MASTER EN SCIENCES INFORMATIQUES

Proposition d'une méthodologie basée sur UML et adaptation d'un outil CASE

Plichart, Patrick

Award date:
2003

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'Informatique
Année académique 2002-2003

Proposition d'une méthodologie basée sur UML et adaptation d'un outil CASE

Patrick Plichart

Résumé

UML est un langage de modélisation neutre qui se doit d'être spécialisé au contexte et à l'expérience de l'organisation développant du logiciel. La première partie de ce mémoire propose une méthodologie basée sur un sous ensemble d'UML jugé utile par les analystes et développeurs de cette organisation. La définition de *"profil"* introduite par l'omg autorise la spécialisation des éléments d'UML par rapport à un sujet particulier. Certains outils CASE vont plus loin dans cette extension d'UML en définissant la possibilité de développer des outils agissant sur le modèle de l'utilisateur. L'organisation peut alors développer des outils conduisant le processus de développement en aidant les acteurs du projet dans leur modélisation. Ces extensions de l'outil CASE permettent entre autres d'offrir des modèles de description ("templates") reconnus par l'organisation, des transformations (semi-)automatiques de modèles, des vérifications par rapport aux bonnes pratiques de l'organisation, des aides diverses, etc. Ce mémoire reprend dans une seconde partie de telles adaptations, portées à un outil CASE, par rapport à la méthodologie proposée.

Mots clés : Cycle de vie, Processus logiciel, développement logiciel, UML, extension, Profils, outils, méthodologie.

Abstract

UML is a neutral modelling language that needs to be specialized to the context and to the experience of the specific organisation developing software. The first part of this work proposes a methodology based on a subset of UML considered as useful by the analysts or developers of this organisation. The description of *"profile"* introduced by the omg allows the specialization of its components with regard to a particular subject. Some CASE tools go further in this extension of UML, allowing the possibility to develop tools acting on the user's model. Therefore, the organisation can develop tools driving software process helping the actors of the project in the modelization. These extensions of the CASE tools give the opportunity to propose templates of description (used by the organisation) or (semi-)automatic transformations of model, checkings in regard to good practises of the organisation, several helps, etc. The second part of this work presents the adjustments brought to a CASE tool in relation to the methodology proposed.

Keywords : Life cycle, software process, software development, UML, extension, Profiles, tools, methodology.

Je tiens tout particulièrement à remercier mon promoteur, le Professeur Naji Habra
pour son suivi attentif de mon mémoire.
Je remercie également l'ensemble des employés du CRPHT qui m'ont
accueilli au sein de leur entreprise
et apporté leur expérience dans le domaine.
Enfin, je tiens à remercier mes proches pour leur soutien.

Table des matières

Table des figures

Table des Acronymes

CASE	Computer Aided Software Engineering
CITI	Centre d'Innovation par les Technologies de l'Information
OMG	Object Management Group
OPAL	Outillage au Processus d'Acquisition Logiciel
PIM	Platform Independant Model
PSM	Platform Specific Model
PQP	Plan Qualité Projet
RAD	Rapid Application Development
SI	Système d'informations
UP	Unified Process

Glossaire

Analyse	Phase du processus de développement ayant pour résultat une description structurée des exigences fonctionnelles et non fonctionnelles quant au développement mais aussi la description statique et dynamique du domaine concerné par le développement.
Architecture	Produit lié à la phase de conception, il a pour but de décrire l'organisation des composants du système global, les relations entre ceux-ci en vue d'accomplir les exigences quant au logiciel. Plus précisément, l'architecture logique représentera l'organisation des composants du système, des relations entre eux en restant indépendant par rapport à la plate-forme, l'architecture physique, décrira cette organisation mais du point de vue des composants physiques mis en oeuvre pour répondre aux exigences.
Article de configuration	"Ensemble de matériels, de services ou sous-ensemble défini de ceux-ci, retenu pour la gestion de configuration et traité comme une seule entité dans le processus de gestion de configuration" [Jac96].
Cahier des recettes	Ensemble des recettes, liste de tests, de façon à permettre au client de vérifier un maximum (jamais exhaustif) la conformité du produit livré avec les attentes du client.
Conception	Phase du processus de développement ayant pour résultat la production de l'architecture logique et physique.
Contrainte (UML)	Relation entre des éléments de modèles spécifiant des conditions ou des propositions devant être maintenues vraies. Elles peuvent être exprimées en langage naturel ou en Object Constraint Language.
Cycle de vie logiciel	Très proche de la notion de processus logiciel, le cycle de vie logiciel déroule le processus logiciel dans le temps depuis la décision de développer jusqu'à l'exploitation finale du logiciel.
Développement logiciel	Ensemble de pratiques destinées à produire le logiciel.
Exigence	"Condition ou capacité requise par un utilisateur en vue de résoudre un problème ou d'accomplir un objectif"[610].
Exigence fonctionnelle	Exigence quant au logiciel à développer portant sur une fonctionnalité attendue, un objectif que doit accomplir le logiciel.

Exigence non fonctionnelle	Exigence quant au logiciel à développer n'ayant pas trait à des fonctionnalités du logiciel mais à des contraintes de conception ou de réalisation ou encore à des critères de mise en service du logiciel, etc.
Extension d'UML	Stéréotype ou Tagged Definition ou Contrainte.
Gestion de configuration	"Discipline de management de projet qui permet de définir, d'identifier, de gérer et de contrôler les articles de configuration tout au long du cycle de développement d'un logiciel"[122].
Livrable	Produit attendu d'une activité ou d'une phase lors du développement.
Maquette	Prototype développé ayant pour objectif d'illustrer les interfaces (fenêtres, pages web, etc.) du logiciel et la navigation entre celles-ci ainsi que leur comportement en réaction aux actions de l'utilisateur. Ce prototype n'implémente en général que les parties visibles de l'application, la logique sous-jacente aux diverses fonctionnalités étant laissée de côté.
Méta-modèle	Modèle destiné à décrire un langage de modélisation. Ce modèle est lui-même réalisé dans le langage décrit.
Méthodologie	Ensemble de méthodes et de techniques d'un domaine particulier, en l'occurence, le développement logiciel. Une méthode étant un ensemble de principes, de règles et/ou d'étapes permettant de parvenir à un résultat.
Modèle	Abstraction de la réalité, le modèle est une vue subjective définissant les caractéristiques essentielles jugées pertinentes de cette réalité.
Outil CASE	Outil logiciel destiné à assister le développement logiciel.
Platform Independent Model	Modèle représentant des comportements, fonctionnalités métiers sans tenir compte des détails issus de choix technologiques.
Platform Specific Model	Modèle représentant des comportements, fonctionnalités métiers tenant compte de choix technologiques.
Processus	Description générique, décomposant sur les plans logique ou temporel l'ensemble des tâches à accomplir. Des processus complémentaires ou spécialisés peuvent servir une méthode en fonction du contexte de sa mise en œuvre.
Processus logiciel	Processus associé au développement logiciel
Profil	Ensemble d'extensions d'UML destinées à personnaliser UML à un domaine particulier

Recette	Opération par laquelle le client reconnaît que le logiciel est conforme au cahier des charges, qu'il est exploitable, et enfin qu'il est opportun de le mettre à la disposition des utilisateurs.
Spécifications	"Document spécifiant, d'une manière complète, précise et vérifiable, les exigences, la conception, le comportement, ou d'autres caractéristiques d'un système ou d'un composant, et souvent , les procédures pour déterminer si ces éléments ont été satisfaits"[610].
Stéréotype	Méta-élément dont la structure coïncide avec un méta-élément UML (constituant une généralisation de ce stéréotype). Par exemple, le méta-élément "Classe" peut posséder des stéréotypes "Objet métier", "Objet persistant" (ceux-ci restent des classes). Le stéréotype se distingue de ce méta-élément par l'usage qui en est fait. A un stéréotype peut-être associé des contraintes ou des "Tagged Definition".
Système	Ensemble de composants organisé pour accomplir une fonction spécifique ou un ensemble de fonctions, dans le présent mémoire, par *"système global"* (dans le contexte du développement logiciel) j'entends système intégrant la notion de système d'information, d'acteurs et de composants collaborant dans l'accomplissement des exigences de l'utilisateur.
Tagged Definition	Les *tag définitions* servent à spécifier de nouvelles propriétés attachées aux éléments stéréotypés des modèles. Si on définit par exemple le stéréotype "manager" sur le méta-élément *classe*, on peut définir une *Tagged definition* sur ce stéréotype : isCompetent de type booléen. Une classe stéréotypée manager pourra alors avoir une *TaggedValue* : isCompetent=True. Ces tag définitions sont très proches de la notion de méta attribut et les tagged values proches de la notion de valeur de cet attribut.
Traçabilité	Aptitude à retrouver l'historique, l'utilisation ou la localisation d'une entité au moyen d'identifications enregistrées. Lindvall et Snadhall [LS96] distinguent la traçabilité verticale offrant la possibilité de retracer les éléments correspondants entre les modèles des différentes étapes du cycle de développement, de la traçabilité verticale offrant la possibilité de retracer les éléments dépendants à l'intérieur d'un modèle.
Workproduct	Un workproduct désigne une étape dans le processus de développement, il est caractérisé par la production d'un livrable particulier à son terme (CITI).

Introduction

Les méthodologies de développement sont l'objet d'une remise en cause permanente, celles-ci évoluent sans cesse en fonction des réels besoins des analystes, développeurs ou chefs de projet. Les méthodologies se doivent de s'adapter également aux nouvelles technologies (comme par exemple les langages orientés objet). On citera par exemple l'apparition des méthodologies Agiles venues bouleverser le domaine du processus logiciel en dénonçant l'excès de méthode, s'avérant souvent inapproprié dans le contexte de petits projets. Ces méthodologies Agile dénoncent encore l'inadéquation entre d'une part la production de documents intermédiaires, utilisée comme témoin d'avancement d'un projet par les développeurs et d'autre part les attentes du client qui, lui, évaluera le projet d'après l'utilité des fonctionnalités développées. Cette remise en cause des méthodologies se reflète dans le choix d'une méthodologie par une organisation. Quels sont les apports et les limites des méthodologies ? Quels sont les réels besoins en terme de méthodologie pour une organisation désireuse de développer ?

Le choix d'une méthodologie de développement pour une entreprise, n'est pas une chose aisée, la diversité des types de développements pour une organisation, l'hétérogénéité dans les aptitudes des développeurs, etc. compliquent ce choix. Par exemple, une méthodologie telle Unified Process (UP) basée sur une modélisation UML exige certaines aptitudes de la part des développeurs mais aussi de la part du client. Car UML, bien que semi-formel, n'est pas toujours jugé aussi intuitif qu'il n'y paraît.

Ce choix est d'autant plus complexe que les méthodologies de développement se complètent souvent. Tandis que l'une effectuera des recommandations plus précises sur la gestion du risque, la constitution des contrats, d'autres décriront de manière plus exhaustive l'élicitation des exigences, confieront des modèles de description, etc.

Les organisations qui développent du logiciel se doivent également d'adapter ces méthodologies, à leur contexte précis, au niveau de maturité de l'organisation, à l'expérience interne de l'entreprise ainsi que celle des développeurs ou encore à la nature du projet. Il reste donc encore à voir comment cette méthodologie choisie s'articulera dans le contexte précis de l'entreprise.

Le Centre d'Innovation par les Technologies de l'Information (CITI) n'a pas encore fait le choix d'une méthodologie ou l'autre pour ses développements au même titre que le choix d'un langage de modélisation. Le CITI est désireux de voir comment s'articulerait une méthodologie basée sur UML au contexte de son entreprise et surtout de voir les possibilités quant aux profils UML par rapport à cette méthodologie.

L'objectif de ce mémoire est double. D'une part, l'idée est de développer une méthodologie basée sur UML dans une version minimaliste, plus simple que, par exemple, Unified Process, en vue d'illustrer à quoi ressemblerait la modélisation UML d'un petit projet de développement interne du CITI (Quels éléments UML utiliser ? à quel moment ?). Il sera ainsi

spécialiser UML à l'expérience, aux bonnes pratiques et aux projets propres du CITI, étant donné le caractère neutre d'UML dans sa spécification de base. Cette méthodologie servirait de support, de cadre à l'adaptation d'un outil CASE.

D'autre part, le second objectif est lié aux possibilités offertes par UML. Il s'agit de développer des outils concrets, un "toolbox", lié à cette méthodologie et aidant les développeurs dans leur modélisation ou proposant de nouveaux types d'outils. Ce "toolbox" servirait à illustrer le potentiel des profils UML, sorte de spécialisation, d'extension d'UML, supportés par certains outils CASE. Des prototypes d'outils ont donc été implémentés en vue d'être présentés à la communauté des développeurs.

Dans le cadre de ces objectifs, il était donc requis une certaine démarche quant aux travaux. L'élicitation des besoins en terme de modélisation pour un projet particulier ; l'identification d'outils pertinents pour le "toolbox" nécessitaient une concertation avec les développeurs du CITI. De plus, ces réunions coïncident avec l'ambition d'illustrer un développement basé sur UML et les possibilités des profils UML. La démarche adoptée est explicitée au sein de ce mémoire.

La proposition d'une méthodologie pour le CITI requiert, dans un premier temps, l'identification du cycle de vie des projets du CITI. Ceci constitue le chapitre 2 de ce mémoire, prenant place après un bref rappel théorique des cycles de vie classiques au sein du chapitre premier. Le chapitre trois effectuera une brève description du langage de modélisation qu'est UML et de ses possibilités d'extension. La méthodologie proposée au CITI, déroulant les différents éléments d'UML qu'elle met en oeuvre prend place au sein du chapitre 4. Le chapitre 5 a pour objet de décrire les outils rendus possibles par les profils UML et ceux qui ont été implémentés spécifiquement pour le CITI en accord avec la méthodologie proposée.

Chapitre 1

Modèles de cycles de vie du développement logiciel

1.1 Introduction

Depuis l'identification du besoin, jusqu'à la livraison finale à l'utilisateur, le produit logiciel subit des changements graduels, des transformations. Ce développement constitue un processus complexe. Le cycle de vie permet de dérouler dans le temps de tels processus en présentant les activités auxiliaires de ceux-ci.

"Le cycle de vie du processus logiciel permet l'organisation du travail (découpe), le partage des responsabilités, le suivi et la mise en évidence des activités auxiliaires" [Hab02]. A ces activités intermédiaires, ou phases sont généralement associés des produits ou livrables.

Le cycle de vie logiciel, tel que défini par la norme iso 12207, englobe la gestion de projet, le pré-développement (identification des besoins de l'utilisateur par opposition aux besoins en logiciel établis en phase de développement), développement, post-développement (installation, maintenance, retrait, etc.) et les processus globaux (gestion de configuration, vérification, validation, documentation et formation)[122]. *"La partie du cycle de vie logiciel consacrée au développement à proprement parler s'appelle le cycle de développement du logiciel (software development cycle) et correspond au processus de développement du logiciel (software development process). Le cycle de développement du logiciel commence avec la décision de développer un logiciel et se termine avec la livraison du logiciel et son installation"*[Str00]. Il est accompagné de processus globaux.

Dans ce chapitre, je présente quelques modèles classiques de cycle de vie de développement logiciel, j'illustrerai quelques uns de ceux-ci par des méthodologies de développement dont le cycle de vie présente des similarités avec ces modèles généraux.

1.2 Les modèles classiques de cycle de vie

1.2.1 Modèle en cascade

Le cycle de vie en cascade est une approche "top-down" du processus logiciel, inspiré du modèle conception/production du matériel. Il s'agit d'une succession de quelques phases principales où l'output d'une étape est l'input de la phase suivante. Ce modèle se décline

Modèle général

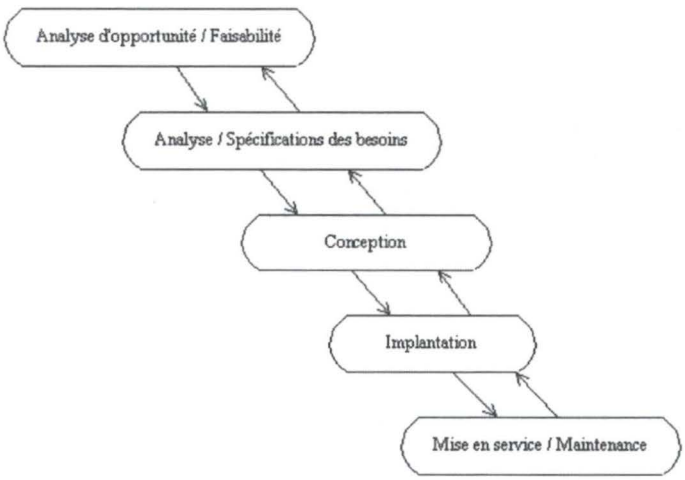


FIG. 1.1 – Modèle de cycle de vie en cascade

Description des phases

Lors de l'*analyse d'opportunité*, le client et les analystes définissent, par le biais d'interviews ou d'études de l'existant, un "cadre général pour une éventuelle solution automatisée, ainsi que les ressources et technologies à mettre en œuvre"[Hab02]. On y étudiera également la faisabilité des solutions envisagées.

Lors de l'*analyse et spécifications des exigences*, les analystes, dans cette phase, structurent et formalisent les exigences fonctionnelles et non fonctionnelles du logiciel décrites de manière non structurées par le client.

Lors de la *conception*, les analystes (ou architectes informatique) définissent une solution logicielle répondant aux diverses exigences. Cette solution est définie, par un ensemble de modules liés entre eux permettant de satisfaire, par leur collaboration, aux diverses exigences fonctionnelles. Le produit de cette phase est une architecture logique, indépendante de la plate-forme, la description détaillée des structures de données, les algorithmes à produire, les interfaces.

Lors de l'*implantation*, on dérive de l'architecture logique une architecture physique en considérant les exigences non fonctionnelles du client ou le savoir-faire de l'architecte informatique. Le produit de cette phase est une architecture physique et le codage par les programmeurs des modules dans la technologie choisie. Cette phase a également pour objectif la vérification des modules par des tests et l'intégration des divers modules devant être testée également.

Lors de la *mise en service et de la maintenance*, le logiciel, une fois implanté sur l'environnement d'exploitation réel, peut être amené à évoluer au sein de son environnement

développé. Enfin, le fournisseur peut, également parfois être amené à assurer les formations au client.

Avantages

- Simplicité, clarté du processus
- A permis une discipline du processus de développement [Hab02]
- Mise en évidence des points de contrôle [Hab02]

Inconvénients

- Tentative de développer un système d'information en un coup.
- Rigidité des phases.
- Pas de prototype des interfaces Hommes-Machines, l'ergonomie n'est découvert par l'utilisateur qu'au terme du processus.
- Sorte de "Trou noir" du début à la fin pour l'utilisateur.
- Abus de certaines sociétés légitimant certains excès de prestations par cette méthode.
- Erreurs découvertes trop tard.
- "Points de contrôle basé uniquement sur la production d'un document" [Hab02]
- Nécessité d'avoir un produit complet au terme d'une phase pour commencer la suivante est parfois peu faisable.
- Eloignement de l'utilisateur.

Un exemple de variante : Merise [Sob97]

Merise est une méthodologie du processus logiciel d'origine française. Le modèle de cycle de vie sous-jacent à cette démarche, tout en étant un modèle en cascade du point de vue de sa forme, diffère sur quelques éléments du modèle général présenté ci-dessus. En tant que méthodologie, Merise effectue des recommandations plus exhaustives pour chacune de ses phases.

pouvant être retirés du projet. On y vérifie que les ressources à mettre en œuvre (humaines, technologiques, financières, etc.) peuvent effectivement être affectées au projet [Sob97].

L'*Etude préalable* a pour objectif la définition des besoins et des objectifs retenus pour le nouveau système. On y décompose le système d'informations en un ensemble de modules et de structures de données.

L'*Etude détaillée* a pour objectif la description détaillée des fonctionnalités au niveau de chaque module.

La *Réalisation* a pour objectif l'élaboration du code des divers modules et l'intégration de ceux-ci. On y effectue également les tests unitaires et/ou les tests systèmes (effectués sur l'ensemble du produit fini et établis par rapport aux exigences initiales).

La *Mise en œuvre et la maintenance* consiste en la phase de déploiement vers le nouveau système et l'éventuelle formation aux utilisateurs. On y retrouve également de la maintenance corrective/adaptative (correction d'erreurs techniques) et évolutive (par rapport aux propositions des utilisateurs).

1.2.2 Modèle avec prototype à jeter

Ce modèle de cycle de vie tente de répondre à certains problèmes cités ci-dessus (éloignement de l'utilisateur) en proposant à l'utilisateur une meilleure visibilité sur les spécifications des exigences. Le prototype est dit jetable car son but est de vérifier la conformité aux exigences, celui-ci est reconstruit en tenant compte des remarques des utilisateurs. La description qui en est faite ici est inspirée de [Hab02].

Modèle général

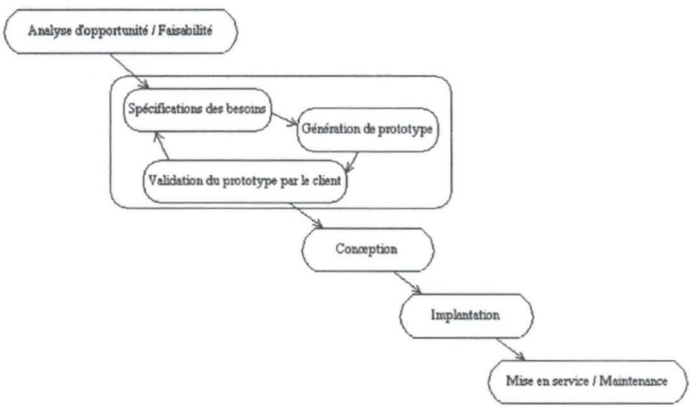


FIG. 1.3 – Modèle de cycle de vie avec prototype à jeter

Avantages

Inconvénients

- Rigidité du Modèle en cascade [Hab02].
- Réalisation du prototype peut s'avérer fastidieuse et coûteuse car relativement tôt dans le cycle de vie.

1.2.3 Modèle en V

Le modèle en V est issu du modèle en cascade. Ce modèle rend explicite les phases de validation des différents produits par les tests et met en correspondance produits et tests associés [Hab02, Sob97].

Modèle général

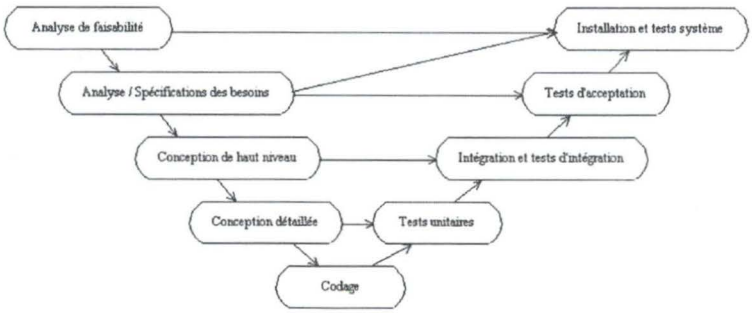


FIG. 1.4 – Modèle de cycle de vie en V

Avantages

- "Validation rendue explicite"[Hab02]
- "Met en évidence la symétrie entre les phases du début de cycle de vie et celles de fin de cycle de vie"[Hab02]

Inconvénients

- Mêmes inconvénients que le modèle en cascade.

1.2.4 Modèle incrémental

Ce modèle est également issu du modèle en cascade , toutefois la conception et l'implantation se fait par incrément. On identifie des sous parties du logiciel pouvant être livrées. Les différentes parties sont développées, testées et livrées selon différentes priorités. La description qui en est faite ici est inspirée de [Hab02, Sob97, Vic00, Roq02].

Modèle général

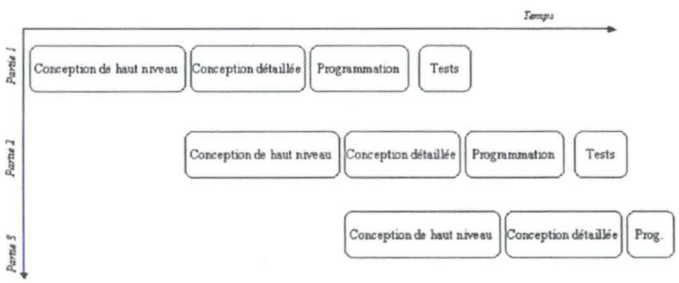


FIG. 1.5 – Modèle de cycle de vie incrémental

Avantages

- Meilleure utilisation des ressources pour le processus logiciel, les architectes informatiques et programmeurs réalisent les mêmes phases de parties différentes.
- L'intégration est incrémentale et donc plus aisée.
- Meilleure visibilité par le client, la livraison s'effectue partie par partie.

Inconvénients

- Architecture logicielle spécifique au modèle incrémental, il ne s'agit pas nécessairement de la meilleure architecture.
- Risque de devoir remettre en cause une partie déjà livrée.

Exemple : Rapid Application Development (RAD)

Rapid Application Development (RAD) est un cycle de vie de développement dont les objectifs sont la rapidité, la qualité et le faible coût du développement. La description qui en est faite ici est inspirée de [Sob97, Vic00].

Les phases d'initialisation, de cadrage et de design, sont similaires au cycle de vie en cascade tandis que la réalisation proprement dite permet un parallélisme du travail, il s'agit d'une approche incrémentale du développement.

- RAD se base sur les **outils CASE**, "la génération de code, la vérification de cohérence et d'intégrité procurée par ceux-ci facilitent le développement RAD"[Sob97]. La génération de prototype par exemple, est facilitée, permettant à l'utilisateur de voir, par ce biais, l'évolution du développement.
- RAD se base sur des **équipes Hybrides** ; ces équipes, constituées de 6 à 7 personnes ont un rôle à la fois d'analyste, de "designer" et de programmeurs.
- RAD se base également sur une **gestion de projet** devant permettre l'extraction rapide des besoins auprès de l'utilisateur. La gestion d'un développement de type RAD se fait par l'utilisation de "timeboxing" (Technique de planification basée sur le respect d'un délai butoir et conditionnant la dimension temporelle du projet RAD).
- RAD se base également sur un **prototypage itératif** impliquant fortement l'utilisateur. Le Rad est en effet caractérisé par une validation permanente garantissant, pour chaque ajout de fonctionnalité, la conformité aux besoins.

Le cycle de vie RAD se décompose en 5 phases : Les chiffres sont issus de [Vic00].

- L'*initialisation* représente environ 6 % du projet en charge, l'objectif est de spécifier les exigences dans leurs grands traits. On organise les groupes de travail auxquels sont assignés des thèmes.
- Le *cadrage* représente environ 9 % du projet. L'objectif est de spécifier les exigences quant à l'application. Ceci se réalise grâce à des réunions Joint Application Development (JAD) où utilisateurs et développeurs se rencontrent dans la cadre d'un brainstorming en vue d'identifier les besoins initiaux pour chaque thème. Lors du Cadrage on verrouille les exigences, les budgets, les délais et la solution globale sur les plans stratégiques, fonctionnels, technologiques et organisationnels.
- Le *design* représente environ 23 % du projet. L'objectif est ici d'élaborer un modèle détaillé des fonctions à développer. Différentes réunions de travail sont organisées en vue de décomposer ces fonctions métiers et de définir le modèle de données associé. Pour les fonctions les plus critiques, un prototype est développé en se basant sur la définition des besoins réalisée à ce niveau du développement. Le travail est réalisé en parallèle par les différentes équipes.
- La *construction* représente 50 % du projet. Lors de cette phase , les développeurs élaborent l'application module par module, l'utilisateur participe toujours au développement en validant les prototypes. Plusieurs sessions itératives sont nécessaires.
- La *finalisation* représente 12 % du projet. L'objectif poursuivi est d'officialiser une livraison globale à l'utilisateur et de transférer le système en exploitation et maintenance. (Mise en œuvre sur le site pilote)

Du point de vue de la qualité du développement, RAD prévoit un suivi rigoureux de la qualité fonctionnelle par les rapport de "Focus" et le suivi des divergences [Vic00]. Les "Focus" constituant les réunions de tous les intervenants du projet, le but en est d'offrir une visibilité maximale sur l'avancement des travaux, de permettre une validation globale des principes et de "lever" les principaux risques. Outre ces réunions de focus, des réunions "Jalons Zero default" peuvent être planifiées entre les focus (revue de code + intégration + validation par l'utilisateur de base). Ceci permet après une étape importante de contrôler l'avancement du projet en terme de qualité et de visibilité.

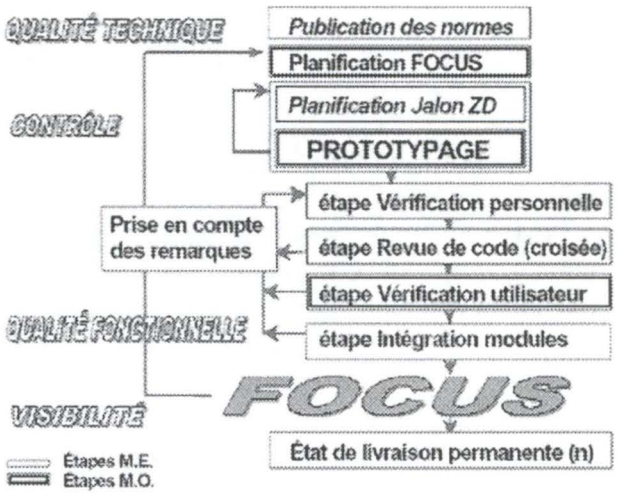


FIG. 1.6 – Suivi de la qualité du développement RAD [Vic00]

Exemple : Unified Process (UP) [Roq02, SA02]

Unified Process ou processus unifié en français est un processus de développement logiciel itératif et incrémental centré sur l'architecture, guidé par les cas d'utilisation et piloté par les risques . UP est plus qu'un simple cycle de vie, il s'agit d'une méthodologie complète liée à un langage de spécification des exigences (UML) et complétée de diverses recommandations. La méthodologie basée sur UML proposée au chapitre 4 est inspirée du processus unifié.

Caractéristiques du Processus Unifié [Roq02]

- *Itératif et incrémental* : le projet est découpé en itérations de courte durée (environ 1 mois) au terme desquelles une partie exécutable du système final est produite et intégrée de façon incrémentale (par ajout).
- *Centré sur l'architecture* : tout système complexe doit être décomposé en parties modulaires afin d'en faciliter la maintenance et l'évolution.
- *Piloté par les risques* : les risques les plus importants du projet doivent être identifiés au plus tôt et levés le plus rapidement possible. Les mesures à prendre dans ce cadre déterminent l'ordre des itérations. (On détermine des priorités sur les parties).
- *Guidé par les cas d'utilisation* : les exigences fonctionnelles de l'utilisateur conditionnent le processus.

Les phases du Processus Unifié Le processus Unifié s'organise autour des 4 phases suivantes :

- La phase d'*Initialisation* a pour objectif de définir la vision du projet, sa portée et sa faisabilité.

- La *Construction* consiste en la conception et l'implémentation des éléments opérationnels.
- La phase de *transition* consiste au déploiement sur le site pilote, à la formation des utilisateurs et aux tests.

Chacune de ces phases est elle-même décomposée séquentiellement en itérations d'une durée de deux à 4 semaines. Le système développé est, au terme de chaque itération, affiné et augmenté. Il existe un certain nombre d'activités réalisées au cours de ces phases, il s'agit du business modeling (Modélisation du métier), de l'étude des besoins, de l'analyse, de la conception, des tests et de l'implémentation, mais il s'agit aussi de la gestion de configuration, de la gestion de projet ou de l'environnement. En phase d'initialisation, il s'agira essentiellement d'activités telles la modélisation du métier concerné ou encore l'étude des besoins qui se verront accomplies plutôt que l'implémentation.

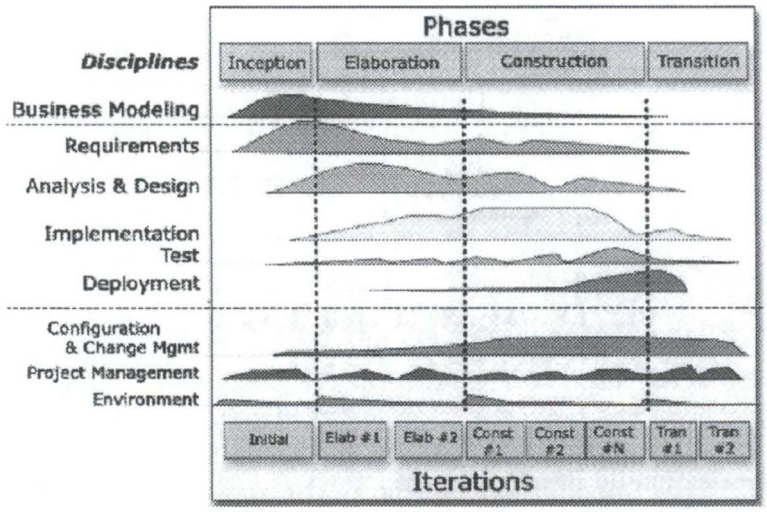


FIG. 1.7 – Les deux dimensions du processus unifié

1.2.5 Modèle transformationnel

Ce modèle est basé sur l'hypothèse qu'il existe un moyen de convertir/transformer une spécification formelle en un programme qui satisfait automatiquement à cette spécification. Il s'agit alors d'élaborer une première spécification formelle, puis par une transformation automatique d'en dériver un code satisfaisant pleinement aux spécifications. Le code peut ensuite être optimisé de telle manière qu'il respecte toujours la spécification. Une expérimentation de ce code permet de déterminer les ajustements pouvant être portés à la spécification formelle établie précédemment et de réitérer les différentes opérations jusqu'à obtenir un code satisfaisant pleinement aux spécifications formelles et aux exigences du client [Str00].

Modèle général

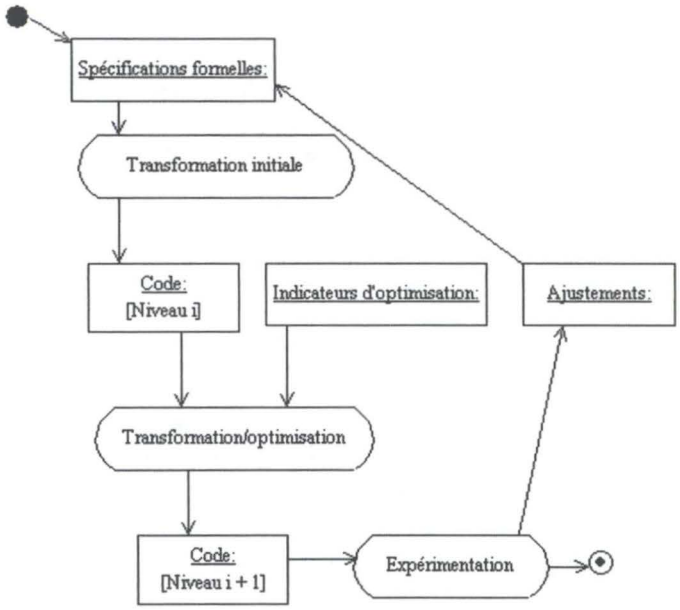


FIG. 1.8 – Modèle de cycle de vie transformationnel

Avantages

- Automatisation du développement.
- Facilités offertes par les outils case.
- Validation du développement par un raisonnement formel.
- "Intégrité du code préservée, celui-ci n'est pas modifié directement"[Str00].

Inconvénients

- Exige des aptitudes particulières de la part des développeurs.
- "Les transformations automatiques ne sont disponibles que pour de petits problèmes"[Str00].

1.2.6 Modèle en spirale

Le modèle en spirale est un modèle cyclique incluant la notion de niveau de produit selon le cycle dans lequel il se trouve. A chaque niveau du produit, l'accent est mis, dans ce modèle sur la gestion du risque. A chaque cycle, on identifie les objectifs, contraintes et alternatives. Ces dernières sont évaluées. On identifie les risques et tentons de les résoudre. Le produit est testé et le prochain cycle planifié. Le produit est au fur et à mesure des cycles ajusté pour correspondre à terme aux exigences du client [Hab02, Sob97].

Modèle général

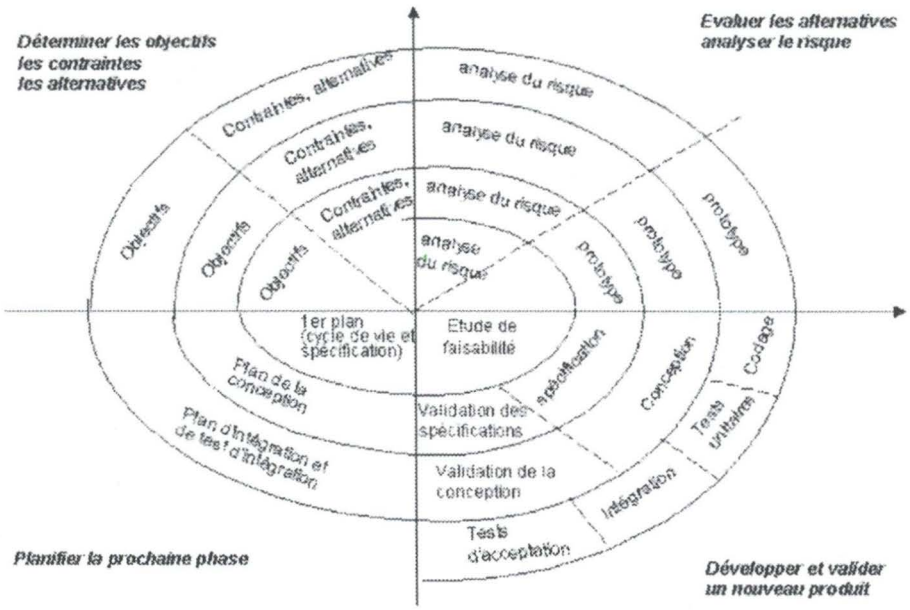


FIG. 1.9 – Modèle de cycle de vie en spirale

Avantages

- Diminution du risque
- Validation à chaque cycle, ajustement avec le client.
- Adapté si les besoins sont peu clarifiés.

Inconvénients

- Inadapté si les besoins sont clairement identifiés.

1.3 Conclusion

Chaque modèle de cycle de vie comporte ses avantages et ses inconvénients, chacun de ces modèles naît de la remise en question d'un autre modèle. Alors que le modèle de cycle de vie en cascade souffrait de sa linéarité et du risque associé ainsi que du manque d'implication de l'utilisateur, le modèle avec prototype tentait d'impliquer plus l'utilisateur. Les modèles itératifs ou incrémentaux, quant à eux permettent une meilleure gestion en affectant plus d'importance à certaine partie et une vue plus aisée de l'avancement du projet ou encore une meilleure utilisation des ressources.

Chapitre 2

Cycle de vie des développements CITI

2.1 Introduction

L'outillage au processus de développement du Centre d'Innovation par les Technologies de l'Information (CITI) requiert la formalisation du cycle de vie utilisé. Ceci en vue d'essayer d'identifier les livrables à produire au terme de chaque phase et de produire éventuellement des modèles de description type (*"template"*), règles méthodologiques ou d'autres outils pouvant guider et par là faciliter les développements.

Le cycle de vie des développements au CITI n'est actuellement pas formalisé. Les développements ne possèdent pas une démarche commune. La difficulté d'une telle formalisation réside dans la nature même des développements, qui, par opposition à des développements industriels, implique une certaine hétérogénéité dans la manière de développer. La nature des projets d'innovation complique la tâche de formalisation d'un cycle de vie.([Bod02]). Toutefois, le cycle de vie, la méthodologie ainsi que les adaptations faites à l'outil CASE concernent de petits projets développés en interne (dont le client est le CITI), et dont la nature relevant de problèmes de gestion comme décrit dans [Déc02] (informatique de gestion) et non pas le cadre général des développements faits pour le compte de clients externes.

Afin de proposer un outillage au processus de développement, il a donc été nécessaire d'essayer d'identifier quelque peu, dans une première phase, des éléments communs à tous les projets dans la démarche utilisée pour développer, en vue d'essayer de proposer un cycle de vie "plus ou moins" adapté aux différents projets selon leur nature. Il est important de préciser ici l'objectif. En effet, la finalité n'est pas de formaliser un cycle de vie des projets de développements du CITI mais bien d'établir un modèle de base, un support en vue d'illustrer les possibilités d'UML, illustrer à quoi pourrait ressembler un projet modélisé en UML et les possibilités d'adaptation d'un outil case en se basant sur ce modèle. Le cycle de vie repris est donc ici une **description des phases et des activités liées** incombant aux acteurs du développement et liées au processus de développement logiciel sans considérer les processus globaux ([122] : gestion de projet, gestion de configuration, etc.) accompagnant le développement. Egalement, ce cycle de vie n'est pas figé sous l'angle d'un ordonnancement particulier de ces activités.

Le cycle de vie devait logiquement se tourner vers une approche plus classique avec une production de livrables intermédiaires bien établis, en effet, un cycle de vie d'une méthodologie Agile, par exemple, ne formalisant pas ses exigences, s'avérerait peu pertinent pour

2.2 Démarche d'identification du cycle de vie

Développer un modèle de cycle de vie et une méthodologie UML pour le CITI requiert une démarche particulière. Il a été, en effet, nécessaire de procéder par interviews et réunions en vue d'identifier les pratiques, habitudes et étapes dans les développements du CITI ou en vue de proposer de nouvelles choses.

2.2.1 Interview

Lors d'une interview avec Marc Krystkowiak, un cycle de vie des développements CITI a été décrit. Il a été utilisé sur le projet Opal (Outillage au Processus d'Acquisition Logiciel) qui se veut être un outil dont l'objectif principal est d'assister le consultant dans la réalisation d'un cahier des charges par une capitalisation des connaissances des consultants. (L'outil permettrait de produire et de réutiliser des modèles types de description d'exigences fonctionnelles ou non fonctionnelles, modèle de cahier des charges ainsi que des fonctionnalités avancées de publication d'appel d'offres et d'évaluation de ceux-ci). Le CITI a cherché à élaborer un cycle de vie relativement formel pour ce projet. Le langage de spécifications des exigences choisi a été UML.

On y retrouve trois grandes caractéristiques à savoir :

- Une scission entre d'une part la formalisation des exigences fonctionnelles et d'autre part, une première idée de réalisation, de solution pour celles-ci. (WP2 et WP5)
- La réalisation d'une maquette et d'une validation par le client
- Une organisation quant à l'implémentation inspirée du RAD

Cycle de vie du projet OPAL

Le déroulement du projet Opal se détermine à partir de 10 "WorkProducts" (WP) linéaires ou entrelacés dont la terminaison est réalisée par la production d'un livrable particulier, spécifique (Il existe certains modèles de description, exemple : un modèle pour le plan qualité du projet). Ceci est issu d'un document interne au CITI [Lei02].

WP0. Conduite et gestion qualité du projet

Cette tâche a pour résultat essentiel le plan qualité projet, ce livrable contient la *"définition et le suivi des dispositions à prendre dans le cadre du projet afin d'en assurer la qualité et d'atteindre les résultats attendus"*[Lei02]. Ce workproduct accompagne le processus de développement proprement dit, il relève de la "gestion qualité logiciel" associée au processus global de validation et vérification [122]

WP1. Lancement projet, Evaluation du travail

Il s'agit ici de décrire le planning du projet, identifier les points d'arrêt où une validation sera nécessaire.

WP2. Spécifications fonctionnelles détaillées

Ce workproduct est associé au développement proprement dit, il s'agit ici d'élaborer la spécification des exigences fonctionnelles de l'outil. Cette description devra être structurée, modélisée en UML.

WP3. Maquettage de l'application OPAL

Une maquette reprenant les interfaces de l'outil sera le résultat essentiel, elle sera réalisée à partir du résultat de wp2.

WP4. Validation de la Maquette OPAL

Le résultat de ce workproduct est la validation des écrans et des principales fonctionnalités par le client ainsi que la définition des grandes orientations stratégiques par rapport à l'outil par le comité de pilotage. Les exigences fonctionnelles ne sont donc pas validées à partir du modèle UML qui en est fait mais plutôt à partir de la maquette.

WP5. Conception détaillée (phase de Design)

Il s'agit ici d'élaborer dans un premier temps l'architecture logique d'Opal puis dans un second temps l'architecture physique.

WP6. Mise en place de l'organisation de développement(implémentation)

Ce workproduct a pour principaux résultats la définition des règles de programmation, la gestion de configuration, le planning de développement et la liste des indicateurs du projet et des modes de surveillance.

WP7. Développement

Principaux résultats : version 1.0 de OPAL L'implémentation s'effectue pour le projet Opal d'après quelques recommandations effectuées par la méthodologie RAD ([Bod02]) concernant le suivi de la qualité comme présenté à la figure 1.6 .

WP8. Documentation

Le résultat de ce workproduct est la documentation du code, la réalisation d'un guide utilisateur.

WP9. Cahier des recettes

Le résultat de ce workproduct est le cahier des recettes. (Ensemble des recettes, liste de tests, de façon à permettre au client de vérifier un maximum (jamais exhaustif) la conformité du produit livré avec les attentes du client, il permet au client d'établir une vérification d'aptitude du logiciel).

Il s'agit également d'une aide à la démarche d'appropriation des utilisateurs par des actions de sensibilisation, des compléments de formation ou encore des mesures de satisfaction.

WP10. Mise en exploitation et Identification des contraintes de maintenance

L'objet de ce workproduct est le packaging de l'outil, la définition de la politique de licence ainsi que les procédures de mise à jour de logiciel.

2.2.2 Réunions : propositions de cycles de vie, remarques, validations : Beatrix Barafort, Jean-Philippe Bodelet, Stefan Leidner

La formalisation du cycle de vie s'est accompagnée de 3 réunions de discussion en vue de retirer les éléments intéressants de normes, de modèles de cycles de vie classiques et de confronter ces éléments avec les expériences des développements au CITI tel Opal.

2.2.3 Groupe de travail (GT) : Réunion avec les développeurs

Lors de ce groupe de travail, j'ai présenté l'ébauche de cycle de vie réalisée, en vue de sensibiliser ceux-ci aux travaux futurs consistant à essayer d'outiller ce cycle de vie utilisé en tant que **support** pour le développement de ces outils.

2.3 Cycle de vie de développement proposé

2.3.1 Résumé

Une première caractéristique de ce cycle de vie est de décomposer les spécifications des exigences fonctionnelles en trois niveaux. Le premier niveau étant une description de l'exigence par les relations qu'aurait l'utilisateur final avec, non pas l'application proprement dite, mais le système dans sa globalité en tant que boîte noire, c'est-à-dire un système intégrant les divers autres acteurs, composants matériels et logiciels pouvant collaborer à la réalisation de l'exigence fonctionnelle. L'objectif de la première étape consistant à identifier correctement les exigences fonctionnelles par le client (qu'est ce qu'on veut ?). Le second niveau étant la description de l'exigence par les interactions entre l'utilisateur et les différents éléments (Acteurs secondaires, appareil particulier, etc.) du système global dont notamment le système d'informations proprement dit, toutefois on ne décrira toujours pas au cours de cette seconde étape, le fonctionnement interne du SI. On cherche à établir ici comment on peut organiser le système, ce qu'on peut demander à l'utilisateur, au SI ou à d'autres composants pour réaliser l'exigence fonctionnelle. Le troisième niveau consistant à ouvrir la boîte noire faite sur le système d'informations, on a alors une description de l'exigence par toutes les relations, interactions mises en oeuvre entre l'utilisateur, les composants externes au SI, les composants du SI. Cette troisième étape correspondant à l'identification d'une solution (Comment va fonctionner le Système global pour répondre à l'exigence fonctionnelle ?).

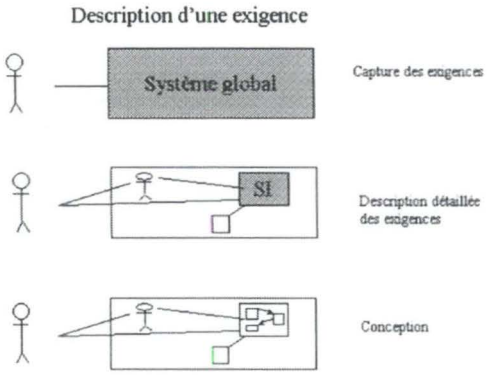


FIG. 2.1 – Description d'une exigence selon la phase

Une seconde caractéristique, tient compte de la nature des projets du CITI dont le contexte requérant une validation par le client implique la création d'un prototype et la prise en compte d'adaptations nécessaires en *"cours de route"*.

Enfin, ce cycle de vie a également la particularité de verrouiller la formalisation des besoins avant l'implémentation. Seule l'implémentation proprement dite se voit découpée selon les priorités (à l'image d'Opal, on spécifie l'ensemble des exigences fonctionnelles en UML puis on établit les priorités et seule l'implémentation est découpée en morceaux plus petits selon les priorités).

Avant de décrire plus en détail le cycle de vie comme support à l'adaptation de l'outil CASE, notons la souplesse qu'apporte une telle adaptation, en effet d'autres solutions sont possibles. Le déroulement proprement dit des activités n'a pas d'importance dans le cadre de l'outillage CASE mais plutôt le contenu des livrables intermédiaires, etc. Si par exemple, au terme d'une première identification des exigences correspondant à la "capture des besoins", on choisit d'effectuer les activités suivantes pour chaque sous-partie et pouvoir livrer par sous-partie, ceci n'est pas gênant du point de vue de l'outillage CASE.

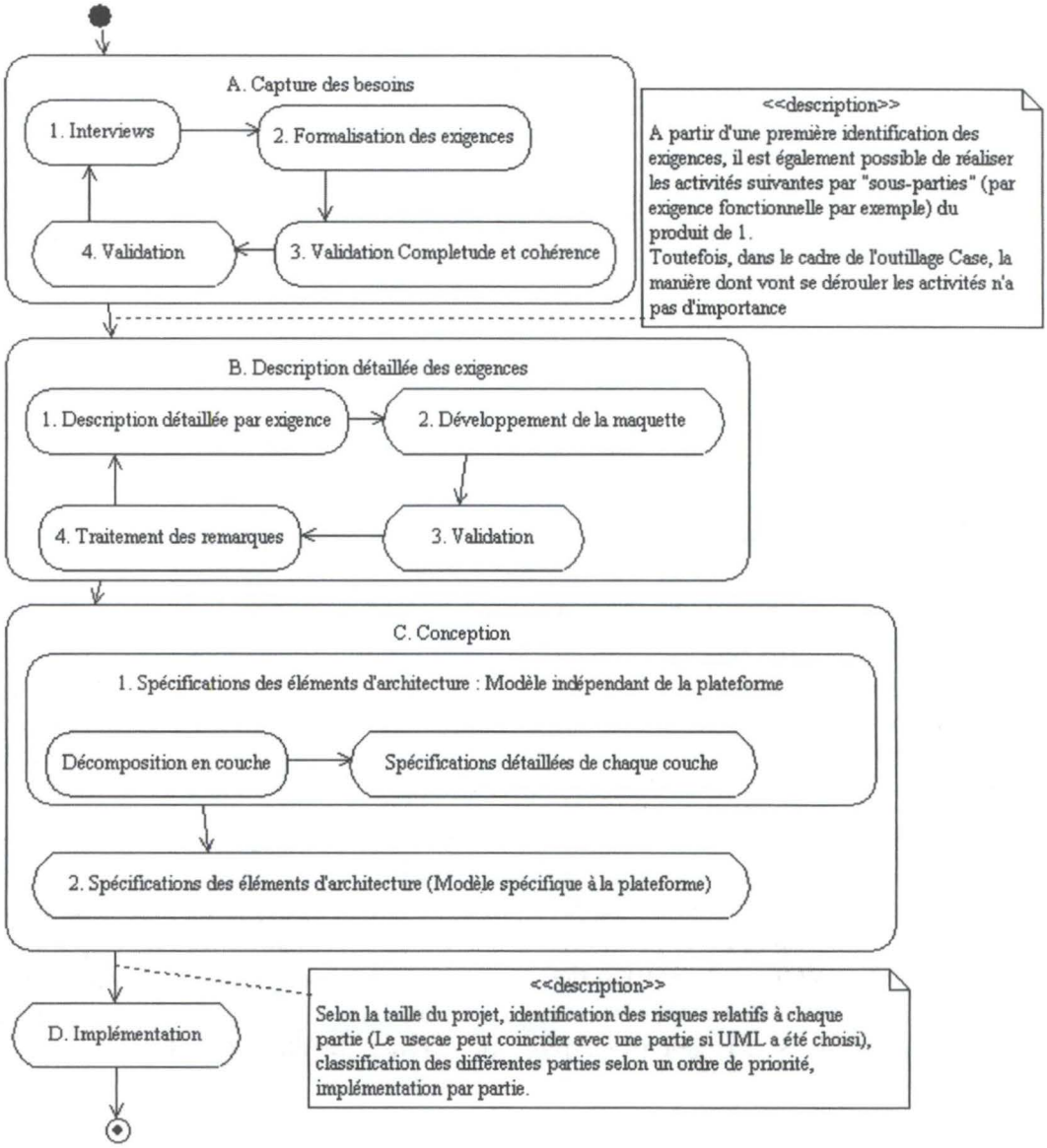


FIG. 2.2 – Modèle de cycle de vie proposé

2.3.2 Description détaillée des phases

A. Capture des exigences

A. 1 Interviews

- **Acteurs :** Client, Experts du Domaine
- **Description :** Cette phase consiste pour l'analyste, grâce à son savoir-faire, à interpréter et comprendre ce que veut le client. L'analyste s'approprie le contexte du projet et le domaine. Il elicite les exigences du client. L'analyste est considéré dans cette tâche comme l'interprète entre le client et le développeur.
- **Techniques :** Interviews, Entrevues avec le client, Observations, Etudes de l'Existant.
- **Résultat :** Liste de besoins, d'exigences sur le système, de connaissances sur le domaine, et de contraintes non fonctionnelles sous forme informelle.

A. 2 Formalisation des exigences

- **Acteurs :** Analyste
- **Description :** Il s'agit ici de déterminer quels services les utilisateurs vont attendre du système et quels services ne feront pas partie du système (Il s'agit d'exigences utilisateur par opposition avec des exigences logicielles). Il s'agit d'une approche "black box" sur le système global, lors de la description des exigences. On décrit les relations entre l'utilisateur et un système global. Par système, on entend système global tel que défini plus haut. L'Analyste devra également décrire le domaine métier concerné par le développement. Les tâches incombant à l'analyste sont donc ici :
  - Le choix d'un langage de spécification.
  - La transcription formalisée et structurée des exigences fonctionnelles et non fonctionnelles (délimitation des frontières du système).
  - Description des acteurs principaux interagissant avec le système global.
  - Description du domaine.
- **Techniques :** Langage de Spécification simple/lisible : UML., la partie Système est vue comme une boîte noire.
- **Résultat :** Le résultat doit être un support de validation pour le client ce qui implique un langage de spécification compréhensible par l'utilisateur, ou un rephrasage lors de la validation
  - Formalisation des exigences fonctionnelles et du domaine.
  - Formalisation des coopérations possibles entre l'utilisateur et le système pour chaque exigence fonctionnelle.
  - Formalisation des exigences non fonctionnelles
  - Description des acteurs.

A. 3 Vérification complétude et cohérence

- **Acteurs :** Analyste
- **Description :** L'Analyste vérifie la complétude et la cohérence, il tente de lever les éventuelles ambiguïtés.
- **Techniques :** Mise en évidence de conflit, de lacunes

A. 4 Validation

- **Acteurs :** Analyste, Client
- **Description :** Validation par le client des documents précédents en terme de conformité avec ses exigences et son domaine.
- **Techniques :** Rephrasage, simulation.
- **Résultat :** Accord de toutes les parties/Remarques -> Réitération. Documents précédents validés.

A. 5 Traçabilité

- **Acteurs :** Analyste
- **Description :** Il est nécessaire d'établir un mécanisme permettant de tracer les éléments de description textuelle élicités lors des interviews et les éléments formalisés. Cette activité accompagne les activités précédentes.
- **Techniques :** Matrice, Outil paramétré, etc.
- **Résultat :** Matrice de traçabilité / Outil paramétré
  Exemple de matrice de traçabilité si on choisit UML comme langage de spécifications [RV00] :

		Exigence 1	Exigence 2	Exigence 3
Use case A	Scénario 1	x		x
	Scénario 2			x
	Scénario 3			x
Use case B	Scénario 1			
	Scénario 2	x		

FIG. 2.3 – Matrice de traçabilité UML

B. Description détaillée des exigences

B. 1 Description détaillée des exigences

- **Acteurs :** Analyste
- **Description :** On passe des exigences sur le système à des exigences logicielles (du SI) et sur d'autres composants du système global (acteur, composant matériel). La description de ces exigences est alors faite par la description des messages échangés entre les différentes composantes du système global pour réaliser l'exigence fonctionnelle. L'Analyste devra également affiner sa description du domaine.
- **Techniques :** Sur base des documents établis précédemment. La partie SI est vue comme une boîte noire.
- **Résultat :**
  - Description du système global et description détaillée des éléments du système global. (Composants, SI, de nouveaux acteurs, (utilisateur)).
  - Description détaillée des objets métiers.
  - Affinement de la description des exigences fonctionnelles faites en A. 2, on décrit chaque exigence fonctionnelle par les interactions entre composants, acteurs et sys-

B. 2 Développement de la maquette
- **Acteurs :** Analyste, Développeurs
- **Description :** Réalisation des IHMs sur papier, Modélisation sous un outil et génération de prototype des interfaces homme-machine
- **Techniques :** Outils de générations de fenêtres.
- **Résultat :** Maquette des IHMs.

B. 3 Validation
- **Acteurs :** Analyste, Client
- **Description :** Validation par le Client de la maquette en terme de fonctionnalités recouvertes.
- **Techniques :** Réunions de démonstration.
- **Résultat :** Rapport des remarques, compte rendu de réunion.

B. 4 Traitement des remarques
- **Acteurs :** Analyste
- **Description :** Prise en compte des remarques éventuelles faites lors de la validation. Adaptation de l'analyse. Réitération si nécessaire.
- **Techniques :** /
- **Résultat :** Documents précédents mis à jour.

B. 5 Traçabilité
- **Acteurs :** Analyste
- **Description :** Mise à jour de la traçabilité
- **Techniques :** Matrice, outil paramétré.
- **Résultat :** Matrice de traçabilité, outil paramétré.

C. Conception

C. 1 Spécifications des éléments d'architecture (Architecture logique)
- **Acteurs :** Architecte informatique
- **Description :** Description détaillée des composants de l'architecture en terme de couches (BusinessData, View, BusinessControl, etc.) présentant une cohérence interne.
- **Techniques :** Savoir-faire de l'architecte informatique.
- **Résultat :** Identification et description formelle de chacun des composants globaux de l'architecture.

Exemple de décomposition de l'architecture :
- Conception détaillée de la modélisation des données. Il peut s'agir d'un schéma Entité/Association, Diagramme de classe (UML), etc.
- Conception détaillée de l'architecture de la couche logicielle de l'IHM (View)
- Conception détaillée de l'architecture de la couche logicielle métier (Business Logic),

- Conception détaillée de la couche applicative. Il s'agit de la couche chapeau, contrôlant les autres couches.

C. 2 Spécification des éléments d'architecture (Architecture physique)

- **Acteurs :** Architecte informatique
- **Description :** Il s'agit ici de décrire l'architecture des composants physiques, des technologies et des outils
  Exemple : Swing -> JSP-> Java-> JDBC-> ORACLE, etc. (<> Couche de données, couche métier, etc.)
- **Techniques :** Lié au savoir-faire de l'architecte informatique. Les choix de technologies peuvent également découler de contraintes de conception identifiée lors de la capture des besoins du client (Exigences non fonctionnelles).
- **Résultat :** Description, modèle représentant l'architecture physique.

C. 3 Traçabilité

- **Acteurs :** Architecte informatique
- **Description :** Description des relations entre les éléments décrivant les exigences, issus de la phase de description détaillée des exigences et les éléments de l'architecture.
- **Techniques :** Matrice de traçabilité, Outil CASE
- **Résultat :** Documents de traçabilité mis à jour, Outil paramétré.

D. Implémentation

Cette partie peut donner lieu à une phase itérative et incrémentale, où après identification des risques, chacune des parties répondant aux exigences du client peut être développée selon un ordre de priorité. Chacune des partie venant se greffer aux précédentes et affinant le système global. L'implémentation ne rentrant pas dans le cadre de l'outillage par les profils UMLs, ce point au même titre que les tests et la maintenance n'a pas été approfondi.

2.4 conclusion

Le cycle de vie proposé est adapté aux projets de développement internes du CITI qui restent relativement de petite taille et où il est nécessaire de pouvoir impliquer à un moment ou un autre le client (que ce soit par la présentation du prototype ou de documents réalisés suite à la phase de capture des exigences). Les caractéristiques de ce cycle de vie sont issues soit des habitudes des employés (ex. : La présentation intermédiaire d'un prototype), ou de nouvelles propositions faites par les membres du CITI (description d'une exigence fonctionnelle en 3 étapes). Ce cycle de vie assez classique (succession de phases aux termes desquelles sont produits des livrables bien définis) constituera un support en vue d'illustrer la méthodologie UML (Comment la spécification itérative d'une exigence fonctionnelle s'articulera-t-elle avec UML ? ou encore comment les phases de prototypage et de conception peuvent elles être réalisées en utilisant UML comme langage de modélisation ?).

Chapitre 3

Unified Modeling Language

3.1 Introduction

L'Unified Modeling Language (UML), traduisez langage de modélisation unifié, est issu de 3 méthodes qui ont le plus influencé la modélisation Objet dans les années 90 (OMT-1 de Rumbaugh [Rum95], OOSE de Jacobson [Jac92] et Booch [Boo94]). De nombreux ndustriels ayant par la suite adopté UML, ont contribué à son développement. En effet, UML est, désormais, sous la responsabilité de l'Object Management Group (OMG), un groupe d'industriels dont l'objectif est la standardisation autour des technologies objet, en vue de garantir l'intéropérabilité des développements.

UML est un langage de modélisation orienté objet, il permet de décrire des besoins, de spécifier , de documenter des systèmes ou encore d'esquisser des architectures logicielles sous une notation graphique standardisée. UML se veut être un standard souple et flexible, il reste en effet assez neutre et doit être complété d'une méthodologie ou de recommandations d'utilisation (Exemple : UP) propres aux besoins d'une industrie, entreprise particulière pour ses développements. La spécification UML est disponible gratuitement sur `http://www.omg.org`.

3.2 Les diagrammes UML

[RV00, Sob97, Déc02, Sof99, Gro03, Roq02]

3.2.1 3 Axes de Modélisation

UML définit 9 diagrammes différents, ceux-ci ont des rôles particuliers quant à la modélisation d'un développement, toutefois on peut les regrouper au sein de 3 axes d'utilisations différentes [Roq02].

La description faite d'UML, ci-après a pour but de redéfinir brièvement les diagrammes UML et leur contexte d'utilisation au lecteur. Elle est basée sur la définition d'UML 1.5 désormais disponible. Une documentation plus exhaustive sur UML est disponible sur le site de l'omg.

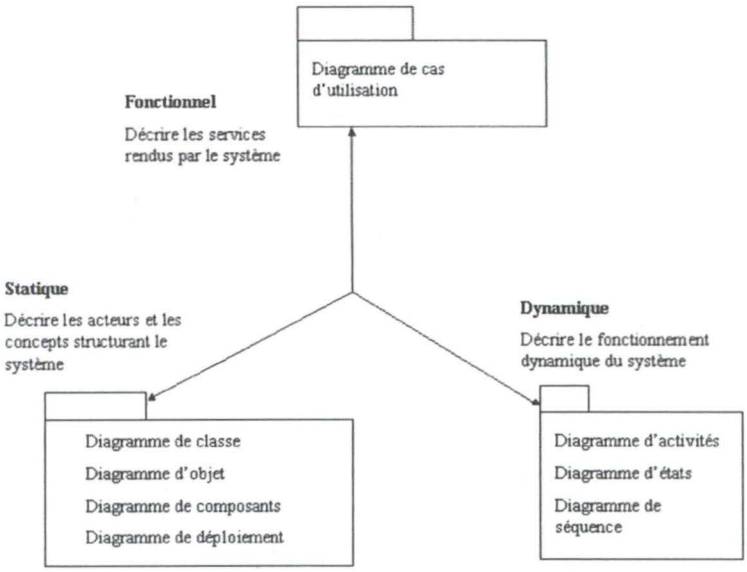


FIG. 3.1 – Les 3 axes de modélisation en UML [Roq02]

3.2.2 Modélisation Fonctionnelle

Le Modèle fonctionnel d'UML s'appuie exclusivement sur le diagramme de cas d'utilisation (Use Case). Ce dernier sert à l'activité de modélisation des besoins, c'est-à-dire des exigences fonctionnelles du système à développer. L'idée de ce diagramme est de définir quels services le système devra accomplir et indirectement quels services le système ne devrait pas réaliser. Le diagramme de cas d'utilisation sert à délimiter les frontières du système. La modélisation des besoins en cas d'utilisation se fait toujours du point de vue de l'utilisateur. La description des use cases ne doit donc en aucun cas décrire des solutions d'implémentation. L'objectif d'un modèle conceptuel des besoins est d'éviter de tomber dans la dérive d'une approche fonctionnelle, où l'on liste les fonctions et sous fonctions à développer pour répondre aux besoins du client. Le usecase, peut être décrit par un diagramme d'interaction reprenant la suite d'échanges que l'utilisateur est prêt à effectuer avec le système en vue d'accomplir un objectif ("goal") particulier. La description d'un cas d'utilisation se complète souvent par une description textuelle ne faisant pas partie d'UML.

Les éléments de base du diagramme de cas d'utilisation sont :

- L'*acteur*, celui-ci n'est pas nécessairement une personne physique, il peut s'agir également d'un composant matériel, d'un système d'information, etc. L'acteur désigne un type de personne (selon le domaine), par exemple : client, administrateur, internaute. L'acteur déclenche un ou plusieurs cas d'utilisation.
- Le *cas d'utilisation*, celui-ci décrit une séquence d'actions réalisées par le système et produisant un résultat pertinent pour un acteur particulier [RV00]. Il correspond à une attente, un besoin du client. Les cas d'utilisation peuvent être structurés en

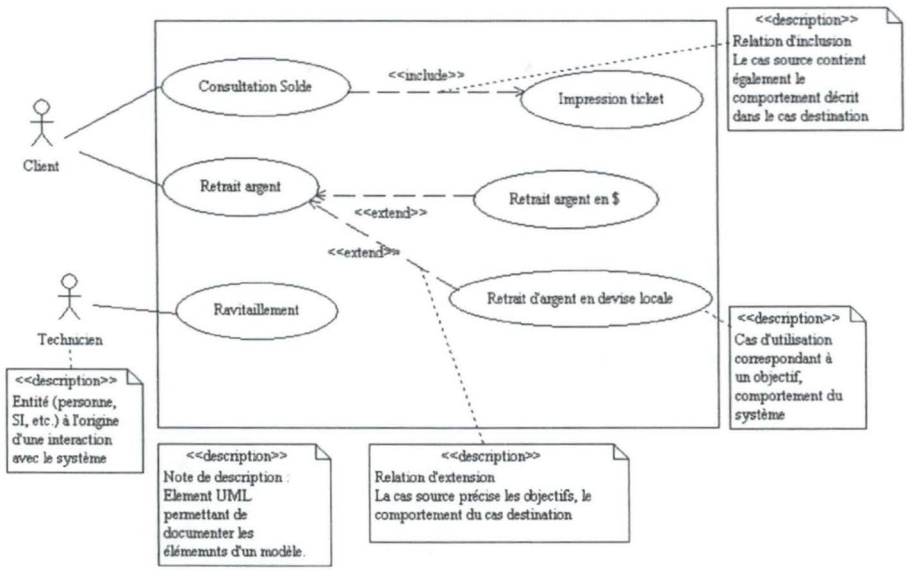


FIG. 3.2 – Gestion d'un distributeur d'argent, exemple de diagramme de usecase

3.2.3 Modélisation Statique

Le mode de représentation statique s'appuie sur le diagramme de classe, d'objets, de composants et de déploiement.

Diagramme de classe et diagramme d'objets

Le rôle du diagramme de classe peut varier selon la phase de développement où l'on se situe. En phase d'analyse, celui-ci peut servir à décrire de manière statique les objets métiers du domaine. Dans une phase ultérieure, il pourra servir à décrire les objets d'une base de données. En conception, le diagramme de classe peut servir à décrire l'architecture ou encore la structure en Packages du code. Les utilisations qui sont faites du diagramme de classe sont nombreuses.

Le diagramme de classe modélise un ensemble de type d'entités, les instances de ceux-ci, les entités proprement dites peuvent être représentées par les objets du diagramme d'objets. Par exemple, lorsqu'on trouvera le concept "client" dans la modélisation du domaine métier d'une banque, on peut modéliser au sein d'un diagramme d'objet, le client "Monsieur Dupond" ainsi que ses occurences de type d'association avec les autres concepts et les valeurs de ses propriétés.

Outre les *classes*, on trouve au sein du diagramme de classes, les *associations* entre classes. Elles ont pour rôle de modéliser une relation bi-directionnelle entre deux classes. Les *associations* au sens UML désignent un type d'association. Les instances de ces types

une relation de composition) ou encore la généralisation (les sous-classes héritent de la description de la super-classe) [BHH+97].

Le type d'association possède un nom mais peut posséder également des noms de rôles joués par chacune des classes liées ou encore des cardinalités indiquant le nombre d'instances des classes liées pouvant participer à une instance de l'association.

Les classes peuvent contenir des attributs, c'est-à-dire des propriétés de ces classes ou encore des opérations portant sur ces classes.

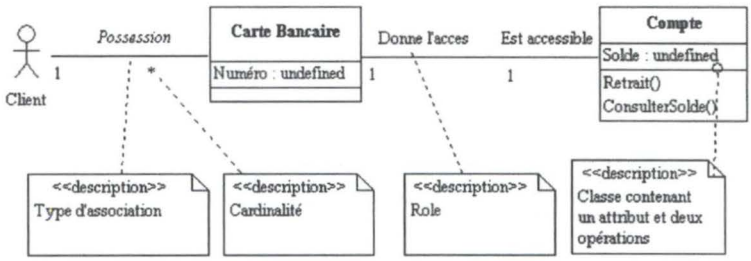


FIG. 3.3 – Gestion d'un distributeur d'argent, exemple de diagramme de classe

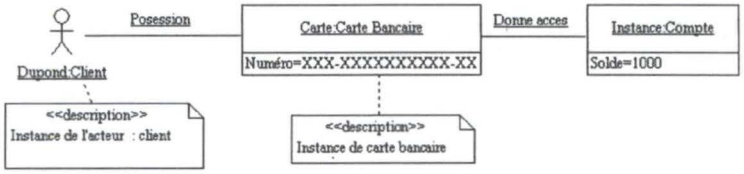


FIG. 3.4 – Gestion d'un distributeur d'argent, exemple de diagramme d'objet

Diagramme de composants et le Diagramme de déploiement

Le diagramme de composants représente les concepts de configuration logicielle. Il sert à représenter la manière dont vont s'agencer les composants, les unités physiques tels que les librairies, les fichiers source ou encore les exécutables. Les dépendances entre composants servent à modéliser des contraintes de compilation ou à mettre en évidence la réutilisation de composants.

Le diagramme de déploiement, quant à lui sert à représenter la disposition physique du matériel informatique, de composants matériels, etc. qui composent le système ainsi que la répartition des composants sur cette infrastructure.

3.2.4 Modélisation Dynamique

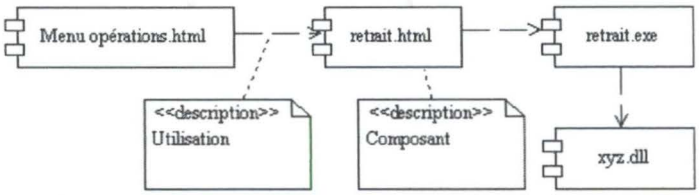


FIG. 3.5 – Gestion d'un distributeur d'argent, exemple de diagramme de composant

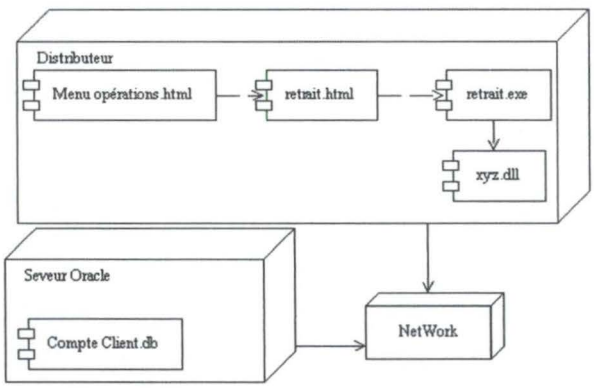


FIG. 3.6 – Gestion d'un distributeur d'argent, exemple de diagramme de déploiement

Diagramme d'état

Un diagramme d'état permet de montrer l'espace des états d'une classe. Un état est une situation, au cours de la vie d'un objet, qui se définit par un certain nombre de conditions à remplir. Ce diagramme modélise également les événements qui vont provoquer les transitions d'un état à l'autre ainsi que les actions qui résultent de cet événement. Il permet de représenter également la notion d'états imbriqués (Si les état B et C sont imbriqués dans l'état A (B et C étant les seuls états imbriqués), un système se trouvant dans l'état A est soit dans l'état B soit dans l'état C) ainsi que les états orthogonaux (Considérons l'état A contenant deux sous états orthogonaux, B et C, un système dans l'état A est dans l'état B et dans l'état C [Khr00].

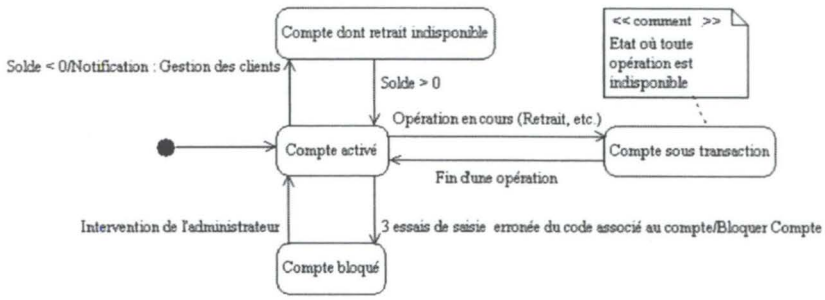


FIG. 3.7 – Gestion d'un distributeur d'argent, exemple de diagramme d'état

Diagramme d'activité

Le diagramme d'activité permet de modéliser des comportements. On l'utilisera notamment pour modéliser le déroulement d'opérations particulières au système ou à un plus haut niveau les flux d'informations propres au métier. Une activité est une séquence d'opérations dont la terminaison provoque la transition vers une nouvelle activité. Le diagramme d'activité peut également contenir des barres de synchronisation modélisant le déclenchement d'une activité que si toutes les activités attachées à la synchronisation sont terminées. Le diagramme d'activité peut-être décomposé en partitions pouvant par exemple correspondre à des unités organisationnelles. On peut représenter au sein de ces diagrammes des objets

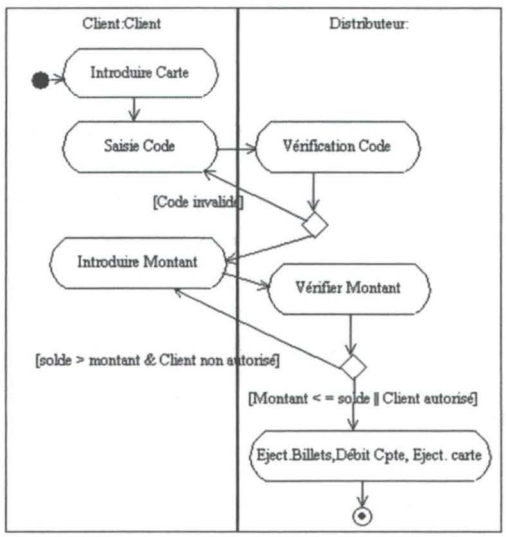


FIG. 3.8 – Gestion d'un distributeur d'argent, exemple de diagramme d'activités

en entrée ou en sortie des activités, ou encore les états d'objets qui découlent d'activité.

d'activités permet de modéliser la séquence d'actions et les conditions. Le diagramme d'état est plus focalisé sur l'évolution des entités suite aux événements.

Diagramme d'interaction

Les diagrammes d'interactions, c'est-à-dire les diagrammes de séquence et de collaboration modélisent une séquence d'interaction entre objets. Ces diagrammes sont très similaires, leur principale différence réside dans leur présentation. Le diagramme de séquence a un apport quant à la ligne de vie d'un objet et à la séquence de ses messages tandis que le diagramme de collaboration présentera plus facilement un grand nombre d'objets et de messages ceux-ci pouvant être positionnés librement. Ces diagrammes permettent de décrire un paquetage, un cas d'utilisation, une classe, une opération ou une instance.

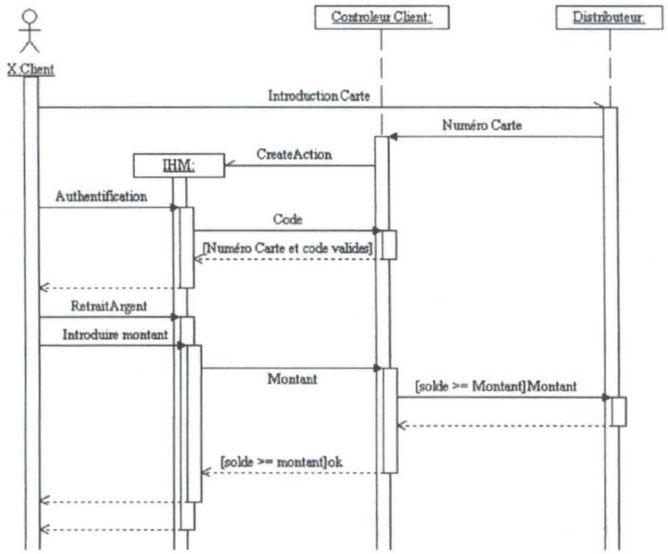


FIG. 3.9 – Gestion d'un distributeur d'argent, exemple de diagramme de séquence

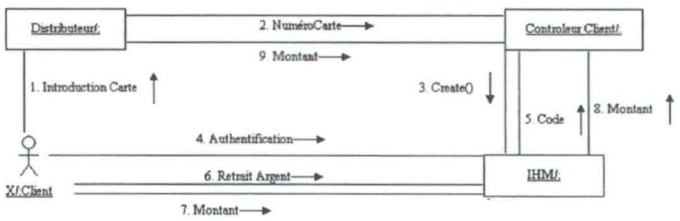


FIG. 3.10 – Gestion d'un distributeur d'argent, exemple de diagramme de collaboration

3.3 Le méta-modèle UML

UML peut être défini grâce à un méta-modèle, c'est-à-dire un modèle représentant les éléments UML, eux-mêmes modélisés grâce au langage UML. Ce méta-modèle décrit alors le formalisme UML, du moins le niveau syntaxique. Le niveau sémantique des éléments UML est, quant à lui, décrit sous forme textuelle. On a pour cela recours au diagramme de classe. Les modèles UML de l'utilisateur sont des instances de ce méta-modèle. Si nous cherchons à modéliser le fait que les classes possédent des attributs et des opérations, nous obtenons la figure 3.11. Ce sous méta-modèle contient des "méta-éléments". Le "méta-élément" "Class" est une classe dont les instances sont des "Class". Pour désigner, par exemple, le méta-élément "Attribute", on utilisera parfois l'appellation méta-Attribute.

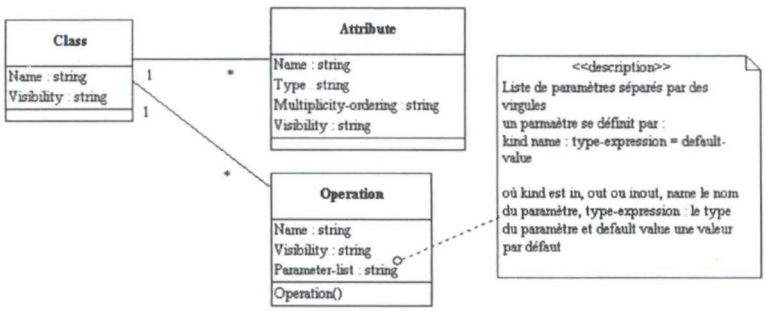


FIG. 3.11 – Exemple de méta-modèle exprimant qu'une classe peut posséder des attributs et des opérations

3.4 Les mécanismes d'extension d'UML

3.4.1 Les profils

Le développement logiciel ne peut se contenter d'une définition d'une notation graphique telle UML. "Le savoir-faire, c'est-à-dire les compétences et l'expérience des intervenants, les procédures et les connaissances initiales existant dans l'entreprise ou dans le projet, demeure la clé du succès (...)"[Sof99].

UML 1.3 a introduit la notion de profil. "Un profil UML est un ensemble cohérent d'extensions défini pour des sujets spécifiques"[Gro03]. Le profil étend le méta-modèle UML sans, toutefois altérer celui-ci [SW01]. Les extensions possibles sont les stéréotypes, les tagged values et les contraintes.

Le Profil UML permet en particulier de définir et maîtriser le processus de développement logiciel avec UML, ce qui représente un bénéfice considérable pour tous les développeurs" [Sof99].

Le profil UML constitue donc un support au processus de développement, il peut conduire

le type de projet de développement). Les profils servent à regrouper tous les mécanismes d'extension spécifiques à ces domaines. Par exemple on trouvera des profils sur le site de l'omg dédiés à des technologies telles CORBA ou les Entreprise Java Beans. Les profils peuvent hériter des extensions d'UML contenues dans d'autres profils. La notion de profil a été consolidée au sein de la version 1.4 d'UML.

Les stéréotypes

Les stéréotypes sont des méta-éléments dont la structure coïncident avec un méta-élément UML (constituant une généralisation de ce stéréotype). Par exemple, le méta-élément "Classe" peut posséder des stéréotypes "Objet métier", "Objet persistant" (ceux-ci restent des classes). Le stéréotype se distingue de ce méta-élément par l'usage qui en est fait. A un stéréotype peuvent être associées des contraintes, par exemple le nom d'un élément stéréotypé «X» ne devrait excéder 8 caractères. La définition d'un stéréotype peut également contenir des tags, c'est-à-dire des types de valeurs, les éléments stéréotypés pourront alors se voir affecter les valeurs associées, le point suivant décrira plus en détail ces *"Tag définition"*. La figure illustre un template pour définir un stéréotype. Des exemples de stéréotypes seront évoqués au sein du chapitre 5. Parent désigne un stéréotype parent. Exemple «internaute»

Stéréotype	Base Class	Parent	Description	Contrainte
<<Internaute>>	Actor	/	Acteur accédant au système (d'information ou général) par le biais d'un site web	/

FIG. 3.12 – Exemple de définition d'un stéréotype

peut être le parent d' «Internaute Identifié» et de «Internaute anonyme» Les contraintes peuvent être exprimées textuellement ou en Object Constraint Language (OCL). Définir un stéréotype correspond à étendre le méta-modèle.

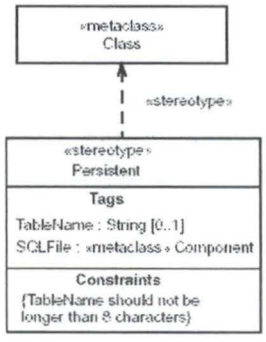


FIG. 3.13 – Extension du méta-modèle par la définition d'un stéréotype sur le méta-élément

Les *Tag définitions*

Les *tag définitions* servent à spécifier de nouvelles propriétés attachées aux éléments des modèles. Si on définit par exemple le stéréotype "manager" sur le méta-élément *classe*, on peut définir une tag définition sur ce stéréotype : isCompetent de type booléen. Une classe stéréotypée manager pourra alors avoir une *TaggedValue* : isCompetent=True, bien que dans le cas d'un booléen, la simple assignation de isCompetent à une classe aura la valeur vraie par défaut. On peut ainsi assigner une liste de propriétés à des méta-éléments. Ces tag définitions sont très proches de la notion de méta attribut et les tagged values proches de la notion de valeur de cet attribut.

Les outils Case se serviront par exemple d'une tag définition assignée au méta-élément "Opération" intitulée Nocode : boolean et l'assignation à une opération de la tagged Value Nocode aura pour effet dans un traitement ultérieur de ne pas générer de code pour cette opération.

Les contraintes (Well-formedness Rules)

Il est également possible de formuler des règles en vue d'exprimer des contraintes sur les éléments du Modèle. Par exemple, tous les éléments du modèles "soumis" à une généralisation doivent être du même stéréotype que le parent (parmi boundary, control et entity). La contrainte est exprimée en Object Constraint Language, un langage défini par l'omg dont un exemple issu de [Gro03] est présenté ci-dessous. *"OCL procure une notation flexible et concise pour exprimer des contraintes"* par opposition à une notation textuelle [RG98].

```
   Context Generalization inv:
(self.parent.stereotype->size>0) implies

   (if(self.parent.stereotype->name->includes("boundary")then(
(self.child.stereotype->name->includes("boundary")and
(self.child.stereotype->name->excludes("control")and
(self.child.stereotype->name->excludes("entity"))

   else (if
(self.parent.stereotype->name->includes("control")then(
(self.child.stereotype->name->includes("control")and
(self.child.stereotype->name->excludes("boundary")and
(self.child.stereotype->name->excludes("entity"))

   else (if
(self.parent.stereotype->name->includes("entity")then(
(self.child.stereotype->name->includes("entity")and
(self.child.stereotype->name->excludes("boundary")and
(self.child.stereotype->name->excludes("control")
```

3.4.2 Les profils vu par Softeam

Certains outils CASE vont plus loin dans la définition de profils, prenons l'exemple de Softeam proposant des règles supplémentaires associées aux éléments UML. Softeam a développé un langage complet, le langage J permettant d'offrir de véritables outils supplémentaires aux différents intervenants du projet. Les outils proposés au sein du dernier chapitre en constituent quelques exemples. Les outils développés au sein de profils peuvent servir notamment :

- A définir la trame du projet (le cycle de vie) au sein de l'outil CASE. On peut lier les produits (Modèles) aux phases.
- A valider des modèles par rapport à une méthodologie, à un standard autre qu'UML, aux règles de bonnes pratiques de l'entreprises, ou du point de vue de leur cohérence.
- A définir une méthodologie complète, un outil imposant le suivi de cette méthodologie. Il y aurait des interfaces de saisie d'informations appropriées à la phase courante du projet, etc.
- A transformer automatiquement des modèles. UML dans sa version 2.0 proposera des transformations automatiques de modèles au sein de profil(s) selon l'approche MDA proposée par l'omg.
- A partager les expériences, les modèles des autres développeurs. Exemple du Toolbox proposé au chapitre 5 (phase de conception)
- ...

Les profils UML sont le garant de la qualité des développements d'une entreprise.

Le langage J dédié Métamodèle et profil

Le langage J est un langage java-Like proposé par Softeam au sein de son outil Objecteering profile Builder en vue de réaliser des profils UML 1.4, c'est-à-dire de créer des outils, des stéréotypes ou des règles capables de manipuler, vérifier, transformer ou encore de proposer des outils supplémentaires au développeur. Le développeur quant à lui modélise et accède à ces outils et crée ses modèles via le logiciel Objeecteering Modeler. Ce langage est intéressant de par son type, en effet, en vue de pouvoir produire les profils tels que définis par l'omg, il est nécessaire d'avoir un langage permettant de parcourir le méta-modèle UML, à la manière d'Object Constraint Language. Mais le langage J va plus loin en permettant non seulement de parcourir ce méta-modèle, mais en permettant d'en modifier ses instances (==> Transformations automatiques au niveau des modèles). Le méta-Modèle quant à lui ne peut être transformé, il peut simplement être étendu ou spécialisé, ceci en vue de respecter le standard UML.

C'est grâce à ce langage que les outils présentés dans le dernier chapitre de ce mémoire ont été développés et proposés au CITI.

- **Le Méta-Modèle Objecteering**
  Softeam a recréé le méta-Modèle UML, celui-ci modélise l'ensemble des éléments UML (dans sa version 1.4) et des relations entre eux au sein d'un diagramme de classe.

et toute une panoplie d'objets graphiques). La présentation d'un tel méta-modèle est faite par une multitude de sous-diagrammes se complétant. Certains font abstraction de détails tandis que d'autres s'avèrent plus exhaustifs.

- **Exemples de code J**

Exemple 1

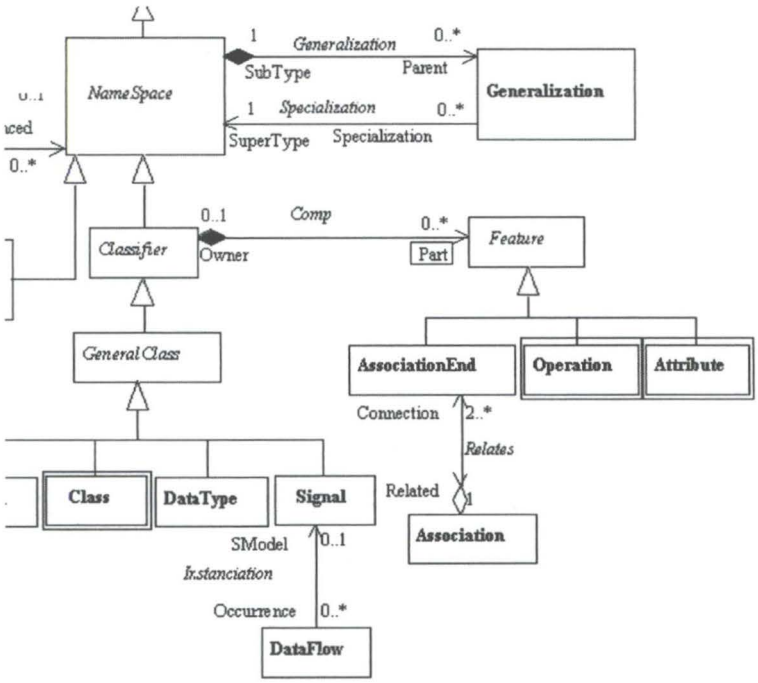


FIG. 3.14 – Sous ensemble du méta-modèle concernant l'exemple 1 et 2

`Class:printAttribute()`
Méthode définie sur la méta-classe "classe".
`{`
`String line;`
`PartAttribute`
Nous sommes au niveau d'une classe (voir figure), par le lien "Part" liant une "Class" à ses "Attribute" (d'ou le PartAttribute), nous naviguons sur le métamodèle et obtenons un ensemble d'Attributes.
`{`
Cette accolade signifie une boucle sur l'ensemble des attributs obtenus.L'objet courant est maintenant 1 Attribut
`String buffer = "," + Name; line = line + buffer`
Name désigne en fait this.Name , this étant l'objet courant (un Attribute) et Name un attribut de la méta-classe Attribute.
`}`

Exemple 2

Cette méthode portant toujours sur l'objet classe ajoute à la classe courante une opération print()

```
Class:addPrint()
{
Operation M ;
sessionBegin("addPrintOperationExample", true) ;
```

Ouvre une session destinée à vérifier la conformité par rapport au standard UML au terme de la session.

```
M = Operation.new ();
M.setName ("print") ;
```

Set : ajoute une propriété et Name est une propriété des opérations.

```
this.appendPart(M) ;
```

Append : sert à ajouter une instance d'un lien , part étant le lien entre les classes et les opérations.

```
sessionEnd ();
```

Vérification de l'intégrité faite au terme de la session.

```
}
```

D'autres exemples se trouvent en annexe. Ce langage permet également une gestion d'interface avec l'utilisateur, une gestion d'entrée sortie, etc.

3.5 conclusion

UML couvre de nombreux aspects du développement, on peut facilement représenter les aspects fonctionnels d'un système d'informations, décrire le domaine de façon statique ou encore dynamique. On pourra encore facilement modéliser l'architecture, les tests, les aspects de maintenance, etc.

Malgré la documentation réalisée par l'omg sur ce langage définissant de manière détaillée les éléments d'UML et dont un bref aperçu est présenté ici. La sémantique ou du moins *"l'interprétation qui en est faite reste parfois ambiguë"* [EBF+98], les utilisations qui peuvent être faites d'un diagramme sont nombreuses. Même si ce langage peut sembler plus intuitif que d'autres langages de description, il reste important de personnaliser ce langage que ce soit par l'extraction d'un sous ensemble de ses éléments, la définition d'une méthodologie adaptée, la proposition de recommandations ou même par certaines spécialisations de ses éléments rendues possibles par les "stéréotypes", "Tag Définition" et "contraintes".

Chapitre 4

Définition d'une méthodologie basée sur UML

4.1 Introduction

Ce chapitre reprend le cycle de vie réalisé dans le chapitre 2 en y développant les recommandations quant aux éléments de spécifications UML pouvant correspondre au produit attendu de chaque phase, dans l'hypothèse où l'on effectue le choix d'UML comme langage de modélisation. Cette méthodologie a été présentée au cours de réunions de groupe de travail déjà citées précédemment.

Les chapitres 4 et 5 ont plusieurs objectifs :

- Montrer l'apport d'UML en terme de modélisation notamment l'aspect graphique par rapport à d'autres langages de description.
- Homogénéiser la méthode de développement des projets réalisés en interne au CITI
- Montrer l'intérêt de pouvoir personnaliser un outil CASE de façon à refléter une méthode de développement propre à une entreprise (Objet du chapitre 5)

4.2 Pourquoi UML ?

L'optique de mes travaux au CITI a été de présenter ce que donneraient des spécifications UML d'un développement plutôt que de fixer UML comme seul langage possible. Le choix d'un langage pour décrire, par exemple, les architectures est une controverse courante. Il est en effet difficile de dire que tel langage est mieux qu'un autre car bien souvent ils sont complémentaires.

En quoi UML est-il adapté comme langage de modélisation pour les projets de développement en interne du CITI ? [Déc02]

- Les projets développés au CITI sont souvent de l'informatique de gestion par opposition à une informatique industrielle. Dans des projets d'informatique de gestion, le client n'est pas nécessairement censé connaître le langage de spécification adopté. Dès lors , UML, se voulant être un langage semi-formel et lisible par le client, semble plus indiqué. Ceci permettra des validations par le client en *"cours de route"*.
- UML est un langage pouvant servir à modéliser les besoins en partant de l'utilisateur, les cas d'utilisation sont souvent décrits en partant de l'acteur, de ce qu'il est prêt à fournir comme information. Ce point est crucial si les besoins quant au logiciel à

4.3 Choix méthodologiques

La méthodologie proposée se trouve à mi-chemin entre UP (Unified Process) et une approche minimaliste, simplificatrice de la méthodologie (à la mode actuellement comme, par exemple, les méthodologies Agile). La méthodologie proposée ne s'attarde toutefois qu'à la description des activités et livrables produits aux termes des phases plutôt qu'à l'ordonnancement des activités, ou des recommandations sur la gestion de risques, de projet, etc. Ceci n'étant pas l'objectif parcouru. La méthodologie servira comme cadre à l'adaptation de l'outil CASE faite au chapitre 5. Cette méthodologie est caractérisée par :

- Son caractère non exhaustif
- La position centrale des Usecases

4.3.1 Une méthodologie non exhaustive

L'objectif d'une méthodologie du développement n'est pas de réaliser une pléthore de documents ou d'anticiper les besoins futurs du client, etc. mais de développer facilement avec une méthode simple et efficace encourageant une documentation succincte mais facilement tenable à jour.

L'idée de cette méthode est également de retenir les éléments UML les plus pertinents pour aider à la construction des produits suivants dans le cycle de vie (Architecture, implémentation) et d'établir correctement les liens entre les spécifications de chacune de ces phases. Le passage de documents d'analyse modélisé en UML vers des documents de conception ou le code lui-même doit être souvent éclairci. La maintenance et la traçabilité en sont d'autant facilitées. Ce qui dans un contexte de développement de projet d'innovation tel qu'au CITI où les besoins sont jugés instables, est un critère fondamental pour la méthodologie. La personnalisation de l'outil faite au chapitre 5 aurait pu aller jusqu'à l'élaboration d'un mécanisme de traçabilité. La méthode citée ci-après n'a toutefois pas la simplicité d'une méthodologie telle XP mais bien par opposition à des méthodes traditionnelles.

4.3.2 Une méthodologie guidée par les Usecase

Il existe de nombreuses définitions possibles du usecase, on trouvera dans [RV00] :
"Un cas d'utilisation (usecase) représente un ensemble de séquences d'actions réalisées par le système et produisant un résultat observable, intéressant pour un acteur particulier. Un cas d'utilisation modélise un service rendu par le système. Il exprime les interactions entre les acteurs et le système et apporte une valeur ajoutée, notable, à l'acteur concerné"

Le usecase conduit le développement, la description des exigences fonctionnelles se fait par les usecase. Les usecase évoluent au fur et à mesure du développement (figure 4.1).

4.4 Modélisation de www.jebouquine.com

En vue d'illustrer l'utilisation d'UML proposée au CITI, je reprends un exemple issu

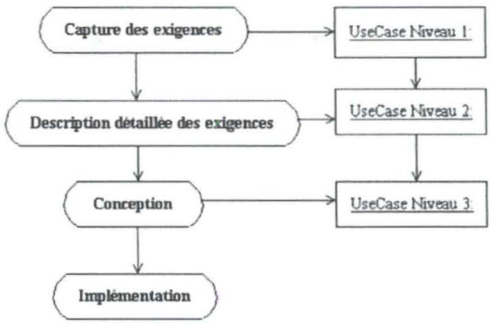


FIG. 4.1 – Evolution du usecase dans un cycle de vie linéaire

par l'expérience des employés du CITI. La méthodologie se devait en effet de correspondre plus ou moins à ce qui se faisait déjà actuellement. L'objectif est ici d'illustrer la méthodologie en modélisant l'exemple de www.jebouquine.com L'élicitation des exigences quant à l'outil, bien que sous-jacente au sein de ce chapitre sera décrite de manière plus détaillée au sein du chapitre 5. Lorsqu'on parlera d'exigence au sein de ce chapitre ou de description de domaine, elles sont relatives au cas de www.jebouquine.com et non à l'outil développé au chapitre 5, ce chapitre a pour objet de décrire la méthodologie et les recommandations faites (quels éléments uml utiliser).

4.4.1 A. Capture des exigences

A. 1 Interviews

Le savoir-faire de l'analyste dans sa relation au client est ici mis à contribution. Supposons qu'au terme de cette succession d'interviews, l'analyste ait rédigé textuellement un résumé des exigences quant au site web d'e-commerce.

Exigences fonctionnelles

Recherche L'internaute doit pouvoir trouver le plus rapidement possible un ouvrage au sein du catalogue. Les méthodes de recherches sont scindées en 2, une recherche par titre et une recherche par thème. Le résultat de cette recherche doit apparaître sur une page particulière.

Découverte Chaque livre contient une description détaillée, ceci permet à l'internaute de pouvoir au terme de sa recherche, faire son choix parmi les livres qui lui sont proposés via une description plus exhaustive. Outre les informations générales quant au livre (Auteur, titre, ISBN, éditeur, etc.), la description d'un livre contient une image, le prix, des commentaires de lecteurs et la table des matières.

Commande

A partir de son panier, le client peut accéder au formulaire de bon de commande, dans lequel il saisit ses coordonnées et les informations nécessaires au paiement et à la livraison.

Exigences non fonctionnelles

- Pour des raisons de performances, le panier de l'internaute a une durée de vie limitée à son temps de visite.
- Suite à une étude de marché , il est prévu qu'il puisse y avoir jusqu'à 1 000 connections simultanées sur le site et environ 1 000 000 d'ouvrages dans le catalogue

A. 2 Formalisation des exigences

A partir du produit des interviews, c'est-à-dire des notes explicitant les exigences fonctionnelles et non fonctionnelles, de manière non structurée, l'analyste va essayer de formaliser celles-ci.

L'objectif pour cette phase est d'obtenir un document décrivant les limites du système à développer. Par système, j'entends le système d'information, des composants matériels, mais aussi des rôles de personnes, il s'agit donc du système global à mettre en place en vue de pouvoir répondre aux exigences de l'utilisateur. Le document doit décrire ce que le système doit pouvoir faire pour l'utilisateur et ce qu'il ne doit pas faire. Il s'agit d'une approche "blackbox" sur le système, c'est-à-dire qu'on reste indépendant par rapport à une solution à mettre en place. On décrit dès lors les exigences fonctionnelles que par les interactions qu'elles génèrent entre l'utilisateur et le système (Diagramme de séquence). Ce système ne sera ouvert que dans les étapes ultérieures.

La première activité consiste en la description des acteurs interagissant avec le système global, celle-ci s'effectue au CITI par le remplissage d'un petit modèle de description reprenant le nom de l'acteur, une brève description, les raisons pour lesquelles on a besoin de cet acteur et en quoi l'acteur est intéressé par le système.

L'analyste devra également décrire le domaine du métier, l'élément UML concerné est le diagramme de classe toutefois les aspects dynamiques pourront être couverts par le diagramme d'activités ou états-transition. A ce niveau de spécifications, toutefois on ne devrait faire référence qu'à des objets métiers, c'est-à-dire des objets relatifs à des concepts connus de l'utilisateur, qu'il manipule dans son métier. Dès lors, chaque classe représentera un "objet métier". Ces objets métiers, à ce niveau de spécifications, ne devraient contenir ni d'attributs ni d'opérations.

Enfin, l'analyste devra décrire les exigences fonctionnelles et non fonctionnelles, l'élément UML concerné est le diagramme de usecase. Chaque usecase représente un objectif que doit accomplir le système. Chacun des usecase de ce diagramme doit être décrit grâce à un diagramme de séquence, comme dit précédemment , ce diagramme ne comprend que deux instances, il s'agit de l'acteur (interessé par l'exigence) et le système global. Il est évident qu'un diagramme de séquence peut parfois s'avérer prématuré à ce niveau de spécification, on peut dès lors avoir recours à une description textuelle plus succincte (moins formelle)

Champ	Description
USE CASE	Nom du usecase
Goal in Context	Objectif plus précis parcouru par le usecase
Scope et Level	On définit la portée (ex : la compagnie) et le niveau (ex :Fonction, sous fonction)
Preconditions	L'"état du monde" pouvant déclencher le usecase
Success End Condition	Postcondition : L'"état du monde" après la terminaison correcte du usecase
Failed End Condition	L'"Etat du monde" si une condition lors du déroulement n'est pas remplie
Primary, Sec. Actors	Liste des acteurs responsables du usecase
Trigger	L'action qui va déclencher le usecase
DESCRIPTION	
Step, action	1 <Description des étapes du usecase> 2 <...>
EXTENSIONS	
Branching Action	Numéro de l'étape <Condition> : <action>
SUB-VARIATIONS	
Branching Action	Numéro de l'étape <Liste des variations proposées >
Priority	Définition de la priorité du usecase
Performance	Le temps idéal d'execution du usecase
Frequency	La fréquence d'occurences du usecase
Channels to actors	Nature du lien avec l'acteur (interactif)
Open Issues	Liste des problèmes
Due Date	Date de livraison
Subordinates	Liens vers les sous use cases.

FIG. 4.2 – Modèle de description d'Alistair Cockburn d'un usecase

Les exigences non fonctionnelles peuvent être modélisées en UML via l'élément "Note" , celui-ci peut contenir une description structurée de l'exigence. Les exigences non-fonctionnelles peuvent être locales à un usecase ou être globales au système. Dans l'exemple ici, les exigences non fonctionnelles se rattachent au usecase de passage de commande pour ce qui est de la gestion panier (pour la durée de vie de celui-ci), la contrainte quant au nombre de livres et de connections, quant à elle, est globale. Les exigences non fonctionnelles interviendront surtout dans les choix technologiques en phase de conception.

La dynamique entre usecases peut être modélisée par un diagramme d'activités. Concrètement, on retrouvera la succession d'activités suivantes :

- Description des acteurs
- Description du domaine par un diagramme de classe (ou bien diagramme d'activités, états pour modéliser la dynamique)
- Modéliser sous forme d'un diagramme de usecases par acteur, les exigences fonction-

- Associer aux usecases concernés les exigences non fonctionnelles sous forme de note, ou les associer au Package reprenant l'ensemble des usecases si il s'agit d'exigences non fonctionnelles globales.
- Diagramme d'activités entre usecase.

Description des acteurs

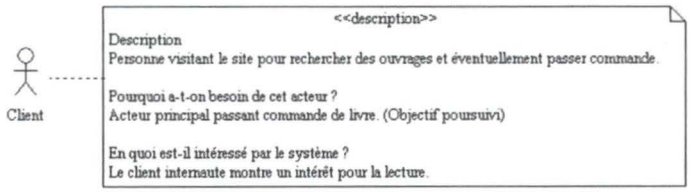


FIG. 4.3 – Description de l'acteur Client

Description du domaine

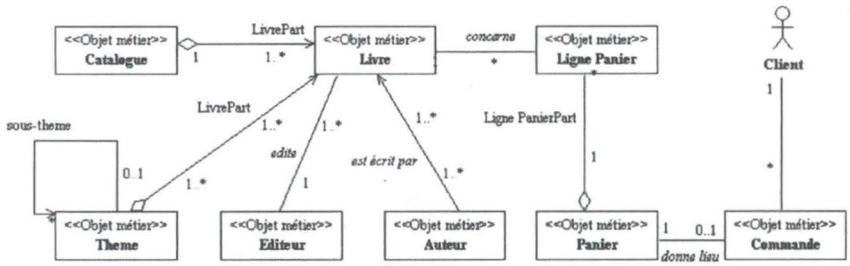
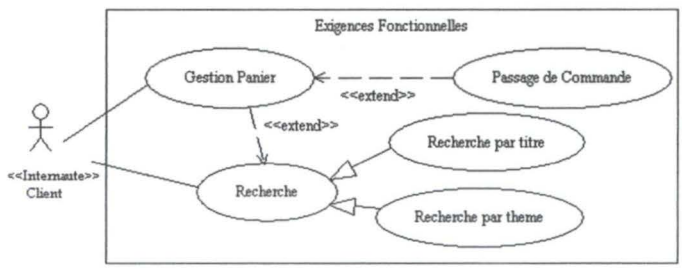


FIG. 4.4 – Description statique du domaine

Description des exigences fonctionnelles



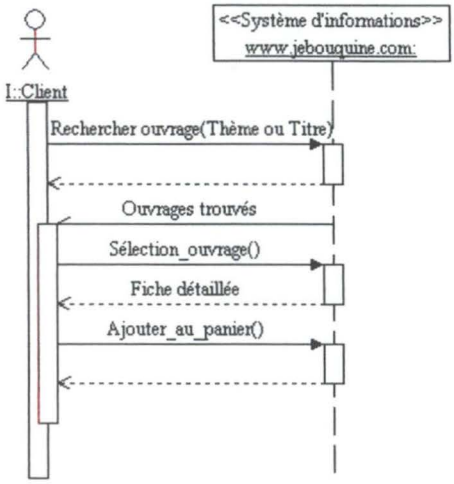


FIG. 4.6 – Description du usecase recherche d'ouvrage

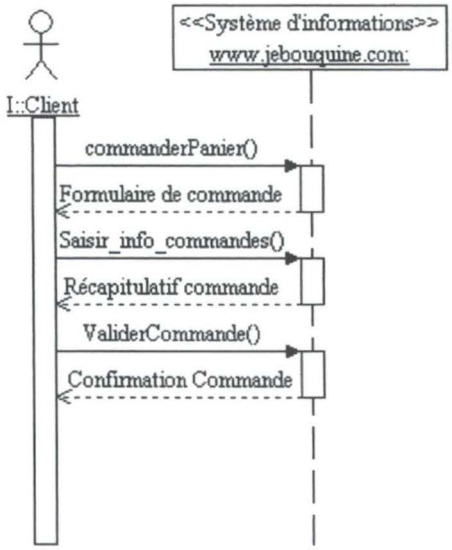


FIG. 4.7 – Description du usecase Passage de commande

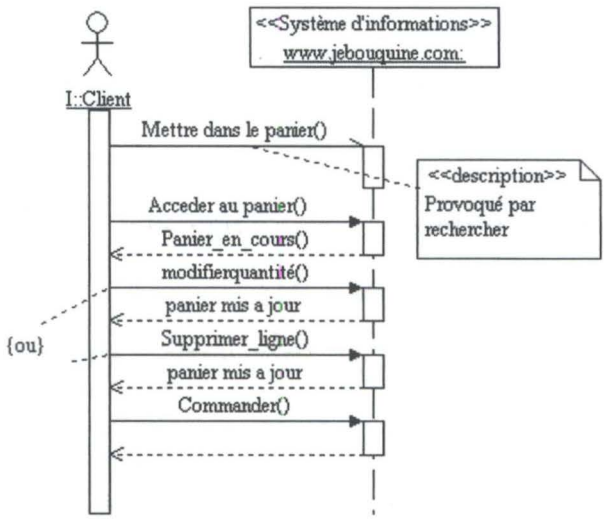


FIG. 4.8 – Description du usecase Gestion panier

Champ	Description
USE CASE	Gestion Panier
Goal in Context	Lorsque le client est intéressé par un ouvrage, il doit avoir la possibilité de l'enregistrer dans un panier virtuel, puis d'ajouter d'autres livres, en supprimer ou bien encore d'en modifier les quantités
Preconditions	Néant
Success End Condition	Néant
Primary, Secondary Actors	Client
DESCRIPTION	
Step, action	1 <Suite à une recherche, le client peut sélectionner un résultat pour le placer au sein du panier> 2 <Le système lui affiche le panier en cours> 3 <Le client peut modifier la quantité commandée d'un livre, ou supprimer totalement la commande d'un livre particulier> 4 <Le système lui affiche le panier mis à jour>
EXTENSIONS	
Branching Action	2,3,4 <Panier non vide> : <Le client fait part de son désir de passer la commande> 4 <Erreur de saisie de quantité : <Le système présente la panier dans dernier état valide>

FIG. 4.9 – Description de gestion panier sous forme textuelle

4.4.2 B. Description détaillée des exigences

B. 1 Description détaillée des exigences

La description détaillée des exigences a pour but d'ouvrir la boite noire faite au premier niveau. On va désormais spécifier les usecases en décrivant les interactions entre l'utilisateur, les composants, les autres acteurs et le système d'information. Ceux-ci collaborent, dans ce second niveau de description de solution, pour répondre aux besoins de l'utilisateur. Il se peut donc que de nouveaux acteurs apparaissent au sein du système global. A ce niveau, on obtient une description du fonctionnement et de l'organisation de la société cliente, du business suite à la mise en place du SI.

La description du domaine peut également s'étoffer, on décrira notamment les attributs de chacune des classes. Si une des classes du domaine est fortement modifiée par une exigence, on peut modéliser le comportement de cette classe par un diagramme d'état. Il se peut également que certains usecases choisis et décrits lors du premier niveau de spécifications soient entièrement résolus sans interactions avec le système d'informations. Également, de nouveaux usecases peuvent apparaître. Certains nouveaux acteurs peuvent apparaître.

Le chaînon manquant entre cette description de classes et la description faite des exi-

Concrètement, on retrouvera la succession d'activités suivante :
- Description du domaine par un diagramme de classe avec attributs (pas d'opérations) + éventuellement diagramme d'activités, état pour des mécanismes particuliers
- Description affinée des usecases par un diagramme de séquence.
- Ajout de nouveaux acteurs
- Diagramme d'état pour les classes changeant de statut.

Description du domaine

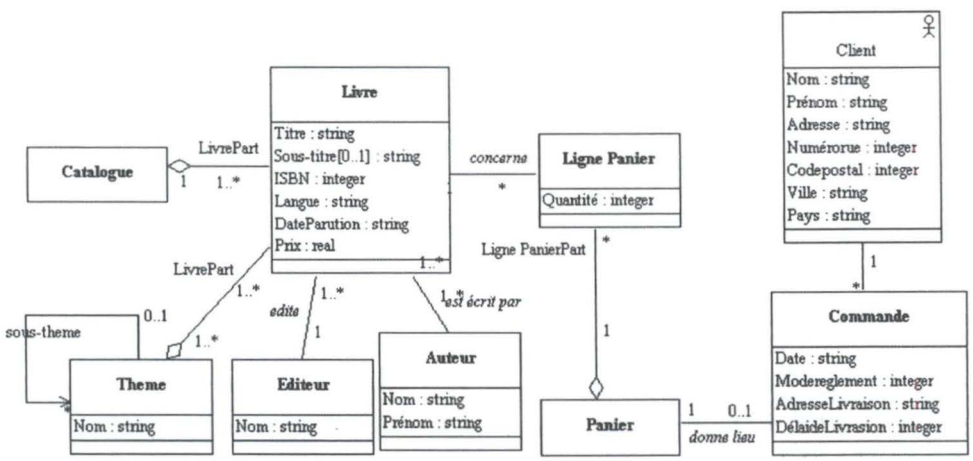
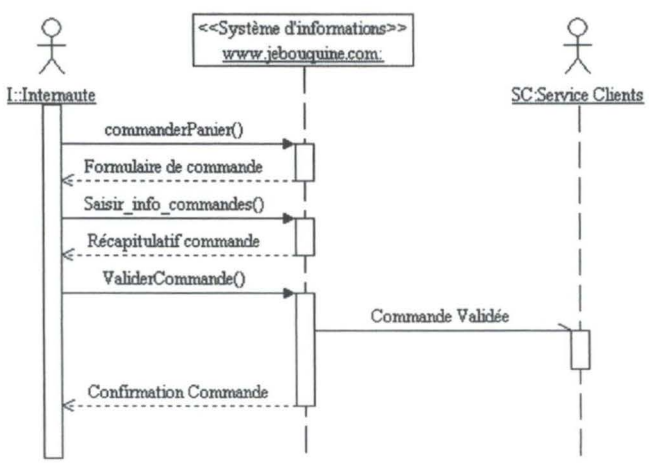


FIG. 4.10 – Description du domaine

Description des exigences fonctionnelles



Description de l'acteur Service client

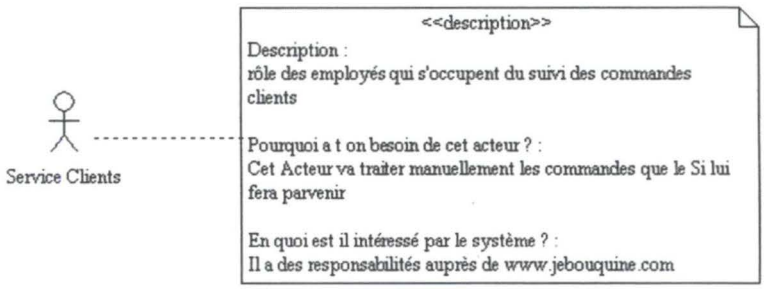


FIG. 4.12 – Description de l'acteur Service client

B. 2 Développement de la maquette

Il a été proposé ici deux modèles complémentaires de représentation des Interfaces Hommes Machines en UML au CITI. Ces modèles sont liés aux possibilités de l'outillage CASE. Il est en effet possible à partir de modèles suffisamment complets de générer du code pour les IHMs. Ceci nécessite une personnalisation de l'outil, nous y reviendrons au chapitre 5. Les IHMs ainsi produites (peu conviviales) serviraient de base à l'élaboration d'un prototype des IHMs plus complets.

Le premier modèle a pour but de représenter la navigation entre les différentes interfaces de l'application (Diagramme d'activités), le second quant à lui a pour objectif de représenter le contenu informationnel de chaque fenêtre, si les informations sont des entrées ou des sorties, leur type, etc[1].

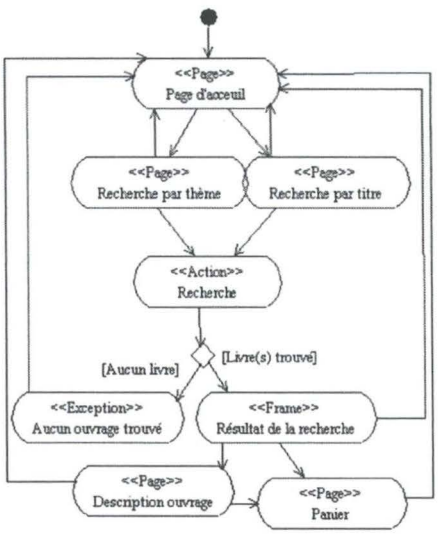


FIG. 4.13 – Modèle de représentation de la navigation entre IHMs

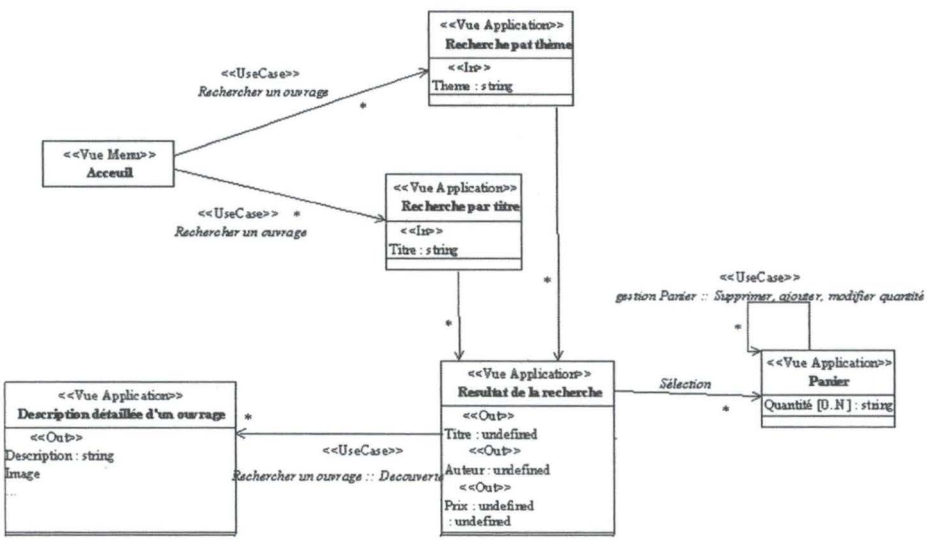


FIG. 4.14 – Modèle de représentation des informations contenues au sein des IHMs

4.4.3 C. Conception

La méthodologie proposée quant à la phase de conception présente quelques similarités avec la méthodologie Model Driven Architecture proposée par l'omg[1].La phase de conception a pour objet de partir d'un modèle logique de l'architecture (Le Platform Independant Model) vers un modèle physique (Platform Specific Model) par une série de transformations successives. La scission établie entre ces modèles porte l'avantage non négligeable du point de vue de la maintenance et de l'évolutivité de l'application par l'indépendance que le modèle logique offre par rapport au coté éphémère des technologies, spécificités des plateformes.

Architecture logique

La modélisation de l'architecture logique se décompose en 2 activités, la première activité consiste à développer la spécification des exigences fonctionnelles en ouvrant la partie système d'informations et en décrivant les divers composants et les relations entre ceux-ci.

Le diagramme de classe ou le diagramme de séquence s'avérent être appropriés (ou encore le diagramme de collaboration). Les objets du domaine décrits précédemment se voient collaborer avec les autres objets (Acteurs, composants, Contrôleurs, Interfaces, etc.) en vue de satisfaire l'objectif de l'utilisateur. Ceci dans la continuité des phases précédentes affinant la spécification des exigences, en effet si l'on reprend la description d'une exigence au cours des phases de développements. on a :

- Capture des exigences : description par les interactions entre l'acteur et le système global.
- Spécifications détaillées des exigences : Description par les relations Acteurs, d'autres acteurs, composants du système global, et système d'informations.
- Conception : description des relations entre les éléments précédents ainsi que les relations entre composants du SI pour réaliser l'exigence. (Diagramme de classe, séquence ou de collaboration)

Une découpe proposée par Ivar Jacobson ([Jac92]) scinde les objets collaborants en trois types que sont les *boundary class* (un tel objet a pour rôle une communication avec l'utilisateur, une conversion de protocole), les *entity class* (modélise les informations manipulées par les usecase, idéalement un entity correspond à un objet du domaine) et les *control class* (modélise le comportement spécifique à un usecase, la logique métier). On retrouve au sein de la méthodologie présentée dans ce chapitre, le même type de découpe avec les stéréotypes "interface" (boundary class), "Control" (control class) et Entity (entity class) auxquels vient s'ajouter le stéréotype "Objet persistant" désignant les informations persistantes indirectement manipulées par les usecases. Outre ces objets, on retrouvera également, les acteurs ainsi que d'éventuels autres composants externes au système d'information mais participant au usecase. Il est intéressant que les objets collaborant contiennent des propriétés ou des opérations (uniquement diagramme de classe) dans leur description. Les propriétés d'un composant correspondant, par exemple, à une *interface* seront les informations qu'elle contient que ce soit des saisies ou des productions d'informations, les opérations d'un contrôleur consisteront aux appels de méthodes qu'il assume. . Sur l'exemple construit, on observe que l'on retrouve les éléments réalisés lors du prototypage des IHMs et de la description

Les relations entre ces objets collaborants sont issues des interactions des diagrammes de séquence réalisés lors des étapes de spécifications antérieures. Une relation entre un acteur et le système d'information décrite en phase de "spécifications détaillées des exigences" se voit éclatée parmi les relations entre les objets collaborants. Si on décide de représenter cette collaboration par un diagramme de classe, ces relations entre composants sont décrites par les opérations des classes et l'association vis à vis d'autres classes (Voir exemple de www.jebouquine.com).

La seconde activité pour modéliser l'architecture logique a pour but de créer une vue plus globale. Comme au sein de chaque exigence, on va retrouver des "objets collaborants" communs, il est intéressant d'avoir une vue globale reprenant l'ensemble des composants et les relations entre eux. Le modèle ainsi produit consiste en un modèle de description de la collaboration entre composants répondant aux diverses exigences fonctionnelles, modèle qui se veut indépendant de toute technologie, choix technique, modèle dit indépendant de la plate-forme ou *architecture logique*. Le diagramme utilisé est alors le diagramme de classe. La méthode employée consiste à regrouper ces composants au sein de "couches" (on peut alors regrouper selon la même scission proposée précédemment : Entity, Control, Boundary).

Spécifications détaillées de la recherche d'un ouvrage

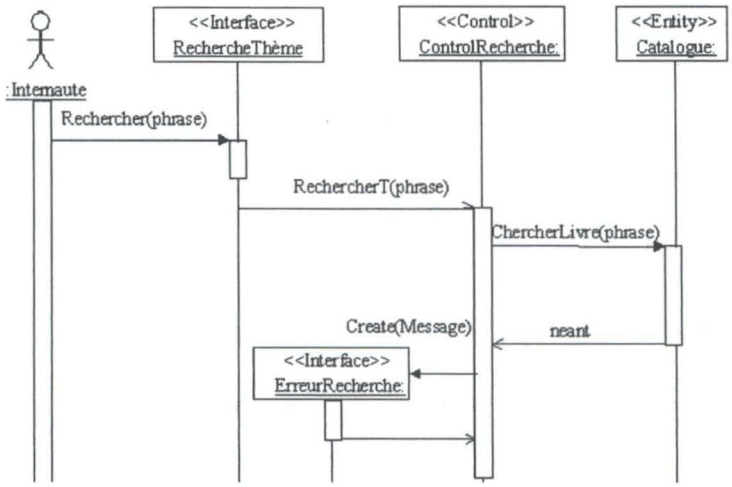


FIG. 4.15 – Description détaillée de "rechercher ouvrage", cas d'une recherche non fructueuse Recherche par thème, Erreur Recherche étant des "interfaces", Control Recherche, un 'Control", P-Catalogue, un "Entity" et Catalogue, un objet persistant.

Le diagramme de séquence a l'avantage de mieux intégrer la dimension temporelle, durée de vie d'un objet et montre les interactions découlant d'un scénario particulier mais ne décrit pas complètement les objets collaborants. Le diagramme de classe tel qu'il est utilisé ci montre les couples attributs-types décrivant les objets collaborants (les types d'objets plutôt que les *instances* présentées au sein du diagramme de séquence ou de collaboration) mais ne décrit pas quelles interactions appartiennent à tel scénario. Les deux diagrammes

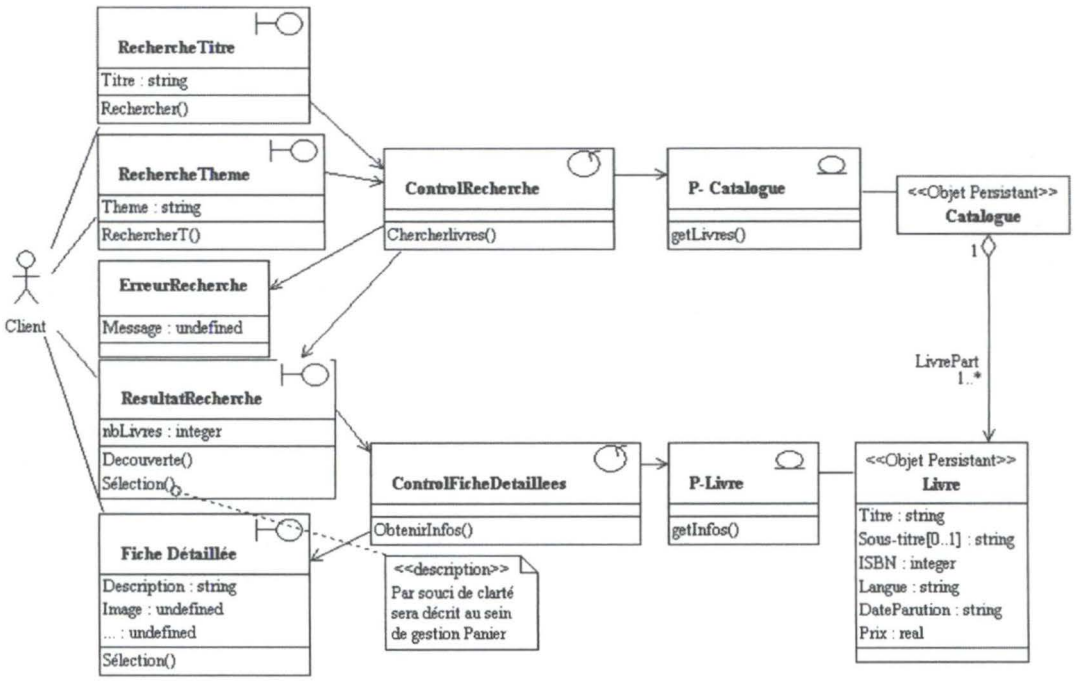


FIG. 4.16 – Description détaillée de "Rechercher ouvrage"

Les paramètres des opérations doivent être également décrits, toutefois ils ne sont pas représentés au sein du diagramme sous l'outil utilisé. A partir de "resulatRecherche", on peut "sélectionner" un livre et l'ajouter au panier. La sélection est faite à partir du usecase "Rechercher un ouvrage".

Spécifications détaillées de Gestion Panier

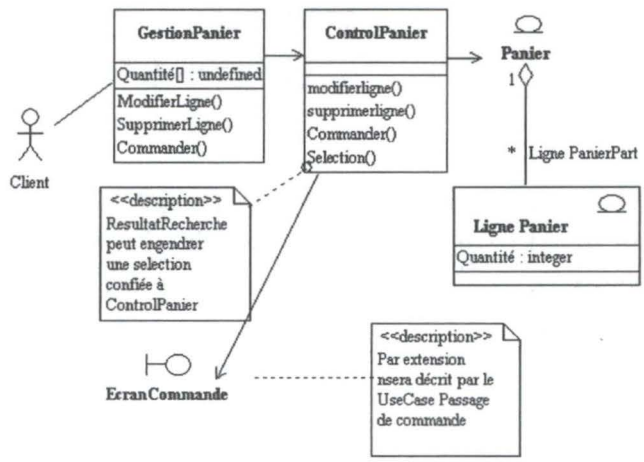


FIG. 4.17 – Description détaillée de "GestionPanier"

Spécifications détaillées de Passage de commande

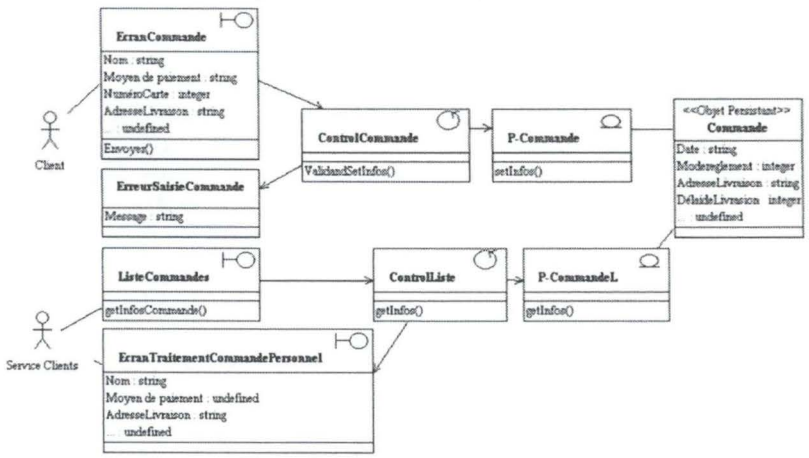
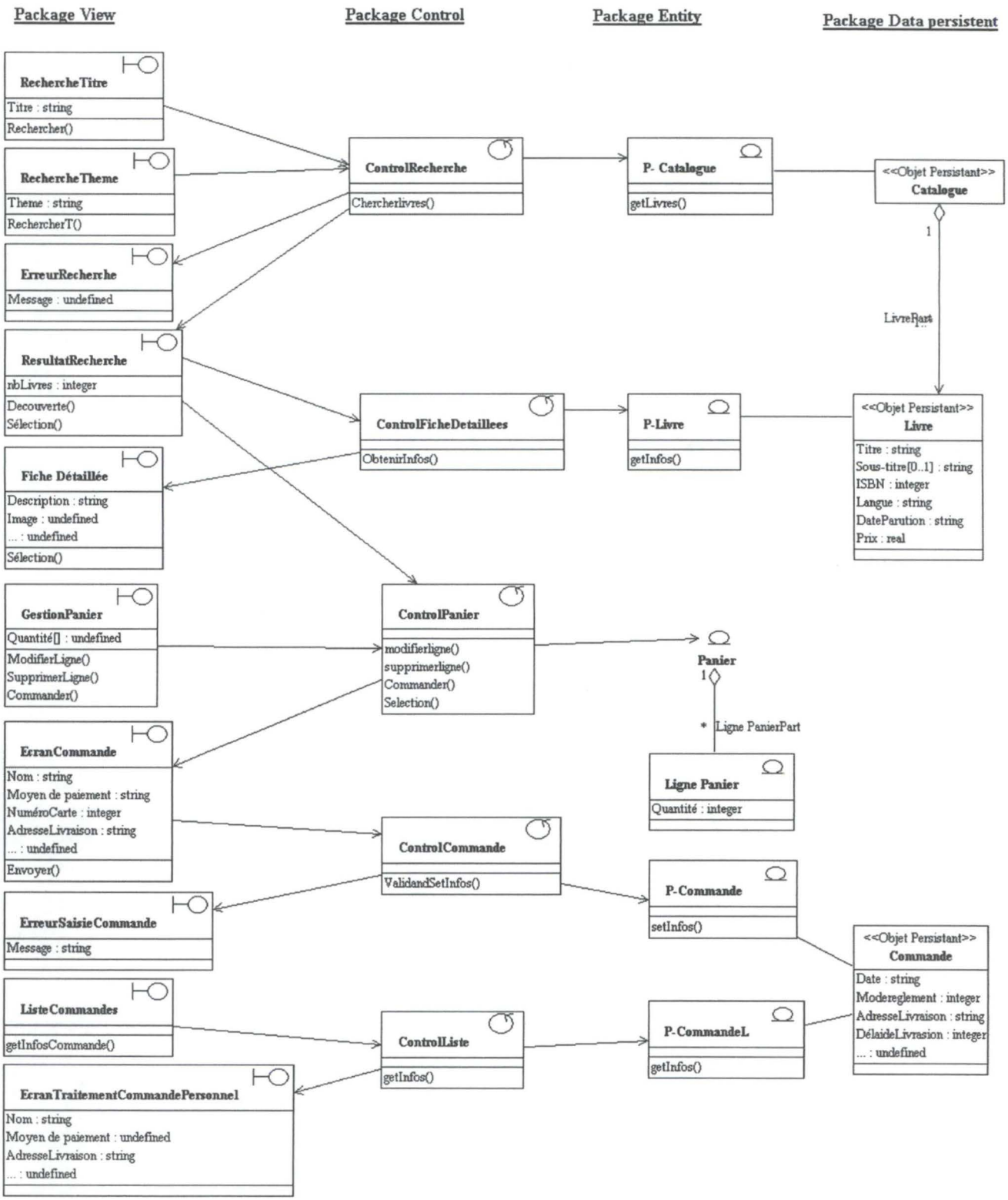


FIG. 4.18 – Description détaillée de "Passage de commande"


Package View
Package Control
Package Entity
Package Data persistent
RechercheTitre
Titre : string
Rechercher()
RechercheTheme
Theme : string
RechercherT()
ErreurRecherche
Message : undefined
ResultatRecherche
nbLivres : integer
Decouverte()
Sélection()
Fiche Détaillée
Description : string
Image : undefined
... : undefined
Sélection()
GestionPanier
Quantité[] : undefined
ModifierLigne()
SupprimerLigne()
Commander()
EcranCommande
Nom : string
Moyen de paiement : string
NuméroCarte : integer
AdresseLivraison : string
... : undefined
Envoyer()
ErreurSaisieCommande
Message : string
ListeCommandes
getInfosCommande()
EcranTraitementCommandePersonnel
Nom : string
Moyen de paiement : undefined
AdresseLivraison : string
... : undefined
ControlRecherche
ChercherLivres()
ControlFicheDetaillees
ObtenirInfos()
ControlPanier
modifierligne()
supprimerligne()
Commander()
Selection()
ControlCommande
ValidandSetInfos()
ControlListe
getInfos()
P- Catalogue
getLivres()
P-Livre
getInfos()
Panier
1
* Ligne PanierPart
Ligne Panier
Quantité : integer
P-Commande
setInfos()
P-CommandeL
getInfos()
<<Objet Persistant>>
Catalogue
1
LivreRart
<<Objet Persistant>>
Livre
Titre : string
Sous-titre[0..1] : string
ISBN : integer
Langue : string
DateParution : string
Prix : real
<<Objet Persistant>>
Commande
Date : string
Modereglement : integer
AdresseLivraison : string
DélaideLivrasion : integer
... : undefined

Architecture physique

Il est évident que l'étape suivante consiste à choisir les technologies d'implémentation, la plate-forme (support réseau, persistance des informations, etc.) en vue de pouvoir procéder à l'implémentation proprement dite des composants.

Le passage de ce modèle indépendant à un modèle spécifique à la plate-forme (architecture physique) se fait par itérations successives. Le savoir-faire de l'architecte informatique ainsi que les exigences non fonctionnelles élicitées auparavant conduiront l'architecte au choix de transformation par exemple : Un objet "IHM" pourra devenir un "Script" puis une 'Page JSP", les contrôleurs pourront éventuellement se transformer en Javabeans, les objets persistants décrits précédemment peuvent devenir une "source de donnée" (le choix d'une base de données particulière n'ayant pas encore été faite). L'architecte informatique choisira encore de transformer les "Entity objet" en des composants EJB, puis effectuera le choix plus particulier d'"EJB Entity" ou "EJB cache" (Gestion Panier devant avoir une durée de vie égale à la visite de l'utilisateur sur le site, un EJB session, n'étant pas lié à une base de donnée persistante, sera plus opportun). L'expérience de l'architecte informatique le poussera à modéliser d'autres composants nécessaires ("Serveur d'application" se transformant par après en "JBoss" dans le case présenté ici), etc. Le choix de ces transformations revient à l'architecte informatique, tantôt il sera contraint par une exigence non fonctionnelle, tantôt il saura orienter ses choix par son expérience personnelle, il est difficile ici d'effectuer des recommandations quant aux transformations. L'outil proposé au chapitre 5 concernant cette activité assistera toutefois l'architecte informatique dans sa tâche.

L'architecture physique est représentée via un diagramme d'Objets comme présenté à la figure 4.20.

4.5 Conclusion

La méthodologie détaillée ici extrait un sous ensemble d'UML, et montre comment ces éléments peuvent servir à décrire les livrables requis pour les différentes phases du cycle de vie proposé. La vision faite ici d'UML reste assez simpliste à dessein. Cette méthodologie se caractérise également par le rôle central que tiennent les usecases (à l'image d'Unified Process) où on retrouve la description d'une exigence fonctionnelle faite en 3 étapes successives proposée dans le cycle de vie. Cette méthodologie a servi de cadre à l'adaptation d'un outil CASE décrite au sein du chapitre suivant.

ControlListe:
Accède
Accède
Accède
EcranCommande:
Localisation=Default
ErreurSaisieCommande:
Localisation=Default
EcranTraitementCommandePersonnel:
Localisation=Default
ListeCommandes:
Localisation=Default
<<EJB Session>>
Panier:
Accède
GestionPanier:
Localisation=Default
Execute
Execute
Execute
Hebergement
Accède
Execute
Execute
<<Servlet Engine>>
Servlet Engine:
Extension
Web Application Server:
Version=3.0.4
comment=Tomcat 4.0.6
Implemente
EJB Container:
Hebergement
Fiche Détaillée:
Localisation=Default
Execute
Execute
Execute
Execute
Execute
Hebergement
Hebergement
Hebergement
ResultatRecherche:
Localisation=Default
ErreurRecherche:
Localisation=Default
RechercheTheme:
Localisation=Default
RechercheTitre:
Localisation=Default
<<EJB Entity>>
P- Catalogue:
<<EJB Entity>>
P-Commande1:
<<EJB Entity>>
P-CommandeL:
<<EJB Entity>>
P-Livre
Accède
Accède
Accède
Accède
Utilisation
Utilisation
Utilisation
Utilisation
ControlRecherche:
ORACLE
pilote
Package :Data -persistent:
<<Connecteur Data>>
Connecteur Data:

Chapitre 5

Adaptation d'un outil CASE

5.1 Introduction

Les outils CASE UML actuels sur le marché offrent souvent la possibilité de créer des profils UML en vue de spécialiser ces outils à des domaines particuliers. La méthodologie UML proposée au sein du chapitre 4 illustre de quelle manière on peut utiliser certains éléments d'UML pour modéliser un projet de développement. En créant un profil UML lié à cette méthodologie, on va pouvoir conduire plus ou moins le processus de développement, en offrant aux différents acteurs du développement des outils particuliers. Il peut s'agir de modèles de description, de conseils méthodologiques ou encore d'outils plus particuliers permettant des transformations semi-automatiques de modèles, etc. Le premier point de ce chapitre présentera globalement la démarche pour établir les attentes quant au toolbox ainsi que les exigences non fonctionnelles. Ces exigences non fonctionnelles constitueront le point 5.3. Je citerai ensuite les outils envisagées pour le toolbox. Enfin, je présenterai le prototype du "toolbox" implémentant quelques uns de ces outils.

5.2 Démarche d'élicitation des exigences

Les exigences quant à l'outil n'étaient pas formellement établies. L'idée de base était de constituer un "toolbox", une "boite à outils" UML.

L'identification des exigences sur l'outil s'est déroulée de manière analogue à la proposition du cycle de vie et de la méthodologie. Au cours de réunions avec les développeurs du CITI, j'ai présenté quelles étaient les possibilités des profils UML, en quoi les profils pouvaient-ils aider au développement. J'ai présenté quelques prototypes d'outils et les développeurs ont effectué leurs remarques concernant ceux-ci, si cela les aiderait vraiment ou non, ou bien ont proposé d'autres outils.

En vue de ne pas introduire de confusion lors de ces réunions, j'ai préféré développer des petits prototypes, plutôt que de présenter des exigences sur l'outil elles-mêmes modélisées en UML. En effet je présentais déjà le cycle de vie, la méthodologie UML avec des exemples lors de ces réunions, un modèle UML de l'outil lui-même aurait pu introduire la confusion. Il est également plus facile d'avoir des remarques sur un prototype que sur un modèle UML. De plus, le travail le plus important ne consistait pas à comprendre le fonctionnement d'un des outils du toolbox (et donc de présenter un modèle de cet outil) mais bien à trouver quels outils on pourrait développer et si ceux-ci sont pertinents.

Globalement ces réunions se sont bien passées, bien qu'au début, les objectifs du toolbox ont été mal perçus. L'idée n'étant pas d'imposer une méthodologie de développement basée sur UML, d'imposer la modélisation des architectures uniquement en UML, d'imposer un outil CASE, etc. mais bien ... "Si on utilisait UML à quoi ressemblerait l'architecture ? Si on utilisait UML, que peuvent apporter les profils ?". La question du choix d'un langage de modélisation est en effet débattue au sein du CITI, car, comme évoqué précédemment, les langages se complètent souvent et seul "le savoir-faire" de l'Analyste ou de l'Architecte informatique permettra de retirer les éléments les plus pertinents de ces langages pour modéliser l'architecture d'un projet particulier. Par la suite, les réunions se sont très bien déroulées, et de nombreuses remarques permettant d'améliorer les outils ou d'ajouter de nouveaux outils selon les besoins ont été formulées.

5.3 Exigences non fonctionnelles sur le toolbox

Tout d'abord le choix technique de l'outil CASE de softeam Objecteering s'est imposé à moi, car même si le CITI n'avait pas définitivement fait le choix d'un outil CASE UML pour ses développements, la courte durée de mon stage a conduit à ce choix personnel. De plus, le CITI possédait déjà une licence de "Objecteering Profile Builder", l'éditeur de profils de softeam.

Une seconde exigence quant au tooolbox était la souplesse de l'outil car en effet, celui-ci ne devait pas imposer une méthodologie ou l'utilisation d'un outil, le développeur devait à tout moment pouvoir faire ce qu'il voulait dans son modèle sans être contraint par le Toolbox. Ceci se reflète au sein du toolbox par la possibilité d'une part de le désactiver et d'autre part de pouvoir utiliser les outils liés à n'importe quelle phase du cycle de vie, à n'importe quel moment.

5.4 Les types d'outils envisagés et leurs apports

Je liste ici quelques outils possibles proposés lors de réunions. Les types d'outils ont pour but de faire refléter la méthodologie, envisagée précédemment, par l'outil CASE. Dans ce cadre, on notera l'importance du chapitre précédent soulevant les différentes caractéristiques lors de la modélisation du développement qui "élicitent" de manière sous-jacente les exigences quant au toolbox.

Un premier outillage se situe au niveau du cycle de vie de développement et de la sélection des éléments UML pertinents pour la phase en cours. Ceci tout en restant cohérent par rapport au critère de souplesse énoncé précédemment. Il est en effet possible par une adaptation de l'outil Case, selon la phase en cours, de ne proposer que les éléments les plus pertinents d'UML pour cette phase. Par exemple, si l'utilisateur fait part de son désir de modéliser des exigences ou d'analyser le domaine, on lui proposera des éléments UML tels Acteur, usecase, diagramme de séquence, etc. ainsi que les outils associés plutôt que des diagrammes de déploiement ou d'autres diagrammes jugés inopportuns pour cette phase.

Un deuxième type d'outil provient de la possibilité de développer des outils permettant de représenter la notion de "modèle de description" (*template*) par des éléments UML et de faciliter le remplissage de ces modèle de description par l'utilisateur. Si l'on prend, par exemple, le cas du CITI, il existe quelques modèles de description incontournables (un acteur, une exigence fonctionnelle possède son modèle). Une adaptation de l'outil CASE permettrait à l'utilisateur, chaque fois qu'il cherche à décrire un utilisateur d'obtenir automatiquement le "modèle de description" associé pour le pouvoir le remplir.

Il est également possible de spécialiser des éléments UML par la notion de stéréotype, en procédant de la sorte les éléments d'UML pourront être beaucoup plus évocateurs pour l'utilisateur. Par exemple, la notion de classe se voulant être un concept assez général, (à bon escient certes), sera beaucoup plus explicite si l'on parle de "classe persistante", d'"objet métier" dans le cadre d'une méthodologie.

Un autre type d'outil envisagé est la possibilité d'effectuer des contrôles sur le(s) modèle(s), supplémentaires à ceux inhérents à UML et aux outils CASE. En effet, les outils CASE se chargent souvent de vérifier la conformité du modèle de l'utilisateur par rapport à un standard (ici UML). On peut ajouter une couche de contrôle au dessus vérifiant la conformité du modèle de l'utilisateur par rapport aux "bonnes pratiques" du CITI. Par exemple, si l'organisation fait le choix de ne pas modéliser des attributs ou opérations dans un diagramme de classe, jugeant ceux-ci peu pertinents pour le client, on peut facilement contrôler cela automatiquement.

Personnaliser un outil CASE peut permettre également de générer du code à partir d'un modèle spécifique. Ceci a été envisagé pour la génération automatique d'IHMs à partir des modèles de représentation des IHMs évoqués précédemment (Chapitre 4).

Des mécanismes permettant d'assurer la traçabilité peuvent également être établis, lors de la modification d'une exigence en amont, l'outil présenterait les répercussions sur l'architecture, etc. Ce type d'outillage ne trouve toutefois pas son équivalent au sein des adaptations faites pour le CITI.

Enfin, les possibilités d'outillage sont grandes, une proposition faite au CITI a été un outil permettant de "faciliter" les transformations permettant de passer d'une architecture logique à une architecture physique en partageant l'expérience interne des architectes informatiques.

L'ensemble de ces outils proposés lors des réunions s'est vu implémenté au sein du toolbox, à l'exception de l'outil destiné à assurer un mécanisme de traçabilité requérant plus de temps.

5.5 Présentation du toolbox

Les sections suivantes décriront plus en détail les outils proposés par le toolbox. L'architecture du code contenant les méthodes "clés" du toolbox et la description de ces méthodes

5.6 Toolbox et outil de gestion du cycle de vie

Une adaptation de l'outil CASE a été faite en vue de conduire le développement par rapport au cycle de vie proposé. Il est en effet important de proposer à l'Analyste et aux Architectes informatiques les outils et éléments UML pertinents par rapport à la phase de développement en cours. Le rôle attendu ici est donc double :

- D'une part, proposer à l'utilisateur une trame, un modèle vide constitué des divers diagrammes pertinents dès l'instant où l'utilisateur fait part de la phase en cours. Il existe un autre intérêt d'ordre "technique" par rapport la création de ce modèle vide, en effet, le "toolbox" attribue un certain statut aux éléments de telle phase ou l'autre et dès lors si un des outils du toolbox doit modifier un diagramme propre à l'analyse, il ne modifiera que ceux-ci.
- D'autre part, ne proposer que les outils propres à la phase en cours. (Si phase d'analyse : modèle de description d'un acteur, d'un usecase ou d'une exigence non fonctionnelle)

Pour refléter les phases quelques stéréotypes ont été créés (figure 5.6).

Stéréotype	Classe parente	Description	Ic
«Capture des exigences»	Package	Package intégrant les informations relatives à la modélisation de la phase de capture des exigences	
«Analyse»	Package	Package intégrant les informations relatives à la modélisation de la phase d'affinement des exigences	
«Maquette»	Package	Package intégrant les informations relatives à la modélisation des IHMs	
«Conception»	Package	Package intégrant les informations relatives à la modélisation de l'architecture	
«Exigences»	Package	Package intégrant les informations relatives aux usecases	
«Domaine»	Package	Package intégrant les informations relatives à la description statique du domaine	
«PSM»	Package	Package intégrant le modèle logique associé à la phase de conception	
«PIM»	Package	Package intégrant les modèles intermédiaires et physiques associés à la phase de conception	

FIG. 5.1 – Définition des stéréotypes associés aux phases de cycle de vie

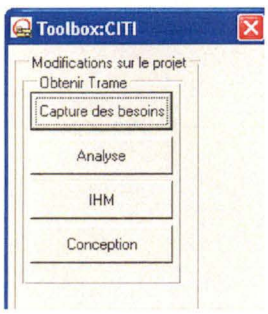


FIG. 5.2 – Gestion des phases du cycle de vie

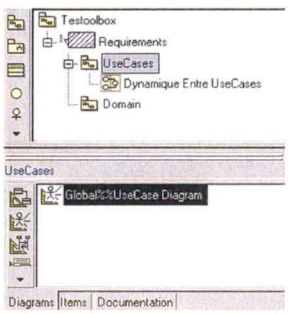


FIG. 5.3 – Création de la structure et des diagrammes vides

capture de besoin, par exemple, aura pour effet de créer la structure de packages stéréotypés et les diagrammes (figure 5.3) et présentera également l'accès aux outils de la phase de capture des exigences(figure 5.4).

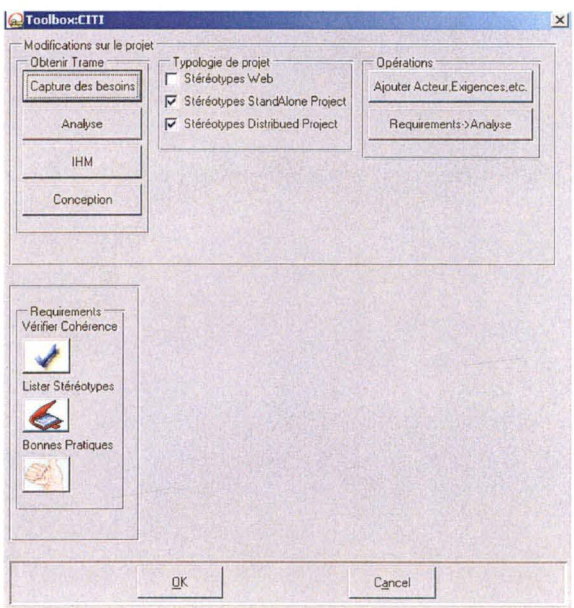


FIG. 5.4 – Accès aux outils spécifiques à la capture des exigences

5.7 Toolbox et outils associés à la phase d'analyse

Le toolbox propose, pour la phase de "capture des exigences" et de "spécifications détaillées des exigences", des modèles de description, des stéréotypes et une vérification par rapport aux bonnes pratiques de l'entreprise.

5.7.1 Description d'un acteur

Concernant la description d'un acteur, le toolbox propose un modèle de description basé sur les informations préconisées dans [Déc02](Les informations requises sont les mêmes que celles fournies lors de l'exemple de www.jebouquine.com). La création d'un acteur se fait via le toolbox(toutefois par respect du critère de souplesse, celle-ci peut toujours s'effectuer normalement). Elle a pour résultat la création d'un diagramme de usecase modélisant cet acteur et où viendront "se placer les usecase" propres à cet acteur. L'acteur est également modélisé au sein d'un diagramme de usecase reprenant l'ensemble des usecases globaux au système[1]. La description de cet acteur, créée également par le toolbox est modélisée via l'élément Note d'UML associée à cet acteur et pouvant être librement masquée.

Le modèle de description reprend, outre le nom de l'acteur, la description de celui-ci, les raisons pour lesquelles on a besoin de cet acteur et en quoi l'acteur est intéressé par le système. Lors de la sélection de l'outil permettant d'ajouter un acteur au sien du toolbox, on obtient la figure 5.5.

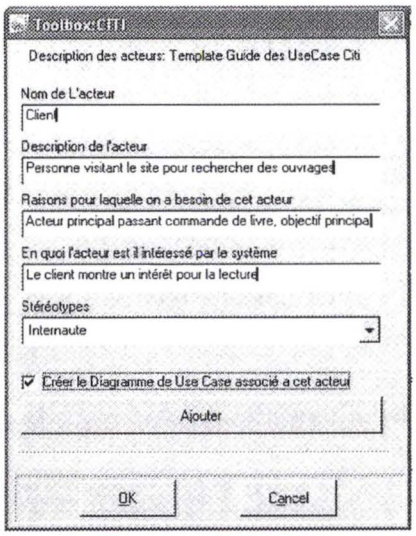


FIG. 5.5 – Modèle de description d'un acteur

L'option finale permet de choisir ou non de modéliser cet acteur ainsi que ses exigences au sein d'un diagramme de usecase spécifique à cet acteur. Le champ stéréotype permet de décrire la nature d'un acteur. Les stéréotypes suivants ont été créés à cet effet : Utilisateur, administrateur, utilisateur privilégié, modérateur, internaute, webmaster, client, acteur interne, acteur externe, mais aussi analyste, développeur, architecte informatique. Seuls les stéréotypes pertinents par rapport à la nature du projet sont proposés (sélection de la typologie du projet sur la fenêtre principale du toolbox parmi projet Web, Standalone, Distribué).

5.7.2 Description d'une exigence fonctionnelle

La description d'une exigence fonctionnelle peut se faire de deux manières (parfois complémentaires), si les informations concernant l'exigence sont déjà avancées, on aura utilement recours au diagramme de séquence associé au usecase modélisant cette exigence, mais le toolbox offre une alternative basée sur le modèle de description d'Alistair Cockburn[Coc98] présenté à la figure 4.2. Toutefois on veillera à refléter l'évolution des spécifications préconisées dans la méthodologie. Par manque de temps, l'outil ne reflète pas vraiment cette évolution

exigences et de spécifications détaillées des exigences. Le modèle reprend également la description d'exigences non fonctionnelles locales au usecase mais un autre outil du toolbox est dédié à la description de telles exigences.

L'interface de l'outil se présente comme suit :

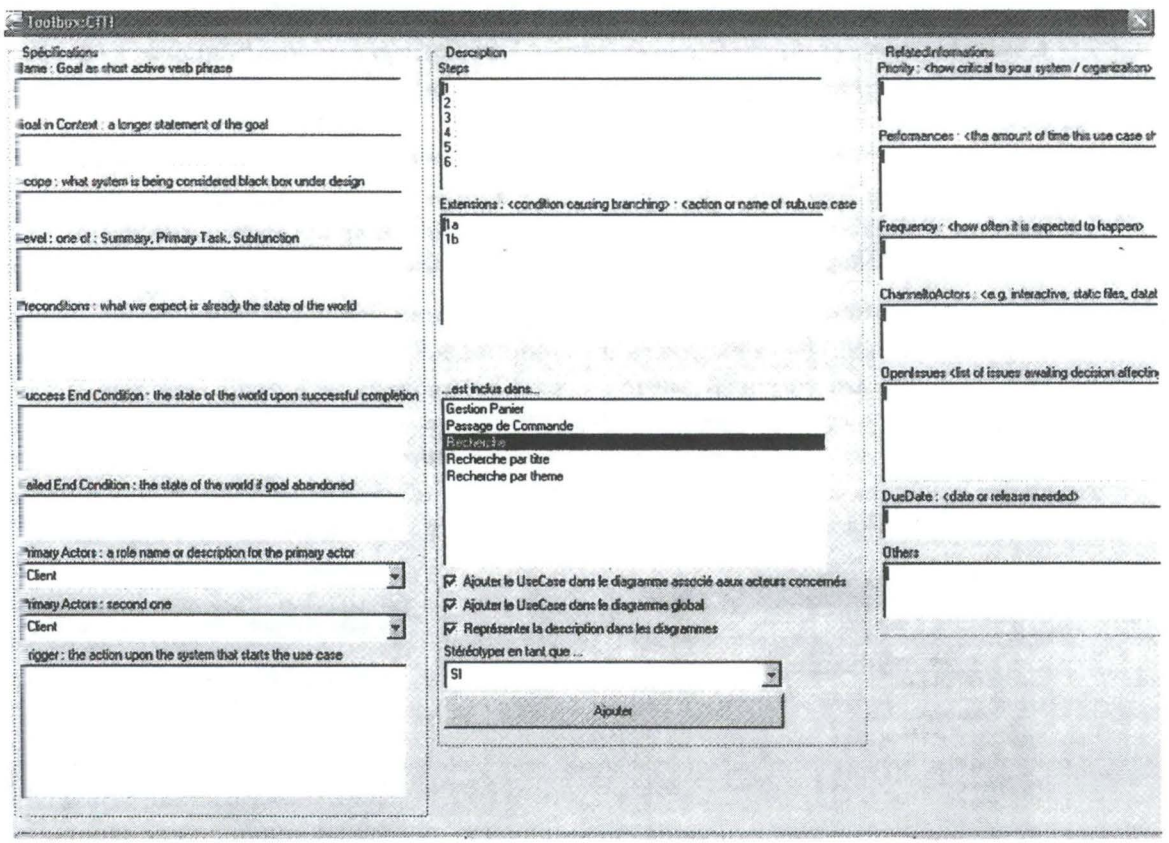


FIG. 5.6 – Modèle de description d'une exigence fonctionnelle

Les champs Acteurs, "...est inclus dans..." proposent les usecase déjà modélisés. Il est également possible de stéréotyper le usecase en SI, non SI et existant. Ces stéréotypes sont intéressants en phase de "spécifications détaillées des exigences" où l'on peut à partir de ce moment spécifier que tel usecase sera manuel, réalisé par le système d'informations ou déjà existant. Les 3 options permettent à l'utilisateur de spécifier s'il veut ou non masquer la description, ne pas représenter le usecase au sein du diagramme principal de usecase, etc.

Le usecase et sa description viendront s'ajouter dans le modèle et seront représentés au sein du diagramme global (reprenant l'ensemble des acteurs et usecase) ainsi qu'au sein du diagramme de l'acteur concerné par une note contenant toutes les informations décrites.

Concernant l'activité de description d'un usecase par un diagramme de séquence re-

d'informations" et "composants" ont été créés sur la méta-classe "ClassifierRole" (Il s'agit d'un "rôle joué par une instance au sein d'une collaboration"[Gro03]" représentée dans les diagrammes de séquence et de collaboration).

5.7.3 Description d'une exigence non fonctionnelle

L'idée de proposer un modèle pour décrire les exigences non fonctionnelles a été également sollicitée lors des réunions, par manque de temps, une veille dans le domaine n'a pas pu être réalisée et il aurait été intéressant d'intégrer quelques éléments du modèle de description de Volere [RR03], etc. Toutefois en vue d'illustrer le principe, le toolbox propose d'ajouter des exigences non fonctionnelles locales à un usecase (modélisée via une note associée au usecase) ou globale au système (modélisée via une note associée au package du projet) grâce à une petite classification ainsi que quelques modèles de description de base.

La modélisation d'une exigence non fonctionnelle se fait via la sélection parmi une classification non exhaustive d'exigences non fonctionnelles (figure 5.7). L'étape consiste à sélectionner le usecase concerné parmi la liste de usecases déjà modélisés que lui propose le toolbox. Laisser ce champ vide aura pour résultat la création d'une exigence non fonctionnelle globale au système. Le toolbox propose alors un petit modèle succinct pouvant reprendre quelques caractéristiques quant au type d'exigence voulu 5.8. Le toolbox crée suite au remplissage une note associée au usecase 5.9 au sein des divers diagrammes.

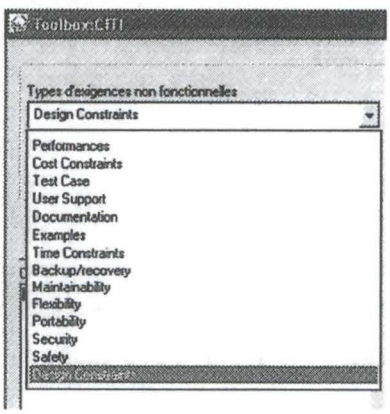


FIG. 5.7 – Sélection d'une exigence non fonctionnelle

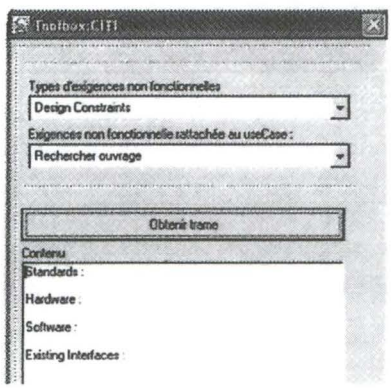
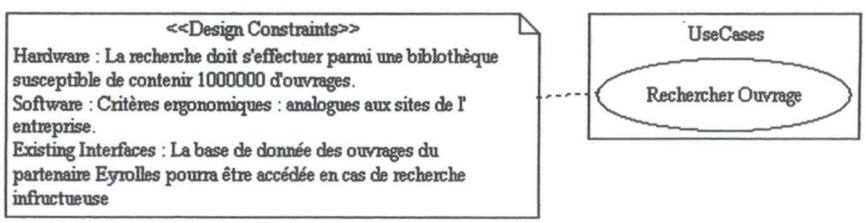


FIG. 5.8 – Le toolbox propose un modèle de description selon le type



En vue de représenter les types d'exigences non fonctionnelles, les stéréotypes suivant ont été créés par rapport à l'élément Note d'UML : Performances, Design constraints (contraintes quant à la conception), Cost constraints (contraintes de coût), Test case (le usecase ou le système devra satisfaire au test suivant), User support (support utilisateur), documentation, Examples, Time constraints (le usecase doit être livré pour telle date), Backup/recovery (contrainte pour assurer la fiabilité des données en terme de sauvegarde et de restauration), Maintainability (Contrainte relative à la maintenance), Flexibility (contrainte de souplesse), Portability, Security, Fiability.

5.7.4 Vérification par rapport aux bonnes pratiques de l'entreprise

Le toolbox propose une couche de vérifications supplémentaires sur le modèle, au dessus des contrôles d'intégrité et de conformité à UML déjà réalisés par l'outil case. Ces vérifications se font par rapport aux "bonnes pratiques" du CITI. Elles ont pour objet essentiellement de vérifier au sein du modèle de l'utilisateur, que l'utilisateur n'a pas employé ce diagrammes inadéquats selon la phase du cycle de vie. La capture des exigences ne devrait pas utiliser de diagrammes de déploiement, etc. Egalement, l'expérience du CITI, veut que lors de la description statique du domaine par un diagramme de classe, les classes ne contiennent d'opérations ni d'attributs en phase de capture des exigences et pas d'opérations en phase de spécifications détaillées des exigences. Par respect du critère de souplesse, le toolbox ne modifie pas le modèle de l'utilisateur mais affiche une liste d'informations, recommandations à ce dernier lorsque celui-ci a demandé une telle vérification (figure 5.10).

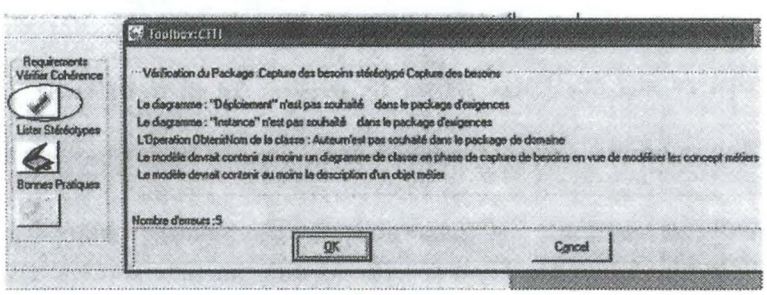


FIG. 5.10 – Vérification du modèle réalisé en phase de capture des exigences

5.7.5 Description du domaine

Concernant l'activité de description du domaine, en vue de représenter des objets métiers, le stéréotype "objet métier" a été créé sur la métaclasse "Class" décrivant un concept bien connu du client dans son métier. Le toolbox, mis à part quelques vérifications décrites précédemment ainsi que la création du diagramme n'offre pas d'outil particulier pour cette activité. L'utilisateur modélise lui-même le domaine et dispose de classes stéréotypées "objet métier".

5.8 Toolbox et outil associé à la modélisation des interfaces

Comme évoqué précédemment, il serait intéressant de choisir un modèle de représentation complet et validé par les développeurs pour les interfaces. A partir de tels modèles, on aurait pu concevoir un outil capable de générer du pseudo-code implémentant un prototype des interfaces. Par manque de temps, ceci n'a pu être réalisé. Toutefois , le toolbox offre des recommandations (issues de [Roq02]) pour modéliser les interfaces par deux modèles complémentaires de représentation. Ceux-ci sont identiques à ceux réalisés dans le chapitre 4 pour le cas de www.jebouquine.com, un premier diagramme d'activité pour modéliser l'enchaînement des interfaces, la dynamique entre ceux-ci et, en second, un diagramme de classe modélisant les informations contenues au sein des fenêtres (Figures 4.13 et 4.14). En poussant un peu plus loin ces modèles on aurait pu prévoir par exemple, des stéréotypes particuliers pour les attributs des classes contenues dans ces modèles, ces stéréotypes auraient alors, par exemple, défini des types lié à un langage d'implémentation (ex : Pour Java : JComboBox, JTextField, etc.).

En vue d'aider l'utilisateur dans sa modélisation, le toolbox propose les quelques stéréotypes suivants issu de [Roq02] pour le cas d'un projet web :

- Stéréotypes associés aux activités : Page, action (de l'utilisateur), Frame, Exception.
- Stéréotypes associés aux classes : Vue Menu (Interface destinée à orienter l'utilisateur), Vue application (Interface destiné à la saisie et/ou la production d'informations)

5.9 Toolbox et outil associé à la phase de conception

5.9.1 Apport du toolbox pour la phase de conception

L'outil doit conduire, aider la méthodologie explicitée en phase de conception au sein du chapitre 4.

Dans cette méthodologie, on passe d'un modèle logique, (Platform indépendant model) vers un modèle physique (Platform specific Model) par une série de transformations successives. On appellera les modèles résultants de ces transformations intermédiaires des modèles intermédiaires (Low-level Transient Model).

Ces modèles représentent un ensemble de composants plus ou moins spécifiques à la plate-forme et leurs relations, ceux-ci collaborant à la poursuite de l'objectif du système global. L'outil proposé partant d'une architecture logique va proposer de transformer les éléments de ces architectures en des composants plus ou moins spécifiques à la plate-forme jusqu'à obtenir une architecture physique. Un composant peut être un fichier XML, une base de donnée, un script PHP, ou encore un connecteur, une source de données, etc. Un composant possède un nom, un stéréotype et des couples [attributs, valeurs] le décrivant. Les exigences fonctionnelles du Toolbox quant à la phase de conception sont les suivantes, quel que soit le modèle en cours de l'utilisateur : L'outil devrait ...

- proposer des composants plus ou moins spécifiques à l'utilisateur, des composants types, fruits de l'expérience des employés dans le domaine des architectures.

- Suite à l'ajout d'un composant, proposer les composants "pouvant être liés" à ce dernier. Par expérience, un développeur sait que s'il ajoute le composant "Fichier XSL", il aura besoin d'un "processeur XSLT", le toolbox devrait refléter ces automatismes et proposer tous les composants pouvant être liés et éventuellement les ajouter au modèle en cours.
- Suite à l'ajout d'un composant, proposer de lier celui-ci avec des composants déjà modélisés. L'ajout d'un "Fichier XSL" peut être lié aux "Fichier XML" déjà modélisés. Seuls les liens "pertinents" doivent être proposés à l'utilisateur.

L'outil devrait donc connaître un ensemble de composants, de liens entre eux et la notion de transformations entre ces composants. (Le modèle d'information est présenté en annexe B). Toutefois, ces informations doivent être issues de l'expérience des employés. Cette expérience s'enrichissant constamment, le toolbox se devait d'être évolutif de ce point de vue, celui-çi doit donc proposer en outre :

- la possibilité d'ajouter de nouveaux composants au toolbox, composants qui seraient alors proposés aux fonctionnalités attendues et citées précédemment.
- La possibilité d'ajouter des liens "possibles" entre composants (déjà existants ou nouveaux) au toolbox, composants qui seraient alors proposés aux fonctionnalités attendues, citées précédemment.
- La possibilité d'ajouter des transformations possibles entre composants au toolbox, composants qui seraient alors proposés aux fonctionnalités attendues et citées précédemment

Il ne faut donc pas confondre au sein de ce chapitre les deux notions :

- La notion d'*ajouter un composant (,lien ou transformation)à une architecture* qui est une fonctionnalité du toolbox permettant à l'utilisateur de choisir un composant(,lien ou transformation) et de l'ajouter dans son modèle
- la notion d'*ajouter un composant au toolbox* qui est la possibilité d'étendre le toolbox de telle façon, que le toolbox dans sa fonctionnalité "d'ajout d'un composant à une architecture", par exemple, propose désormais ce nouveau composant. L'ajout d'un composant au toolbox est également une fonctionnalité du toolbox.
  De plus une telle extensibilité du toolbox me paraissait plus judicieuse étant donné qu'en développant le toolbox, je ne pouvais me permettre d'y placer des composants, liens ou transformations si celui-ci restait statique n'ayant pas suffisamment d'expérience dans le domaine, les composants proposés initialement n'auraient peut-être pas été pertinents. L'ensemble de composants initiaux peut donc être étendu mais aussi modifié.

5.9.2 Prototype de l'outil du toolbox lié à la phase de conception

Un prototype de l'outil a été développé et intégré au toolbox, celui-ci implémente les fonctionnalités requises, il a été présenté lors d'une réunion aux différents développeurs. Concernant l'activité de modélisation d'une architecture logique, les stéréotypes suivants ont été créés sur la métaclasse "ClassifierRole" ; "Object Entity","Interface","Objet persistant" et "Control" correspondant aux concepts manipulés dans cette même acti-

Dans un premier temps, l'outil veille à créer les diagrammes adéquats vides. La première étape pour l'utilisateur consiste à modéliser son architecture logique au sein du diagramme correspondant et semblable à l'architecture logique présentée au chapitre 4 4.19. Un outil intéressant pour le toolbox aurait été de proposer à l'utilisateur de développer une première esquisse d'architecture possible à partir des informations relatives aux exigences contenues au sein des diagrammes de séquence ou de collaboration réalisés dans les phases précédentes. Le toolbox aurait alors proposé par exemple un modèle en 3 couches (Interface, Control, Entity, objet persistant) en recherchant les éléments stéréotypés de la sorte au sein des différents diagrammes.

Du modèle logique vers un premier modèle intermédiaire

Le toolbox propose une première étape consistant à transformer son modèle logique en un modèle intermédiaire dont les composants seront plus ou moins spécifiques à la plate-forme. La figure 5.9.2 représente l'outil dans cette première étape.

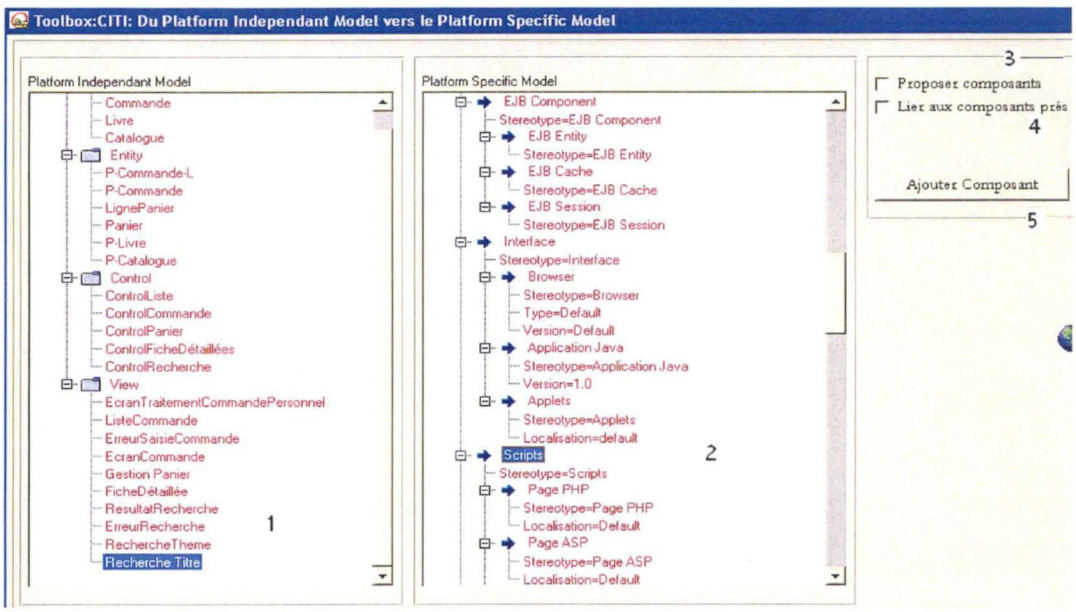


FIG. 5.11 – 1er niveau de transformation du modèle logique

- 1 : Le toolbox présente dans cette partie les éléments du modèle logique à l'utilisateur. Sélection d'un élément du modèle logique (package ou classe).
- 2 : Un ensemble de composants est présenté à l'utilisateur, celui-ci est représenté par un arbre où la relation entre noeud et feuille désigne la transformation du composant au niveau du noeud en un composant plus spécifique associé au niveau de la feuille. Selection parmi ces composants d'un composant jouant le rôle de l'élément sélectionné dans le modèle logique.
- 3 : Ce bouton radio signifie que le toolbox devra proposer tout les éléments pertinents pouvant être associés au composant sélectionné en 2.

L'utilisateur choisit dans une première étape de transformer une interface "Recherche Titre" en un script. Sans aucune des options en 3 et 4 cochées. Le toolbox transforme alors ce composant sur le modèle intermédiaire. Supposons maintenant qu'il décide de transformer l'élément Entity "P-Catalogue" en un "EJB component", c'est avec cette fois ci l'option 3 sélectionnée. La figure 5.12 illustre les conséquences du choix de l'option 3. La figure 5.13 montre le diagramme intermédiaire produit par le toolbox.

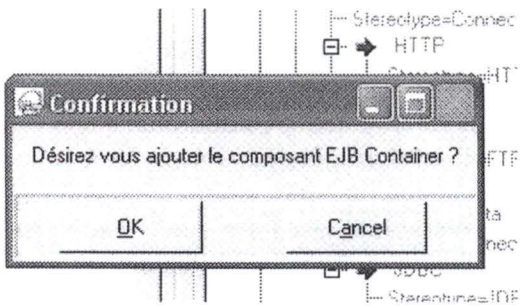


FIG. 5.12 – Proposition des composants liés

Le toolbox dans ce cas-ci va proposer successivement les composants suivants, supposons les réponses de l'utilisateur entre crochets : EJB Container[OK],JBoss[Cancel], JDBC[Cancel], Connecteur Data[OK].
Le toolbox propose successivement "JDBC" et "Connecteur Data", car un connecteur Data peut être lié à un "Ejb Component" et "JDBC" est une transformation spécifique de "Connecteur Data". (Sorte d'héritage du lien, toutefois, l'héritage n'est pas géré par le toolbox , les 2 liens entre "JDBC" et "EJB component" ainsi qu'entre "Connecteur Data" et "EJB component" sont enregistrés tout deux au sein du toolbox)

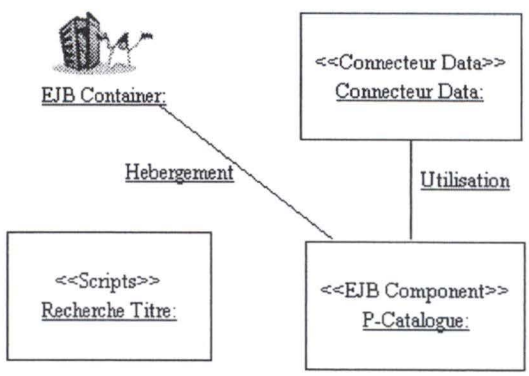


FIG. 5.13 – Modèle intermédiaire produit par le toolbox

Ces premiers choix de transformations effectués par l'architecte informatique restent peu spécifiques à la plate-forme, d'autres transformations viendront s'effectuer sur ces premiers composants produisant à terme un modèle physique, spécifique à la plate-forme. L'architecte informatique pourra encore ici spécifier que ses éléments du package "persistent" constitueront une "source de données" (il peut sélectionner tout un package). L'option 4 sur la figure 5.9.2 a pour objet de stipuler au toolbox d'essayer de lier le nouveau composant produit aux composants déjà présents dans le modèle intermédiaire. Supposons en effet que l'architecte informatique désire transformer les objets persistants en une source de données avec l'option 4 activée. Le toolbox trouve un lien pertinent entre une "source de données" et un "connecteur data" comme illustré à la figure 5.14. Le résultat est présenté à la figure 5.15

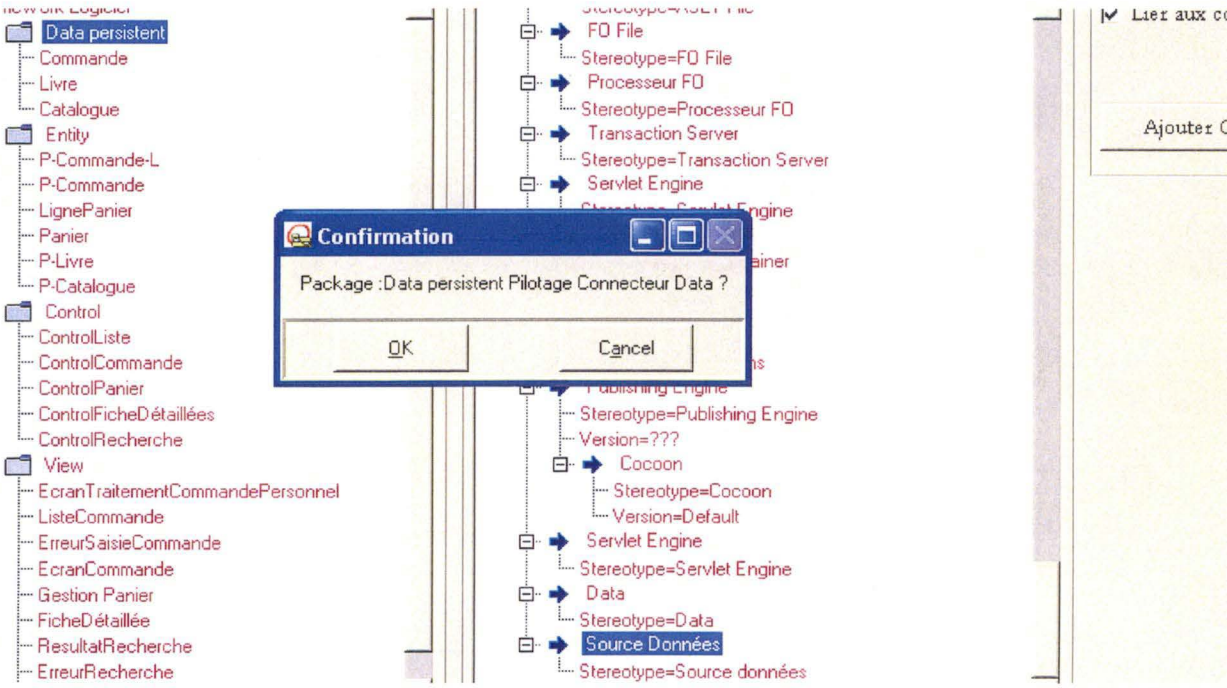


FIG. 5.14 – Transformation des objets persistants en "une source de données", proposition d'un lien

Le toolbox a trouvé un "connecteur data" dans le modèle intermédiaire or celui-ci peut être lié à la source de données d'où la proposition. Toutefois, le toolbox vérifie que ce connecteur data n'est pas déjà lié par le même type de lien (même nom) avec une source de données déjà modélisée auquel cas, il ne propose pas de lier avec la nouvelle source de données en vue de limiter les propositions faites à l'utilisateur pouvant s'avérer dans certains cas peu pertinentes.

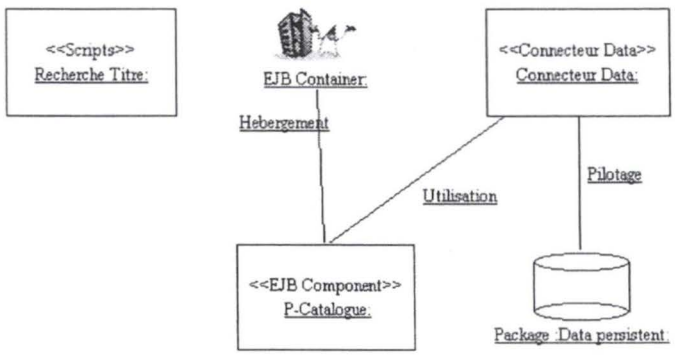


FIG. 5.15 – Résultat suite à la transformation de "data persistent" en source de données

Ajouter les composants liés à un composant existant sur le modèle intermédiaire

Une option du toolbox permet à tout moment de prendre un composant déjà existant du modèle intermédiaire et d'ajouter tout les composants pouvant être liés. Ceci est analogue à l'option 3 de la figure 5.9.2. Supposons cette fonctionnalité effectuée sur l'objet Script, le toolbox trouve alors qu'un script peut éventuellement être lié à un "Servlet Engine", il propose ce choix à l'utilisateur qui le valide et obtient le modèle présenté à la figure 5.16

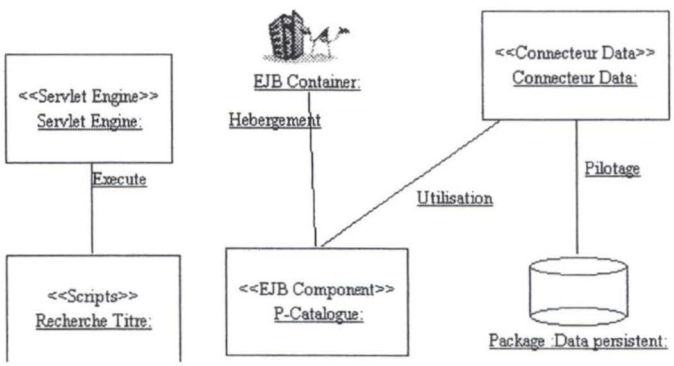
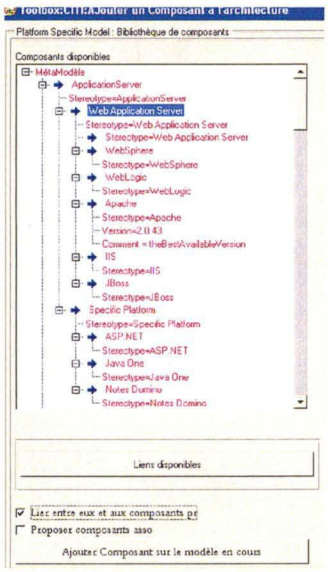


FIG. 5.16 – Ajout d'un *Servlet Engine*

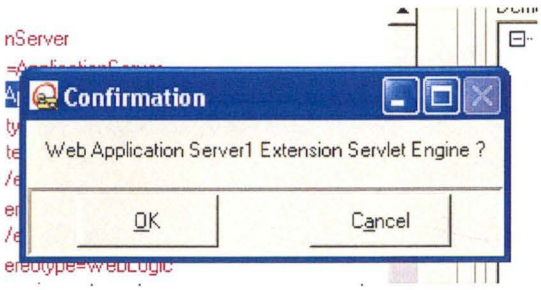
Ajouter un composant au modèle au modèle intermédiaire

Certains composants ne sont pas dérivés directement de composants du modèle logique. L'ajout de composant est rendu possible par le toolbox comme présenté à la figure 5.9.2



Il est possible de sélectionner plusieurs composants et de les ajouter en un coup. Tous les composants sélectionnés peuvent posséder des liens pertinents entre eux, ceux-ci sont créés par le toolbox si l'option "Lier entre eux et aux composants du modèle" a été sélectionnée. Le choix de cette option a également pour conséquence de créer les liens pertinents entre les composants ajoutés et les composants déjà existants dans le modèle intermédiaire (du moins ces liens sont proposés à l'utilisateur).

FIG. 5.17 – Ajout d'un composant *Web Application Server* au modèle intermédiaire



Le toolbox propose de lier le composant "Servlet Engine" au nouveau composant *Web Application Server* car un lien entre les deux est pertinent. Il proposera ensuite un lien avec l'EJB Container existant dans le modèle intermédiaire.

FIG. 5.18 – Proposition d'un lien

Le fait de choisir de "proposer les composants liés" est analogue à l'option 3 de la figure 5.9.2 mais est répété pour tout les composants sélectionnés par l'utilisateur dans la fenêtre d'"ajout d'un composant", toutefois, si deux des composants sélectionnés peuvent être liés à un composant Y, Y ne sera proposé qu'une seule fois à l'utilisateur.

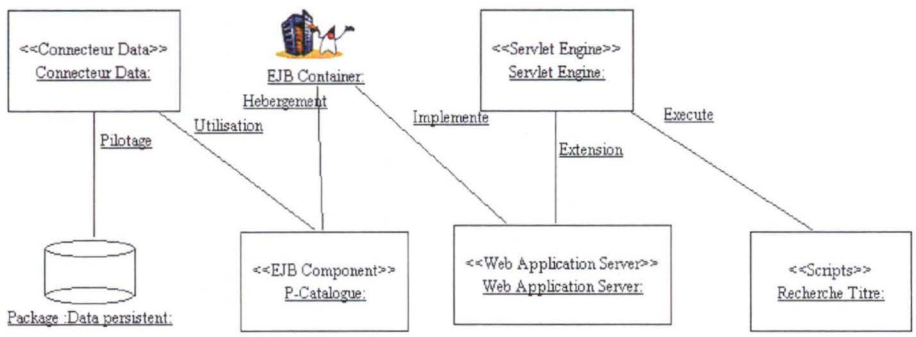


FIG. 5.19 – Resultat de l'ajout du *Web Application Server* au modèle de la figure 5.16

Transformations du modèle intermédiaire

L'utilisateur peut également continuer à transformer ses composants par des choix technologiques issus de sa propre expérience ou d'exigences non fonctionnelles formalisées lors des étapes précédentes. Supposons le modèle intermédiaire issu de la figure 5.19. L'interface liée à la transformation de modèle est présenté à la figure 5.20.

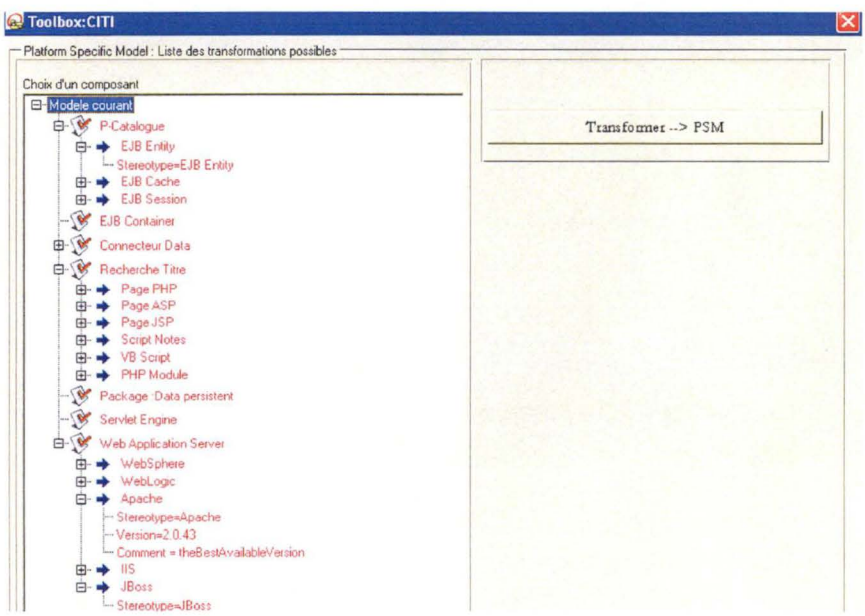


FIG. 5.20 – Toolbox et transformations

L'outil présente d'une part les composants du modèle actuel au sein d'un arbre , les fils de la racine représentant chacun de ces composants. Le toolbox propose alors pour chacun de ces composants les transformations possibles et ceci récursivement jusqu'au niveau terminal (transformations de composants transformés, etc.). Il s'agit ici pour l'architecte informatique soit de tenir compte de son propre savoir-faire, soit de tenir compte des éventuelles exigences non fonctionnelles identifiées précédemment pour choisir correctement ces transformations. L'expérience de l'architecte aidant, il sera amené à transformer le "Web Application Server" implémentant des "EJB container" en "JBoss". Il convertira également le "script" Recherche Titre en "Page JSP", l'"EJB Component" P-Catalogue en EBJ Entity.

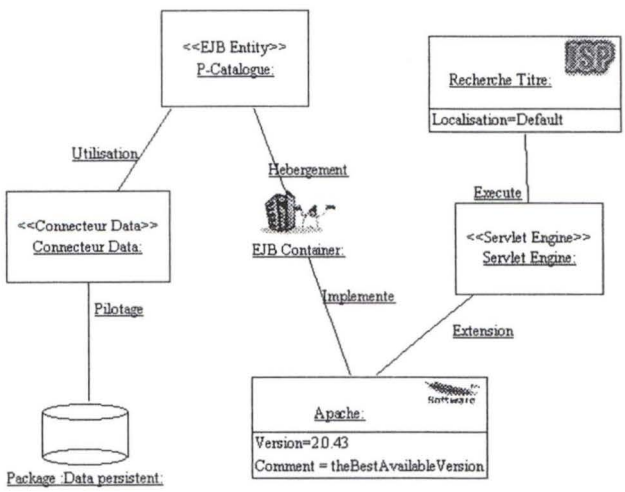
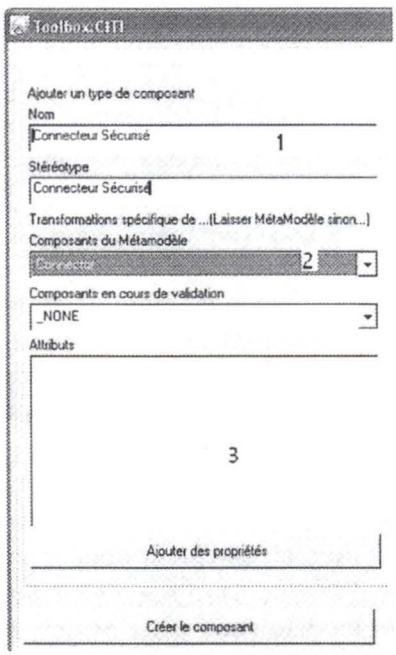


FIG. 5.21 – Modèle intermédiaire de l'utilisateur après les transformations évoquées à la figure 5.20

Soumission de nouveaux éléments au toolbox

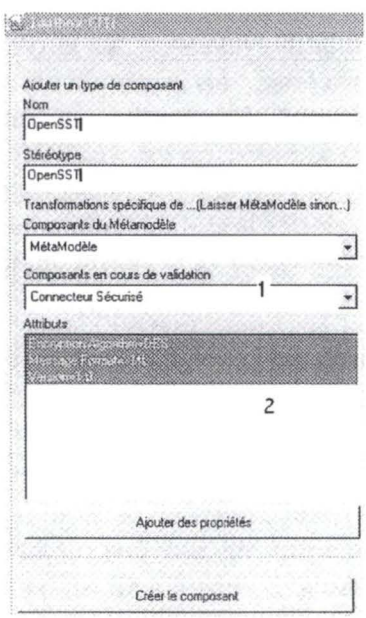
Les utilisateurs du Toolbox peuvent soumettre des nouveaux composants, liens et transformations. Ces nouveaux éléments pourront être utilisés au même titre que les autres composants déjà introduits dans le toolbox. En vue de contrôler l'ajout de tels éléments, ceux-ci doivent être validés par un administrateur du toolbox, celui-ci étant en possession d'un outil supplémentaire (un autre profil UML) pour pouvoir valider les diverses propositions (Voir section "Administration du toolbox"). Cette extensibilité du toolbox constitue une mise en commun de l'expérience des architectes informatiques du CITI.

Supposons qu'un architecte cherche à soumettre un composant "Connecteur sécurisé" qui est une transformation du composant "connector" déjà présent dans le toolbox. Il cherche également à spécifier "OpenSST", transformation spécifique de "Connecteur sécurisé". Il cherche également à proposer le lien "input-output" entre "XML file" (déja existant dans le toolbox) et "OpenSST".



- 1 : Saisie d'un nom par défaut pour le composant et d'un stéréotype désignant ici le type du composant. Il est à préciser ici que normalement il n'est pas possible de créer de nouveaux stéréotypes sans passer par l'éditeur de profils , le toolbox simule donc ici le stéréotype par une "tagged Value" ce qui explique également l'impossibilité d'associer une image aux nouveaux composants.
- 2 : Une liste des composants disponibles est dressée, l'utilisateur spécifie celui dont le nouveau composant est une transformation spécifique
- 3 : Définition d'attributs et de valeurs par défaut

FIG. 5.22 – Soumission d'une demande du nouveau composant "Connecteur sécurisé"



- 1 : Une liste des composants en cours de validation, l'utilisateur spécifie le composant existant dont le nouveau composant est une transformation spécifique
- 2 : Une petite interface permet d'ajouter le type et la valeur d'une propriété.

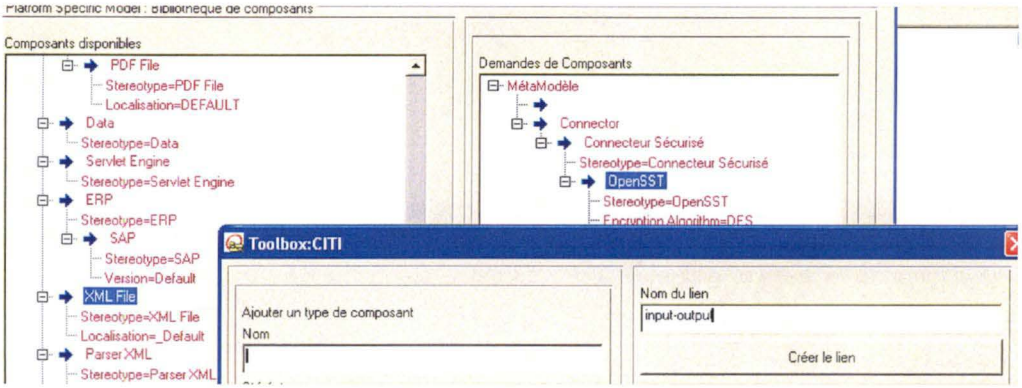


FIG. 5.24 – Soumission d'une demande de lien "OpenSST" input-output "XML File" En arrière-plan se trouve l'interface dédiée à l'ajout de composants et de saisie de demande de nouveaux composants, liens et transformations. En vue d'ajouter un lien, l'utilisateur sélectionne deux composants au sein de cette fenêtre (Existants ou en cours de validation) et indique un nom.

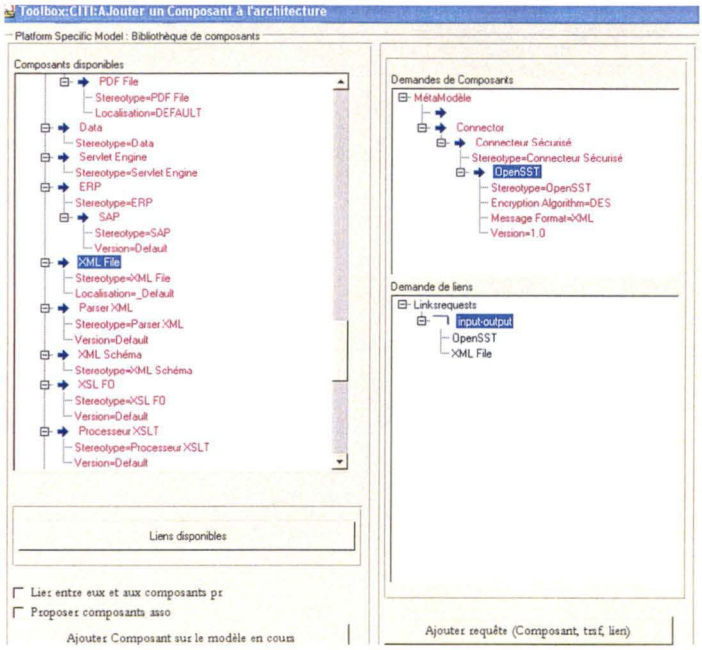


FIG. 5.25 – Liste des composants, transformations et lien en cours de validation

Administration du toolbox

L'administrateur dispose d'un profil UML supplémentaire, il peut accéder à l'outil "Administration", celle-ci est constituée de l'outil présenté à la figure 5.26

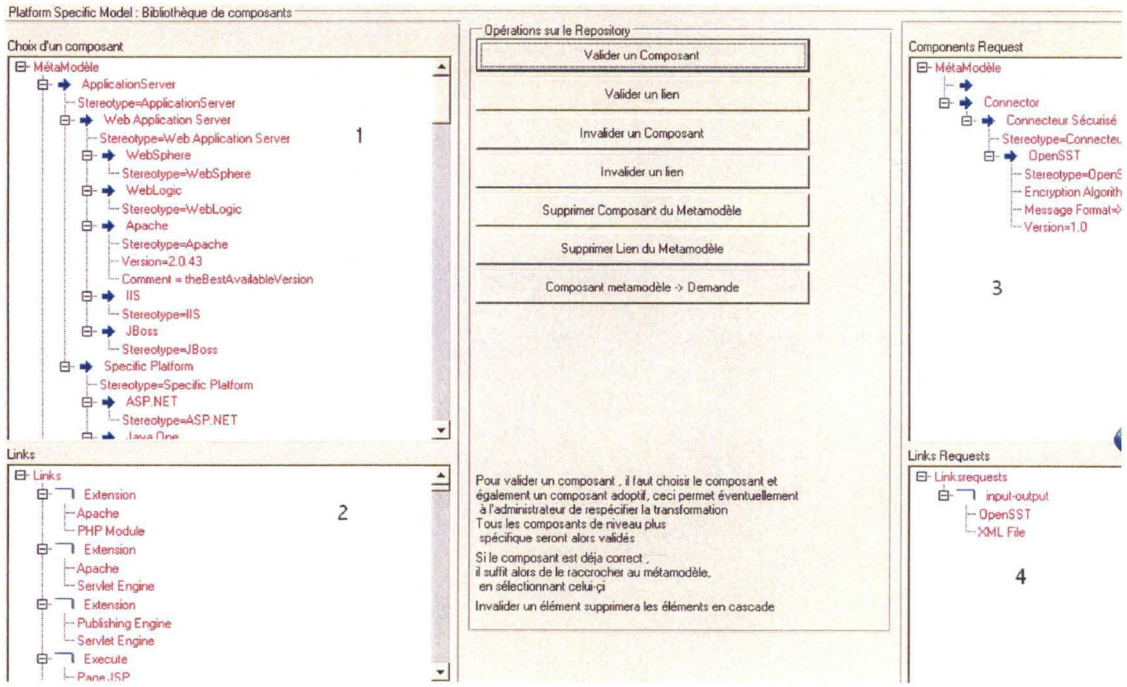


FIG. 5.26 – Administration du toolbox

- 1 : Liste des composants et transformations présents dans le toolbox
- 2 : Liste des liens présents dans le toolbox
- 3 : Liste des composants, et transformations soumis par les utilisateurs
- 4 : Liste des liens soumis par les utilisateurs

L'administrateur peut valider (figure 5.9.2) ou invalider une demande d'un composant ou d'un lien, il peut également valider une demande de composant mais en respécifiant la transformation, par exemple spécifier qu "openSST" serait une transformation d'un autre composant. Il peut supprimer des composants ou des liens existants, il peut également changer le statut d'un composant (et la transformation associée) en demande de composants ou changer le statut d'un lien en demande de lien.

5.9.3 Remarques

Je reprends ici quelques remarques personnelles ou proposées lors de réunions en vue d'améliorer le toolbox.

- Concernant l'ergonomie des interfaces, celle-ci est peu conviviale, certains éléments graphiques se voient atrophiés. Ceci ne peut se résoudre que par une nouvelle version

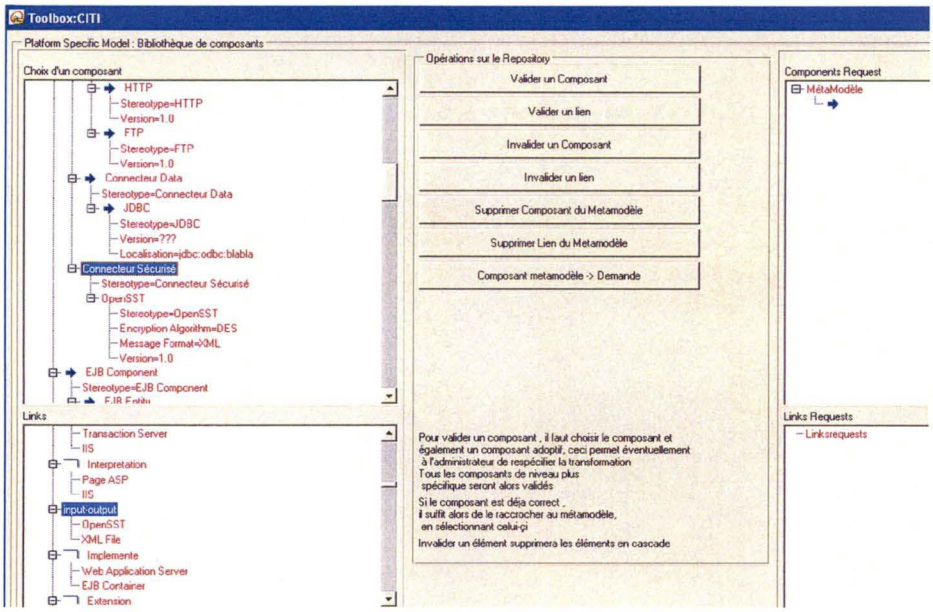


FIG. 5.27 – Résultat de la validation des demandes

Les composants, liens et transformations ajoutés peuvent maintenant être l'objet des fonctionnalités du toolbox pour tous les projets et utilisateurs.

premier modèle intermédiaire et d'autre part les transformations des modèles intermédiaires qui suivent. Ces transformations étant de même nature, la fonctionnalité devrait être la même. Par rapport à l'exemple cité au chapitre 4, après avoir ajouté les composants "Entity", "Control", "Boundary"(interface) au toolbox ainsi que les transformations "Boundary" -> "scripts", on effectuerait des transformations directement à partir du modèle logique puis éventuellement d'autres transformations en passant toujours par cette même fonctionnalité de transformation. Le fait d'avoir scindé la première étape de transformation et les autres étapes de transformations découle du fait qu'une architecture logique est souvent modélisée par un diagramme de classe (au CITI), et une architecture physique par un diagramme d'objet. Les fonctionnalités ne sont donc pas tout à fait identiques du point de vue du code.

- Il a été proposé lors des réunions d'ajouter un mécanisme de mémorisation de "justifications associées aux transformations"

5.10 conclusion

Les outils proposés au CITI ont l'intérêt majeur d'illustrer les possibilités quant à la réalisation de profils UML adaptant l' outil CASE "objecteering" qu'il s'agisse de stéréotypes, de templates de description, ou d'outil de transformation de modèle, ceux-ci montrent comment le développement peut être guidé, suivant les recommandations, les habitudes de l'entreprise et pouvant même ici faire partager les connaisssances

Conclusion

Les modèles de cycles de vie se doivent d'être adaptés au contexte de l'entreprise ou même d'un projet. Le cycle de vie proposé au CITI n'est à considérer que (est figé) sous l'angle des activités, des livrables intermédiaires associés aux phases plutôt que sous l'angle de l'ordonnancement des phases, de l'attribution des ressources aux activités ou de la gestion du projet. Le cycle de vie est ici un support en vue d'illustrer les possibilités d'UML et d'adaptation d'un outil case. C'est pourquoi le cycle de vie proposé reste assez classique avec une succession de livrables intermédiaires. Etablir un modèle de cycle de vie moins classique, de type Agile, par exemple, aurait été moins intéressant du point de vue de l'outillage CASE.
La méthodologie UML est issue de la description de ces activités et de ces livrables proposée dans le cycle de vie. La méthodologie extrait un sous ensemble d'UML, sélectionne quelques diagrammes qui s'avèrent adéquats pour modéliser les livrables attendus au sein de chaque phase du cycle de vie identifiée auparavant. et énonce quelques bonnes pratiques quant à la modélisation d'un projet. En définissant une méthodologie adaptée aux habitudes des développeurs, on pallie à l'ambiguïté sémantique d'UML auxquels les développeurs ou le client sont parfois confrontés en utilisant UML.
Grâce aux possibilités offertes par les profils uml, on peut adapter un outil CASE. On peut alors offrir aux acteurs d'un projet différents outils associés aux phases du cycle de vie. Le toolbox illustre cette démarche et offre à l'utilisateur quelques modèles de description d'éléments faisant partie intégrante des livrables associés aux phases du cycle de vie d'un développement, il offre également des outils de vérification par rapport aux bonnes pratiques de l'entreprise, des outils de transformations (semi-)automatiques de modèle peuvent également aider le développeur dans la réalisation de l'architecture.
L'apport de ce mémoire par rapport au CITI réside sur plusieurs points.
Tout d'abord, par la proposition d'un cycle de vie et d'une méthodologie, les techniques de développement et les modèles produits au CITI se voient uniformisés, ceci rentre dans un objectif d'apprentissage et de sensibilisation d'UML aux développeurs.
Un second apport réside dans la personnalisation d'UML qui est faite où, par la définition de la méthodologie et de quelques extensions d'UML, on "désambiguise" ce langage quant à sa sémantique, on le rend plus compréhensible par le client et les développeurs.
Enfin, par rapport à l'objectif initial d'illustrer les possibilités offertes par les profils UML, le toolbox développé montre comment l'adaptation d'un outil CASE peut permettre de conduire le développement d'après une certaine méthodologie, de faciliter la vérification et la validation des produits. L'outil CASE adapté par ces profils devient alors le garant de la qualité du développement logiciel pour une entreprise, il peut également accroître la rapidité. Il homogénéise les pratiques, les techniques de

toolbox.
Il aurait été intéressant pour cet outillage au processus de développement d'adopter une démarche plus importante que celle entreprise ici, d'effectuer une sensibilisation plus marquée des développeurs par rapport à l'intérêt du toolbox. Un tel outillage au processus de développement exige d'arriver à un certain consensus de la part des développeurs, une certaine homogénéisation des pratiques et des habitudes ce qui dans le contexte de petits projets développés en interne peut encore paraître faisable, mais qui dans le contexte plus général des projets d'innovation peut s'avérer fastidieux. Des travaux plus importants de veille sur les besoins en terme de méthodologie pour les projets du CITI permettraient d'améliorer le toolbox.
Avec plus de temps accordé, il aurait été intéressant de tester ce toolbox sur des projets en cours au CITI, afin d'obtenir un certain retour d'expérience et éventuellement adapter le toolbox. D'autres perspectives intéressantes restent la réalisation d'un outil veillant à assurer un mécanisme de traçabilité des éléments UML (mais dont la portée resterait toutefois au niveau de l'outil case concerné), d'un outil chargé de générer de petits prototypes d'IHMs à partir d'un modèle suffisamment complet, ou encore de profils spécialisant l'outil CASE à un domaine de recherche particulier.

Bibliographie

Livres

[Boo94] G. Booch. Object Oriented Analysis and Design with Application. Addison-wesley edition, 1994. Seconde édition.

[Jac92] I. Jacobson. Object-Oriented Software Engineering : A Use Case driven Approach. Addison-wesley edition, 1992.

[Jac96] D. Jacquin. Maîtriser votre gestion de configuration logicielle. Paris, masson edition, 1996.

[Roq02] P. Roques. UML - Modéliser un site e-commerce. France, eyrolles edition, Septembre 2002.

[RV00] P. Roques and F. Vallée. UML en action. Eyrolles edition, 2000.

[Sob97] M. Soberman. Développement rapide d'application. Paris, France, hermes edition, 1997.

Thèses

[BD02] J.-P. Bodelet and B. Desaintghislain. Proposition d'une démarche dédiée à la conduite de projets d'innovation. PhD thesis, Facultés Universitaires Notre Dame de la Paix, Namur, 2002.

[Khr00] I. Khriss. Vers un paradigme transformationnel dans le développement orienté objet. PhD thesis, Université de Montréal, Montréal, mai 2000. Chapitre 2.

Articles

[BHH+97] R. Breu, U. Hinkel, C. Hofman, C. Klein, B. Paech, B. Rumpe, and V. Thurner. Towards a formalization of the unified modeling language. Lecture Notes in Computer Science, 1241 :344–356, 1997.

[LS96] M. Lindvall and K. Snadhal. Pratical implications of traceability. Software – Practise Experience, 26 :1161–1180, 1996.

[Rum95] J. Rumbaugh. Omt : The development process. Journal of Object Oriented Programming, 8 :8–76, 1995.

[SA02] S. Si Alhir. Uml : Understanding the unified process. Methods & Tools, Volume 10 :Page 2–18, Mars 2002. on-line : http ://www.methodsandtools.com/PDF/dmt0102.pdf.

[Déc02] C. Décosse. Guide d'expression et d'analyse des exigences avec les usecase, 2002.

[Lei02] S. Leidner. Outillage du processus d'acquisition d'un logiciel plan qualité de projet, octobre 2002.

En ligne

[Coc98] A. Cockburn. Template de usecase. Octobre 1998. `http://members.aol.com/acockburn/papers/uctempla.doc` (consulté le 28-05-2003).

[Gog98] M. Gogolla. Uml for the impatient. Technical report, University of Bremen, 1998. `http://citeseer.nj.nec.com/89077.html` (consulté le 02-08-2003).

[Gro03] Object Management Group. Omg unified modeling language specification, Mars 2003. Version 1.5, `http://www.omg.org` (consulté le 02-08-2003).

[otHKSAR02] The Government of the Hong-Kong Special Administrative Region. An introduction to rapid application development. 2002. `http://www.itsd.gov.hk/itsd/english/itgov/download/g47a.pdf` (consulté le 28-05-2003).

[RR03] J. Robertson and S. Robertson. Volere requirements specification template. 2003. `http://www.volere.co.uk` (consulté le 02-08-2003).

[Sof99] Softeam. Profiles uml et langage j : Contrôlez totalement le développement d'applications avec uml, 1999. `http://www.softeam.fr` (consulté le 28-05-2003).

[Str00] A. Strohmeier. Cycle de vie du logiciel. 2000. `http://lglwww.epfl.ch/teaching/case\_tools00/doc/cycle-de-vie.pdf` (consulté le 28-05-2003).

[Vic00] J.-P. Vickoff. Methode rad, elements fondamentaux. 2000. `http://www.rad.fr` (consulté le 02-08-2003).

Conférences

[EBF+98] Andy Evans, Jean-Michel Bruel, Robert France, Kevin Lano, and Bernhard Rumpe. Making UML precise. In Luis Andrade, Ana Moreira, Akash Deshpande, and Stuart Kent, editors, Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why ? How ?, 1998.

[RG98] Mark Richters and Martin Gogolla. On formalizing the UML object constraint language OCL. In Tok-Wang Ling, Sudha Ram, and Mong Li Lee, editors, Proc. 17th International Conference on Conceptual Modeling (ER), volume 1507, pages 449–464. Springer-Verlag, 1998.

[SW01] Ansgar Schleicher and Bernhard Westfechtel. Beyond stereotyping : Me-

Autres

[122] ISO/IEC 12207 :1995. Information technology - software life cycle processes.

[610] IEEE Standard 610.12-1990. Ieee standard glossary for software engineering.

[Hab02] Naji Habra. Génie logiciel la discipline et le cours, 2002.

Annexe A

Architecture du toolbox

Je reprends ici l'architecture représentant l'organisation du code du toolbox, il s'agit là d'une documentation post-développement, cette architecture ayant évolué au fur et à mesure des exigences élicitées lors des réunions. Elle ne reprend que les éléments clés du toolbox, de nombreux éléments étant liés aux particularités de ce langage. Le profil créé quant à l'administration, plus simple, n'est pas décrit ici.

A.1 Architecture associée aux outils proposés pour l'analyse

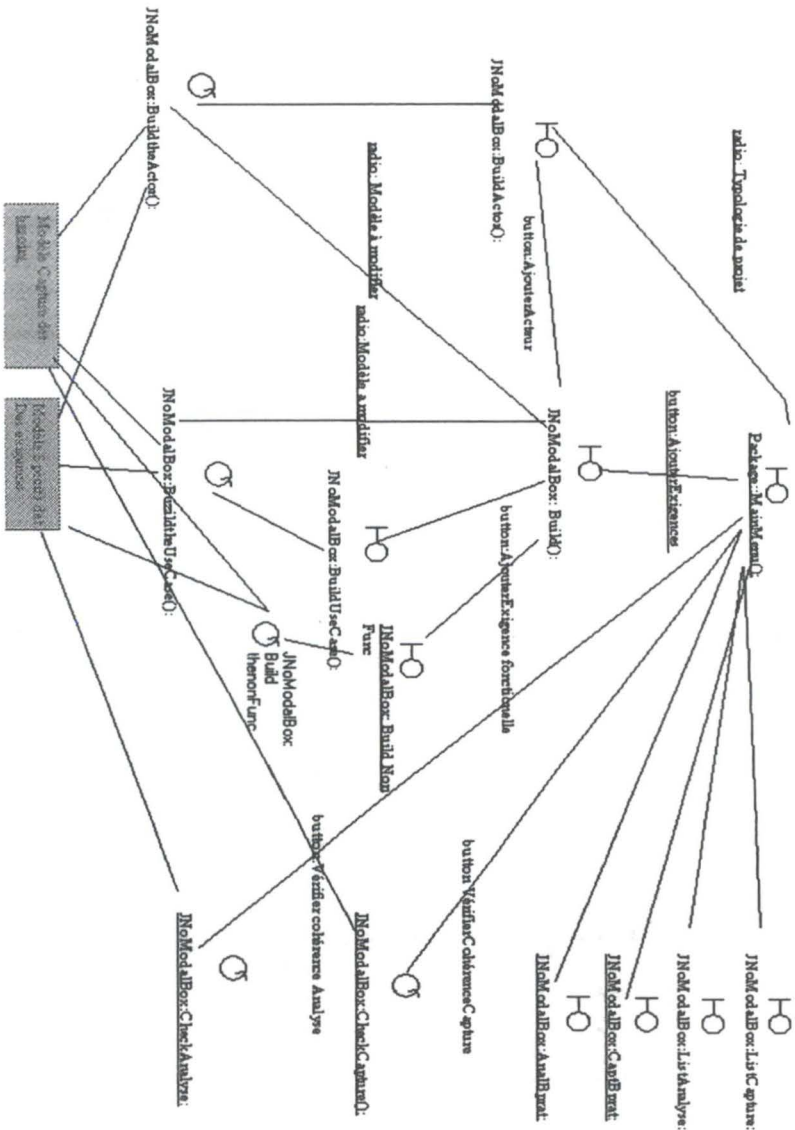


FIG. A.1 – Architecture associée aux outils proposés pour l'analyse

A.2 Architecture associée aux outils proposés pour la conception

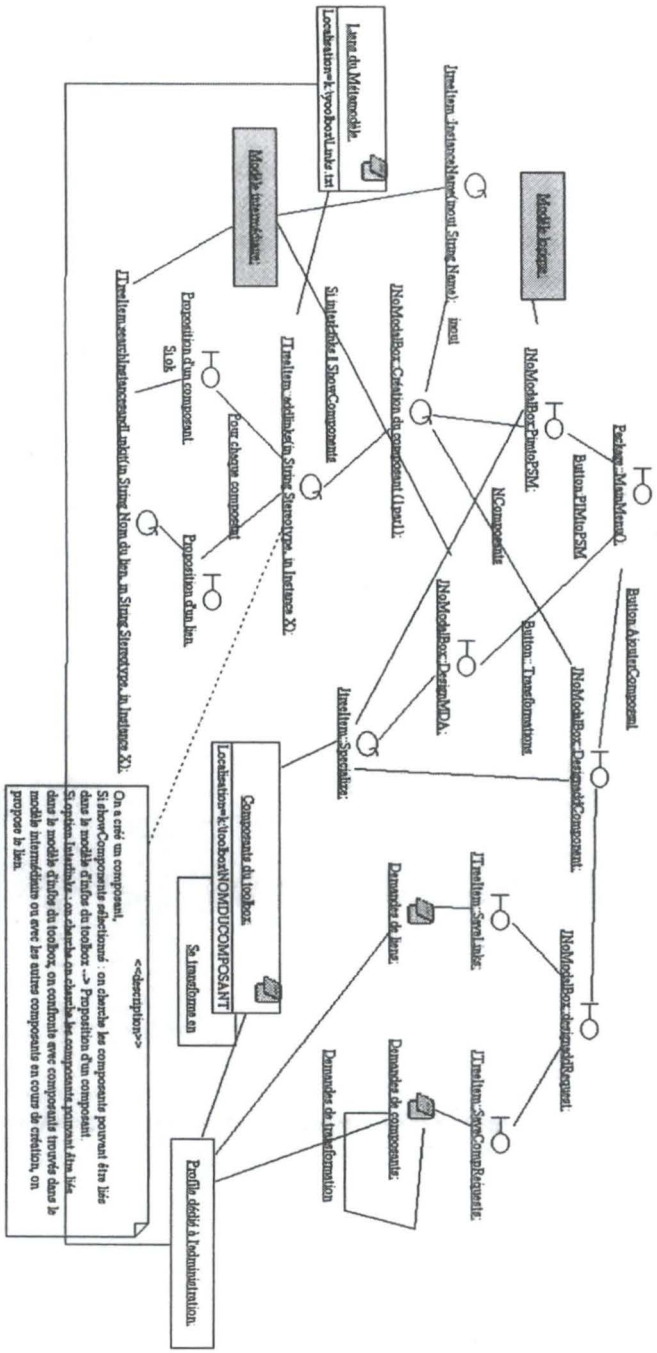


Fig. A.2 – Architecture associée aux outils proposés pour la conception

A.3 Objet des composants de l'architecture

Je fais ici une brève description de quelques méthodes clés dans un but de maintenance éventuelle bien que normalement il s'agit d'un prototype et est donc susceptible d'être jeté, les objets et méthodes spécifiques à des petits éléments graphiques ne sont pas repris ici (fastidieux), également les mécanismes d'interaction avec le modèle de l'utilisateur ne sont pas repris (facilités en langage J). Les composants de cette architecture sont des méthodes, ces méthodes sont déclarées sur des méta-éléments UML ou des méta-éléments propres à Softeam. Les paramètres de ces méthodes ne sont pas tous rendus explicites, en effet, ces méthodes utilisent des informations saisies dans certaines fenêtres ou modifient ces informations sans que ces paramètres soient décrits dans l'en-tête.

A.3.1 package :MainMenu()

Main Menu est une méthode propre au Package, un projet de l'utilisateur étant un package (contenant des diagrammes ou des packages).
Pré : un projet est initié
Objet de la méthode : L'interface principale du Toolbox est créée, celle-ci donne accès aux outils du toolbox (Cette fenêtre est un Objet "JNoModalBox"). Elle commence par l'affichage d'un menu demandant la phase en cours du projet en vue de proposer tous les outils associés à cette phase, cette fenêtre évolue donc en fonction de la phase. A la fin d'un projet, elle présente l'ensemble des outils possibles comme présenté à la figure A.4. L'utilisateur a toujours accès normalement à son projet. L'appel de cette méthode se fait via une commande l'appelant . L'utilisateur peut initier cette commande comme présenté à la figure A.3, l'accès aux outils d'administration se fait via un profil particulier.

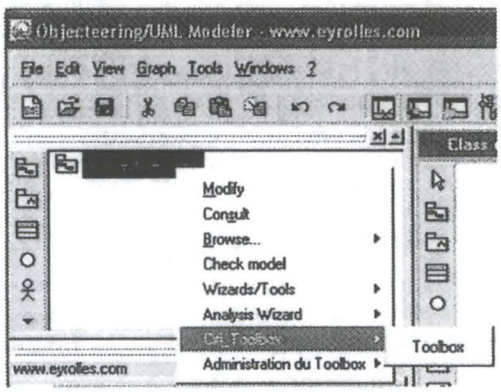


FIG. A.3 – Lancement du Toolbox

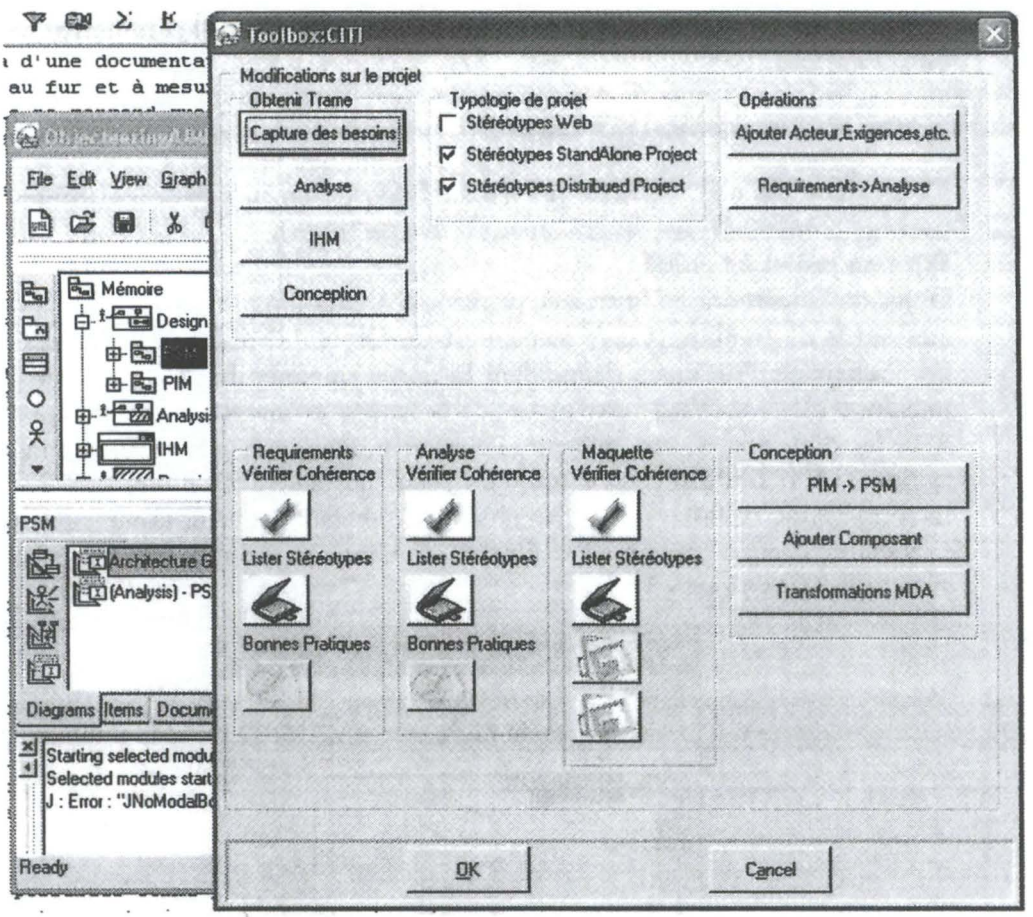


FIG. A.4 – Fenêtre principale du toolbox en fin de projet

A.3.2 JNoModalBox :Build()

Outil accessible en phase de Capture des besoins et des exigences. Cette méthode est appelée par le bouton "Créer acteur, exigences, etc." de la fenêtre créée par MainMenu().
Pré : Le projet possède les documents (vides ou non) dédiés à la phase de capture des exigences ou de spécifications détaillées des exigences.
Objet de la méthode : Elle a pour but de créer une interface dédiée au choix des outils de saisie d'une exigence fonctionnelle(BuildUseCase()), non fonctionnelle(BuildNonFunctionn ou de création d'un acteur(BuildActor()).
Cet outil étant analogue pour la phase de capture des exigences ou de spécifications des exigences, il est proposé indifféremment pour les deux phases, les boutons "radio" servant à spécifier si l'on désire ajouter l'exigence ou l'acteur dans le modèle de la première phase ou de la seconde ou encore au sein des deux.

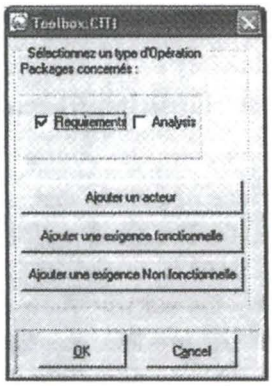


FIG. A.5 – Interface dédiée à l'accès à la saisie d'une exigence fonctionnelle, non fonctionnelle, ou d'un acteur

A.3.3 JNoModalBox :BuildActor()

Cette méthode a pour but de créer un objet JNoModalBox dédié à la saisie du modèle de description d'un acteur.
Pré : Le projet possède les documents (vides ou non) dédiés à la phase de capture des exigences ou de spécifications détaillées des exigences.
Objet de la méthode : Le modèle de description d'un acteur est présenté à l'utilisateur.

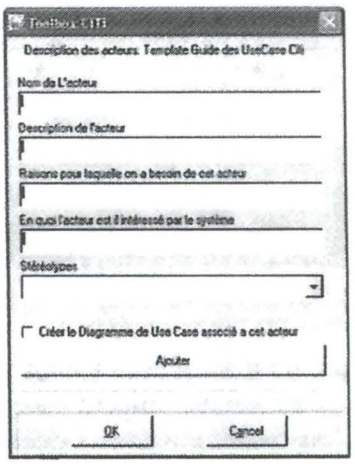


FIG. A.6 – Modèle de description d'un acteur

A.3.4 JNoModalBox :BuildtheActor()

Pré : Le projet possède les documents (vides ou non) dédiés à la phase de capture des exigences ou de spécifications détaillées des exigences.
Objet de la méthode : Le modèle de description d'un acteur, d'où est appellée cette méthode, se voit modélisé dans le projet de l'utilisateur en fonction du choix effectué au sein de Build() (capture des exigences ou spécifications détaillées des exigences).
L'acteur sera créé au sein du modèle, il sera représenté avec sa description au sein du diagramme de usecase global (reprenant tous les acteurs et toutes les exigences).
Selon les choix faits par l'utilisateur au sein de la fenêtre créée par BuildActor(), un diagramme spécifique à cet acteur sera créé et l'acteur et sa description y seront également représentés. La description est modélisée via l'élément uml Note stéréotypée «Description» associée à cet acteur. L'acteur sera éventuellement stéréotypé si l'utilisateur stipule un stéréotype particulier.

A.3.5 JNoModalBox :BuildUseCase()

Cette méthode a pour but de créer un objet JNoModalBox dédié à la saisie du modèle de description d'une exigence fonctionnelle.
Pré : Le projet possède les documents (vides ou non) dédiés à la phase de capture des exigences ou de spécifications détaillées des exigences.
Objet de la méthode : Le modèle de description d'une exigence fonctionnelle est présenté à l'utilisateur. Celui-ci présente également automatiquement l'ensemble des acteurs déjà modélisé (pour le choix d'un acteur primaire ou secondaire déclenchant cette exigence) ainsi que l'ensemble des usecase déjà modélisé, en vue de spécifier si l'exigence fonctionnelle est incluse dans un de ces use cases.

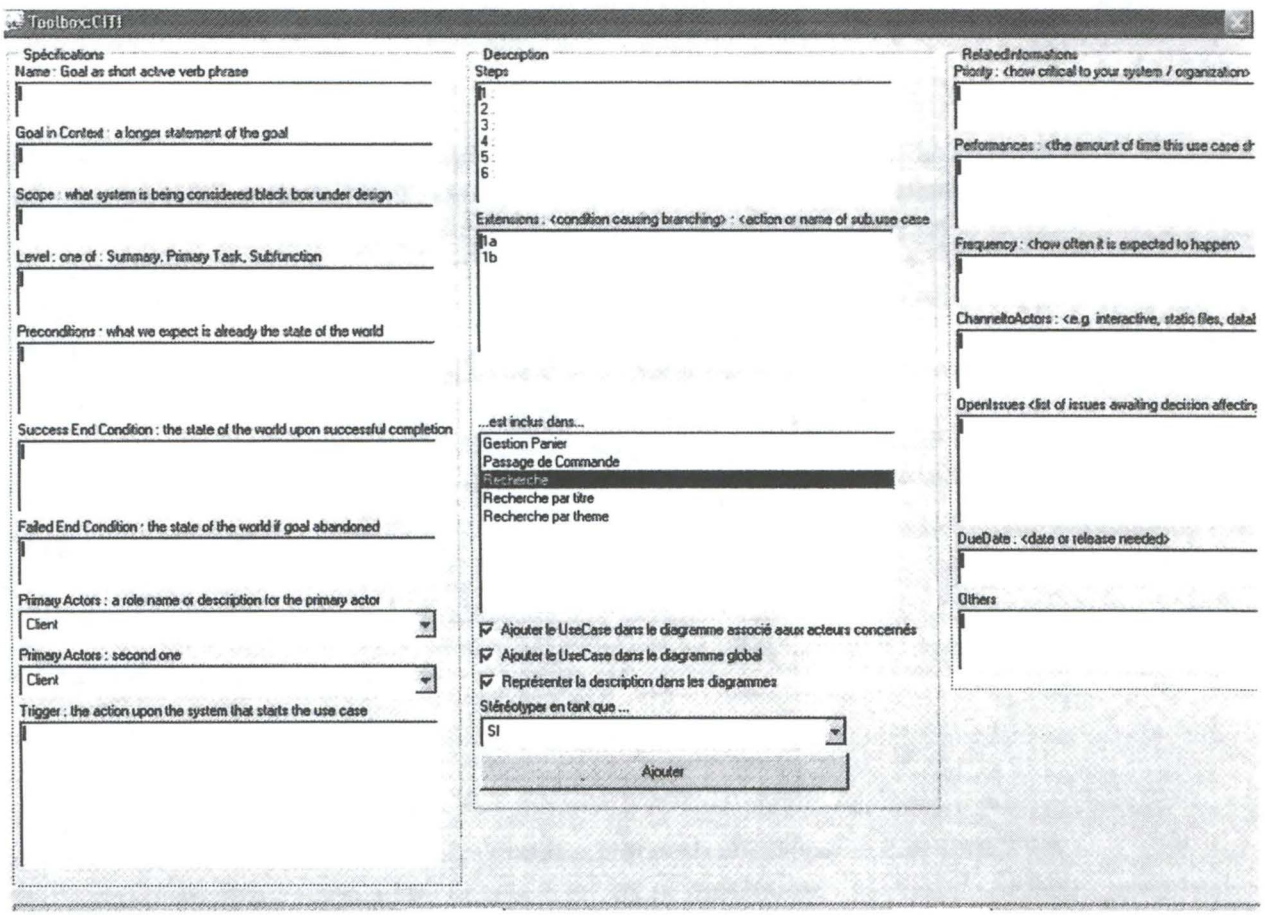


FIG. A.7 – Modèle de description d'un usecase

A.3.6 JNoModalBox :BuildtheUseCase()

Pré : Le projet possède les documents (vides ou non) dédiés à la phase de capture des exigences ou de spécifications détaillées des exigences.
Objet de la méthode : Le modèle de description appelant cette méthode se voit modélisé au sein du projet l'utilisateur en fonction du choix effectué au sein de la fenêtre créée par Build().
Selon les choix faits par l'utilisateur au sein de BuildUseCase(), l'exigence fonctionnelle sera créé au sein du modèle, elle sera représentée avec sa description via l'élément uml "usecase" au sein du diagramme de usecase global (reprennnant tous les acteurs et toutes les exigences), ce usecase sera éventuellement représenté au sein du/des diagrammes spécifiques associé à l'/aux acteur(s) déclenchant ce usecase. La description est modélisée via l'élément uml Note stéréotypée «Description» associée à ce usecase.

A.3.7 JNoModalBox :BuildNonFunc()

Pré : Le projet possède les documents (vides ou non) dédiés à la phase de capture des exigences ou de spécifications détaillées des exigences.
Objet de la méthode : Une classification d'exigence non fonctionnelle est présentée à l'utilisateur. Le choix d'un type d'exigence non fonctionnelle affichera automatiquement un petit modèle de description associé à ce type. Une liste de usecase déjà modélisés est présentée à l'utilisateur, celle-ci permet de spécifier à quel usecase se rapporte l'exigence non fonctionnelle.

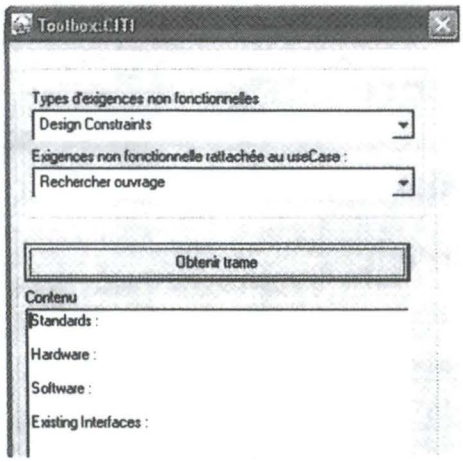


FIG. A.8 – Modèle de description d'une exigence non fonctionnelle

A.3.8 JNoModalBox :BuildtheNonFunc()

Pré : Le projet possède les documents (vides ou non) dédiés à la phase de capture des exigences ou de spécifications détaillées des exigences.
Objet de la méthode : La description de l'exigence non fonctionnelle est crée sous forme d'une note UML associée au use case (si locale à un usecase) ou au Package d'exigences (si globale au projet) et représentée au sein des diagrammes reprenant le usecase concerné. La "Note" est stéréotypée comme «TYPE de l'exigence non fonctionnelle».

A.3.9 JNoModalBox :ListCapture(), ListAnalyse(), CaptBprat(),AnalB

Ces méthodes créent des fenêtres ayant pour but de produire les informations à l'utilisateur quant aux stéréotypes disponibles pour chaque phase, ainsi que les bonnes pratiques associées aux phases.

A.3.10 JNoModalBox :checkCapture(),CheckAnalyse()

Objet de la méthode : Une interface est crée produisant à l'utilisateur la liste des problèmes rencontrés dans son modèle par rapport aux bonnes pratiques de l'entreprise et selon la phase.
(Présence de diagrammes inopportuns, présence d'opérations dans la description du domaine, aucune exigence fonctionnelle, etc.)

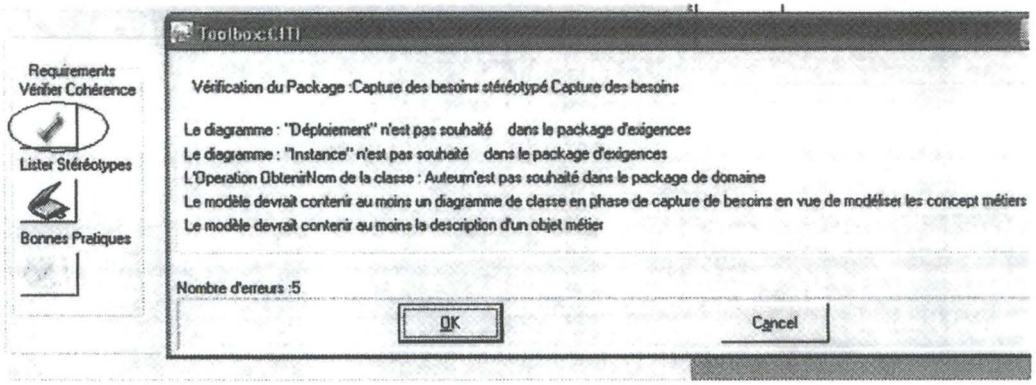


FIG. A.9 – Vérification de modèle

A.3.11 JNoModalBox :PimtoPSM()

Pré : Le projet possède les documents (vides ou non) dédiés à la phase de conception.
Objet de la méthode : Une interface reprenant dans la partie gauche, l'ensemble des éléments du modèle logique de l'utilisateur et dans la partie droite l'ensemble des composants plus ou moins spécifiques est proposée.

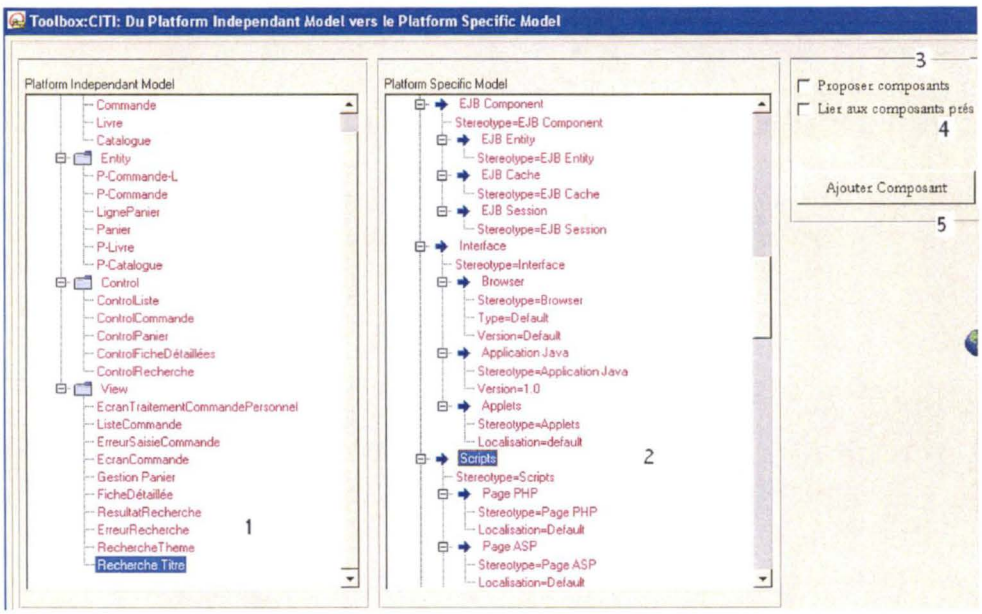


FIG. A.10 – Du modèle logique vers un premier modèle intermédiaire

A.3.12 JNoModalBox :InstanceName(inout String Name)

Pré : Le projet possède les documents (vides ou non) dédiés à la phase de conception. Objet de la méthode : Cette méthode prend un nom de composant en paramètre. Si un composant possède déjà le même nom au sein du modèle intermédiaire, elle ajoutera à celui-ci le suffixe " 1", et si un tel type de suffixe existe déjà, elle incrémentera la valeur numérique. Le nouveau nom est renvoyé à la méthode appellante.

A.3.13 JNoModalBox :Créationducomposant(1par1)(in JTreeItem[])

Pré : Le projet possède les documents (vides ou non) dédiés à la phase de conception. Objet de la méthode : Cette méthode reçoit une liste de JTreeitem, références vers des composants sélectionnés par l'utilisateur et destinés à l'ajout sur le modèle intermédiaire. Elle ajoute ceux-ci au modèle intermédiaire en ayant pris soin de vérifier que de tels composants n'existent pas déjà(appel à Instance Name), selon les options de l'utilisateur, les composants pouvant être associés seront créés et/ou les composants seront liés entre eux et avec ceux du modèle intermédiaire (appel à addlinks(in String Stereotype, in Instance X)).

A.3.14 JTreeItem : :addlinks(in String Stereotype, in Instance X)

Pré : Le projet possède les documents (vides ou non) dédiés à la phase de conception.

Si l'option showComponents ("Proposer les composants pouvant être liés") est sélectionnée : on cherche les composants pouvant être liés dans le modèle d'informations du toolbox –> Proposition d'un composant.
Si l'option Interlinks ("Essayer de lier entre eux et aux composants déjà présents") est sélectionnée : on cherche les composants pouvant être liés dans les informations du toolbox, on confronte avec les composants trouvés dans le modèle intermédiaire ou avec les autres composants en cours de création, on propose le lien.
Ces options se trouvant sur la fenêtre créée par la méthode ayant appelé la création du composant(et indirectement addLinks(...)) sinon, elles sont considérées à faux.
JTreeItem :searchInstancesandLinkit(in String Nom du lien, in String Stereotype, in Instance X) a pour rôle d'éventuellement créer le lien entre le composant nouvellement créé (Instance X) et le composant "compatible" ou bien de créer un nouveau composant (et le lien) pouvant être lié au composant nouvellement créé (Instance X).

A.3.15 JNoModalBox : :DesignMDA()

Pré : Le projet possède les documents (vides ou non) dédiés à la phase de conception.
Objet de la méthode : Cette méthode va créer l'interface proposant les transformations possibles des composants du modèle intermédiaire. Elle extrait donc des informations du modèle intermédiaire et les transformations possibles par (specialize()). La transformation est assurée par la méthode non représentée ici JtreeItem :MDAit().

A.3.16 JNoModalBox : :DesignAddcomponent()

Pré : Le projet possède les documents (vides ou non) dédiés à la phase de conception.
Objet de la méthode : Cette méthode a deux rôles , premièrement produire une interface présentant l'ensemble des composants, transformations et liens du toolbox en donnant la possibilité d'ajouter un composant au modèle intermédiaire (L'ajout d'un composant est assuré par l'appel de la méthode "Créationducomposant(1par1)".) et d'autre part représenter les demandes de nouveaux composants, liens et transformations, cette interface donne encore accès à la fenêtre créée par JNoModalBox : :designaddRequest() permettant la saisie d'une demande de composant, lien et transformation.

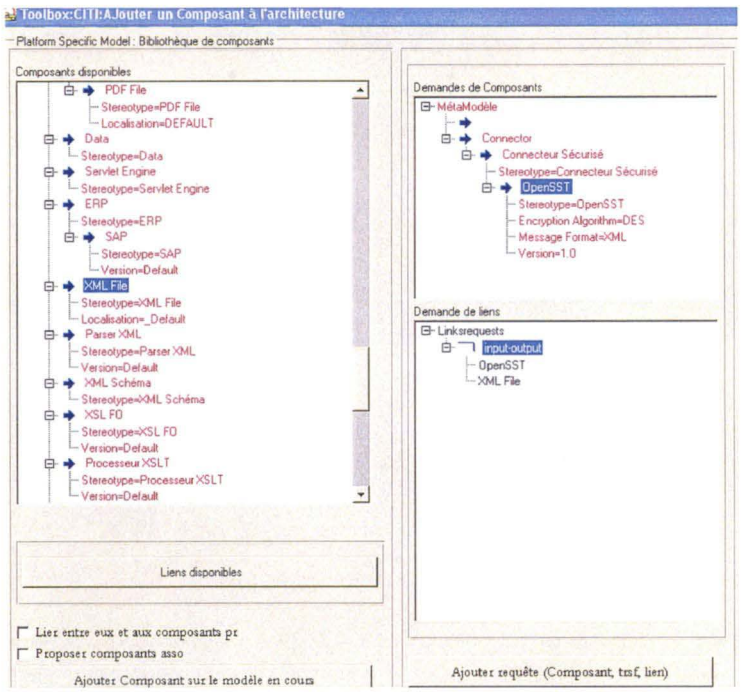


FIG. A.11 – Fenêtre crée par DesignAddcomponent()

A.3.17 JtreeItem : :Specialize(in String nom, inout RootItem Arb2)

Pré : /
Objet de la méthode : Cette méthode permet au toolbox d'alimenter les arbres de ses interfaces relatifs aux composants, demandes de composants, et transformations à partir de fichiers textes. Le code de celle-ci est repris dans la section consacrée aux extraits de code. "nom" comprend le nom du fichier racine à utiliser (Permet de distinguer s'il faut générer un arbre représentant les composants ou les demandes de composants) et arb2 est l'arbre à modifier.
Elle est également utilisée par le profile dédié à l'administration.

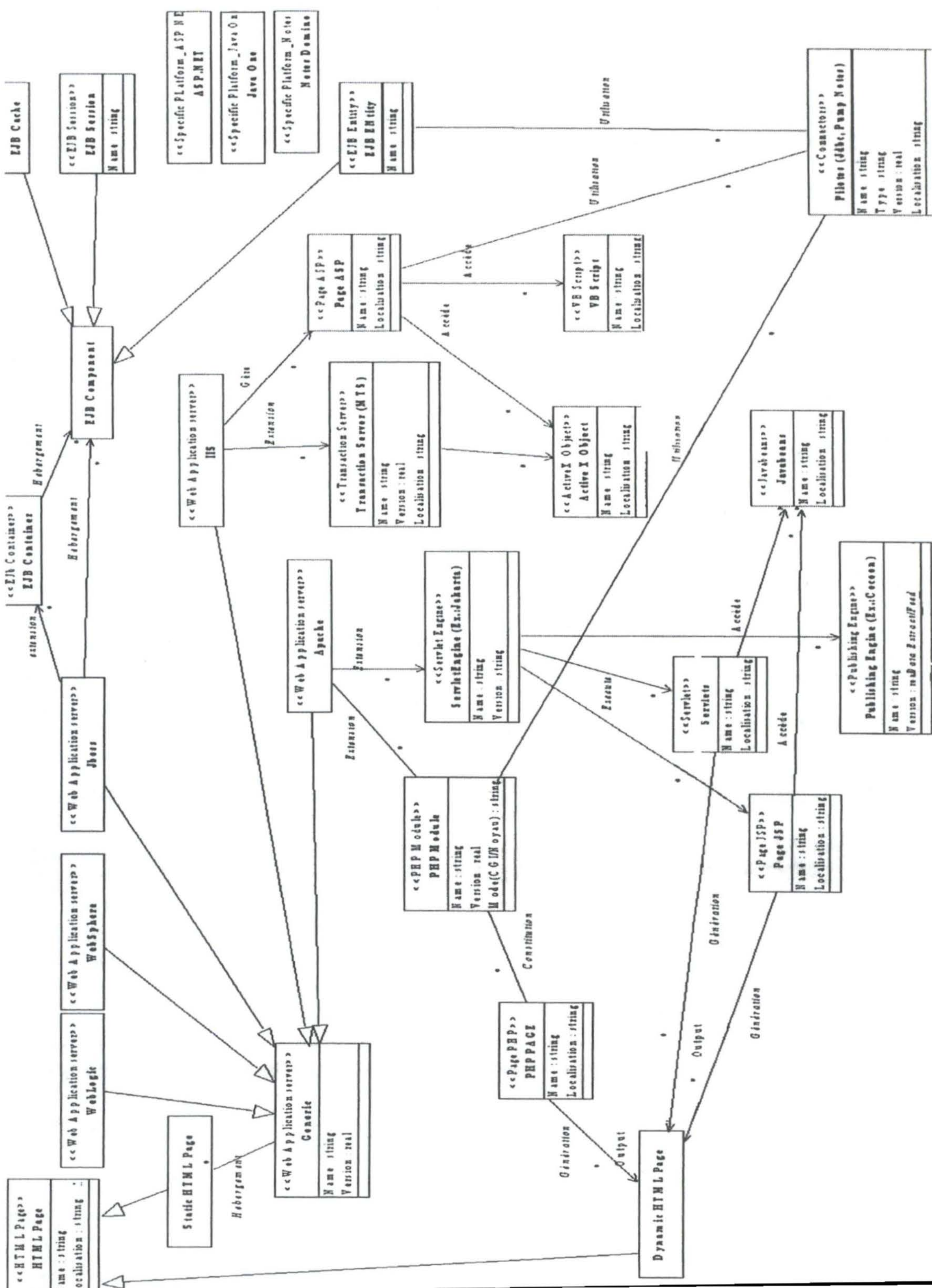
A.3.18 JNoModalBox : :designaddRequest

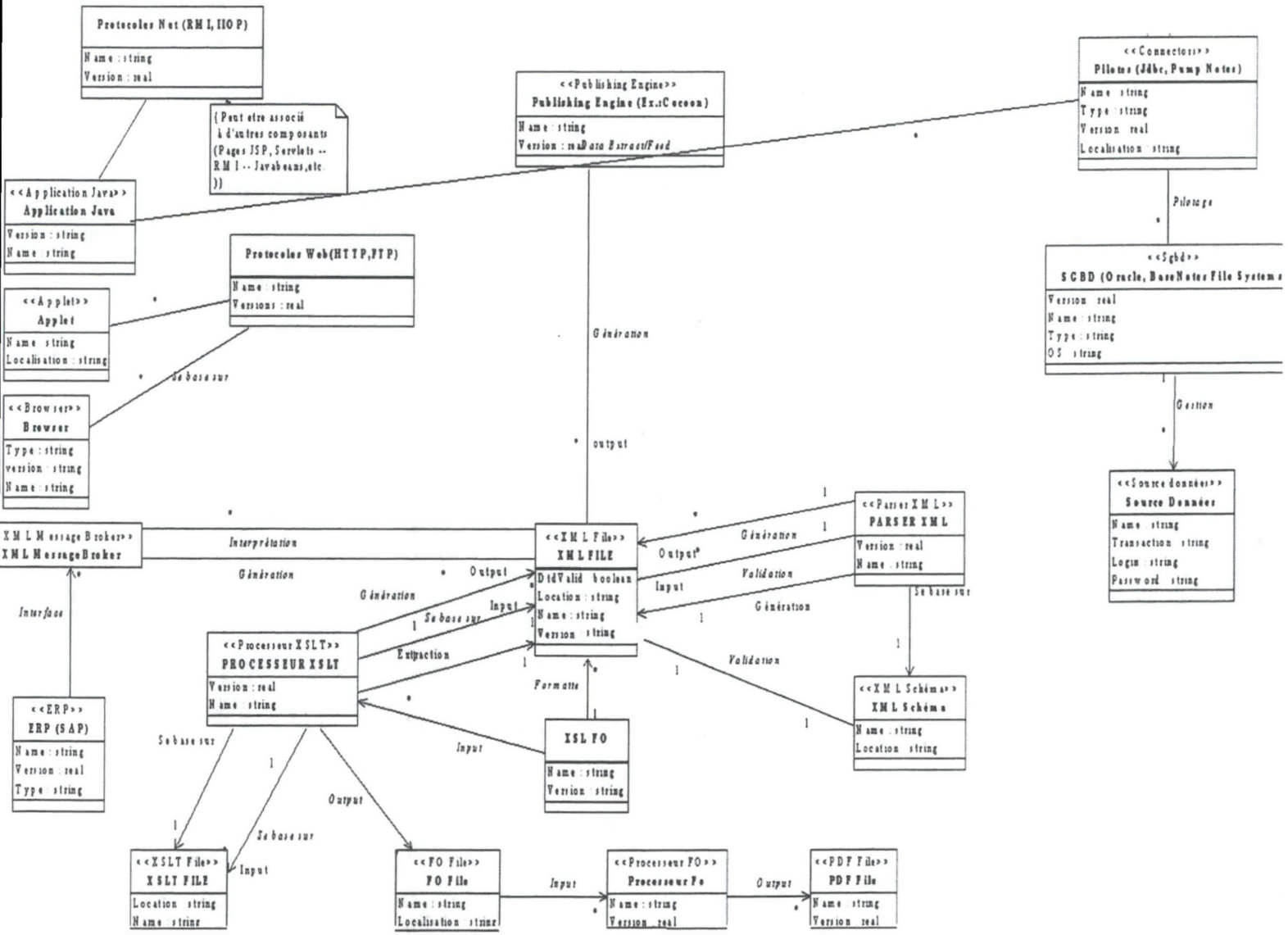
Pré : /
Objet de la méthode : Proposer l'interface de saisie de nouvelles demandes de composants, transformations et liens. Cette méthode propose à l'utilisateur de sélectionner deux composants (en cours de validation ou non) au sein de la fenêtre DesignaddComponent() et de saisir un nom en vue de créer la demande de lien. De façon analogue, l'utilisateur spécifiera la description d'un composant, ainsi que le composant (en cours de validation ou non) dont le nouveau composant est une transformation. Cette méthode créera les demandes via les appels aux méthodes JTreeItem : :SaveLinks et Jtreeltem : :SaveCompRequests.

Annexe B

Modèle d'information utilisé par l'outil proposé en phase de conception

Je décris ici le modèle d'information c'est-à-dire la représentation des informations relatives aux composants, transformations et liens entre composants sur lequel le toolbox va se baser pour offrir les outils évoqués. Ce modèle ne va donc pas servir à mémoriser des informations relatives aux composants que l'utilisateur place sur son modèle (rôle de l'outil CASE) mais à mémoriser la description de (nouveaux) composants, liens et transformations en vue de proposer ceux-ci à l'utilisateur lorsqu'il modélisera une architecture.
L'ensemble de composants, liens et transformations peut être décrit par un diagramme de classe, en associant une certaine sémantique à quelques uns de ces éléments. La sémantique est la suivante : une classe représente un composant(1Nom, 1 stéréotype, 1 ensemble de couple attributs, type), un lien représente un lien de type "Le composant x *peut* être lié au composant Y" (Un lien est déterminé par un nom), une généralisation représente une transformation possible entre composants. Pour être plus précis, une classe représente un type de composant par exemple un "Fichier XML" (Avec comme attributs : un nom, un stéréotype, un chemin) tandis qu'une instance de cette classe, un composant représente un composant particulier, par exemple "toto.xml".
Un ensemble de composants, liens et transformations provisoires placés dans le toolbox à titre illustratif est repris aux figures complémentaires B.1, ceux-ci ont servi essentiellement à la démonstration du toolbox.


EJB Cache
<<EJB Session>> EJB Session
<<Specific Platform_ASP.NET>> ASP.NET
<<Specific Platform_Java One>> Java One
<<Specific Platform_Notes>> Notes Domino
<<EJB Entity>> EJB Entity
<<Connectors>> Pilotes (Jdbc, Pump Notes)
EJB Component
<<EJB Container>> EJB Container
<<Web Application server>> IIS
<<Page ASP>> Page ASP
<<VB Script>> VB Script
<<Transaction Server>> Transaction Server (MTS)
<<ActiveX Object>> Active X Object
<<Javabeans>> Javabeans
<<Web Application server>> Jboss
<<Web Application server>> Apache
<<Servlet Engine>> ServletEngine (Ex:Jakarta)
<<Publishing Engine>> Publishing Engine (Ex:Cocoon)
<<Servlet>> Servlets
<<PHP Module>> PHP Module
<<Page JSP>> Page JSP
<<Web Application server>> WebSphere
<<Web Application server>> WebLogic
<<Web Application server>> Generic
<<Page PHP>> PHP PAGE
Static HTML Page
<<HTML Page>> HTML Page
Dynamic HTML Page
Utilisation
Accède
Gère
Extension
Hebergement
Execute
Génération
Output
Constitution


Protocoles Net (RMI, IIOP)
Name : string
Version : real
(Peut etre associé à d'autres composants (Pages JSP, Servlets -- RMI -- Javabeans,etc.))
<<Publishing Engine>>
Publishing Engine (Ex.:Cocoon)
Name : string
Version : real
Data Extract/Feed
<<Connecteur>>
Pilotes (Jdbc, Pump Notes)
Name : string
Type : string
Version : real
Localisation : string
Pilotage
<<Application Java>>
Application Java
Version : string
Name : string
Protocoles Web(HTTP,FTP)
Name : string
Version : real
<<Sgbd>>
SGBD (Oracle, BaseNotes File System
Version : real
Name : string
Type : string
OS : string
<<Applet>>
Applet
Name : string
Localisation : string
Se base sur
Génération
Gestion
<<Browser>>
Browser
Type : string
version : string
Name : string
output
<<Source données>>
Source Données
Name : string
Transaction : string
Login : string
Password : string
<<Parser XML>>
PARSER XML
Version : real
Name : string
<<XML Message Broker>>
XML MessageBroker
Interprétation
Génération
<<XML File>>
XML FILE
DtdValid : boolean
Location : string
Name : string
Version : string
Output
Input
Validation
Génération
Se base sur
Interface
<<Processeur XSLT>>
PROCESSEUR XSLT
Version : real
Name : string
Extraction
Formatte
Validation
<<XML Schéma>>
XML Schéma
Name : string
Location : string
<<ERP>>
ERP (SAP)
Name : string
Version : real
Type : string
XSL FO
Name : string
Version : string
Input
Output
<<XSLT File>>
XSLT FILE
Location : string
Name : string
<<FO File>>
FO File
Name : string
Localisation : string
<<Processeur FO>>
Processeur Fo
Name : string
Version : real
<<PDF File>>
PDF File
Name : string
Version : real

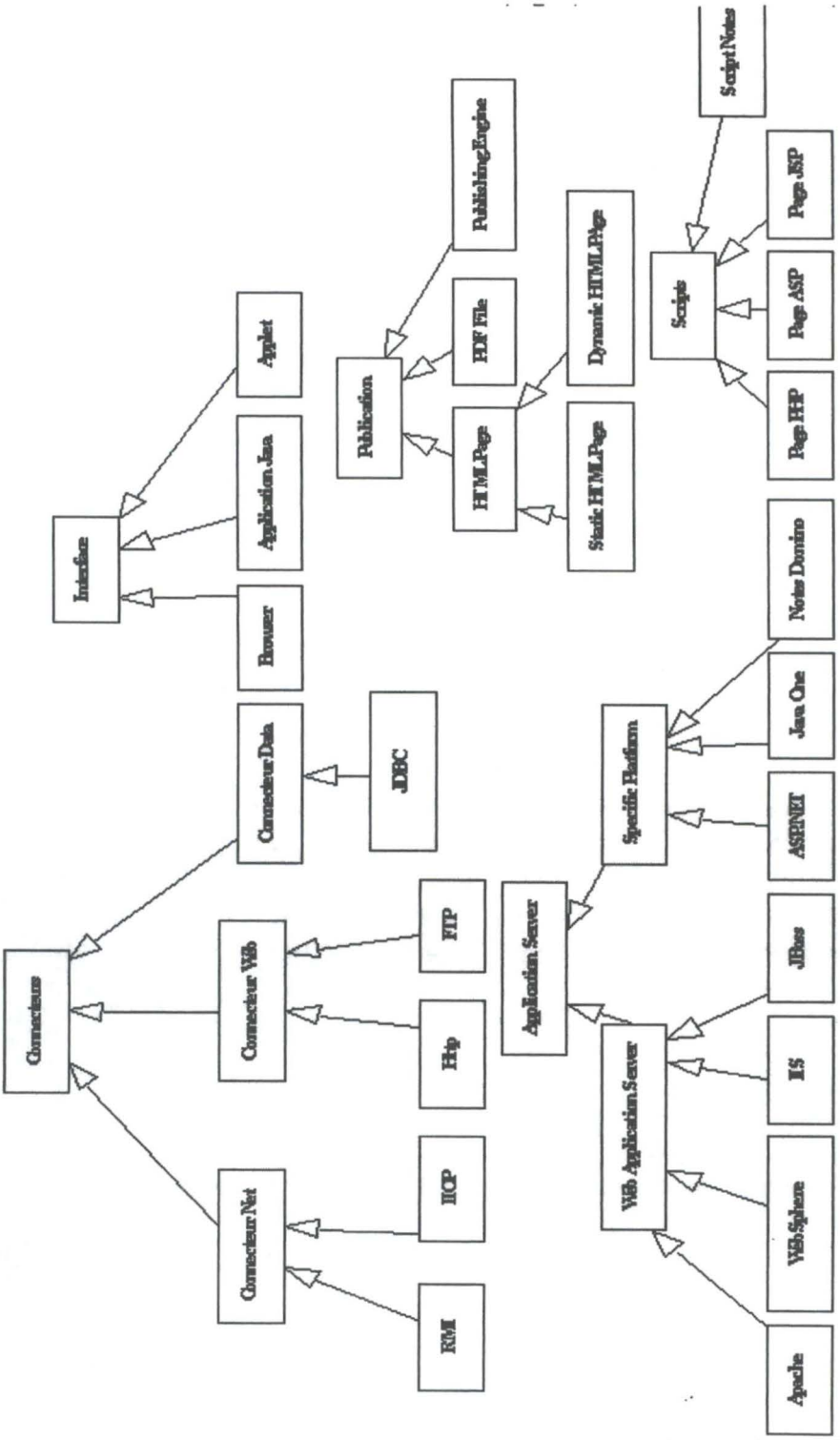


Figure B.1 (suite)

Ajouter des composants, liens et transformations au toolbox correspondra à l'ajout de classe, liens et généralisation au sein du modèle interne.
Toutefois, d'un point de vue physique, les informations ne pourront être modélisées par un diagramme de classe, en effet, il aurait été intéressant que le Toolbox travaille directement avec un tel diagramme, mais ceci est impossible techniquement étant donné qu'un diagramme de classe est spécifique à un projet et que le toolbox doit pouvoir être étendu à partir de n'importe quel projet et que ces extensions du toolbox doivent pouvoir être exploitées à partir de n'importe quel projet. Chaque projet,sinon, aurait au bout d'un certain temps un toolbox possédant ses propres composants, liens et transformations.
Il est également plus opportun de manipuler, non des classes mais des objets qui permettent, outre la description d'un type de composant, d'avoir également des valeurs par défaut pour les attributs. La figure B.2 reprend un exemple de description où le nom du type de composant est apporté par le stéréotype, un "XML File", le nom du fichier est une valeur par défaut, sa localisation, etc. Ceci est intéressant dans la mesure où on modélise des composants tel "Apache" dont le CITI possède une version particulière à telle adresse sur le serveur, etc. On peut facilement associer des informations aux composants.

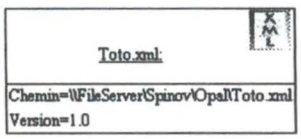


FIG. B.2 – Exemple de description d'un composant

Le modèle physique représentant ces informations sera donc particulier. Le toolbox devra traiter des fichiers textes placés sur un serveur en respectant quelques conventions. Par exemple, on peut choisir comme convention qu'un fichier texte décrit un composant (le stéréotype, l'ensemble couple de valeurs), un fichier texte peut contenir une ou plusieurs références (un chemin et un nom) vers d'autres fichiers textes décrivant d'autres composants. Cette notion de référence traduisant physiquement la relation de transformation possible entre composant. Un fichier particulier contiendra les informations relatives aux liens.

Annexe C

Quelques extraits de codes

C.1 JtreeItem : :Specialize(in String nom, inout Jtree Arb2)

```
inStream MyFile; boolean existe=false; JTreeItem tr = this;
```

Tr contient la référence vers la feuille de l'arbre à partir de laquelle sera construite la représentation des composants et transformations ou demandes de composants

```
JTreeItem str; SetOfString seg; SetOfString sseg;

int j=0; String theName="";

String Voir;

MyFile.existFile ("K:\Toolbox\"+nom+".txt",existe);

if (existe) {

MyFile.open("K:\Toolbox\"+nom+".txt"); MyFile.read(Voir);
MyFile.close();
```

Ces dernières instructions ayant pour but de vérifier l'existence d'un tel fichier racine de nom "nom" et d'en placer le contenu au sein de la variable Voir. Ce fichier racine contient la description des composants dits de haut niveau, les composants indépendants de la plate-forme et les références vers les fichiers textes associés.(ceux-ci étant des fichiers textes au même titre que le fichier racine, on va récursivement appeler la méthode en vue de trouver les transformations spécifiques de chacun de ces composants)

```
if (Voir!=""){ seg = Voir.segment(NL+"--"+NL);
```

Boucle sur chaque composant

```
seg {
sseg = this.segment("%%"+NL);
j=0;
```

Boucle sur chaque détails de description du composant (nom, stéréotype, couple attribut-valeur)

```
sseg
   {
   if (j!=0) {Arb2.addItem (str,this, "",false, j+28);j=j+1;}
   else {
        str = Arb2.addItem (tr,this, "",false,j+28);j=j+1;theName=this;
```

pouvant transformer ce composant.

```
        }
    }
    str.specialize(theName,Arb2);
```

Appel récursif vers un composant pouvant transformer le composant courant. Appel réalisé sur str, sous feuille du composant transformable.

```
}

}
```

C.2 JtreeItem : :Specialize(in String nom, inout Jtree Arb2)

```
Project projo=getCurrentProject(); Package rootpack; Package[]
rootpacks; Diagram[] diags; Package[] pack; Package[]
spack;Package Temp; Actor[] act; Communication com; Communication
com2; Diagram diagtemp; Diagram diaglobal; Diagram
diagactor;Diagram diagactor2; Actor theActor; Actor
thetwoActor;String[] seg; UseCase X; UseCase inclu; SetOfString
UCI; UseCase[] tempuc; Note nota; Package Temp2; String star
="";Stereotype st; String ZZZ;
```

Les déclarations suivants ont pour rôle de contenir les informations contenues dans l'interface dédiée à la saisie du modèle de description d'une exigence fonctionnelle

```
 String togX; String togY; String
togZ; String tog1; String tog2; String Nameof; String Goal; String
Scope; String Level; String Preconditions; String
SuccessEndCondition; String FailedEndCondition; String Actors;
String Actors2; String Trigger; String Steps; String Extensions;
String Priority; String Performances; String Frequency; String
ChanneltoActors; String OpenIssues; String DueDate; String Others;
String SuperOrdinates; String SubOrdinates; UseCaseDependency UCD;
UseCaseDependency[] ListUCD;
```

Récupération de la fenêtre du modèle de description d'une exigence fonctionnelle et récupération des informations. On récupére les choix de l'utilisateur quant aux documents à modifier (phase de capture des exigences ou de spécifications détaillées des exigences sur la fenêtre crée par Build())

```
JSet js = getJSet ("OP"); js.getValue("tog1",tog1);
js.getValue("tog2",tog2); getValue("Nameof",Nameof);
getValue("togX",togX); getValue("togY",togY);
getValue("togZ",togZ); getValue("Goal",Goal);
getValue("Scope",Scope); getValue("Level",Level);
getValue("Préconditions",Preconditions); getValue("Success End
```

```
getValue("Steps",Steps); getValue("Extensions",Extensions);
getValue("Priority",Priority);
getValue("Performances",Performances);
getValue("Frequency",Frequency);
getValue("ChanneltoActors",ChanneltoActors);
getValue("OpenIssues",OpenIssues); getValue("DueDate",DueDate);
getValue("Others",Others);
getValue("SuperOrdinates",SuperOrdinates);
getValue("SubOrdinates",SubOrdinates); getValue("star",star);
```

Ouverture d'une session destinée à à la modification du modèle de l'utilisateur

```
sessionBegin ("begin2",true);

if ((tog1=="true") && (tog2=="true")) {tog1="false";}

if (tog1=="true") {tog1="Capture des besoins";}

if (tog2=="true") {tog2="Analyse";}

getValueList ("Included", true, UCI);
```

Navigation dans le projet en vue de retrouver les documents concernés par les modifications

```
rootpack = projo.ModelPackage; rootpacks = rootpack.getPackages();
```

Boucle sur chaque Package du projet

```
rootpacks

{

Temp=this; act = getActors ();
```

L'objet courant de la boucle étant un package, getActors renverra la liste des acteurs du Package, l'instruction suivante sert à voir de quelle manière sont stéréotypés ces Packages , de façon à retrouver le package traitant de la capture des exigences et/ou de la phase de spécifications détaillées des exigences.

```
ExtensionStereotype {
          if ((Name == tog1) || (Name == tog2))
            {
              spack = Temp.getPackages();
              spack
              {
              Temp2=this;
              ExtensionStereotype
                   { if (Name=="Exigences")
```

Création du message

```
tempuc = Temp2.getUseCases() ;
```

Création de la dépendance entre ce use case et les usecases l'incluant et définition du stéréotype "include" sur cette dépendance.

```
UCI
 {
  ZZZ=this;
  tempuc
   {if (this.Name==ZZZ) {inclu=this;}}
  UCD = X.createAndAddUseCaseDependency ("", inc
  st=findStereotypeInProject ("include",
  "UseCaseDependency");
  UCD.appendExtension(st);
  ListUCD.addElement(UCD);
 }
```

Définition du stéréotype associé au usecase

```
if (star!="")
 {
 st=findStereotypeInProject (star,"UseCase");
 X.appendExtension(st);
 }
```

Création d'un note de description contenant les informations du modèle de description

```
nota = Temp2.createNote ("", "description",
"Goal"+NL+Goal+NL+
"Scope"+NL+Scope+NL+
"Level"+NL+Level+NL+
"Préconditions"+NL+Preconditions+NL+
"Success End Condition"+NL+SuccessEndCondition+N
"Failed End Condition"+NL+FailedEndCondition+NL+
"Trigger"+NL+Trigger+NL+
"Steps"+NL+Steps+NL+
"Extensions"+NL+Extensions+NL+
"Priority"+NL+Priority+NL+
"Performances"+NL+Performances+NL+
"Frequency"+NL+Frequency+NL+
"ChanneltoActors"+NL+ChanneltoActors+NL+
"OpenIssues"+NL+OpenIssues+NL+
"DueDate"+NL+DueDate+NL+
"Others"+NL+Others+NL
, "Actor"); X.appendDescriptor(nota);
```

Création du lien entre le usecase et les acteurs le déclenchant

```
act
{ if (Name==Actors) {
                    theActor=this;
```

```
                                    }
                    if (Name==Actors2) {
                                    thetwoActor=this;
                                    com2 = this.<create
                                    AndAddCommunication("", X);
                                     }
            }
```

Cette seconde partie a pour but de représenter ces divers éléments au sein du/des diagrammes des acteurs concernés et le diagramme global

```
diags = Temp2.getDiagrams();
diags
{
diagtemp=this;
seg=Name.segment("%%");
```

Le nom du diagramme permet de retrouver l'acteur concerné par ce diagramme

```
 seg
 { if (this=Actors) {diagactor=diagtemp;}
   if (this=Actors2){diagactor2=diagtemp;}
 }

 if (Name == "Global\%\%UseCase Diagram")
 {diaglobal=this;StdOut.write(diaglobal.Name);}
}
```

La partie du code suivant a été pour des raisons de présentation, décalée sur la gauche bien qu'étant toujours imbriquée dans les boucles précédentes, togY contient la valeur quant au choix de représenter ou non le usecase au sein du diagramme global

```
if (togY=="true")

{
```

Ouverture d'une session destinée à la modification du modèle de représentation graphique des informations du projet

```
diaglobal.open();
```

Ouverture du diagramme global. Représentation du lien de déclenchement entre l'acteur et le usecase, représentation de la note et du usecase selon le choix de l'utilisateur.

```
if (notVoid(theActor))
{diaglobal.createAddAndMoveViewBox(theActor,
30,30,100,100);diaglobal.createAndAddViewLink(com) ;
}
```

```
{diaglobal.createAddAndMoveViewBox(nota,
   500,200,70,200);
}
```

Représentation du usecase au sein du diagramme global

```
diaglobal.createAddAndMoveViewBox(X, 200,80,70,40);
```

Représentation du second acteur et de lien de déclenchement avec le nouveau usecase au sein du diagramme global

```
if (notVoid (thetwoActor))
{
diaglobal.createAddAndMoveViewBox(thetwoActor, 30,120,100,100);
diaglobal.createAndAddViewLink(com2);
}
```

Représentation du lien include au sein du diagramme global

```
ListUCD {diaglobal.createAndAddViewLink(this) ;}
diaglobal.close();
}
```

Si l'utilisateur a fait le choix de représenter le usecase ou non au sein des diagrammes associés aux acteurs concernés.

```
if (togX=="true")

{

    if ((notVoid (theActor)) && (notVoid (diagactor)))
    {
        diagactor.open(); diagactor.createAndAddViewLink(com) ;
```

Ouverture du diagramme représentant les exigences fonctionnelles de l'acteur concerné par le nouveau usecase, l'instruction suivant a pour but de représenter les liens "include" avec le nouveau usecase

```
        ListUCD { diagactor.createAndAddViewLink(this) ; }
        if ((notVoid (thetwoActor)) && (notVoid (diagactor2)))
            {
                diagactor2.open();
                ListUCD {diagactor2.createAndAddViewLink(this) ; }
                diagactor2.createAndAddViewLink(com2) ;
```

Représentation du usecase

```
                diagactor2.createAddAndMoveViewBox(X, 200,80,70,40);
            }
        if (togZ=="true")
           {
```

```
                (diagactor2))){diagactor2.createAddAndMoveViewBox(nota,
                                500,200,70,200);}
            }
        diagactor.createAddAndMoveViewBox(X, 200,80,70,40);
        diagactor.close();

        if ((notVoid (thetwoActor)) &&
            (notVoid(diagactor2))){diagactor2.close();}
    }

}
```

Fin de la condition (togx=true), de représentation de l'exigence au sein des diagrammes

```
                                                    }
```

Fin du code associé à la valeur vraie de la condition :
(Package.Extensionstereotype.Name="Exigences")

```
                                }
```

Fin de la boucle sur les stéréotypes associés au Package

```
                            }
```

Fin du code associé à la valeur vraie de la condition (((Name == tog1) || (Name == tog2))) désignant un package.ExtensionStereotype.Name=="Capture des besoins" ou un package.ExtensionStereotype.Name=="Spécifications détaillées des exigences"

```
 }
```

Fin de la boucle sur les Packages à la racine du Projet

```
sessionEnd();
```