception d'un message sur la porte G.	Réception d'un message via l'objet G.
P1[a,b] **[ ]** P2[c,d]	L'exécution de P1 **[**a,b**]** ou P2**[**c,d**]** dépend de l'environnent ainsi que l'état de leurs portes.	Selon un événement extérieur sur l'une des portes a, b, c, d le thread sollicité est lancé.
P1 ‖ P2	exécution parallèle sans synchronisation: P1est indépendant de P2.	P1 et P2 deux threads s'exécutant en parallèle.
P1[a,b] |**[**B**]**| P2[c,d]	Exécution parallèle avec synchronisation sur la porte B.	P1 et P2 deux threads s'exécutant en parallèle, et se synchronisant via la porte B
P1 >> P2	Exécution séquentielle, P2 commence son exécution à la fin de P1.	A la fin de l'exécution du thread P1, le thread P2 est lancé.
P1 **[**> P2	Perturber : P1 peut être interrompu à tout moment avant son arrêt par P2.	P1 et P2 deux threads s'exécutant en parallèle. P2 peut à tout moment interrompre P1.
A ***;*** P	L'exécution d'un processus P préfixée par une action A.	Avant que le thread P soit lancer l'action A est exécuté.
Stop	processus inactif.	Un thread inactif.
Exit	processus qui peut mettre fin à son exécution ou se transformer en un processus **stop**.	Un thread qui peux mettre fin a son exécution, ou passé a l'état inactif.

Implémentation des règles de transformation :

- ✓ au niveau des interactions entre les classes :

On crée une classe principale comportant la méthode *main*, qui implémente le comportement des classes entre elles, elle a la structure suite :

```
Class <Nom du Process principal> {
 /* Déclaration de l'ensemble des portes préfixé
     avec le modificateurs d'Attributs static */
/* Déclaration d'une liste ''EtatThreads'' qui contiendra l'état des threads
     avec le modificateurs d'Attributs static */
/* Déclaration du vecteur ''EtatPortes'' qui contiendra l'état des threads
    avec le modificateurs d'Attributs static */
        Public static void main(String []args){
        /* pour  l'ensemble classes Ci*/
        EtatThread.add(  new Ci()) ;
           }
       Public Object getThread(String t){
      /* retourne le thread 't'*/
       }
      Public Object getPorte (String p){
      /* retourne la porte 'p'*/
       }

}
```

Transformation des structures de contrôles de LOTOS.

- A ; P / A ∈ {ε, PA}, P ∈ {ε, PA, M}

Supposant que l'opération A ; P est défini dans le processus principal.

```
Class <Nom du Process principal> {
 /* attributs */
        Public static void main(String []args){
        /* ensemble d'instruction  implémenter */
          A ;
          getThread(''p'').start () ;
           }
}
```

- P1 >> P2 / P1, P2 ∈ { Pi }

Supposant que l'opération P1 >> P2 est défini dans le processus principal.

```
Class <Nom du Process principal> {
  /* attribute
    Public static void main(String []args){
    /* ensemble d'instruction  implémenter */
       P1.start() ;
      While (getThread(''P1'').proc != null) ;
      P2.start () ;
            }
}
```

- P1 ||| P2 / P1, P2 ∈ { Pi }

Supposant que l'opération P1 ||| P2 est définie dans le processus principal.

```
Class <Nom du Process principal> {
  /* attributs */
        Public static void main(String []args){
        /* ensemble d'instruction  implémenter */
          getThread(''P1'').start() ;
          getThread(''P2'').start () ;
            }
}
```

- P1[a,b] **[ ]** P2[c,d] / P1, P2∈{ Pi }

Supposant que l'opération P1[a,b] **[ ]** P2[c,d] est définie dans le processus principal.

```
Class <Nom du Process principal> {
  /* attributs */
        Public static void main(String []args){
        /* ensemble d'instruction  implémenter */
                {
                   Si (événement reçu sur port a ou b) alors
                        getThread(''P1'').start ();
                   Si (événement reçu sur port c ou d) alors
                        getThread(''P2'').start ();
                }
          }
}
```

- P1[a,b] |**[B]**| P2[c,d] / P1, P2∈{ Pi }

Supposant que l'opération P1[a,b] **[ ]** P2[c,d] est définie dans le processus principal.

```
Class <Nom du Process principal> {
  /* attributs */
          Public static void main(String []args){
          /* ensemble d'instruction  implémenter */
           synchronized  getPorte(''B'') ;
             }
}
```

- P1 [> P2 / P1, P2 ∈ { Pi }

Supposant que l'opération P1 [> P2 est définie dans le processus principal.

```
Class  P1  implements Runnable{
          Thread proc =
               new Thread(this){
                    public void run(){  }
              };
       public P1 () {
                proc.start ();  /* lancer l'exécution du thread  PROC  */
       }
        void interompu(){
        /* récupérer l'instance du thread P2 depuis la classe Main */
         P2.interrupted();
       }
  }
```

✓ Au niveau d'une classe :

Chaque classe prendra la forme suivante.

```
Class <Nom du Process> implements Runnable{
    Boolean fin = false;
    Thread proc =
      new Thread(this){
        public void run(){
   while(!fin){/*  implémenter le comportement de la classe <Nom du Process> */
            }
      }
      };
      public <Nom du Process>  () {
                proc.start ();  /* lancer l'exécution du thread <Nom du Process> */
      }
}
```

Transformation des structures de contrôles de LOTOS.

- A ; P / A ∈{ε, Pa}, P ∈ {ε, Pa, M}

Supposant que l'opération A ; P est défini dans le process G.

```
Class G  implements Runnable{
        Thread proc =
            new Thread(this){
                public void run(){   /* ensemble d'instruction implémenter */
                    A ;
                  /* récupérer l'instance du thread P depuis la classe Main */
                    p.start () ;
                }
            };
      public G () {
              proc.start ();  /* lancer l'exécution du thread  PROC  */
      }
  }
```

- P1 >> P2 / P1, P2 ϵ { Pi }

Supposant que l'opération P1 >> P2 est défini dans le processus G.

```
Class G  implements Runnable{
        Thread proc =
            new Thread(this){
                public void run(){   /* ensemble d'instruction  implémenter */
               /* récupérer l'instance du thread P1et P2 depuis la classe Main */
                    P1.start() ;
                    While (P1.proc != null) ;
                    P2.start () ;
                    }
            };
      public G () {
              proc.start ();  /* lancer l'exécution du thread  PROC  */
      }
  }
```

- P1 ||| P2 / P1, P2 ∈ { Pi }

Supposant que l'opération P1 ||| P2 est définie dans le processus G.

```
Class G  implements Runnable{
        Thread proc =
            new Thread(this){
                public void run(){   /* ensemble d'instruction implémenter */
               /* récupérer l'instance du thread P1et P2 depuis la classe Main */
                    P1.start() ;
                    P2.start () ;
                    }
            };
      public G () {
              proc.start ();  /* lancer l'exécution du thread  PROC  */
      }
  }
```

- P1[a,b] **[ ]** P2[c,d] / P1, P2 ∈ { Pi }

Supposant que l'opération P1[a,b] **[ ]** P2[c,d] est définie dans le processus G.

```
Class G  implements Runnable{
        Thread proc =
            new Thread(this){
                public void run(){   /* ensemble d'instruction implémenter */
                /* récupérer l'instance du thread P1et P2 depuis la classe Main */
                    /*  récupérer l'état des portes depuis la classe Main */
                    {
                        Si (événement reçu sur port a ou b) alors
                            P1.start ();
                        Si (événement reçu sur port c ou d) alors
                            P2.start ();
                    }
                }
            };
      public G () {
              proc.start ();  /* lancer l'exécution du thread  PROC  */
      }
 }
```

- P1[a,b] **[B]** P2[c,d] / P1, P2 ∈ { Pi }

Supposant que l'opération P1[a,b] **[ ]** P2[c,d] est définie dans le processus G.

```
Class G  implements Runnable{
        Thread proc =
            new Thread(this){
                public void run(){   /* ensemble d'instruction implémenter */
                /* récupérer l'instance du thread P1et P2 depuis la classe Main */
                    /* récupérer la porte  B depuis la classe Main */
                synchronized  B ;
                  }
            };
      public G () {
              proc.start ();  /* lancer l'exécution du thread  PROC  */
      }
 }
```

- **Exit**

Supposant que l'opération **Exit** est définie dans le processus G.

```
Class G  implements Runnable{
        Thread proc =
            new Thread(this){
                public void run(){  }
            };
      public G () {
              proc.start ();  /* lancer l'exécution du thread  PROC  */
      }
     void Exit(){
       proc = null;
      }
     void Stop(){
       proc.stop();
      }

  }
```

- P1 [> P2 / P1, P2 ∈ { Pi }

Supposant que l'opération P1 [> P2 est définie dans le processus G.

```
Class  P1  implements Runnable{
        Thread proc =
            new Thread(this){
                public void run(){  }
            };
      public P1 () {
              proc.start ();  /* lancer l'exécution du thread  PROC  */
      }
void interompu(){/* récupérer l'instance du thread P2 depuis la classe Main */
            P2.interrupted();
     }
  }
```

- **Stop**

Supposant que l'opération **Stop** est définie dans le processus G.

```
Class G  implements Runnable{
        Thread proc =
            new Thread(this){
                public void run(){  /* ensemble d'instruction implémenter */
                 proc.stop() ;
                }
            };
      public G () {
      proc.start ();  /* lancer l'exécution du thread  PROC  */
      }
  }
```

Etude de cas : Transformation de la spécification LOTOS du client-serveur Vers JAVA

Process Pattern-Client-Serveur : pour ce processus il va représenter en Java la **classe** « PatternClientServeur » qui contient la méthode **main** du programme.
Son rôle est de déclencher l'exécution des trois processus :

```
Class  PatternClientServeur {
Pubivc static java.lang.Object CCS ;
Pubivc static java.util.List EtatThreads = new java.util.list() ;
Pubivc static java.lang.Object etatCCS= null ;

Public static void main(String []args){
     EtatThread.add(  new  CommunicationClientServeur()) ;
     EtatThread.add(  new  Client ()) ;
    EtatThread.add(  new  Serveur ()) ;

    synchronized  getPorte(''CCS'')
      }
Public Object getThread(String t){
/* retourne le thread 't'*/
  }
Public Object getPorte (String p){
/* retourne la porte 'p'*/
  }
}
```

Process Communication-Client-Serveur: pour ce processus il va représenter une classe en Java qui implémente l'interface **Runnable**.

```
Class CommunicationClientServeur implements Runnable{
/* les attributes de la class CommunicationClientServeur */
  Thread proc =
  new Thread(this){
  public void run(){
  ajouter-service-communication ();
  synchronized  PatternClientServeur.getPorte (''B'') ;
  supprimer-service-communication ();
  }
  };
  public CommunicationClientServeur() {
  proc.start ();
}
  void ajouter-service-communication (){
  /*…..*/
  }
  void supprimer-service-communication (){
  /*…..*/
  }
/* les méthodes de la class CommunicationClientServeur */
}
```

Process Client : pour ce processus il va représenter une classe en Java qui implémente l'interface **Runnable**.

```
Class Client implements Runnable{
      /* les attributes de la class Client*/
    Thread proc =
          new Thread(this){
             public void run(){
                demander-service ();
                appeler-serveur () ;
                While (proc != null) ;
                 getThread("Serveur").start () ;
                 traiter-résultat () ;

              }
         };
    public Client () {
       proc.start ();
    }
    void Exit(){
        proc = null;
      }
     void Stop(){
        proc.stop();
      }
```

Process Serveur: pour ce processus il va représenter une classe en Java qui implémente l'interface **Runnable**.

```
Class Serveur implements Runnable{
      /* les attributs de la class Client*/
    Thread proc =
          new Thread(this){
             public void run(){
                 exécuter-service ();
                 retourner-service ();
                 appeler-client ();
                While (proc != null) ;
                 getThread("Client").start () ;
              }
         };
    public Serveur () {
       proc.start ();
    }
    void Exit(){
        proc = null;
      }
     void Stop(){
        proc.stop();
      }
```

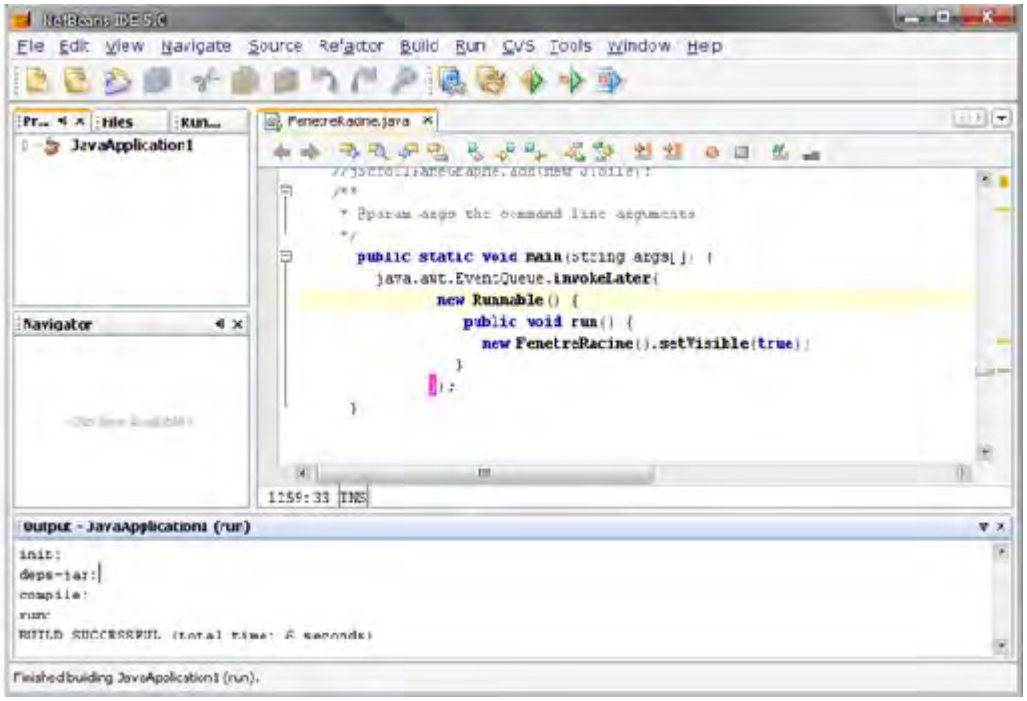
B2. Présentation de l'environnement de développement.

Apres l'étape d'analyse et de conception vient l'étape de développent qui nécessite des outils technologiques adaptés aux fenêtrages.

Comme toutes applications la partie visible est très importante, elle doit être soumise à certaines contraintes d'ergonomie, pour satisfaire ces contraintes on a choisi d'utiliser NetBeans comme Environnement de Développement.

NetBeans est un environnement de développement intégré (IDE) pour Java, placé en open source par Sun en juin 2000 sous licence CDDL (Common Development and Distribution License). En plus de Java, NetBeans permet également de supporter différents autres langages, comme Python, C, C++, XML et HTML. Il comprend toutes les caractéristiques d'un IDE moderne (éditeur en couleur, projets multi-langage, refactoring, éditeur graphique d'interfaces et de pages web).NetBeans est lui-même développé en Java.

Fenêtre de travail :



NetBeans permet aux programmeurs une grande facilité de développement, en prenant en charge des difficultés et des parties secondaires de vos projets. En utilisant NetBeans, vous allez épargner du temps, de l'argent, et libérer votre esprit. Par conséquent, vous pouvez vous concentrer facilement sur votre code.

1. environnement de développement

jGRASP *est un* Environnement de développement qui supporte les langages Java, C, C++ et Ada, et pouvant être configuré pour travailler avec d'autres langages. Cet outil est compatible avec toutes les plates-formes qui supportent une machine virtuelle Java. Simple éditeur à ses débuts, **jGRASP** est aujourd'hui un environnement de développement complet.

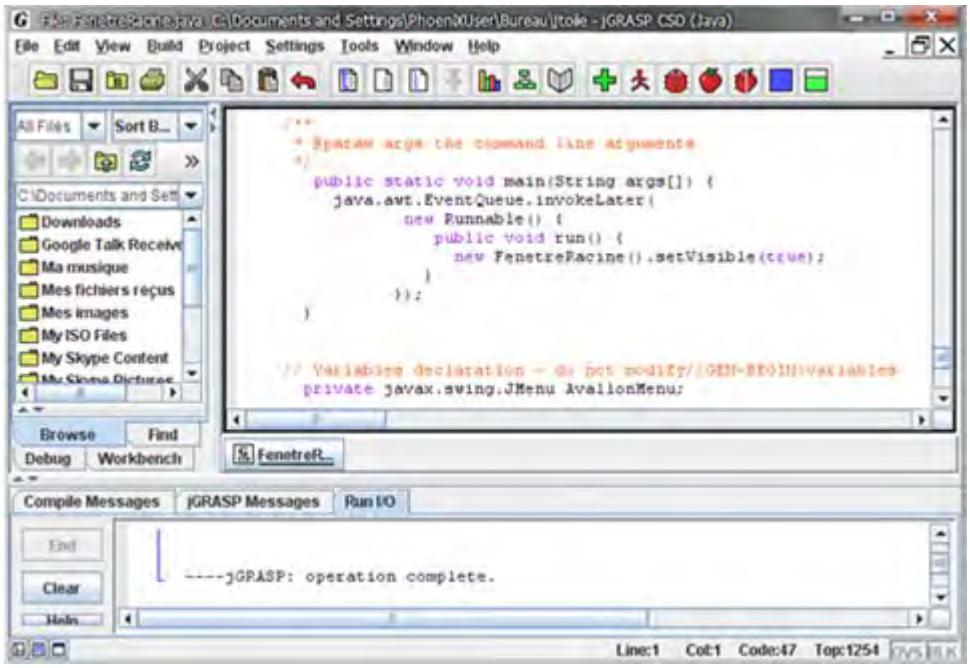


Figure B1- L'environnement de développement

Présentation de l'application

1. Lancement de SLD pattern

Nous présentons dans cette partie une version d'un environnement de développement de systèmes à l'aide de patrons de conception implémentés en JAVA. La particularité de cet environnement est de permettre la saisie de spécifications LOTOS décrivant le comportement des systèmes et permet de générer automatiquement du code JAVA à partie de ces spécifications.

2. Différentes vues de l'application

Dans ce qui suit nous présentons les différents vues (fenêtres) de notre application, que nous nommons «SLD (System Language Description) ».

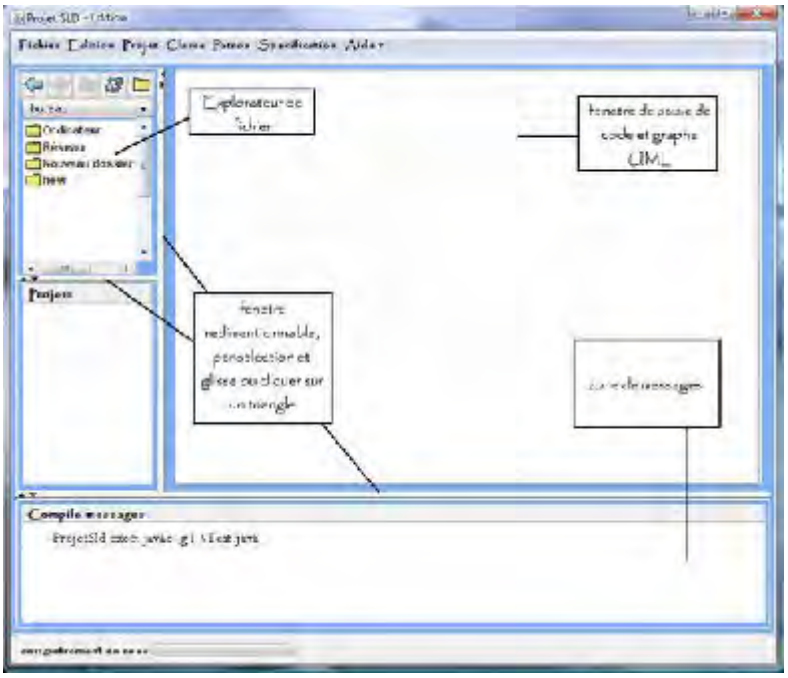


Figure B.2- SLD fenêtre principale.

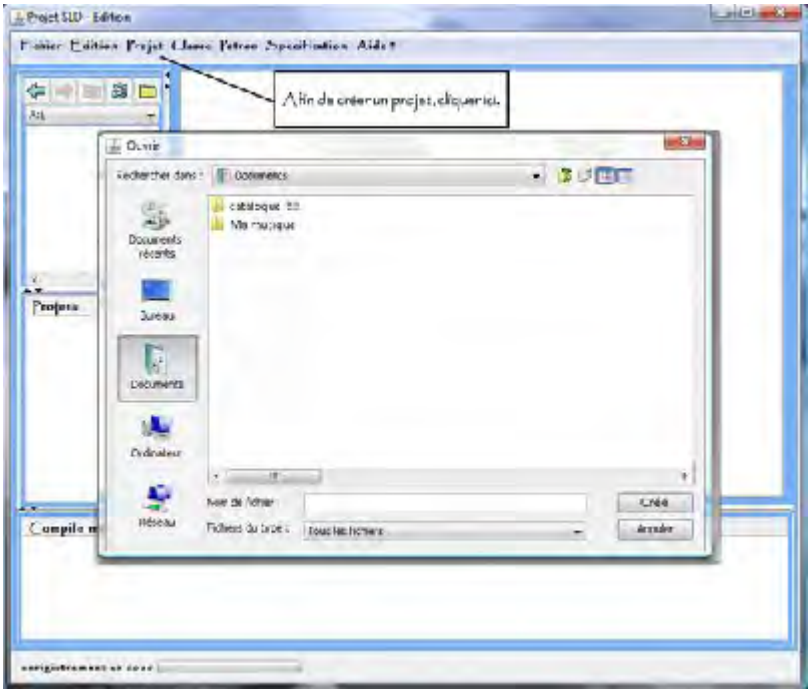


Figure B.2- SLD création de projet.

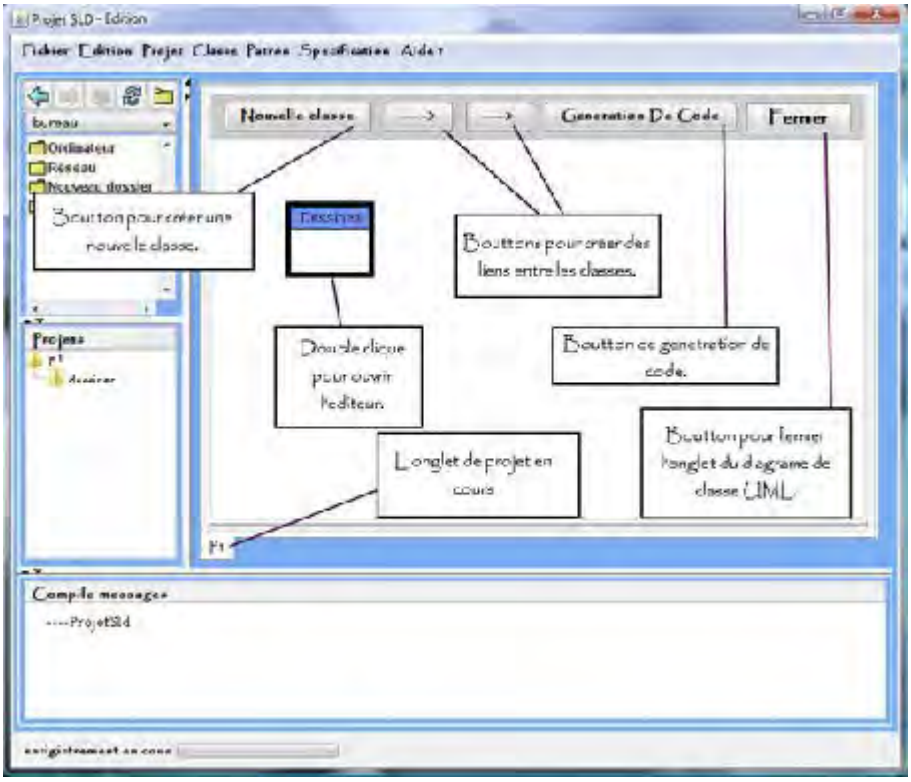


Figure B.3- SLD fenêtre interactions.

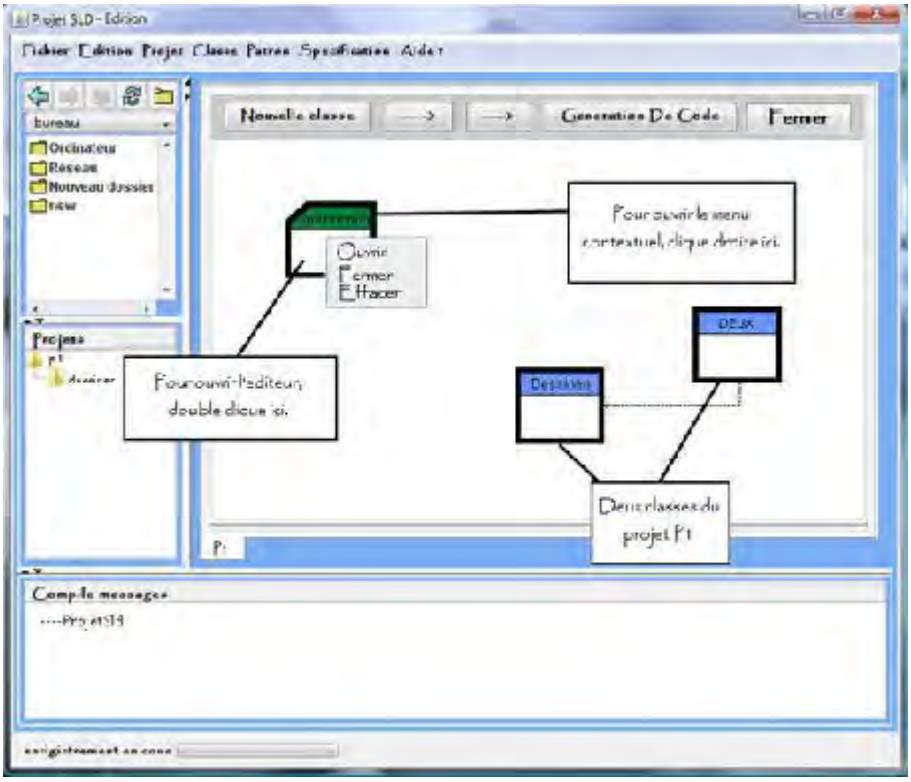


Figure B.4 -SLD fenêtre spécification LOTOS.

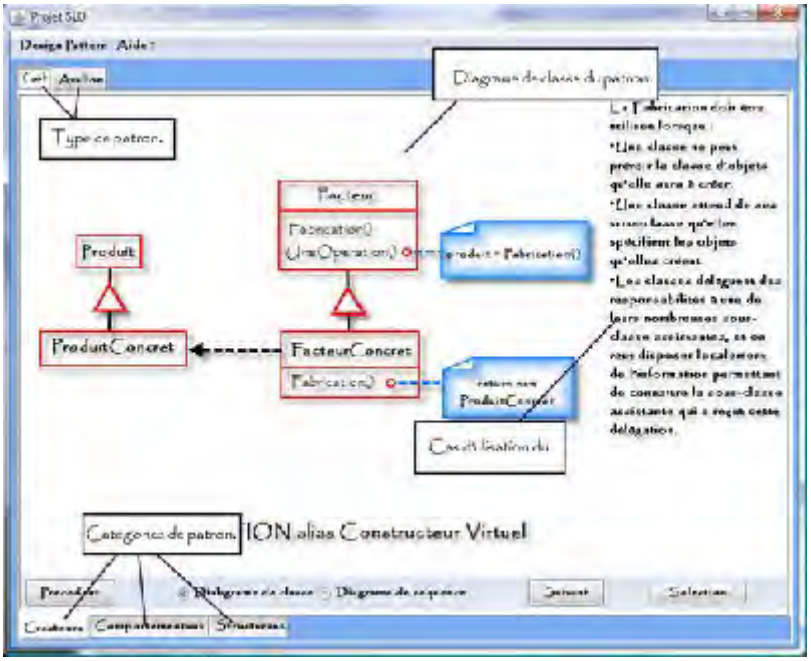


Figure B.5 : SLD fenêtre des patrons.

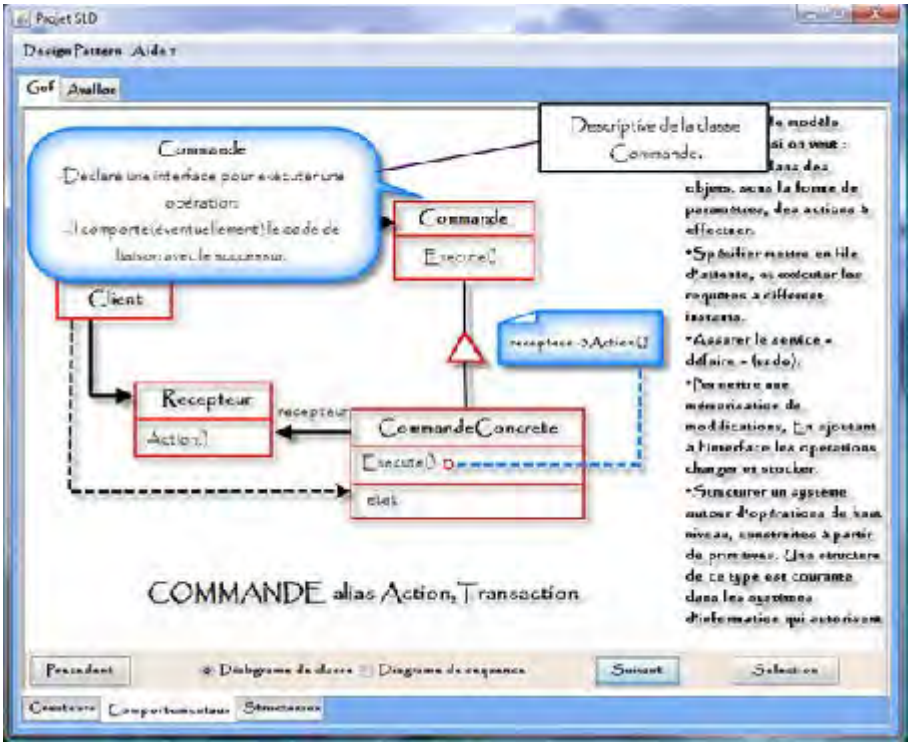


Figure B.6 - SLD fenêtre diagramme de classe.

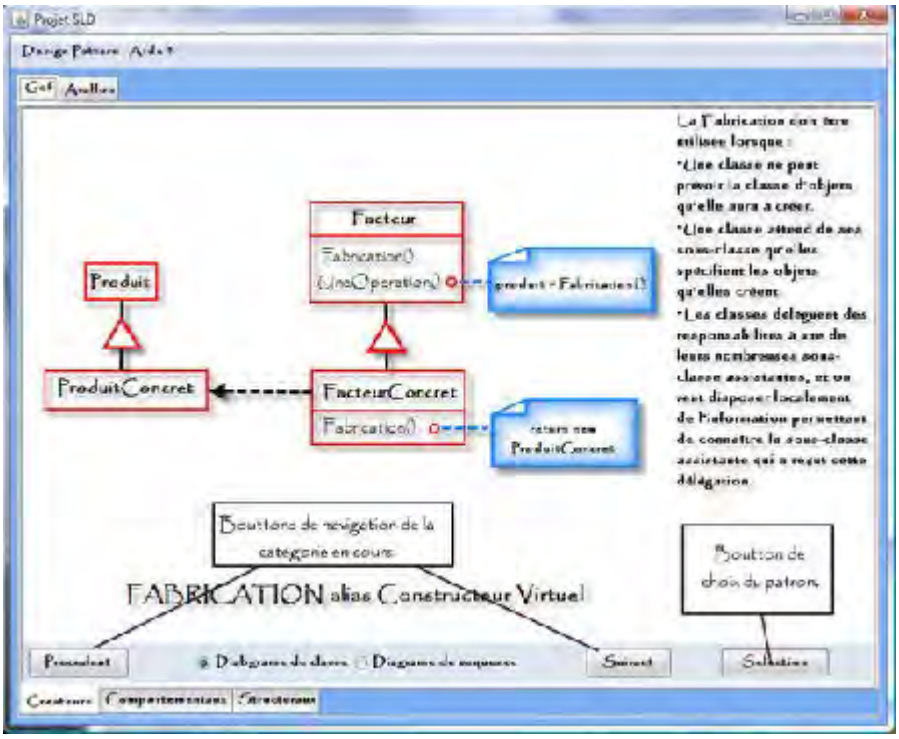


Figure B.7- SLD fenêtre navigation patrons.

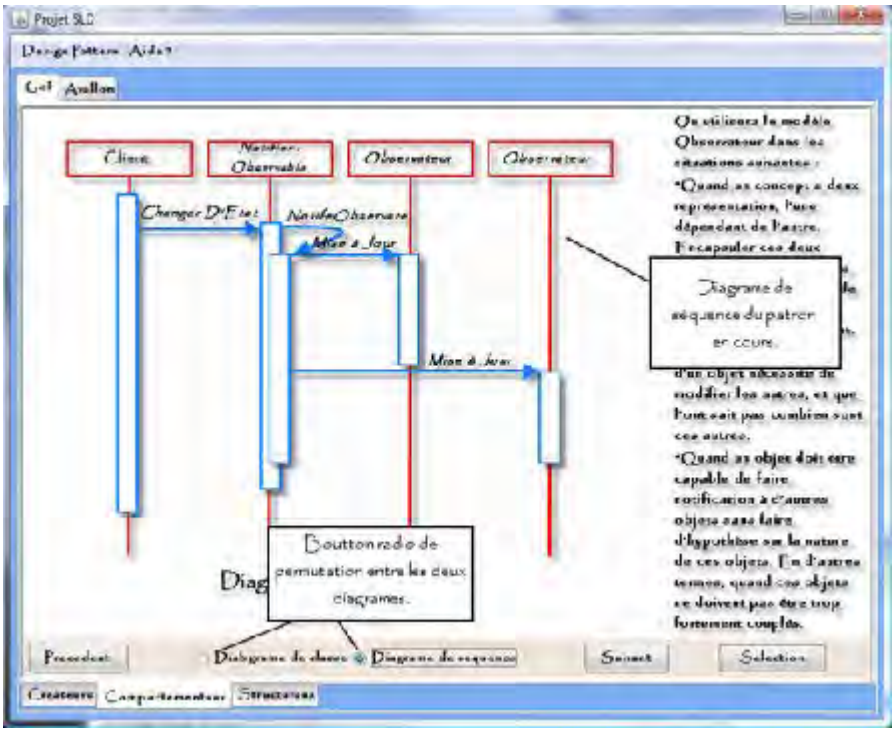


Figure B.8-SLD fenêtre diagramme de séquence.

Abstract

This thesis falls within the global framework of information systems engineering. More precisely, it concerns the use and integration of design patterns coupled with formal methods.
Toward this direction, we provide a contract formal approach, enabling developers to modelize components at the conception stage.
In order to achieve this goal, we describe a method of specification of patterns, integrating the two paradigms, the UML method (semi formal) and the LOTOS language of formal specification.
In addition, we propose a language for architecture description ADL-LOTOS. The ADL-LOTOS language allows one to specify system static and dynamic aspects in a unified way. It is expressive, thanks to its simple syntax. Beside, it promotes the components definition with various way of communication (synchronous and/or asynchronous) at a high level of abstraction. The description of the language is illustrated by some examples and a case study.

Key words: Design Pattern, formal methods, LOTOS, Information Systems, description languages.

ملخص

تندرج هذه الأطروحة ضمن النظام العام لاستعمال الطرق الرياضية لهندسة الأنظمة المعلوماتية. وتهتم هذه الأطروحة بصفة خاصة بإدماج و استعمال موصفات النماذج بواسطة الطرق نصف و كاملة القطعية بواسطة LOTOS و UML.

في هذا الاتجاه وفرنا مقاربة قطعية على أساس العقد كما إقترحنا لغة وصف LOTOS-ADL. و تعد هذه اللغة مميزة بنحو بسيط يحتوي على رسوم بيانية. إضافة إلى ذلك فان هذه اللغة تسمح بوصف مركبات الأنظمة ومختلف الاتصالات فيما بينها على درجة عالية من التجريد.

كلمات مفتاح:

موصفات النماذج، الطرق القطعية، **LOTOS** ، أنظمة المعلوماتية، لغة وصف.