UNIVERSITÉ DU QUÉBEC À MONTRÉAL

VERS UNE APPROCHE AUTOMATIQUE POUR L'EXTRACTION DES RÈGLES D'AFFAIRES D'UNE APPLICATION

MÉMOIRE
PRÉSENTÉ
COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN INFORMATIQUE
(GÉNIE LOGICIEL)

PAR
GINO CHÉNARD

OCTOBRE 2007

REMERCIEMENTS

Au terme de ce mémoire, je tiens à adresser mes remerciements et à témoigner toute ma reconnaissance aux personnes qui ont participé de près ou de loin à la réalisation de cette recherche.

Mes remerciements vont tout d'abord à mon codirecteur Ismaïl Khriss, professeur à l'Université du Québec à Rimouski pour m'avoir donné la chance de poursuivre mes études au niveau de la maîtrise en me proposant ce projet de recherche. Et surtout pour m'avoir soutenu tout au long de ce projet par sa disponibilité, sa patience, ainsi que ses nombreuses idées, ses remarques et son soutien.

Je remercie aussi mon directeur Aziz Salah, professeur à l'Université du Québec à Montréal pour m'avoir permis de poursuivre ce projet à l'UQAM. Je le remercie aussi pour ses remarques et son aide dans la rédaction de ce mémoire et son soutien.

Je tiens également à remercier les professeurs en informatique de l'UQAM pour la qualité de leur enseignement lors de ma maîtrise. Tout comme je tiens à remercier le personnel du département de mathématique, d'informatique et de génie de l'UQAR pour m'avoir accueilli et accordé des contrats durant les mois passés avec mon codirecteur à Rimouski et pour leur enseignement lors de mon baccalauréat.

Je remercie aussi les membres du jury qui par leurs commentaires éclairés m'ont aidé à améliorer la qualité de ce mémoire.

Finalement et non les moindres, je tiens à remercier mes parents qui m'ont toujours aidé et soutenu lors de mes études.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

ADL	Langage de description d'architecture (Architecture Description Language)
API	Interface de programmation (Application Programming Interface)
AST	Arbre syntaxique abstrait (Abstract Syntax Tree)
BMP	Persistance gérée au niveau du Bean (Bean Management Persistence)
CASE	Génie logiciel assisté par ordinateur (Computer Aided Software Engineering)
CDIF	Format d'échange entre environnements CASE (CASE Data Interchange Format)
CMP	Persistance gérée au niveau du conteneur (Container Management Persistence)
CVS	Système de contrôle de versions (Concurrent Versions System)
DAO	Objets d'accès aux données (Data Access Object)
EJB	Enterprise Java Beans
GFRN	Réseau générique de raisonnement flou (Generic Fuzzy Reasoning Nets)
HTML	Langage de balisage d'hypertexte (HyperText Markup Language)
JSP	Java Server Page
MDB	EJB basé sur les messages (Message Driven Bean)
OFBS	Service de courtage financier d'OTN (OTN Financial Brokerage Service)
OMT	Méthode OMT (Object Modeling Technique)
OSI	Modèle de référence OSI (Open System Interconnection)
OTN	Oracle Technology Network
PME	Petite et Moyenne Entreprise
RPC	Appel de procédure à distance (Remote Procedure Call)
SQL	Langage SQL (Structured Query Language)
UCM	Méthode UCM (Unified Change Management)
UML	Langage UML (Unified Modeling Language)
VSM	Centre d'achat virtuel (Virtual Shopping Mall)
WSL	Langage WSL (Wide Spectrum Language)
XMI	Langage XML (XML Metadata Interchange)
XML	Standard d'échange d'informations UML basé sur XML (EXtensible Markup Language)

RÉSUMÉ

Les compagnies font face à d'énormes coûts pour maintenir leurs applications informatiques. Au fil des ans, le code de ces applications a accumulé des connaissances corporatives importantes (règles d'affaires et décisions de conception). Mais, après plusieurs années d'opération et d'évolution de ce code, ces connaissances deviennent difficiles à récupérer. Les développeurs doivent donc consacrer beaucoup de leur temps à l'analyser : une activité connue sous le nom de « compréhension du logiciel ». Comme il a été estimé que cette activité accapare entre 50 % et 90 % du travail d'un développeur, simplifier le processus de compréhension du logiciel peut avoir un impact significatif dans la réduction des coûts de développement et de maintenance.

L'une des solutions au problème de compréhension du logiciel est la rétro-ingénierie. Celle-ci est le processus d'analyse du code source d'une application pour (1) identifier les composantes de l'application et les relations entre ces composantes et (2) créer une représentation de haut niveau de l'application. Plusieurs approches ont été proposées pour la rétro-ingénierie ; cependant, la représentation abstraite du code source extraite par la plupart de ces approches combine la logique d'affaires de l'application et son architecture (ou son infrastructure).

Dans ce mémoire, nous présentons une nouvelle approche qui permet d'analyser le code source d'une application orientée objet afin d'en extraire un modèle abstrait ne décrivant que les règles d'affaires de cette application. Ce modèle prend la forme d'un diagramme de classes UML, présentant les classes d'affaires de cette application ainsi que les relations entre ces classes. Cette approche a été validée sur plusieurs systèmes (écrits en Java) de différentes tailles. L'approche donne de bons résultats pour les systèmes possédant une bonne architecture et un bon style de programmation. Dans le cas contraire, les résultats sont moins convaincants.

INTRODUCTION

Les compagnies font face à d'énormes coûts pour maintenir leurs applications informatiques du fait qu'après plusieurs années d'opération, les règles d'affaires et les décisions de conception que contiennent ces applications deviennent difficiles à récupérer. Les développeurs doivent consacrer entre 50 % et 90 % de leur temps à analyser le code de ces applications (Pressman, 2001), une activité connue sous le nom de « compréhension du logiciel ». Une approche qui permet de simplifier le processus de compréhension du logiciel peut donc avoir un impact significatif dans la réduction des coûts de développement. Une solution possible à la compréhension du logiciel est d'employer la rétro-ingénierie afin de créer une représentation abstraite du logiciel (Chikofsky, 1990).

MOTIVATION

Une des abstractions intéressantes pouvant être obtenues grâce à la rétro-ingénierie représente les règles d'affaires d'une application. Dans un contexte général, les règles d'affaires sont définies comme des formulations concises de la structure et du comportement d'affaires décrivant le fonctionnement d'une organisation (Earls, Embury et Turner, 2002). Un exemple de règle d'affaires serait : « À la fin de chaque mois, un client est marqué comme inactif s'il n'a pas fait d'achat depuis douze mois ». D'un point de vue plus informatique, les règles d'affaires sont une description des manipulations de données exprimée en termes de domaine d'entreprise ou logiciel (Sneed et Erdos, 1996). Très peu de travaux ont tenté d'extraire les règles d'affaires d'une application. La difficulté vient du fait qu'il est difficile de distinguer entre le code relié à cette logique d'affaires de celui ajouté par les besoins de l'infrastructure utilisée pour implanter cette application. Cette infrastructure est le résultat de l'adoption d'une architecture et du choix de la plateforme d'implantation.

Généralement, l'adoption du premier (c'est-à-dire l'architecture) a une incidence sur le choix du second (c'est-à-dire la plateforme d'implantation).

OBJECTIFS ET CONTRIBUTIONS

Le but de ce mémoire consiste à extraire les règles d'affaires d'une application orientée objet. Pour cela, nous proposons une nouvelle approche de rétro-ingénierie qui permet d'analyser son code source afin d'en extraire un modèle abstrait ne décrivant que les règles d'affaires de cette application. Ce modèle prend la forme d'un diagramme de classes UML présentant les classes d'affaires de cette application ainsi que les relations entre ces classes.

L'approche est fondée sur l'analyse des identificateurs contenus dans le code source de l'application traitée. L'analyse du nom des classes, de leur hiérarchie et de leurs méthodes permet de distinguer le code relié à la logique d'affaires de celui ajouté par les besoins de l'infrastructure utilisée pour implanter cette application. Un prototype d'outil supportant notre approche a été développé et accepte présentement les systèmes écrits en Java. Le prototype a été validé sur plusieurs systèmes de différentes tailles et donne de bons résultats pour ceux possédant une bonne architecture et un bon style de programmation.

ORGANISATION DU MÉMOIRE

Dans le premier chapitre, nous discutons de la pratique de la compréhension du logiciel. Nous commençons tout d'abord par présenter la problématique rendant nécessaire cette compréhension du logiciel. Ensuite, nous montrons comment cette compréhension permet de représenter un logiciel et les rôles qu'elle peut jouer. Ce chapitre se termine par les enjeux auxquels la compréhension du logiciel doit faire face.

Le second chapitre est consacré à l'allié naturel de la compréhension du logiciel, la rétro-ingénierie. Il débute par un cadre de classification permettant de cerner cette

pratique. Le chapitre se concentre à décrire en détail les différentes dimensions de ce cadre, qui sont : les données en entrées, les traitements effectués, les langages intermédiaires utilisés pour l'analyse et les résultats. Nous concluons ce chapitre par un résumé des exemples présentant le cadre de classification vue au cours de ce chapitre.

En suite logique au second chapitre, le troisième chapitre est consacré à présenter diverses approches de rétro-ingénierie afin de montrer un large éventail de ce qu'est effectivement cette pratique en regard de la compréhension du logiciel. Après avoir présenté ces approches, nous concluons ce chapitre par une discussion sur ces approches mettant en évidence aussi bien leurs points faibles que leurs points forts et l'utilité de certaines de ces approches par rapport à nos objectifs.

Au quatrième chapitre, nous abordons la notion d'architecture dans les logiciels, car elle est centrale à l'approche que nous proposons dans le chapitre suivant. Le chapitre débute par quelques définitions sur l'architecture. Par la suite, nous discutons les représentations possibles d'une architecture. En particulier, nous expliquons comment nous voyons la relation qui existe entre l'architecture et des notions comme les stratégies et les mécanismes architecturaux, les patrons d'architecture et de conception, ainsi que les cadres d'application. Nous concluons ce chapitre par une présentation, comme exemple d'illustration, de l'architecture J2EE.

Au cinquième chapitre, nous présentons la raison d'être de ce mémoire, l'approche que nous avons développée. Ce chapitre débute par la présentation d'un exemple d'une petite application qui est utilisée pour illustrer notre approche. Le code source de cet exemple est présenté à l'appendice A. Nous présentons par la suite quelques définitions utiles pour la compréhension de cette approche. Finalement, nous expliquons en détail ses étapes. Le pseudo-code de l'algorithme, supportant notre approche, est donné à l'appendice B.

Nous présentons au sixième chapitre le processus de validation. En premier lieu, nous présentons l'outil que nous avons développé pour implanter notre approche. Par la suite, nous parlons du protocole que nous avons suivi lors de notre processus de validation. Puis, nous donnons les résultats obtenus sur un banc d'essai composé de plusieurs systèmes de différentes tailles. Nous concluons ce chapitre par une discussion de quelques conclusions tirées de notre processus de validation.

Nous terminons ce mémoire par une conclusion générale où nous faisons une synthèse de nos recherches et décrivons ce qui sera à faire comme suite à nos travaux.

CHAPITRE I

LA COMPRÉHENSION DU LOGICIEL

1.1 INTRODUCTION

Un logiciel, à l'image de l'environnement dans lequel il s'intègre, demeure rarement dans un état fixe tout au long de sa vie. Il doit la plupart du temps faire face à une certaine évolution. Comme, pour le moment, les logiciels ne peuvent évoluer par eux-mêmes, ce sont les humains qui sont les acteurs de cette évolution. Or ces logiciels sont souvent très complexes et leur évolution n'est pas une tâche facile. Les acteurs du domaine ont donc besoin d'aide afin de comprendre les logiciels auxquels ils s'attaquent (Storey, 2005).

Dans ce chapitre, nous allons d'abord mettre en évidence les problématiques rendant aujourd'hui nécessaire la compréhension du logiciel. Ensuite, nous allons montrer les formes qu'elle peut prendre et les aspects des logiciels qu'elle peut mettre en valeur. Nous conclurons ce chapitre en présentant les enjeux auxquels elle doit faire face.

1.2 PROBLÉMATIQUE

Au cours de leur vie, les logiciels sont amenés à évoluer. Cette évolution peut être notamment rendue nécessaire afin de corriger des erreurs ou pour s'adapter à de nouvelles exigences ou de nouvelles technologies. Un exemple très connu d'évolution qui a touché plusieurs logiciels est le fameux bogue de l'an 2000. À première vue, la correction de ce problème pouvait paraître facile. Il s'agissait de s'assurer que les logiciels comptent bien les années avec quatre chiffres. Anciennement, il était fréquent pour économiser de la mémoire de les enregistrer

avec seulement deux chiffres par exemple 99 pour 1999. Si la problématique était si claire, sa correction l'était moins. En effet pour la mettre en œuvre, il fallait souvent comprendre des logiciels datant souvent de quelques décennies et écrits en employant des langages et des techniques que peu de gens maîtrisaient encore.

Dans un contexte idéal, les logiciels sont supposés être bien documentés. Par conséquent, la tâche de compréhension du logiciel est aisée. Malheureusement en pratique, la documentation est souvent incomplète, insuffisante ou elle n'est pas à jour. En l'absence d'une documentation adéquate, la maintenance logicielle devient difficile. Pour combler le manque de documentation, les intervenants passent alors un temps non négligeable à comprendre les logiciels. Cette activité accapare entre 50 % et 90 % du travail d'un développeur (Pressman, 2001; Richner et Ducasse, 1999). Pour diminuer l'effort lié à la maintenance, il est donc nécessaire de simplifier cette tâche.

Un premier moyen pour simplifier la compréhension est de faciliter la navigation dans le code source du logiciel. Il est reconnu qu'il est la source de documentation la plus disponible et qu'il est le reflet le plus fiable de l'implantation. Les environnements de développement modernes tels qu'Eclipse (D'Anjou *et al.*, 2004) et Visual Studio.Net (Platt, 2002) offrent de telles fonctionnalités pour faciliter la navigation dans le code source. Ils permettent par exemple de localiser rapidement la déclaration d'une fonction. Malheureusement, cette pratique présente des limites pour les grands logiciels. La raison principale est la taille de leur code source.

Devant cette quantité d'information, il faut donc chercher à simplifier et à cibler l'information à afficher. En revenant au problème du bogue de l'an 2000, nous pouvons constater par exemple qu'il serait pratique de ne pouvoir afficher que les parties du code traitant les dates, un modèle représentant les données traitées et leur format ou bien encore un diagramme illustrant les manipulations faites sur les variables représentant des dates.

C'est donc devant ces limites que la pratique de la compréhension du logiciel prend son importance. Rugaber, Stirewalt et Wills (Rugaber, Stirewalt et Wills, 1995) la définissent comme le processus d'acquisition de connaissances au sujet d'un logiciel. Cette connaissance permet des tâches telles que la correction d'erreurs, l'amélioration, la réutilisation et la documentation.

1.3 REPRÉSENTATIONS D'UN LOGICIEL

La compréhension du logiciel passe en général par une représentation du logiciel analysé. Souvent, il s'agit d'une représentation architecturale de celui-ci. Une représentation architecturale d'un logiciel illustre ses composantes, les relations entre celles-ci et un ensemble d'attributs caractérisant ces composantes et leurs relations (Kazman et Carrière, 1999). En Java, les composantes sont par exemple les classes, les interfaces ou les méthodes. Par exemple, une fonction appelle une autre fonction `n` fois, la classe `A` hérite de la classe `B`. Nous reviendrons en détail sur le concept d'architecture logicielle au chapitre 4.

Les représentations peuvent afficher des informations statiques au sujet d'un logiciel. Il s'agit en général d'informations tirées directement du code source de celui-ci. Les représentations architecturales statiques en UML peuvent prendre la forme de diagrammes de classe, de diagrammes d'objets, de diagrammes de composantes ou de diagrammes de déploiement. Une représentation peut aussi être dynamique. Dans ce cas, elle se base aussi sur les composantes du logiciel, mais utilise en plus des informations sur le fonctionnement du logiciel comme des traces de l'exécution, des informations sur l'exécution en parallèle, l'usage de la mémoire, la couverture du code source. Les représentations dynamiques en UML peuvent prendre la forme de diagrammes de séquence, de diagrammes de collaboration, de diagrammes d'états ou de diagrammes d'activités. Il est aussi possible de retrouver des représentations combinant les aspects statiques et dynamiques. Le fait de combiner des informations statiques et dynamiques dans une seule représentation permet notamment d'identifier

les relations entre les différentes composantes d'un logiciel qui ne se produisent qu'au moment de l'exécution (Claudio et Rodriguez, 2002). Par exemple, une représentation peut servir à montrer quels objets d'un logiciel instancient quels objets lors de son exécution. Ainsi, si on découvre que pour un logiciel une instance d'une classe crée la majorité des autres instances lors de l'exécution, on en déduira que cet objet est central (Ducasse, Lanza et Bertuli, 2004).

Les représentations peuvent donner des vues globales sur un logiciel. Mais, il est souvent plus utile d'en avoir une représentation plus ciblée. Par exemple, il est possible de représenter un logiciel par son diagramme de classes dans la notation UML. Mais si le système est moindrement gros ou complexe, la représentation ne serait pas tellement plus facile à comprendre que le code source. C'est ici que le concept d'abstraction prend son importance. Faire une abstraction consiste à retirer les détails inutiles à une tâche de compréhension particulière. Par exemple, la représentation pourra chercher à filtrer les classes utilitaires et ne conserver que celles concernant la logique d'affaires. Ceci représente une abstraction d'un niveau plus élevé.

1.4 OBJECTIFS DE LA COMPRÉHENSION DU LOGICIEL

Dans cette section, nous allons décrire des rôles généraux pouvant être réalisés avec l'aide de la compréhension du logiciel et qui sont : la redocumentation, la redécouverte de la conception, la détection des règles d'affaires, l'aide à la maintenance, l'aide à la formation et les aspects légaux.

1.4.1 Redocumentation

La redocumentation a pour but de créer une représentation sémantiquement équivalente et au même degré d'abstraction que la représentation originale (Chikofsky, 1990). Comme exemple de redocumentation, il est possible de prendre un diagramme de classes créé à partir d'un outil comme Together Architect (Borland,

2005) pour représenter le code source d'un logiciel. Il sera sémantiquement équivalent et représentera le même niveau d'abstraction, car ce diagramme ne fait que représenter le code source sous une autre forme.

1.4.2 Redécouverte de la conception

L'objectif de la redécouverte de la conception est de créer des représentations à un plus haut niveau d'abstraction (Chikofsky, 1990). Selon Biggerstaff, elle recrée une abstraction de la conception à l'aide du code source, de la documentation de conception existante (si disponible), de l'expérience personnelle, des connaissances générales au sujet du problème et du domaine du logiciel (Biggerstaff, 1989). Donc en plus de décrire comment le logiciel est conçu, ce type de compréhension du logiciel utilise des techniques permettant de créer des abstractions afin de faciliter la compréhension. Par exemple, notre approche analyse le code source d'une application afin de découvrir les objets d'affaires qu'il contient en identifiant les différentes classes représentant son implantation.

1.4.3 Détection des règles d'affaires

Les règles d'affaires contenues dans un logiciel sont une abstraction intéressante à identifier. Dans un contexte général, les règles d'affaires sont définies comme des formulations concises de la structure et du comportement d'affaires décrivant le fonctionnement d'une organisation (Earls, Embury et Turner, 2002). Voici un exemple d'énoncé de règle d'affaires : « À la fin de chaque mois, un client est marqué comme inactif s'il n'a pas fait d'achat depuis douze mois ».

D'un point de vue plus informatique, les règles d'affaires sont une description des manipulations des données exprimée en termes de domaine d'entreprise ou logiciel (Sneed et Erdos, 1996). La règle d'affaires de l'exemple précédent, serait identifiable dans un logiciel par le fait qu'un traitement sur chaque client est effectué à minuit le premier de chaque mois. Nous aurions dans le logiciel du pseudo-code semblable à

celui de la figure 1.1. Lors de ce traitement si la date de la dernière commande de ce client remonte à plus de douze mois, son champ inactif est placé à vrai.

En sortant du monde des affaires, il est possible d'affirmer que les règles d'affaires représentent ce que fait le logiciel. Parfois dans le contexte de l'entreprise, elles ne sont décrites nulle part ailleurs que dans le code source. Elles sont aussi intéressantes à identifier, car elles aident à comprendre les fonctionnalités d'un logiciel sans se soucier de ses détails d'implantation.

1.4.4 Aide à la maintenance

Nous avons déjà vu que la compréhension d'un logiciel peut aider à détecter ses composantes. Parfois, ces composantes peuvent être isolées assez précisément pour être réutilisées dans d'autres logiciels. La compréhension du logiciel permet aussi l'analyse de l'impact des changements en aidant à détecter les entités affectées (Desclaux et Ribault, 1991). De même, elle peut proposer des solutions aux problèmes soulevés lors de la modification d'un logiciel, un outil peut ainsi proposer des changements à effectuer sur les composantes affectées par ces changements. Par exemple, un outil d'aide à la compréhension du logiciel pourrait utiliser une base de connaissances lors du portage d'un logiciel d'une bibliothèque graphique à une autre, souligner les composantes du code devant être changées et même suggérer comment effectuer ces changements (Desclaux et Ribault, 1991).

```
IF isFirstDayOfMonth() and isMidnight()
 FOR each client in clientList DO
  IF
   client.getLastCommand().getAgeInMonth
   s() > 12
  THEN
   client.setInactive(true)
  ENDIF
 ENDFOR
ENDIF
```

Figure 1.1 Pseudo-code représentant une règle d'affaires

La compréhension du logiciel permet aussi la traçabilité des exigences en partant d'une définition existante des exigences et en tentant de découvrir les parties du code où elles ont été implantées (Antoniol *et al.*, 2002). Ainsi, en cas de changements des exigences, cette traçabilité permettrait de connaître quelles parties du code doivent être modifiées.

La compréhension d'un logiciel permet de mieux organiser un logiciel (Chikofsky, 1990). Cette tâche est parfois appelée réorganisation structurelle, restructuration ou consolidation. Cette tâche consiste à transformer la structure d'un logiciel tout en restant au même niveau d'abstraction sans en changer le fonctionnement externe (Lu *et al.*, 2002; Magdalena *et al.*, 1999). Il est évident que si on ne comprend pas le logiciel, cette tâche est difficile à réaliser. Le réusinage (« refactoring ») est la forme objet de la réorganisation structurelle (Fowler, 1999; Mens et Tourwé, 2004). Par exemple, il peut être utilisé afin de redistribuer entre les classes du logiciel les diverses responsabilités. Car, il est reconnu qu'il est une mauvaise pratique d'attribuer trop de fonctions à une seule classe (« god class ») (Riel, 1996).

1.4.5 Aide à la formation

La compréhension du logiciel peut aussi aider à la formation. Des outils permettant d'explorer le code source d'un logiciel sont très utiles afin de faciliter l'adaptation à un nouveau logiciel. De même, la compréhension du logiciel peut aider les étudiants à acquérir des connaissances en informatique en leur permettant de comprendre des logiciels existants (Eilam, 2005; Hajitaheri, Gaffar et Kline, 2003).

1.4.6 Aspects légaux

La compréhension du logiciel peut même aider à détecter des crimes. Grâce à elle, il devient possible de détecter le plagiat ou l'utilisation de code volé (Ducasse, 1999; Eilam, 2005). Et bien entendu, une certaine forme de compréhension du logiciel se retrouve du côté illégal. Par exemple, certains se servent d'outil de compréhension du

logiciel afin de percer les protections contre la copie de logiciel. Malheureusement, c'est cette forme qui est la plus connue du grand public.

Si elle peut détecter des crimes, elle permet aussi d'améliorer la sécurité d'un logiciel ou de détecter des atteintes à sa sécurité (Eilam, 2005). Par exemple, un usage utile de la compréhension du logiciel se pose dans le cas des virus et des logiciels malveillants. En étudiant la structure et le fonctionnement de tel code, les spécialistes peuvent prévoir des dommages présents et futurs que ces logiciels peuvent créer et trouver des façons afin de les détecter et les annihiler.

1.5 LES ENJEUX DE LA COMPRÉHENSION DU LOGICIEL

La compréhension du logiciel n'est pas un domaine simple. Elle doit faire face à plusieurs enjeux, qui peuvent varier selon l'objectif recherché et le domaine auquel elle est appliquée. Nous allons présenter six enjeux auxquels doit faire face la compréhension du logiciel ; à savoir l'hétérogénéité des logiciels, la complexité des nouveaux logiciels, l'absence de l'information de conception, le niveau de détails de l'information, l'intégration aux environnements de développement, et finalement l'adaptation aux étapes du cycle de vie d'un logiciel.

1.5.1 Hétérogénéité des logiciels

Il est difficile de créer la méthode ou l'outil ultime, car il y a une multitude d'anciens logiciels basés sur des technologies diversifiées (Aiken, Muntz et Richards, 1994). Une technique fonctionnant bien avec un langage de programmation ne sera pas nécessairement adaptée à un autre langage. De plus, un même logiciel peut-être construit avec plusieurs technologies ce qui complique encore la tâche.

Certaines personnes pensent que l'aide à la compréhension du logiciel est particulièrement utile aux anciens logiciels. Mais en fait, des logiciels de tout âge peuvent en nécessiter. Ainsi, il y a beaucoup de segments de code récent qui

manquent cruellement de documentation. Comme les anciennes méthodes sont adaptées aux anciennes technologies, les nouvelles technologies vont continuellement nécessiter de nouvelles méthodes.

1.5.2 Complexité des nouveaux logiciels

Les nouveaux logiciels sont souvent orientés objet. Or, contrairement à ce qui est pensé couramment, les logiciels orientés objet peuvent à l'occasion compliquer la compréhension du logiciel. Par exemple, les aspects dynamiques comme le polymorphisme ne peuvent être détectés facilement avec une analyse statique. Ils imposent de procéder à une analyse dynamique (Systä, 1999).

De plus, les nouveaux logiciels incorporent des fonctionnalités plus diversifiées. Lors de leur analyse, il faut donc faire face à une plus grande quantité d'informations à décortiquer. C'est ce qu'évoque Booch (Booch, 2004) en affirmant qu'une des raisons de la crise permanente que vit le monde du logiciel est la fuite en avant vers la complexité des applications. La complexité intrinsèque de chaque nouvelle génération de systèmes est plus grande de sorte que les solutions techniques ne sont plus adaptées. Par exemple, décrire toutes les fonctions que peut réaliser un ordinateur sous Windows XP en 2006 demanderait plus de description que la description des fonctionnalités d'un autre ordinateur qui fonctionnait sous MS-DOS en 1982.

1.5.3 Absence de l'information de conception

Le code source souvent ne conserve pas entièrement l'information de conception. Même si la conception de départ est excellente, des détails à son sujet peuvent être perdus lors de l'implantation. En conséquence, il peut être difficile de recréer la conception originale en se basant sur le code source. Par exemple, des caractéristiques pouvant être simplement décrites en UML ne peuvent pas être reflétées dans le code source. Ainsi, l'agrégation et la composition peuvent être

implantées de façon similaire. Les cardinalités peuvent être aussi perdues dans le code même si elles sont présentes dans les documents de conception. De même, des détails du code source peuvent ne pas retrouver leur représentation dans le langage de plus haut niveau. Par exemple, les propriétés du C# qui combinent des aspects à la fois des champs et des méthodes (Archer et Whitechapel, 2002). Le fait que le code source soit souvent la seule source d'information fiable rend ce problème beaucoup plus complexe (Paul *et al.*, 1991).

1.5.4 Niveau de détails de l'information

L'information qui doit être présentée à l'utilisateur doit être limitée et ne doit être ni trop complexe ni trop simpliste. Le but de la compréhension du logiciel est de présenter de l'information à l'utilisateur. Mais, il est important de ne pas l'assaillir sous un déluge d'informations. Il faut donc cibler l'information à présenter, ce qui est le but de l'abstraction.

En effet, l'information présentée doit être compréhensible pour l'utilisateur moyen. Si elle est trop complexe et nécessite un niveau d'expertise trop élevé pour être utilisée, la compréhension du logiciel n'a pas atteint son but. À l'inverse, il est inutile de simplifier trop l'information. Par exemple à l'extrême, nous pourrions dire d'un logiciel : « il imprime des factures ». Ce serait vrai, mais s'il n'y a pas de représentations plus détaillées, ça ne sera pas d'une très grande aide à l'utilisateur. Ainsi, il faut que l'information présentée et son utilisation soient à la portée de l'utilisateur (Lanza et Ducasse, 2003; Wuyts, 1998).

1.5.5 Intégration aux environnements de développement

Les outils doivent aussi être bien intégrés aux environnements de développement afin qu'ils soient couramment utilisés. Par exemple, si un outil est visible sous la forme d'un bouton dans la barre d'outils d'un environnement de développement, l'utilisateur va être plus enclin à l'utiliser. Ensuite, il sera plus facile d'indiquer à

l'outil les données à utiliser pour son analyse. De plus, il est souhaitable d'harmoniser son utilisation à celle de l'environnement de développement. L'intégration des outils de compréhension du logiciel avec les outils de développement nécessite une normalisation favorisant les échanges entre les outils (Hajitaheri, Gaffar et Kline, 2003; Müller *et al.*, 2000).

1.5.6 Adaptation aux étapes du cycle de vie d'un logiciel

La compréhension du logiciel doit s'adapter et être utile aux différentes étapes du cycle de vie du logiciel. Elle permet d'abord l'approche circulaire du génie logiciel (« Round trip engineering »). Cette approche consiste à automatiser la transformation de modèles en code source et l'inverse (Kollman *et al.*, 2002). L'assurance-qualité peut aussi tirer parti de la compréhension du logiciel afin notamment de détecter des pratiques nuisibles dans les méthodes de programmation (Magdalena *et al.*, 1999). Particulièrement si on utilise le concept d'anti-patrons (Brown, Malveau et McCormick, 1998). De plus, la traçabilité tout au long du cycle de vie du logiciel est une pratique à encourager qui minimiserait le recours à la compréhension du logiciel ou l'amoindrirait. Ce qui peut impliquer de faire de la « micro compréhension du logiciel » à chaque étape plutôt que d'attendre d'être face à un logiciel complexe avec aucun lien apparent entre ses différents artefacts (Müller *et al.*, 2000). Dans la suite de cette sous-section, nous traiterons de quelques apports de la compréhension du logiciel dans différentes étapes du cycle de vie du logiciel.

Premièrement, à l'étape de spécification des exigences qui consiste à déterminer ce que le logiciel devra faire, la compréhension du logiciel peut être utile. Par exemple, dans le cas où un ancien logiciel effectuait en totalité ou en partie la tâche qu'un nouveau doit faire, nous pouvons employer la compréhension du logiciel afin de comprendre ce que l'ancien effectuait afin de l'implanter dans le nouveau. De même si le logiciel a à interagir avec des logiciels déjà en place, elle permettra d'aider à

connaître les tâches que le nouveau logiciel aura à faire pour communiquer avec eux (Eilam, 2005).

Deuxièmement, à l'étape de conception où l'architecture est créée, documentée et vérifiée par rapport aux exigences, la compréhension du logiciel peut notamment permettre de découper le projet afin de le diviser le travail entre différentes équipes (Müller *et al.*, 1993).

Troisièmement, il y a l'étape d'implantation, où le logiciel est implanté d'après la conception établie à l'étape précédente. Lors de cette étape, la compréhension du logiciel peut servir au programmeur à s'assurer que le code qu'il vient d'écrire fait vraiment ce qu'il devrait. Elle peut aussi permettre de vérifier que la structure du logiciel respecte bien ce qu'elle devrait être d'après la conception. Par exemple, un programmeur déboguant un logiciel pourrait identifier les variables avec des valeurs incorrectes et ensuite isoler les parties du logiciel qui affectent la valeur de ces variables. Ce qui permet au développeur de trouver plus facilement où les valeurs incorrectes sont affectées aux variables (Sward et Chamillard, 2004).

Quatrièmement, à l'étape de test, les résultats de l'étape précédente sont évalués et intégrés. Ils sont ensuite vérifiés afin de s'assurer qu'ils respectent bien les exigences. La compréhension du logiciel intervient ici, car elle permet de vérifier ce respect des exigences. Un outil pourrait permettre de vérifier si le logiciel fait bien les opérations demandées, en permettant de s'assurer que les exigences se reflètent bien dans le code source. Certains outils peuvent résumer les opérations effectuées par certaines parties du logiciel (Abd-El-Hafiz et Basili, 1996). Dans un cas idéal, les spécifications des exigences auxquelles le logiciel doit répondre auraient été écrites dans un langage compréhensible par un outil. Ce même outil pourrait extraire du logiciel implanté les spécifications vraiment réalisées. Ce qui permettrait de vérifier que les deux correspondent. Ainsi, l'outil REV-SA (Cooper *et al.*, 2004) extrait des spécifications de l'implantation d'un logiciel qu'il compare avec celles de conception.

Cinquièmement, il y a l'étape d'installation et de validation. À cette étape, le logiciel est intégré dans son environnement de production et testé dans cet environnement afin de vérifier qu'il se comporte selon les exigences. La compréhension du logiciel peut être utilisée dans le cas où un logiciel aurait été modifié afin de déterminer quelles parties ont été modifiées et afin de déterminer ce qui doit être installé, configuré et testé à nouveau (Tip, 1995).

La sixième et dernière étape du cycle abordée ici est la maintenance. À cette étape, le logiciel est modifié après sa livraison afin de corriger des problèmes, améliorer certaines de ses caractéristiques ou l'adapter à des changements de son environnement. Il est évident que l'aide directe qu'apporte la compréhension du logiciel à la maintenance est la compréhension de ce que le logiciel réalise et la façon dont il le fait. La compréhension du logiciel favorise la maintenance adaptative, par exemple en facilitant l'identification de composantes réutilisables (Chikofsky, 1990; Lu *et al.*, 2002) ou en permettant de détecter les impacts et ce qui adviendrait lors de changements potentiels (Chikofsky, 1990). Elle peut être aussi un atout lors de maintenance correctrice, car en permettant une meilleure compréhension du logiciel, il est plus facile de détecter et de corriger les erreurs. De même, la maintenance d'amélioration perfective peut en tirer profit avec des outils facilitant notamment la restructuration du code (Magdalena *et al.*, 2000) (parfois en l'automatisant), la détection de faiblesses de conception (détection d'anti-patrons) (Brown et Wallnau, 1996; Brown, Malveau et McCormick, 1998; Bruce, Mike et Patrick, 2003; Smith et Williams, 2003) et la détection (parfois élimination) de code dupliqué (clones) (Magdalena *et al.*, 2000). Lors de la maintenance évolutive, la compréhension du logiciel permet de repérer les changements à apporter au logiciel, de même que leurs conséquences et la manière de les effectuer (Desclaux et Ribault, 1991). Finalement, durant l'étape de retrait, la compréhension du logiciel peut être utile afin d'aider à développer ses successeurs.

1.6 CONCLUSION

Dans ce chapitre, nous avons mis en évidence l'importance de la compréhension du logiciel. Nous avons aussi montré les formes qu'elle peut prendre et les aspects qu'elle peut mettre en valeur. Finalement, nous avons présenté les enjeux auxquels la compréhension des logiciels doit faire face afin d'y répondre. Dans le chapitre suivant, nous allons présenter la pratique de la rétro-ingénierie pour la mise en œuvre de la compréhension du logiciel.

CHAPITRE II

CADRE DE CLASSIFICATION DES APPROCHES DE RÉTRO-INGÉNIERIE

2.1 INTRODUCTION

Dans le chapitre précédent, nous avons abordé des problématiques présentant l'importance de la compréhension du logiciel. Nous avons vu les enjeux et les objectifs auxquels cette pratique doit faire face.

Dans ce chapitre, nous aborderons la rétro-ingénierie : une pratique permettant d'exercer la compréhension du logiciel. Nous la présenterons au travers de notre cadre de classification des approches de rétro-ingénierie. Ce cadre contient quatre dimensions de la rétro-ingénierie. La première dimension représente les données en entrées, c'est-à-dire les sources pouvant servir au processus de rétro-ingénierie. La deuxième dimension est constituée des traitements effectués sur ces données. Elle se base en partie sur la classification des méthodes de rétro-ingénierie de Gannod et Cheng (Gannod et Cheng, 1999). La troisième dimension englobe les représentations intermédiaires utilisées pour la rétro-ingénierie. La quatrième et dernière dimension englobe les objectifs et les tâches ciblées par la rétro-ingénierie.

La rétro-ingénierie du logiciel est, selon Chikofsky (Chikofsky, 1990), le processus d'analyse d'un logiciel ayant pour objectif d'identifier les composantes et les relations entre les composantes de ce logiciel et d'en créer une représentation de haut niveau (à un niveau plus élevé d'abstraction). La pratique de la rétro-ingénierie est donc précieuse pour la compréhension du logiciel. Il est intéressant de remarquer que si l'objectif est seulement de faciliter la compréhension, et non une restructuration. La précision n'a pas toujours à être totale (Anquetil et Lethbridge, 1998).

La figure 2.1 résume notre cadre de classification de la rétro-ingénierie tout en l'illustrant de plusieurs exemples. À gauche, nous retrouvons les données en entrées, en bas il y a les traitements, en haut nous voyons les représentations intermédiaires et à droite sont montrés les objectifs. Les flèches représentent le flux des différents échanges entre ces dimensions.

2.2 LES DONNÉES EN ENTRÉES

La première dimension de notre cadre de classification représente les données utilisées comme source d'information primaire. La rétro-ingénierie peut se baser sur plusieurs sources d'information afin de réaliser ses objectifs. Elle peut aussi combiner différentes sources d'information primaires afin de faire ressortir d'autres informations. Ces sources de données peuvent prendre plusieurs formes. Dans la suite de cette section, nous allons présenter six exemples de données en entrées : le code source, les bases de connaissances, les traces d'exécution, l'exécutable, la documentation et finalement, les intervenants dans la vie d'un logiciel tels que les acteurs et les experts du domaine.

2.2.1 Code source

Le code source est une description du logiciel écrite dans un langage de programmation spécifique. Cette description sert à la création du logiciel exécutable. Le code source est considéré généralement comme la représentation la plus fiable du logiciel contrairement à d'autres, comme les documents de conceptions qui n'ont peut-être pas été respectés lors de l'implémentation. C'est pourquoi il est la source d'information la plus fréquemment employée. Parmi les langages employés se retrouvent : C/C++, Java, C# et Visual Basic.

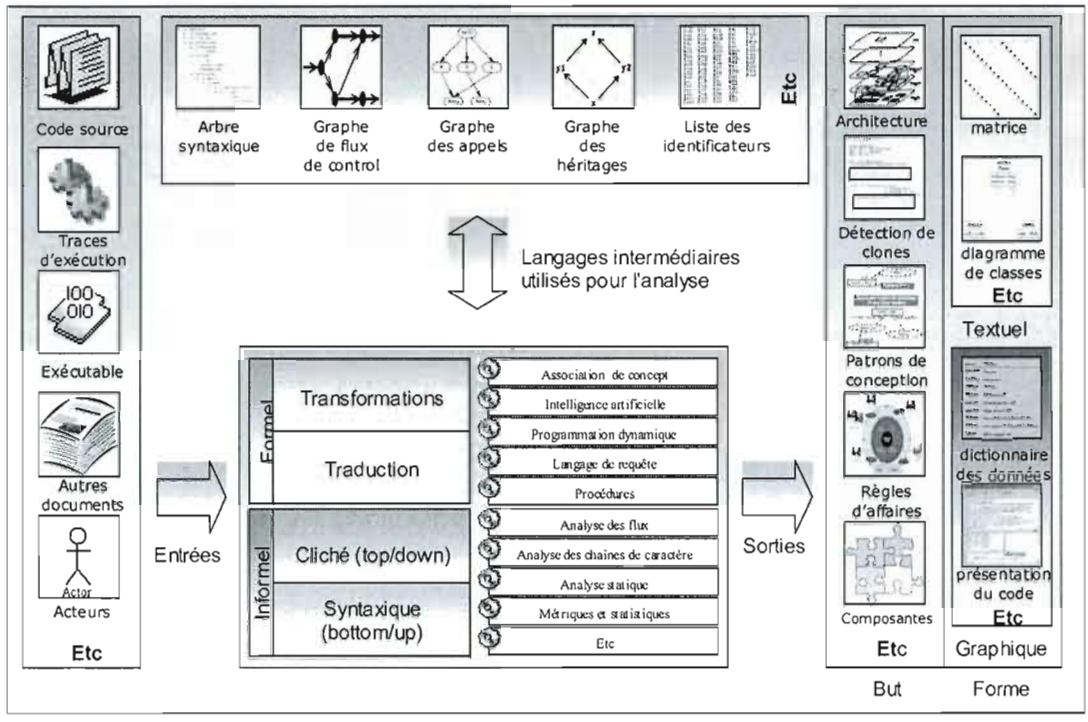

Figure 2.1 Cadre de classification de la rétro-ingénierie

D'autres données reliées au code source, autres que celui-ci, peuvent contenir des informations intéressantes. Par exemple, le nom des fichiers sources et les commentaires (Anquetil et Lethbridge, 1997).

2.2.2 Bases de connaissances

Certains algorithmes de rétro-ingénierie se fondent sur des bases de connaissances afin de guider leurs analyses. Ces bases peuvent notamment prendre la forme de connaissances sur des aspects de la programmation. Par exemple, une telle base de connaissances pourrait contenir des informations sur l'impact d'un changement effectué sur une composante. Ces connaissances peuvent aussi être concentrées sur un domaine particulier non nécessairement relié à l'informatique, mais au domaine du logiciel. Par exemple, un outil de rétro-ingénierie pourrait avoir une connaissance du

domaine des feuilles de paie qu'il cherche à relier au code source d'un logiciel traitant la paie.

2.2.3 Traces d'exécution

Les traces d'exécution sont une autre source d'information souvent employée. Elles peuvent par exemple enregistrer d'une façon ordonnée la création des objets ou les appels des fonctions. L'analyse des traces d'exécution peut être particulièrement utile pour retrouver certaines informations concernant le polymorphisme (Lanza et Ducasse, 2003).

L'inconvénient majeur de cette source de données est qu'il est difficile de s'assurer que toutes les exécutions possibles sont reflétées dans les traces. Par contre, la représentation sera toujours exacte pour la séquence d'exécution analysée. Cela peut être utile dans le cas où nous voudrions mettre en évidence certaines fonctionnalités du logiciel. Par exemple, dans le cas du tranchage (« slicing ») – qui est une pratique visant à cacher les parties du code n'étant pas en relation avec une composante du logiciel – il est intéressant de n'afficher que les lignes du code d'un logiciel pouvant affecter une certaine variable (Ducasse, Lanza et Bertuli, 2004).

2.2.4 Exécutable

L'exécutable lui-même est une source d'information parfois utile pour la rétro-ingénierie. L'exécutable est le fichier écrit en langage machine qui permet l'exécution du logiciel. Mieux que le code source, il reflète exactement le logiciel. Par contre, il est en général plus difficile à analyser que le code source. Une exception notable est le code octet Java (« byte code ») dont la structure le rend relativement facile à analyser (Bowman, Wgodfrey et Holt, 1999).

2.2.5 Documentation

La documentation en rapport direct avec le logiciel peut être une source utile d'information. Elle peut notamment prendre la forme de documents d'analyse ayant servi à la création du logiciel (Layzell, Freeman et Benedusi, 1995), de la documentation plus générale du logiciel comme les guides d'utilisateurs (Lu *et al.*, 2002) ou les requêtes de changement qui permettent de suivre l'évolution du logiciel (Layzell, Freeman et Benedusi, 1995).

Diverses documentations du domaine n'étant pas en rapport direct avec la conception du logiciel peuvent aussi être utiles à sa compréhension. Ainsi, l'utilisation de documents assez généraux en rapport avec l'organisation comme le rapport aux actionnaires a été testée. Ces documents permettent d'isoler des concepts du domaine pouvant être associés à des composantes du logiciel (Layzell, Freeman et Benedusi, 1995).

2.2.6 Intervenants dans la vie d'un logiciel

Les différents intervenants dans la vie d'un logiciel incluant les acteurs, les experts du domaine, les experts des technologies de l'information et les utilisateurs sont aussi une excellente source d'information (Layzell, Freeman et Benedusi, 1995). L'étude de l'interaction entre les humains et le logiciel est parfois aussi employée. Cela consiste à identifier la façon dont le logiciel est utilisé par les acteurs. Des diagrammes de transitions d'états peuvent ainsi être produits (Liu, Alderson et Qureshi, 1999).

2.3 LES TRAITEMENTS

Les sources de données doivent être manipulées afin d'être utiles. C'est ce que représente la deuxième dimension de notre cadre, à savoir les traitements. Cette dimension vise à déterminer la manière dont les approches sont implantées afin

d'obtenir les résultats souhaités. Les traitements sont divisés suivant deux critères. Le premier critère est le type de traitement employé, alors que le second est la technique employée par le traitement afin d'arriver à sa fin (Lu *et al.*, 2002).

2.3.1 Types des traitements

Le type de traitement se base en partie sur la classification de Gannod et Cheng (Gannod et Cheng, 1999). Cette dimension cherche à montrer une classification possible de méthodes employées afin de manipuler les données. Les types de traitements sont soit informels soit formels. Ces deux catégories sont à leur tour divisées en deux sous-catégories comme nous allons voir par la suite.

Les approches informelles sont fondées sur un ensemble de méthodes de détection de plans (ou clichés) ou sur des techniques de regroupements basées sur la structure syntaxique du code, c'est-à-dire en analysant le code tel qu'il est écrit.

Les approches informelles basées sur la détection des plans reposent sur la recherche de patrons dans le code afin d'identifier des plans (Gannod et Cheng, 1999). Un plan représente un concept lié au domaine d'application du logiciel (par exemple la comptabilité) ou au domaine de la programmation. Ainsi, nous pouvons chercher dans un code l'implantation d'un cliché particulier. C'est ce qui est appelé les techniques descendantes (« top-down »). Cela consiste à analyser le domaine du logiciel pour y découvrir des concepts et ensuite rechercher des parties du code correspondant à ces concepts. Un exemple de concept, au niveau du domaine de programmation, serait une liste chaînée. Au niveau du domaine du logiciel, un exemple de concept serait dans le cas d'un logiciel de graphisme, un zoom sur une section de l'image. Au niveau de la conception, un exemple de concept serait le patron de conception « Observer ».

La deuxième sous-catégorie d'approches informelles est basée sur l'analyse syntaxique (Gannod et Cheng, 1999). Pour entrer dans cette sous-catégorie, une approche doit analyser les différents constituants d'un logiciel et tenter d'assembler

ses composantes afin de créer une représentation à un plus haut niveau d'abstraction. Il s'agit d'une approche ascendante (« bottom-up »). Par exemple, au lieu de chercher dans le code des instances de patrons de conception précisés dans une liste, les patrons présents dans le code pourraient être identifiés simplement en analysant la structure du code sans aucune connaissance des patrons à chercher. Cette analyse peut aussi consister à analyser le code et tenter de regrouper les parties contenant les mêmes concepts, donc le partitionner selon certains concepts.

Les méthodes formelles utilisent des techniques analytiques afin de s'assurer que des spécifications dérivées sont conformes en regard des spécifications originales. Une technique de rétro-ingénierie est formelle si ses étapes ont une base mathématique formelle. Ces techniques ont pour objectif de dériver des spécifications formelles du code source. Les approches formelles sont basées soit sur les transformations, soit sur la traduction.

La traduction signifie le changement d'un programme en spécifications formelles, à un niveau atomique (Gannod et Cheng, 1999). Par exemple, nous pourrions chercher à traduire, un code écrit en C vers une spécification Z. La transformation, quant à elle, effectue le même travail, mais en cherchant une certaine abstraction (Gannod et Cheng, 1999). Elle peut ainsi chercher à regrouper un ensemble d'instructions du code source dans des expressions équivalentes plus concises. Par exemple au lieu de transformer simplement un code non-objet en spécifications Z, il pourrait être converti en Z objet, c'est-à-dire qu'en plus d'exprimer le code directement en spécification, on y identifie des objets.

2.3.2 Techniques employées

Les types de traitements vus au point précédent peuvent être mis en œuvre selon différentes techniques de programmation. Par exemple, la recherche de clichés dans un code peut se faire par une programmation séquentielle standard ou en employant des techniques d'intelligence artificielle. Dans cette section, nous allons voir des

exemples de techniques employées à savoir l'association de concepts, l'intelligence artificielle, la programmation dynamique, l'utilisation d'un langage de requêtes, l'analyse des flux, les procédures manuelles, l'analyse des chaînes de caractères, l'analyse statique et finalement les métriques et statistiques.

2.3.2.1 Association de concepts

L'association de concepts (Biggerstaff, Mitbander et Webster, 1993) consiste à associer des concepts du domaine d'affaires ou informatique aux composantes d'un logiciel. L'association de concepts peut se faire selon deux approches. Avec la première approche, il s'agit d'abord d'identifier un ensemble de concepts par leurs caractéristiques et d'en rechercher des instances dans le code source. Avec la seconde approche, il s'agit d'analyser le code source et d'identifier les concepts qui y sont contenus.

Un exemple d'association de concepts est la détection de patrons de conception. Les patrons de conception à rechercher sont d'abord décrits. Ensuite, des instances en sont recherchées dans le code source en se basant sur ces descriptions.

2.3.2.2 Intelligence artificielle (ou analyse du raisonnement)

L'emploi de l'intelligence artificielle en rétro-ingénierie est une idée intéressante (Li, Yang et Chu, 2000; Richner et Ducasse, 1999; Wuyts, 1998). Elle représente l'utilisation de méthodes de programmation tentant de reproduire le raisonnement humain. Elle est en lien avec les techniques de compréhension du logiciel, car plusieurs se basent sur le raisonnement humain.

Le raisonnement classique (Balmas, 1997) représente la manière de réfléchir classique selon la logique des prédicats. Par exemple *si a est vrai et b est vrai alors a et b sont vrais*. Il s'agit de raisonnement par déduction. Ainsi, il est possible de définir dans un moteur d'inférence que la présence d'un certain cliché dans le code se

concrétise par la présence de deux composantes. Ainsi, une règle définirait que si une première composante est présente dans le code et une seconde aussi l'est, nous sommes alors en présence de ce cliché.

Le raisonnement par induction est notamment utilisé pour l'apprentissage par l'exemple (Cohen, 1995; Lu *et al.*, 2002). Par exemple, le comportement du logiciel peut être analysé en tentant de généraliser les comportements observés afin de décrire le comportement général du logiciel. Ainsi, nous pourrions observer que devant la présence de telles conditions le logiciel produit tels résultats et déduire que c'est le cas pour d'autres situations plus ou moins analogues. Par exemple, lors de l'exécution d'un logiciel, si nous constatons toujours qu'en entrant un montant négatif nous recevons un message d'erreur, nous pouvons induire que ce sera le cas pour tous les montants négatifs.

En logique floue, les énoncés sont exprimés avec un degré de précision de 0,0 à 1,0. Au contraire de la logique binaire où un énoncé est ou bien vrai ou bien faux. Par exemple, en logique floue, il est possible, d'affirmer que si une première composante est présente dans le code et une seconde aussi, nous sommes à 60% en présence d'un cliché (et non dans 60 % des cas). C'est-à-dire qu'elle représente à 60% ce cliché et non à 60% des chances de le représenter.

2.3.2.3 Programmation dynamique

La programmation dynamique est semblable à l'approche « diviser pour régner » par le fait qu'un problème est divisé en sous-problèmes. Toutefois, dans cette approche lorsque de petites parties du problème sont résolues, le résultat est stocké afin d'être réutilisé plus tard s'il est à nouveau nécessaire. Il n'a donc pas à être recalculé (Neapolitan et Naimipour, 2004). Par contre, s'il y a un gain en vitesse, l'utilisation de la mémoire s'en trouve augmentée.

Par exemple, dans le cas de recherche de clones, nous pourrions calculer les caractéristiques d'une partie de code pour la comparer avec celles d'une autre. S'il y a *n* parties et s'il faut comparer chaque partie avec chacune des autres parties, ces caractéristiques seront utilisées *n* fois. Il est plus efficace de les calculer une fois et de garder le résultat en mémoire que de les calculer à chaque comparaison.

2.3.2.4 Utilisation d'un langage de requêtes

L'utilisation d'un langage de requêtes est une approche souvent employée (Lu *et al.*, 2002). Il s'agit en général d'enregistrer des informations sur la structure du logiciel à analyser dans une base de données. Par la suite, l'utilisation de requêtes peut se faire par l'outil de rétro-ingénierie qui extrait l'information recherchée à l'aide des requêtes.

Par exemple, Richner et Ducasse (Richner et Ducasse, 1999) enregistrent, dans une base de faits, des informations dynamiques et statiques au sujet d'un logiciel. Ces informations sont par la suite exploitées par des requêtes afin de produire des graphiques répondant à diverses interrogations. Ainsi, un utilisateur pourra créer des représentations sur demande répondant à son questionnement.

2.3.2.5 Analyse des flux

Il est intéressant d'analyser comment les données sont manipulées lors de l'exécution du logiciel (Kontogiannis *et al.*, 1994). Il est aussi intéressant d'analyser l'ordre où les différentes composantes du logiciel sont appelées lors de l'exécution du logiciel. Ces composantes peuvent être les objets, les lignes de codes ou des sous-systèmes identifiables dans le logiciel.

Ainsi, en analysant quels objets créent effectivement quel objet lors de l'exécution, il est possible de détecter l'importance réelle de chaque objet et même de les regrouper selon certains critères.

2.3.2.6 Procédures manuelles

Il existe aussi des procédures qui ne sont pas des outils permettant de réaliser des tâches de rétro-ingénierie d'une façon plus ou moins « manuelle ». Ces procédures peuvent aussi guider l'implantation d'outil de rétro-ingénierie. Elles peuvent indiquer les caractéristiques des cibles recherchées dans le code, comment les rechercher étape par étape en passant de la détection des sources d'information jusqu'à l'affichage final des résultats.

Ainsi, Earls, Embury et Turner (Earls, Embury et Turner, 2002) présentent une méthodologie pour extraire les règles d'affaires d'un logiciel sans outil spécifique. Leur méthode se concentre notamment sur l'identification des portions de codes ayant le plus de chance de contenir ces fameuses règles afin de minimiser les pertes de temps. Heuzeroth *et al.* (Heuzeroth *et al.*, 2003) démontrent comment, à partir de caractéristiques statiques et dynamiques d'un patron de conception, nous pouvons définir des algorithmes afin de les détecter.

2.3.2.7 Analyse des chaînes de caractères

Une analyse simplement textuelle des chaînes de caractères contenues dans le code source peut être utilisée afin d'en extraire de l'information. Ces chaînes de caractères peuvent être manipulées avant leur traitement (par exemple, remplacer les variables par des jetons). Des langages de requêtes utilisant des expressions régulières peuvent être employés afin d'effectuer des recherches.

Par exemple afin de détecter des clones, Ducasse, Rieger et Demeyer (Ducasse, Rieger et Demeyer, 1999) comparent chaque chaîne de caractère du code d'un logiciel entre elles à l'aide d'un tableau de hachage.

2.3.2.8 Analyse statique

L'analyse statique implique en général l'analyse des informations obtenues directement en examinant le code source du logiciel visé (Tilley, 2000). Cette analyse permet d'examiner les liens entre les composantes du code source. L'analyse de l'arbre syntaxique abstrait d'un logiciel est souvent la méthode employée pour effectuer une analyse statique.

2.3.2.9 Métriques et statistiques

L'analyse d'un logiciel et de ses composantes permet la prise de mesures quantifiables. Ces mesures permettent aussi d'en calculer d'autres. Dans ce qui suit, nous allons aborder quelques applications possibles.

On peut d'abord employer des mesures de complexité comme le nombre de lignes de code ou le point de fonction (Albrecht. , 1979; April, Merlo et Abran, 1997) afin de mesurer la complexité du logiciel à analyser et ainsi évaluer l'effort à consacrer à la tache de compréhension. Ces mêmes mesures peuvent aussi notamment servir à détecter des clones en comparant leur similarité avec ces mesures (Buss *et al.*, 1994).

Aussi, en calculant les appels entre les instances des classes lors de l'exécution d'un système. Il est possible de faire ressortir des propriétés intéressantes des différentes classes comme le fait Ducasse, Lanza et Bertuli (Ducasse, Lanza et Bertuli, 2004).

Deux autres métriques souvent employées par des approches de rétro-ingénierie sont l'entrance (« fan-in ») et la sortance (« fan-out »). Elles analysent le nombre de liens (par exemple un appel entre classes) sortants ou entrants entre des composantes d'un système (Hamou-Lhadj *et al.*, 2005). De même, on peut mesurer le couplage et la cohésion de sous-systèmes. On peut ainsi mesurer la qualité de la partition d'un système en sous-système. On voudra que les éléments d'un sous-système aient un nombre de liens élevé (forte cohésion) et que les sous-systèmes aient un nombre peu élevé de liens entre eux (couplage faible) (Mitchell, Mancoridis et Traverso, 2002).

Un autre type de couplage est le couplage logique qui permet par exemple de déterminer la correspondance entre les composantes de différentes versions d'un même logiciel (D'Ambros et Lanza, 2006).

2.4 LANGAGES INTERMÉDIAIRES UTILISÉS POUR L'ANALYSE

Afin de pouvoir traiter les données en entrée, celles-ci doivent tenir en mémoire et de préférence sous une forme plus facile à traiter. Si nous ne désirions pas d'abstraction, nous pourrions nous contenter de l'information contenue dans ces représentations. Mais leur forme n'est en général pas plus utile que le code source pour la compréhension du logiciel étant donné qu'elles sont au même niveau d'abstraction (Lu *et al.*, 2002).

Parmi ces langages intermédiaires, nous retrouvons ceux qui conservent une information brute non retravaillée. Les arbres syntaxiques abstraits représentent la structure d'un logiciel telle que retrouvée dans son code source. Des métalangages ont été développés afin de décrire les informations sur un logiciel. Il est aussi possible d'en employer d'autres plus généralistes à cette fin. Certains sont employés à l'aide de requêtes. Parmi ces langages, se retrouvent : REFINE (Ward et Bennett, 1995), SCA (Paul et Prakash, 1994), FAMIX (Ducasse et Lanza, 2005), MOOSE (Lanza et Ducasse, 2003), Prolog (Kramer et Prechelt, 1996) et SOUL (Wuyts, 1998).

Il y a aussi des algorithmes qui conservent une information plus limitée et non retravaillée. De cette façon, un algorithme peut conserver une liste d'identificateurs. Cette liste peut contenir par exemple, le nom de toutes les classes, des méthodes et des identificateurs d'un logiciel. Cette liste pourrait être étendue aux attributs des variables, des fonctions, des classes et des informations comme leur type et leur visibilité.

Il y a aussi des représentations d'informations plus ciblées et parfois déjà retravaillées pouvant être extraites du code source ou des traces d'exécution. Ainsi, le graphe de

flux de contrôle représente les différents chemins d'exécution possibles dans le logiciel. Le graphe des appels peut représenter les appels entre différentes composantes d'un logiciel, comme les classes et les fonctions. Le graphe des héritages permet de représenter la hiérarchie entre les différentes classes d'un logiciel.

L'information sur les logiciels peut être aussi emmagasinée sous forme architecturale. Les langages de description d'architectures (Sartipi, 2003) sont employés à cette fin. Ils permettent de décrire l'architecture d'un logiciel. Parmi ceux-ci, se retrouvent : UML et OMT (Kramer et Prechelt, 1996). Il est à noter que ces langages ne sont pas les plus appropriés pour décrire l'architecture et c'est pourquoi d'autres langages sont utilisés. Ces autres langages, à l'instar de Wright (Allen et Garlan, 1994) et UniCon (Shaw *et al.*, 1995), sont d'ailleurs appelés langages de description d'architecture (en anglais « Architecture Description Language »).

2.5 SORTIES

Le but du traitement des données brutes emmagasinées par le processus de rétro-ingénierie est en général de fournir une abstraction du logiciel. Les sorties représentent aussi cette dimension du processus. En général, le but de la rétro-ingénierie est de créer une représentation à un niveau d'abstraction plus élevée, mais aussi parfois de permettre une transformation du logiciel. Dans ce qui suit, nous discuterons de cette dimension suivant deux angles : les formes de l'information affichée et les techniques d'affichage.

2.5.1 Formes de l'information affichée

L'information affichée à l'utilisateur d'un outil de rétro-ingénierie peut prendre plusieurs formes incluant l'architecture, les clones, les patrons de conception, les exigences et le tranchage (« slicing »).

2.5.1.1 Architecture d'un logiciel

Une forme courante de la rétro-ingénierie consiste en l'extraction d'une architecture d'un logiciel. Nous consacrons d'ailleurs le quatrième chapitre à l'architecture du logiciel. L'objectif est de faire ressortir une structure du logiciel. Il s'agit de démontrer comment le logiciel est construit en démontrant les liens entre ses diverses composantes.

Par exemple, il est courant de chercher à diviser un logiciel en sous-systèmes. Ainsi, Anquetil et Lethbridge (Anquetil et Lethbridge, 1999) ont cherché à diviser un logiciel en sous-systèmes en se basant sur les noms de ses fichiers.

2.5.1.2 Clones

La détection de clones a fait aussi l'objet de plusieurs études (Magdalena *et al.*, 2000). Elle a pour objectif de trouver des parties d'un code source qui se répètent. Pour des raisons de simplification de la maintenance, il est souvent avantageux de supprimer les clones. De plus, les clones peuvent représenter un concept (Baxter *et al.*, 1998). Un tel exemple de concept serait la répétition d'un ensemble de lignes de code écrivant les mêmes variables dans un fichier. Elle pourrait représenter un concept de sauvegarde et aurait intérêt à être remplacée par une fonction.

Les outils peuvent fournir de l'information sur le degré de similarité, la quantité, la taille et la localisation des clones. Ils peuvent aussi corriger les problèmes des clones automatiquement. En remplaçant par exemple les clones par des appels de fonctions, par des macros ou même par une restructuration du code.

2.5.1.3 Patrons de conception

La détection de patrons de conception est un autre objectif possible de la rétro-ingénierie. L'identification d'instances de patron de conception peut être utile afin de

vérifier la qualité d'implantation d'un logiciel. Il est aussi possible de détecter des instances de patrons représentant des pratiques négatives (anti-patrons) (Brown, Malveau et McCormick, 1998). Il peut aussi être utile de détecter des patrons de conception afin d'isoler plus précisément la logique d'affaires contenue dans le code, car ces patrons sont souvent employés pour implanter ces règles, comme nous les verrons au quatrième chapitre. Donc, la représentation d'une règle d'affaires se retrouve répartie dans les différentes classes représentant ce patron. Il est aussi utile de détecter des patrons de conception dans l'approche circulaire du génie logiciel (« round trip engineering »).

Deux types d'extraction des patrons de conception existent. Le premier contient les méthodes identifiées comme « a priori ». Elles se présentent lorsque l'outil possède une liste de patrons qu'il tente de retrouver dans le code. Le deuxième contient les méthodes identifiées comme « post priori ». Elles se présentent lorsque l'outil tente de trouver des patrons qu'il ne connaît pas nécessairement (Pinali, 1999).

2.5.1.4 Exigences d'un logiciel

L'extraction des exigences (Gannod et Cheng, 1999; Lu *et al.*, 2002) est une autre utilisation intéressante de la rétro-ingénierie. Liu, Alderson et Qureshi (Liu, Alderson et Qureshi, 1999) distinguent quatre types d'exigences. D'abord les exigences fonctionnelles qui décrivent ce que le logiciel doit faire. Il y a aussi les exigences non fonctionnelles qui sont des contraintes techniques sous lesquelles le logiciel doit fonctionner comme la performance, la fiabilité, la taille et des facteurs environnementaux. De plus, il y a les exigences de « discipline » qui sont des normes et lois s'appliquant au domaine du logiciel. Et finalement, il y a les règles d'affaires (Earls, Embury et Turner, 2002).

Comme parfois celles-ci ne se retrouvent que dans le code source, il est intéressant de les retrouver. Notamment dans le cas où il serait nécessaire de réimplanter un logiciel

avec les mêmes fonctionnalités. Le fait de bien comprendre toutes ces exigences aidera à comprendre non seulement ce que fait le logiciel, mais aussi pourquoi il le fait de telle manière.

2.5.1.5 Tranchage d'un logiciel (« slicing »)

Une avenue intéressante de la rétro-ingénierie est le tranchage du logiciel (« slicing ») (Xu *et al.*, 2005). Cette pratique consiste à mettre en évidence, certaines parties d'un logiciel impliquées lors d'une opération précise ou contenant une entité du code.

Par exemple, il est parfois pratique d'isoler tous les traitements effectués sur une variable précise. Ainsi lorsqu'une variable affiche un résultat erroné lors de l'exécution, il serait pratique d'avoir un outil affichant les traces d'exécutions où cette variable est traitée et les lignes de code associées. Le développeur pourrait voir le code ayant induit la valeur erronée.

2.5.2 Techniques d'affichage

L'information recueillie lors d'un processus de rétro-ingénierie peut être affichée soit sous une forme graphique, soit sous une forme textuelle.

Plusieurs types d'affichage graphiques peuvent être employés. Ils peuvent prendre notamment la forme de graphe des appels (Desclaux et Ribault, 1991) (il peut avoir été filtré, par exemple pour enlever les appels aux fonctions utilitaires (Hamou-Lhadj *et al.*, 2005)), de graphe de contrôle (Desclaux et Ribault, 1991), de diagramme de classes (Tonella et Potrich, 2005), de diagramme d'objets (Tonella et Potrich, 2005), de diagramme d'interactions (Tonella et Potrich, 2005), de diagramme d'états (Tonella et Potrich, 2005), de diagramme de paquetages (Tonella et Potrich, 2005) ou de matrice (Ducasse, Rieger et Demeyer, 1999).

Les résultats du processus de rétro-ingénierie peuvent s'afficher aussi sous forme de rapports textuels. Le dictionnaire des données (Desclaux et Ribault, 1991; Pressman,

2001) en est un exemple. Il peut consister en une liste contenant la description de toutes les données que le logiciel manipule.

L'affichage textuel peut aussi prendre la forme de l'amélioration de la présentation du code et l'aide à sa navigation. Par exemple, ceci peut se faire en reformatant sa présentation par des moyens typographiques et de mise en page. Ces moyens peuvent consister à mettre en gras les mots-clés ou à améliorer la tabulation afin de bien refaire sortir les différents blocs logiques du code. La navigation hypertexte (Rajlich, 1997) permet la navigation dans le code entre les composantes liées. L'approche par synopsis est une approche textuelle ayant pour but de résumer ce que font certaines parties du code. Par exemple, Balmas (Balmas, 1997) a présenté une façon de faire ressortir les propriétés du parcours d'une liste dans le listage d'un logiciel afin de simplifier la lecture. Abd-El-Hafiz et Basili (Abd-El-Hafiz et Basili, 1993) présentent une méthode permettant d'afficher les propriétés des boucles d'un logiciel sous forme d'arbre.

2.6 RÉSUMÉ DU CADRE DE CLASSIFICATION

La figure 2.2 résume notre cadre de classification sous forme d'arborescence.

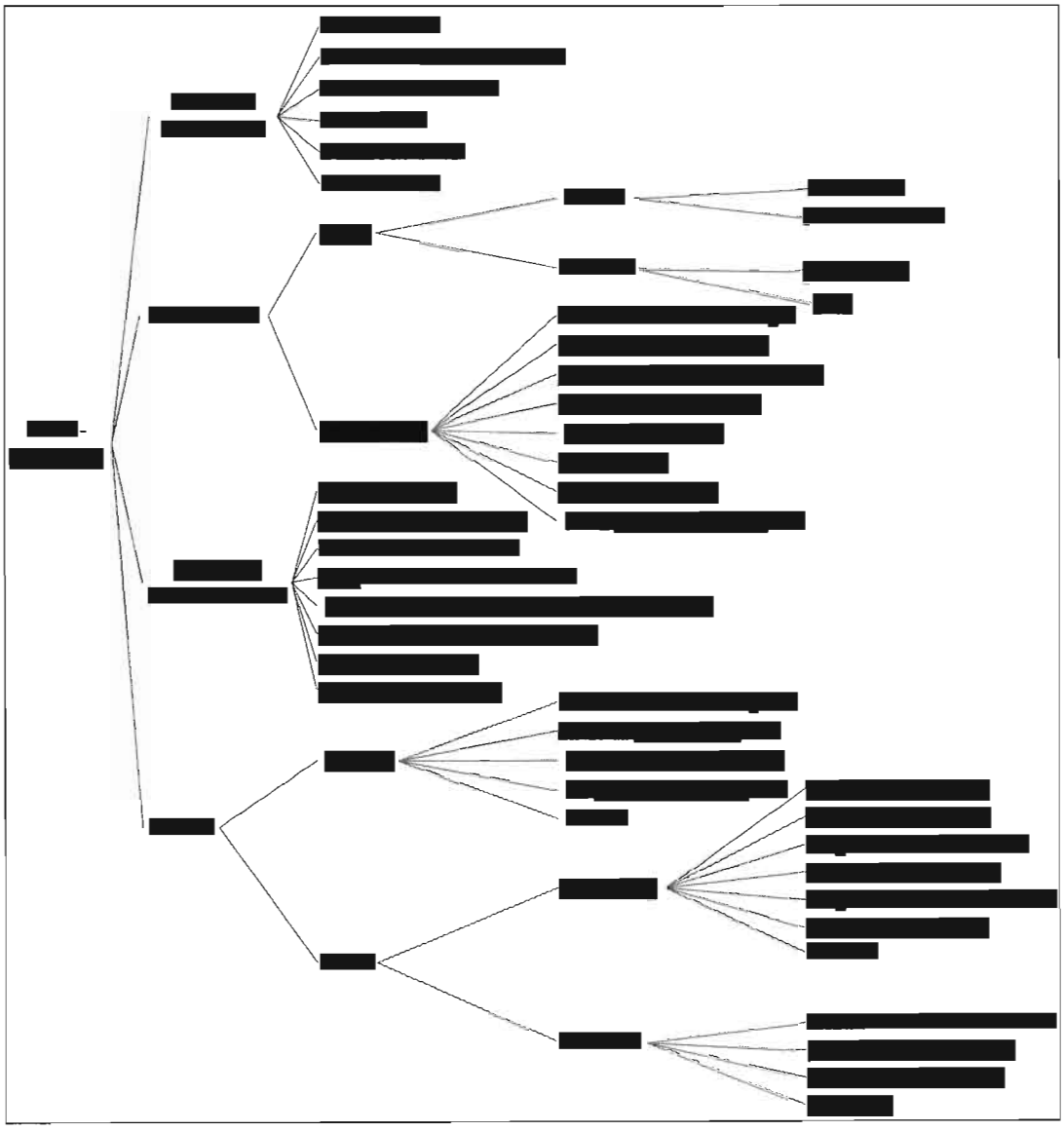

Figure 2.2 La rétro-ingénierie

2.7 CONCLUSION

Dans ce chapitre, nous avons présenté un cadre de classification des approches de rétro-ingénierie ayant quatre dimensions. La première dimension représente les données en entrées. La deuxième dimension est constituée des traitements effectués

sur ces données. La troisième dimension englobe les représentations intermédiaires utilisées pour la rétro-ingénierie. La quatrième dimension englobe les formes et les techniques de présentation des résultats produits par la rétro-ingénierie. Nous avons conclu ce chapitre en présentant une arborescence qui résume notre cadre de classification. Ce cadre démontre que ce domaine est vaste et difficile à classer, notamment car souvent une même approche en exploite plusieurs parties à la fois. Mais le cadre permet de se faire une bonne idée du domaine. Dans le chapitre suivant, nous allons présenter un large échantillon des différentes approches de rétro-ingénierie.

CHAPITRE III

APPROCHES DE RÉTRO-INGÉNIERIE

3.1 INTRODUCTION

Dans le chapitre précédent, nous avons décrit notre cadre de classification de la rétro-ingénierie. Dans ce chapitre, nous allons présenter un échantillon d'approches proposées dans ce domaine. Nous avons réparti ces travaux en deux grandes familles : les approches formelles (transformation et traduction) et les approches informelles (détection de cliché et analyse syntaxique). Nous conclurons ce chapitre par une discussion sur les approches tout en les classant dans un tableau suivant notre cadre de classification.

3.2 APPROCHES FORMELLES

Notre cadre de classification distingue deux sous-catégories d'approches formelles : les approches transformationnelles et les approches basées sur la traduction.

3.2.1 Approches transformationnelles

Dans cette section, nous allons parler d'approches transformationnelles. Rappelons qu'une transformation consiste à transformer un logiciel d'une façon formelle tout en cherchant une certaine abstraction. Elle peut ainsi chercher à regrouper un ensemble d'instructions du code source dans des expressions équivalentes plus concises.

3.2.1.1 Approche de Bowen, Breuer et Lano

Bowen, Breuer et Lano (Bowen, Breuer et Lano, 1993) présentent l'utilisation des méthodes formelles pour l'approche circulaire du génie logiciel. Le but de leur approche est de réaliser la maintenance de spécifications formelles et non de code source. Leur approche commence par extraire des spécifications Z++ du code source COBOL ou Fortran, mais au lieu de traduire simplement le code en spécifications, ils tentent d'isoler des concepts comme les fichiers et les tableaux du code source qui deviennent alors des objets. Puisqu'une abstraction est effectuée, il s'agit donc d'une transformation. Ensuite, si des modifications au logiciel sont nécessaires, le développeur modifie ces spécifications qui sont ensuite régénérées automatiquement en code source.

3.2.1.2 Approche de Ward

Ward (Ward, 2004) décrit un système permettant d'effectuer la transformation de code assembleur en code C. Afin de faire cette transformation, le code assembleur est d'abord traduit dans un langage formel nommé WSL « Wide Spectrum Language ». À cette étape, il s'agit de la traduction, mais il ne se contente pas de spécifications représentant le code assembleur tel quel. Car avant de traduire le tout en code C, il applique à ces spécifications un ensemble de règles de restructuration et de simplification. Il analyse notamment les flux de contrôle (pour découvrir des boucles et des structures de contrôles,) les flux de données (pour éliminer les assignations redondantes sur les registres et les drapeaux) et les constantes. Une fois les spécifications transformées, elles sont réimplantées en code C avec un autre outil.

3.2.2 Approche basée sur la traduction

Les approches basées sur la traduction effectuent la même tâche qu'une transformation, mais sans rien changer par rapport à l'implantation. C'est-à-dire que

chaque instruction du code source se trouve représentée dans les spécifications formelles générées. L'intérêt dans le cadre de la compréhension du logiciel est donc plus limité. Voici pourquoi les travaux représentant cette approche sont plus rares. C'est pourquoi nous nous limitions à un seul travail. En effet, les travaux faisant de la transformation utilisent souvent la traduction en première étape comme nous avons pu le constater.

3.2.2.1 Approche de Gannod et Cheng

Gannod et Cheng (Gannod et Cheng, 1996) présentent un outil permettant de dériver des spécifications formelles représentant le code source C, en analysant l'arbre syntaxique abstrait du logiciel et en se basant sur la notion des plus fortes post-conditions (« strongest post condition (Dijkstra et Scholten, 1990) »). Ils utilisent pour ce faire la logique de Hoare. Cette logique permet d'exprimer dans un triplet de termes l'évolution d'un logiciel avec une pré et une poste-condition et une commande (une action sur le programme) avec des axiomes et des règles d'inférences représentant les instructions de base d'un programme impératif (Hoare, 1969).

3.3 APPROCHES INFORMELLES

Les approches informelles sont divisées en deux sous-catégories : celles basées sur la détection de clichés et celles basées sur l'analyse syntaxique.

3.3.1 Approches basées sur la détection des clichés

Dans cette section, nous allons présenter un large éventail d'approches pour la rétro-ingénierie basées sur la détection des clichés.

3.3.1.1 Approche de Fiutem *et al.*

L'approche de Fiutem *et al.* (Fiutem *et al.*, 1996) a pour but de créer un modèle architectural hiérarchique d'un système dans lequel il est possible de naviguer d'une composante à l'autre. Ce qui signifie qu'il est possible de zoomer d'un niveau d'abstraction à l'autre dans les différents modèles représentant le code. La représentation hiérarchique commence au plus haut niveau par le système lui-même. Au second niveau se retrouvent les logiciels qu'il contient. Au niveau suivant, les programmes sont représentés par un ensemble de modules et leurs relations. Un module est une entité qui exécute un traitement. Ces modules sont ensuite à leur tour divisés en un ensemble de composantes (comme les sous-routines et les variables) et leurs liens. Ces composantes sont définies par des diagrammes d'appel et de flux de données.

Afin d'identifier les logiciels constituant le système, l'approche commence par analyser le fichier « makefile » du système afin de détecter les logiciels qu'il contient. Leurs arbres syntaxiques abstraits sont analysés par des « reconnaisseurs architecturaux » (Harris, Reubenstein et Yeh, 1995). Ces reconnaisseurs analysent l'arbre syntaxique abstrait afin de détecter des constructions au niveau du code source signalant la présence de composantes et de leurs relations. Pour ce faire, l'approche cherche la présence de patrons dans l'arbre représentant des clichés architecturaux. Ces patrons sont définis dans le langage architectural de REFINE (Ward et Bennett, 1995), lesquels sont utilisés dans la création de modèles architecturaux. Les reconnaisseurs arrivent à détecter des relations comme des fichiers partagés entre les programmes, les espaces mémoire partagés entre programmes, les canaux de communication (« pipes »), les appels à un service RPC, les connexions client-serveur via une interface de connexion (« socket ») Unix et les invocations entre deux programmes et d'autres.

3.3.1.2 Approche de Heuzeroth *et al.*

Heuzeroth *et al.* (Heuzeroth *et al.*, 2003) cherchent à détecter les patrons de conception de code source Java. Leur approche est divisée en deux grandes étapes (voir la figure 3.1) : l'une est statique et l'autre dynamique. Ils se basent sur le fait qu'un patron de conception possède des propriétés statiques et dynamiques.

Pour chaque patron de conception à isoler, ils définissent deux spécifications sous forme d'algorithmes : une pour les propriétés statiques et l'autre pour les dynamiques. L'approche est donc très dépendante du patron à isoler. À l'analyse statique, ils détectent les sections du logiciel pouvant potentiellement faire partie d'un patron de conception.

Ces parties sont par la suite filtrées lors de l'analyse dynamique. Par exemple, pour détecter le patron visiteur, du côté statique, ils commencent par chercher les classes « `composantes` » qui sont visitées par les méthodes « `visite` » des classes « `visiteurs` ». Ces classes « `composantes` » sont identifiables par le fait qu'elles doivent avoir une méthode « `Accept` » ayant comme paramètre une classe « `visiteur` ». Cette classe « `visiteur` » doit être appelée avec une « `composante` » en paramètre. « `Composante` » qui sera appelée ensuite par elle, via cette méthode. En général, l'argument est du type de la superclasse du visiteur. Du côté dynamique, ils vérifient si les visiteurs et les composantes potentielles correspondent bien à ce qui est vraiment constaté lors de l'exécution et que les appels se font comme attendu.

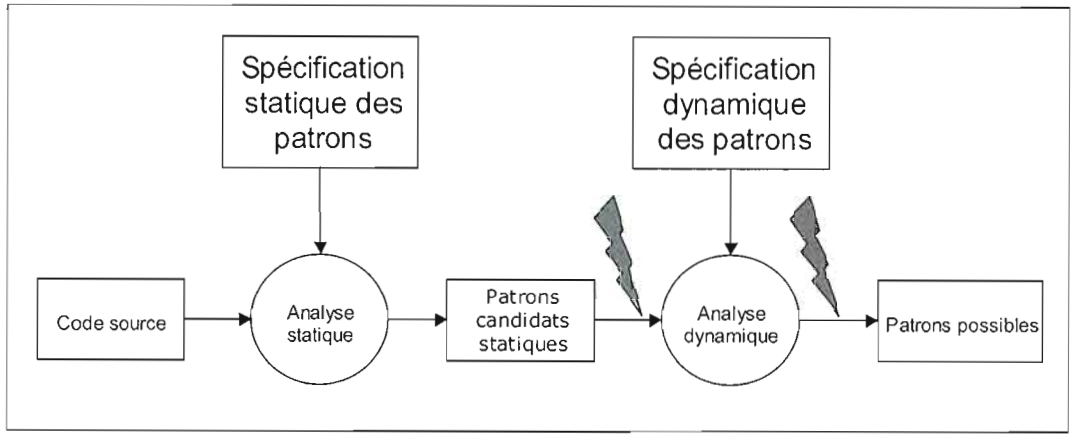

Figure 3.1 Les différentes étapes de l'approche proposée par Heuzeroth *et al.* (adaptée de Heuzeroth *et al.*, 2003)

3.3.1.3 Approche de Huang *et al.*

Huang *et al.* (Huang *et al.*, 2005) cherchent aussi à détecter des patrons de conception en examinant leurs aspects dynamiques et statiques. Pour faire cela, ils analysent d'abord un logiciel Java afin de l'exprimer sous forme de faits dans le langage Prolog, autant de façon statique que dynamique. Quant à eux, les patrons de conceptions à rechercher sont aussi exprimés en Prolog ou avec une notation pouvant exprimer les aspects temporels ou séquentiels représentant le comportement du logiciel qui est ensuite traduite en Prolog. L'application de ces règles aux faits permet de déterminer de nouveaux faits représentant les patrons de conceptions présents dans le logiciel analysé.

3.3.1.4 Approche de Jahnke, Schäfer et Zundorf

Jahnke, Schäfer et Zundorf (Jahnke, Schäfer et Zundorf, 1997) présentent l'utilisation de la logique floue afin de convertir des bases de données relationnelles en bases de données objet. Le processus de migration proposé se résume en trois étapes : migration des schémas relationnels à un équivalent objet, migration des données d'un

modèle à l'autre et migration des logiciels utilisant les anciens modèles vers le modèle objet. Le travail présenté ici se concentre sur la première étape.

Afin d'exprimer les règles permettant de spécifier les transformations, ils utilisent une notation graphique « Generic Fuzzy Reasoning Nets (GFRN) ». La figure 3.2 montre un exemple de notation GFRN d'une règle. Elle signifie que si le code du logiciel inclut un énoncé « `select` » avec le mot-clé « `distinct` » qualifié par un ensemble d'attributs `a`, alors il peut être déduit avec une certitude ϕ que `a` et tous ses sous-ensembles `b` ne représentent pas des clefs candidates.

Les différentes connaissances connues dans le domaine du logiciel peuvent ainsi être décrites dans des règles en GFRN. Les différentes règles sont alors placées dans un réseau en spécifiant un degré de certitude.

En employant un moteur d'inférence adapté, l'approche soumet les prémisses connues, ici les informations sur la base de données relationnelle existante. Elles comprennent aussi les requêtes trouvées dans le code du logiciel l'utilisant. Le moteur se chargera alors d'analyser cette information afin de déduire les faits les plus satisfaisants et ainsi obtenir le meilleur modèle objet possible. Ainsi grâce aux réseaux, les clefs, les cardinalités, les relations d'agrégations et d'héritage pourront être identifiées.

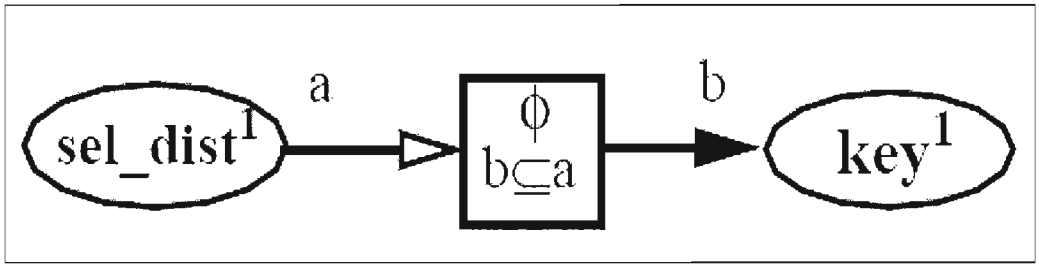

Figure 3.2 Notation GFRN de la règle R (tirée de Jahnke, Schäfer et Zundorf, 1997)

3.3.1.5 Approche de Kramer et Prechelt

Krämer et Prechelt (Kramer et Prechelt, 1996) présentent une méthode pour identifier des patrons prédéterminés dans un code C++ en utilisant le langage Prolog. Ils s'intéressent particulièrement aux patrons architecturaux. Les patrons à rechercher sont d'abord représentés en OMT. Par exemple, la figure 3.3 montre le diagramme OMT du patron Adaptator.

Ces structures sont ensuite converties en Prolog. La figure 3.4 montre la description en Prolog de la figure 3.3.

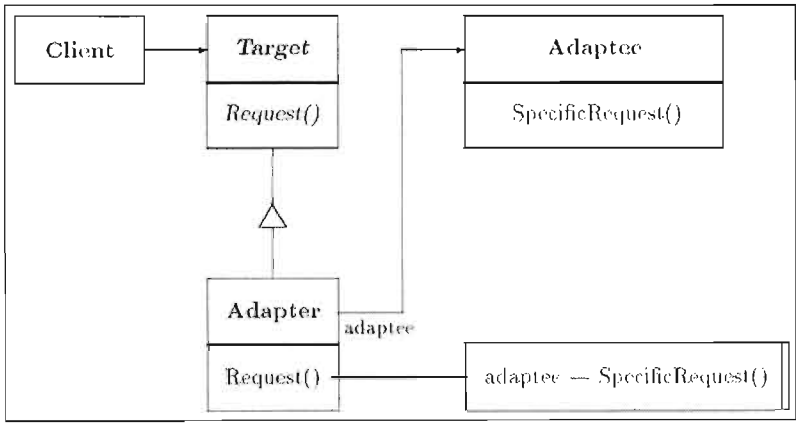

Figure 3.3 Diagramme OMT d'Adaptator (tirée de Kramer et Prechelt, 1996)

```
adaptor(Target,Adapter,Adaptee):-
    class(_,Target),
    class(concrete,Adapter),
    class(concrete,Adaptee),
    operation(_,_,Target,Request,_,_,_),
    operation(_,_,Adapter,Request,_,_,_),
    operation(_,_,Adaptee,
              SpecificRequest,_,_,_),
    inheritance(Target,Adapter),
    association(Adapter,Adaptee).
```

Figure 3.4 Conversion du diagramme de la figure 3.3 en Prolog (tirée de Kramer et Prechelt, 1996)

Ensuite, un outil fait une analyse syntaxique des fichiers en-têtes (« .h ») et récupère la structure du logiciel à analyser. Cette structure est ensuite convertie en Prolog comme le montre la figure 3.6 pour la classe `zPane` qui est décrite dans la figure 3.5.

Finalement, des requêtes détectent les instances des patrons. En appliquant aux descriptions des patrons spécifiés en Prolog les faits extraits du code source, la détection des patrons se fait automatiquement.

```
class zPane:public zChildWin {
    zDisplay* curDisp;
    /* ... */
public:
    virtual void show(int=SW_SHOWNORMAL);
    /* ... */
};
```

Figure 3.5 Exemple d'en-tête C++ (tirée de Kramer et Prechelt, 1996)

```
class(concrete, zPane).
inheritance (zChildWin, zPane).
attribute (zPane, curDisp).
operation(virtual, selector, zPane, show,
          public, "int,", "void").
```

Figure 3.6 Conversion de la figure 3.5 en Prolog (tirée de Kramer et Prechelt, 1996)

3.3.1.6 Approche de Mitchell, Mancoridis et Traverso.

Mitchell, Mancoridis et Traverso (Mitchell, Mancoridis et Traverso, 2002) ont présenté une méthode afin de diviser un logiciel en sous-systèmes. Ils utilisent à la fois une approche descendante et une autre ascendante. Leur approche est supportée par deux outils : BUNCH et ARIS.

Dans un premier temps, leur outil BUNCH fait ressortir l'architecture existante d'un logiciel. Comme montré à la figure 3.7, il effectue une analyse statique du logiciel. Ceci crée un graphe de dépendance. Dans ce graphe les liens entre chaque module du logiciel analysé sont affichés. Ensuite, BUNCH fait une analyse afin de trouver un moyen de diviser ces modules en sous-systèmes. Au centre de l'illustration, toutes les partitions possibles du logiciel en sous-systèmes sont affichées.

L'espace de recherche peut devenir rapidement très grand. C'est pourquoi les auteurs emploient trois heuristiques de recherche afin de trouver une assez bonne solution. Afin d'appliquer ces heuristiques, ils emploient une formule évaluant chaque partition sur le nombre d'arêtes internes et externes. Ils utilisent trois types d'algorithmes de recherche. Un premier algorithme de recherche est de type escalade. Ils commencent par partitionner le logiciel de façon aléatoire et ils tentent par la suite d'améliorer chaque partition afin d'en trouver des meilleurs. Ils arrêtent quand l'algorithme ne trouve pas de meilleures partitions.

Un second algorithme de recherche est exhaustif. C'est-à-dire qu'il tente toutes les partitions possibles du logiciel. Évidemment, cette recherche est possible seulement avec de petits logiciels.

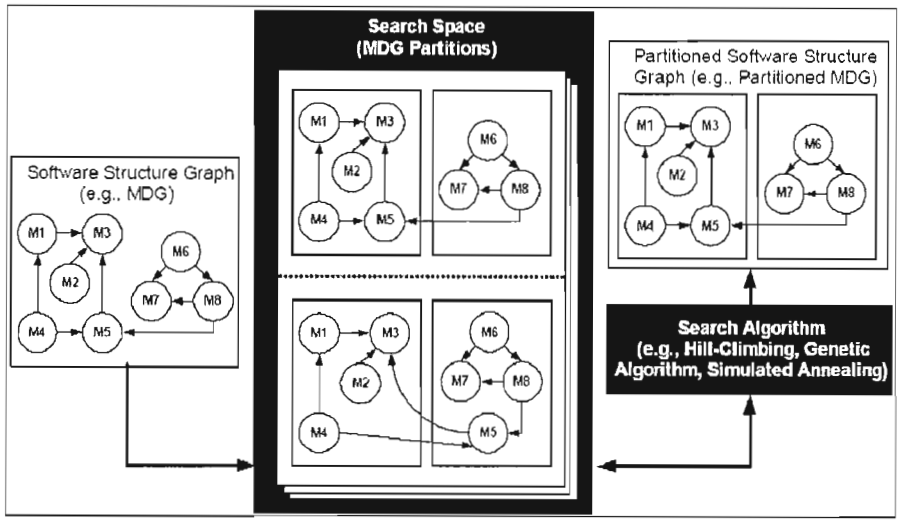

Figure 3.7 Bunch (tirée de Mitchell, Mancoridis et Traverso, 2002)

Un troisième algorithme de recherche est de type génétique (Doval, Mancoridis et Mitchell, 1999). Les algorithmes génétiques appliquent des idées inspirées de la théorie de la sélection naturelle afin de naviguer dans de grands espaces de recherche. Ces algorithmes opèrent sur un ensemble (population) de chaînes de caractères (les individus). La qualité de chaque chaîne est calculée en utilisant une fonction objective. Des règles probabilistes, plutôt que déterministes, sont employées pour diriger la recherche. Dans la terminologie des algorithmes génétiques, chaque itération de la recherche s'appelle génération. À chaque génération, une nouvelle population est créée en tirant profit des individus les plus intéressants de la génération précédente.

Finalement, un graphe affiche la meilleure partition déterminée par BUNCH. Une fois que cet outil a fait son travail, l'autre outil, à savoir ARIS, permet de définir formellement des relations attendues dans le logiciel. Par exemple, ARIS pourrait détecter les relations d'exportation manquantes d'un sous-système pour un module en utilisant une définition de la relation d'exportation afin d'induire les définitions manquantes comme le montre la figure 3.8.

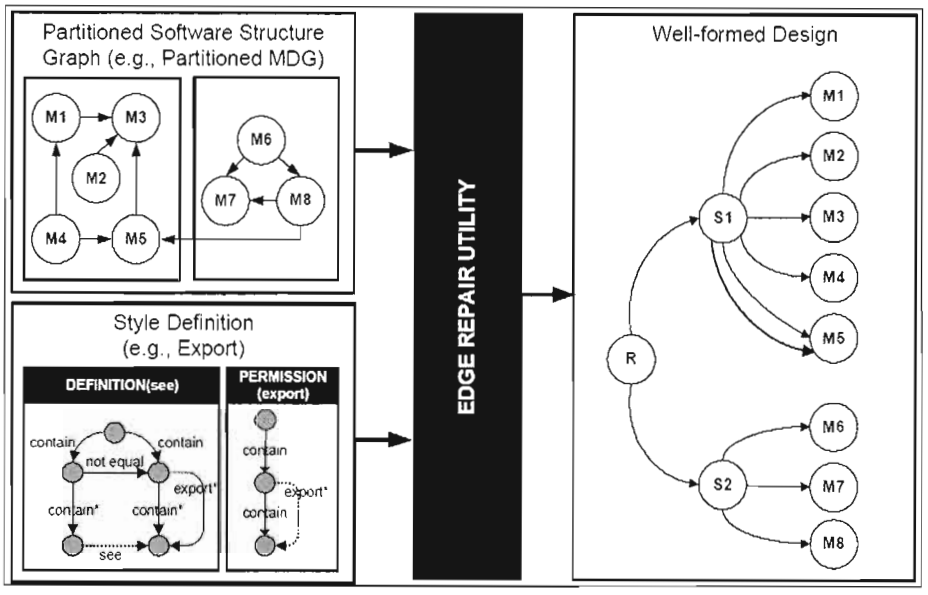

Figure 3.8 Aris (tirée de Mitchell, Mancoridis et Traverso, 2002)

3.3.1.7 Approche de Moha et Guéhéneuc

Moha et Guéhéneuc (Moha et Guéhéneuc, 2005) présentent une étude au sujet de la détection des défauts architecturaux dans une conception orientée objet. D'abord, ils définissent trois types de défauts pouvant être trouvés dans une architecture. Les premiers sont des anti-patrons, qui sont une solution répandue à un problème de conception qui génère des conséquences négatives. Ensuite viennent les défauts de conceptions, qui sont une mauvaise forme d'implantation de patrons de conception. Et finalement les « Code smells », qui sont des symptômes de problèmes dans un code qui peuvent servir d'indice à la détection des deux autres types de défauts. Par exemple, il peut s'agir de code dupliqué, de méthodes trop longues ou de classes ayant trop de responsabilités.

Ensuite, ces défauts sont classés en trois sous-catégories : intra classes, inter classes et comportementales. Afin de les détecter, les auteurs proposent de les décrire de façon formelle. Ils soulèvent le fait que décrire de façon formelle un patron de conception

est plus facile qu'un anti-patron. Pour le moment, ils en sont encore à étudier le formalisme qu'ils emploieront. Les techniques actuelles de détection sont plutôt des techniques manuelles comme l'inspection du code.

3.3.1.8 Approche de Niere, Wadsack et Zündorf

Niere, Wadsack et Zündorf (Niere, Wadsack et Zündorf, 2001) s'attaquent à la récupération des informations d'un système afin de permettre l'approche circulaire du génie logiciel avec leur environnement de génération de code. La première partie consiste à récupérer les informations statiques sous forme d'un diagramme de classes, ce qui est relativement facile. Ensuite, ce qui est plus compliqué, il y a la récupération de diagrammes de collaboration représentant son fonctionnement.

Afin de créer les diagrammes de collaboration représentant les interactions entre les entités des diagrammes de classes, ils utilisent des clichés définis dans des diagrammes de collaboration qui sont ensuite traduits en GFRN. Ceux-ci sont appliqués à l'arbre syntaxique abstrait qui est annoté lorsqu'un patron y est détecté. À partir de ces annotations, des diagrammes de collaboration sont générés.

La figure 3.9 montre un bout de code Java qui a été annoté lors de la détection des clichés. Ces annotations ont ensuite été utilisées afin de créer le diagramme de collaboration aussi donné à la figure 3.9. Nous voyons que la ligne 35 utilise la méthode `getIdUnit`. Elle a été annotée comme la méthode d'accès en lecture pour une association entre la classe `Switch` et la classe `IdentificationUnit`. Ceci nous permet d'interpréter la ligne 35 comme une opération « link look up ». Dans un diagramme de collaboration, cette opération est montrée comme une ligne identifiée par le nom de l'association correspondante. Cette ligne relie deux boîtes représentant la source et la variable ciblée. Dans l'exemple, ce sont les variables `this` et `idU`, respectivement. De même, les annotations au sujet des méthodes d'accès permettent d'interpréter la ligne 39 et 40 comme le « look up » d'une association qualifiée avec

une cardinalité 0..1. Les lignes 42 et 44 montrent un cliché typique pour le « look up » d'une association à plusieurs. Dans un diagramme de collaboration, ce type de « look up » est montré comme une ligne entre les objets appropriés. Finalement, les annotations nous permettent d'interpréter, que la ligne 49 crée un lien entre l'objet s et l'`idU` et la ligne 51 crée un lien entre `wantsTo` et `t1`. Le diagramme de collaboration montre la création de lien en utilisant la couleur grise et le stéréotype « create ».

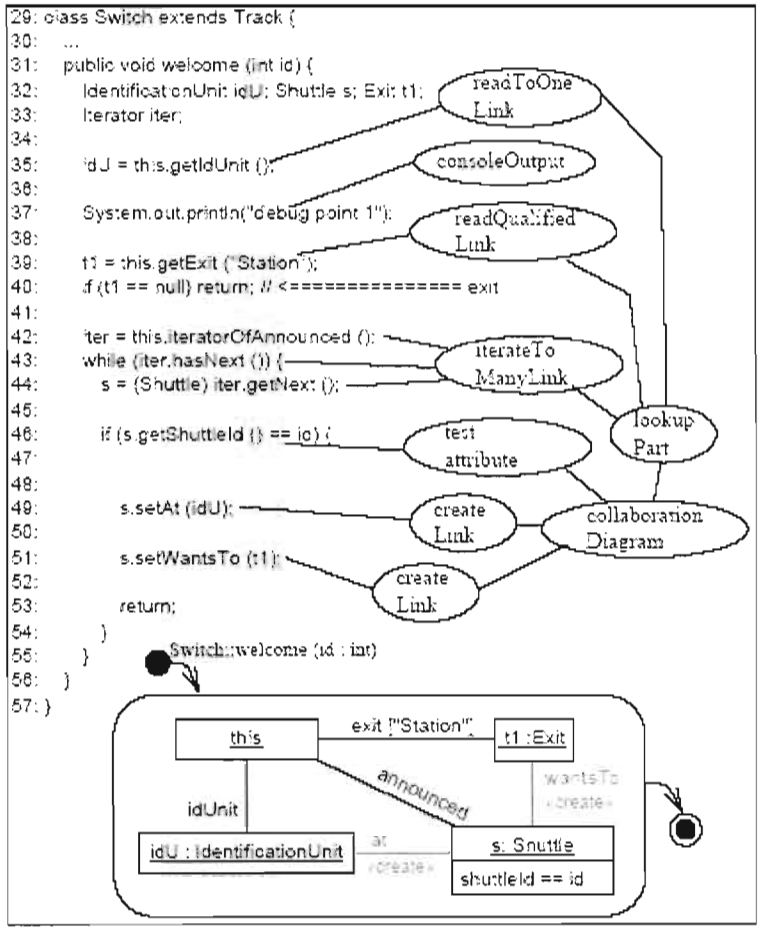

Figure 3.9 Diagramme de collaboration (figure tirée de Niere, Wadsack et Zündorf, 2001)

3.3.1.9 Approche de Paul et Prakash

Paul et Prakash (Paul et Prakash, 1994) ont développé un outil nommé SCRUPULE. C'est un outil consacré à la recherche de patrons dans du code source. Dans ce cas, un patron représente une structure textuelle particulière dans le code source. Pour ce faire, ils utilisent un langage de description de patron. Ils affirment que ce langage contrairement à d'autres techniques de recherches textuelles, comme les expressions régulières, permet de repérer facilement des constructions spécifiques à la programmation. Par exemple, il est possible de trouver une partie de code contenant un « `while` » suivie plus loin d'un « `for` », tout en spécifiant s'il est ou non imbriqué. Le langage de SCRUPULE est conçu pour être semblable au langage du code source visé afin d'en faciliter l'usage. Lors d'une recherche, le code est converti en arbre syntaxique abstrait. La requête que l'utilisateur désire effectuer sur le code source est quant à elle transformée en un automate, auquel est appliqué l'arbre syntaxique abstrait. Si l'arbre syntaxique abstrait contient le patron, l'automate atteindra un état permettant de le localiser.

3.3.1.10 Approche de Philippow *et al.*

Philippow *et al.* (Philippow *et al.*, 2005) présentent une approche de recherche de patrons de conception. L'approche se fonde sur un ensemble de critères positifs et négatifs décrivant ces patrons. Alors que les critères positifs identifient les occurrences possibles des patrons de conceptions, les critères négatifs servent à enlever les faux positifs détectés par les règles positives. Ils donnent en appendice la liste des critères pour 23 patrons de conception du livre de Gamma *et al.* (Gamma *et al.*, 1995).

Par exemple, voici les critères pour identifier le patron « `Strategy` ». Pour les critères positifs, la classe racine doit être abstraite, toutes les classes secondaires doivent avoir la même interface publique que leur super classe et il doit y avoir une

référence à la classe de base. Pour les critères négatifs, aucune classe stratégie concrète ne doit contenir une référence à la classe stratégie abstraite, aucune stratégie concrète ne doit détenir une référence à une autre stratégie concrète et il ne doit pas y avoir de références aux stratégies concrètes.

3.3.1.11 Approche de Pinali

Pinali (Pinali, 1999) présente dans son mémoire, une utilisation de l'intelligence artificielle appliquée à la recherche de patrons de conception. L'approche s'intéresse à la détection de patrons prédéfinis. Elle commence donc par la description de ces patrons. Les patrons sont donc d'abord définis en utilisant le langage UML avec quelques enrichissements. Par exemple, à la figure 3.10 nous voyons la description du patron « `AbstractFactory` » en UML.

Ce patron est ensuite traduit dans un prototype le représentant en code Java. La figure 3.11 représente la traduction de la figure 3.10.

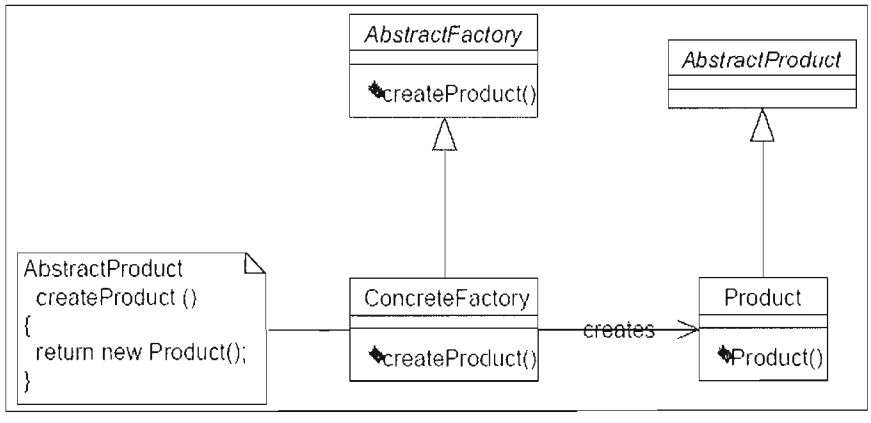

Figure 3.10 Abstract Factory en UML (tirée de Pinali, 1999)

```
abstract public class AbstractFactory
{
        abstract public AbstractProduct createProduct ();
}

public class ConcreteFactory extends AbstractFactory
{
        public AbstractProduct createProduct ()
        {
                return new Product();
        }
}

abstract public class AbstractProduct
{
}

public class Product extends AbstractProduct
{
        public Product ()
        {
        }
}
```

Figure 3.11 Traduction de la figure 3.10 en code Java (tirée de Pinali, 1999)

Une fois ce code obtenu, il est traduit en faits dans le moteur d'inférence. Le logiciel est aussi traduit en faits dans le moteur à l'aide de son arbre syntaxique abstrait. En appliquant les règles aux faits, il est alors possible de détecter des patrons de conception.

3.3.1.12 Approche de Rich et Wills

Rich et Wills (Rich et Wills, 1990) présentent une approche pour identifier des clichés dans le code d'un logiciel. L'intérêt de leur travail est de présenter le fait qu'une représentation intermédiaire puisse faciliter la tâche de la rétro-ingénierie. Ils traduisent donc le code à analyser en « plan calculus » : une représentation indépendante qui combine les flux de données et de contrôle. Cette conversion présente plusieurs avantages. Des codes syntaxiquement différents peuvent avoir une représentation similaire. Des parties de codes non contingentes qui traitent les mêmes données se retrouvent regroupées. Une implantation différente du même cliché peut ainsi devenir un même « plan calculus ». De même pour le chevauchement de clichés,

qui est ainsi démêlé, car on retrouve parfois plusieurs clichés dont le code les implantant qui se chevauchent et la traduction de ce code peut permettre de les isoler.

Le « plan calculus » est créé à partir du code source et est transformé en graphe de flux. La bibliothèque de clichés est aussi encodée sous forme de « plan calculus » et est transformée en graphe de flux. L'analyse de ces graphes donne un arbre de conception représentant les différents clichés trouvés dans le code original. Ce graphe est ensuite traduit dans un pseudo-code s'approchant du langage naturel.

3.3.1.13 Approche de Riva *et al.*

Riva *et al.* (Riva *et al.*, 2006) s'intéressent à l'identification de patrons de conception. Ils proposent une approche plus manuelle qu'automatique divisée en trois phases.

La phase de rétro-ingénierie se concentre sur la création d'un modèle architectural. Elle se base de préférence sur l'utilisation d'un outil de rétro-ingénierie pour créer un diagramme de classes en analysant le code source. Ce diagramme sera ensuite raffiné d'après la documentation. Si les sources ne sont pas disponibles, mais que le système est bien documenté, la documentation servira à la création de ce diagramme.

La phase d'identification des sous-systèmes vise à identifier les sous-systèmes qui abordent une problématique spécifique et les rapports entre eux. Le résultat est un modèle architectural réorganisé en sous-systèmes par une problématique.

La phase d'identification des patrons de conceptions utilise le modèle précédent afin d'identifier des instances d'un groupe de patrons connus. La méthode décrit pour les patrons « Flyweight », « hybrid Mediator-Observer » et « Strategy » les traces de leur présence afin de les détecter. Finalement, toujours en analysant le modèle précédent, ils tentent de découvrir des patrons candidats non connus qui sont une solution répondant à une problématique et qui ne sont pas des compositions des patrons classiques. Ces patrons candidats sont placés dans un ensemble de patrons candidats et adoptés seulement s'ils sont retrouvés dans d'autres systèmes.

3.3.1.14 Approche de Wills

Wills (Wills, 1993) fait la recherche de clichés en se basant sur les diagrammes de flux. Le code source est d'abord converti en diagramme de flux. Une bibliothèque de clichés est élaborée et encodée dans un ensemble de contraintes applicables à un diagramme de flux. En analysant les sections du diagramme de flux correspondant à ces contraintes, il est possible d'identifier les instances des clichés. Une fois ces instances de clichés identifiées, le code contenant un cliché peut en contenir d'autres. Il est donc possible de bâtir un arbre identifiant les liens entre ces clichés.

3.3.2 Approches basées sur l'analyse syntaxique

Dans cette section, nous allons présenter plusieurs travaux basés sur l'analyse syntaxique.

3.3.2.1 Approche de Anquetil et Lethbridge

Anquetil et Lethbridge (Anquetil et Lethbridge, 1999) ont développé une approche pour partitionner un logiciel en sous-systèmes. Pour atteindre leur but, ils se basent sur le nom des différents fichiers contenant le code source du logiciel. L'analyse de ces noms est compliquée par le fait qu'ils peuvent contenir beaucoup d'abréviations. Abréviations qui sont, ou ne sont pas, divisées par des séparateurs et qui peuvent même se chevaucher. Par exemple, le nom « mnut » peut signifier «MeNU+ T » ou « Monitor+ Utility ».

Les auteurs ont exploré notamment les techniques suivantes :

- Analyser les commentaires contenus dans le fichier dont ils cherchent à identifier le nom.
- Utiliser des dictionnaires afin d'associer des mots de la langue anglaise à des abréviations possibles trouvées dans le nom des fichiers.

- Comparer le nom des identificateurs avec différentes métriques.
- Comparer le nom des identificateurs des méthodes et des attributs retrouvés dans le fichier avec celui du nom du fichier.

L'approche commence donc par la division de chaque nom de fichiers en sous-patrons en employant les abréviations qui le constituent. Ceci est déjà un problème en soi. Pour ce faire, ils créent un ensemble d'abréviations qu'ils s'attendent à trouver dans le nom d'un fichier. Par exemple, ils cherchent des noms dans le texte du fichier ou dans un dictionnaire. Ensuite, ils choisissent les abréviations importantes de l'ensemble, les cherchent dans le nom des fichiers et tentent d'identifier la décomposition la plus probable pour chaque nom de fichier. Après avoir identifié les abréviations dans le nom des fichiers, ils créent un groupe pour chaque abréviation trouvée à l'étape précédente. Ils placent un nom de fichier dans un groupe s'il correspond à l'abréviation.

3.3.2.2 Approche de Baxter *et al.*

Baxter *et al.* (Baxter *et al.*, 1998) présentent un outil de détection de clones plus ou moins identiques. Le code source est d'abord placé dans un arbre syntaxique abstrait dont les sous-arbres sont ensuite classés dans une table de hachage selon leurs caractéristiques. Une fonction de hachage plus ou moins floue (selon le degré de similarité désiré) permet de regrouper les parties de codes qui se ressemblent et ainsi de découvrir des clones. Ensuite, l'outil tente de regrouper les clones trouvés afin d'en découvrir de plus gros. Les clones pouvant être regroupés sont alors enlevés de la liste des clones trouvés. Les résultats sont affichés et une macro est proposée pour remplacer les instances de clones dans le code source C.

3.3.2.3 Approche de Bowman, Wgodfrey et Holt

Bowman, Wgodfrey et Holt (Bowman, Wgodfrey et Holt, 1999) ont développé trois outils de création de diagrammes de classes UML pour la représentation de logiciels Java. Le but était de comparer trois techniques d'extraction de l'information nécessaire à la construction de ces diagrammes. Les trois techniques explorées étaient l'analyse du code source, l'analyse du code octet (« byte code ») et l'analyse des traces d'exécution.

Les auteurs ont conclu que l'analyse du code permet de retrouver toutes les informations statiques qui y sont contenues. Par contre, cette analyse n'est pas une tâche simple. Sa mise en œuvre est complexe et le risque d'erreur est donc grand. L'analyse du code octet, quant à elle, est plus facile à mettre en application. En effet, le gros du travail d'analyse du code source a déjà été effectué par le compilateur et cette information est facilement récupérable dans l'exécutable. Par contre, l'optimisation effectuée par le compilateur peut enlever de l'information ou transformer des informations. Le compilateur peut ajouter des choses qui ne reflètent pas la conception originale et par conséquent ces artefacts peuvent demander du travail pour être enlevés. Finalement, utiliser l'analyse des traces pour enrichir l'information découverte statiquement permet la découverte d'information non discernable statiquement, comme le polymorphisme. Par contre, cette analyse est dépendante de l'exécution analysée. Il est donc nécessaire de s'assurer que les traces d'exécutions représentent une exécution convenable du logiciel analysé afin de couvrir toute l'étendue du code ou des fonctions désirées dans le cas de tranchage.

3.3.2.4 Approche de Caprile et Tonella

Caprile et Tonella (Caprile et Tonella, 1999) s'intéressent aux identificateurs contenus dans un logiciel. Ils les analysent d'une manière lexicale, syntaxique et grammaticale. Ils proposent d'utiliser les identificateurs afin d'identifier une

grammaire standard représentant les identificateurs dans un logiciel et pour adapter ceux ne la respectant pas. Ils suggèrent que l'approche puisse aussi servir à enlever les synonymes dans les noms des identificateurs. Les résultats pourraient être aussi utiles pour la documentation, la traçabilité, l'assurance-qualité par l'utilisation de métriques et aider la migration vers l'orienté objet. À noter que dans leur cas, ils analysent du code C.

Les premières étapes de leur approche consistent à extraire le dictionnaire des mots contenus dans les identificateurs. Le dictionnaire des mots d'un logiciel est défini comme le plus petit ensemble nécessaire de mots afin de segmenter tous les identificateurs. Ils commencent avec un dictionnaire vide. Un logiciel écrit en LISP tente la segmentation des mots. S'il en est incapable, il demande l'aide d'un humain. Les mots segmentés sont ajoutés au dictionnaire.

Ensuite, un dictionnaire typé est construit. Ce dictionnaire associe les mots avec la fonction qu'ils exécutent dans l'identificateur. Par exemple la plupart du temps le terme « free » est associé à un verbe. Les termes sont classés de façon lexicale selon 7 classes : formes anglaises, abréviations, acronymes, chaînes de caractères spéciaux (par exemple, les séparateurs en langage C), nombres et autres. Grammaticalement, ils tentent à partir du dictionnaire typé, de construire une grammaire représentant les identificateurs trouvés dans le logiciel.

Dans un autre article, Caprile et Tonella (Caprile et Tonella, 2000) présentent une autre approche de restructuration des identificateurs d'un logiciel basée sur celle-ci (figure 3.12). Ils commencent par extraire les identificateurs de l'arbre syntaxique abstrait et créer le dictionnaire, en s'assurant de garder un lien entre les composantes dans l'arbre et ceux dans le dictionnaire. Les traitements décrits précédemment sont appliqués au dictionnaire. Ceci permet d'éliminer les problèmes dans les identificateurs comme les synonymes et le non-respect de la grammaire découverte. Une fois les identificateurs corrigés, ils sont replacés dans l'arbre syntaxique abstrait qui est ensuite employé afin de générer le code source.

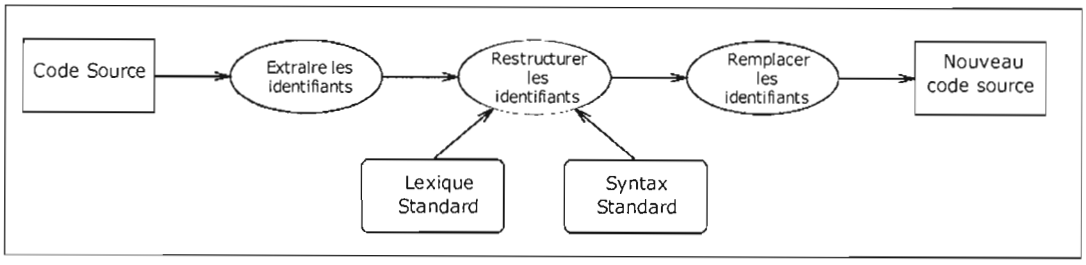

Figure 3.12 Procédure de restructuration des identificateurs (adaptée de Caprile et Tonella, 1999)

3.3.2.5 Approche de Chen, Nishimoto et Ramamoorthy

Chen, Nishimoto et Ramamoorthy (Chen, Nishimoto et Ramamoorthy, 1990) décrivent une approche pour le stockage des composantes d'un logiciel dans une base de données standard et son utilisation en compréhension du logiciel avec un système pour analyser la structure de programmes en C.

Ils commencent par extraire la structure d'un logiciel écrit en C et la traduisent dans un modèle exploitable dans une base de données. Cette base de données comprend cinq types d'objets : fichiers, macros, variables globales, types de donnée et fonctions. Chaque objet aura des attributs comme le fichier le contenant et des relations d'utilisation comme « inclus dans » ou « se réfère à ». Les relations pourront aussi avoir des attributs.

Le système permet d'abord des représentations textuelles comme l'information sur un objet précis, les relations entre deux objets et le code source relié à un objet. Mais plus intéressant, des représentations graphiques permettant de voir les appels de fonctions et les dépendances entre les fichiers. Nous pouvons aussi extraire des sous-systèmes en analysant les dépendances de la fonction que nous voulons extraire. Ils présentent un concept de couche en créant un arbre ayant au premier niveau les fonctions qui ne sont pas appelées, au second niveau celles qui sont appelées

seulement par celles de premier niveau, et ainsi de suite. Aussi, l'analyse de cette base permet de détecter le code mort et la cohésion entre deux objets.

3.3.2.6 Approche de Ducasse, Rieger et Demeyer

Ducasse, Rieger et Demeyer (Ducasse, Rieger et Demeyer, 1999) présentent une méthode visuelle de détection de clones qui est indépendante du langage. Comme composante à analyser, ils ont choisi d'analyser des lignes de code. Afin d'éliminer simplement le plus de variations possibles inutiles, ils commencent pour chaque ligne de code par enlever les espaces et les commentaires. Comme comparer chaque ligne avec chaque ligne demande un temps de calcul important, ils utilisent une table de hachage. Cette table permet de regrouper des lignes possédant des points communs. Une comparaison de chaque entité de chaque compartiment du tableau de hachage est effectuée. Le résultat est alors placé dans une matrice où chaque ligne de code est à la fois une ligne et une colonne. Ensuite en analysant la matrice il est possible de trouver des informations.

La figure 3.13 montre des matrices représentant le code d'un logiciel. Les colonnes et les lignes représentent une même ligne de code. Il y a une diagonale au centre, car une ligne est identique à elle-même. Dans la matrice a, les autres diagonales indiquent du code identique. Dans la matrice b, l'espace dans la diagonale indique un code cloné qui a été modifié. Dans la matrice c, la diagonale cassée signifie que du code a été ajouté au clone. Dans la matrice d, les petites lignes qui se répètent (« rectangles ») indiquent une occurrence périodique du même code.

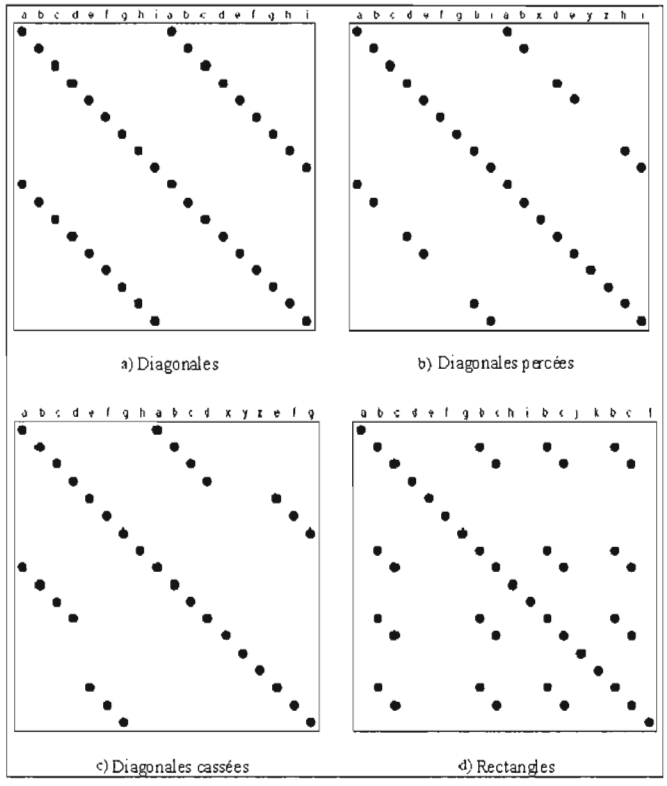

Figure 3.13 Matrice de détection de clones (tirée de Ducasse, Rieger et Demeyer, 1999)

3.3.2.7 Approche de Ducasse, Lanza et Bertuli

Ducasse, Lanza et Bertuli (Ducasse, Lanza et Bertuli, 2004) s'intéressent au cycle de vie des objets lors de son exécution. Ils cherchent à répondre à des questions telles que : Quelles sont les classes les plus instanciées ? Quelles sont les classes qui créent des objets ? Comment les classes communiquent-elles ? Quel est le pourcentage des méthodes d'une classe qui est employé ?

En se basant sur les traces d'exécution d'un logiciel, ils proposent une approche de visualisation interactive qui permet de répondre à ces questions et à d'autres. Les informations récupérées dans les traces sont notamment : le nombre de méthodes appelées, le rapport entre le nombre de méthodes appelées et celles non appelées, le

nombre d'invocations internes d'une méthode par une classe, le nombre de méthodes (statiques) appelées pour une classe, le nombre d'objets créés d'une classe, le temps d'exécution d'une méthode, le nombre d'invocations d'une méthode.

Leur approche permet de condenser l'information des traces d'exécutions dans une représentation plus compréhensible. Comme nous le voyons à la figure 3.14, leur graphe peut afficher l'information par : la position du nœud en x et y, sa couleur, sa largeur, sa hauteur et les liens avec les autres.

En employant les informations des traces, ils peuvent créer de représentations affichant par exemple : les classes qui sont créées et utilisées durant l'exécution du logiciel, l'intercommunication entre les instances durant l'exécution ou la création des classes durant l'exécution. Par exemple, la figure 3.15 représente l'usage des instances d'une classe. Les petits rectangles blancs représentent les classes qui n'ont pas été instanciées. Les rectangles pâles et étroits représentent des classes invoquées, mais avec peu d'instances. Les rectangles pâles et allongés sont les classes qui ont été instanciées souvent, mais peu utilisées. Les rectangles foncés et plats représentent les classes qui ont été instanciées souvent et assez utilisées et dont les méthodes ont été beaucoup invoquées. Finalement, les gros rectangles indiquent les classes qui ont été beaucoup instanciées et dont les méthodes ont été très employées.

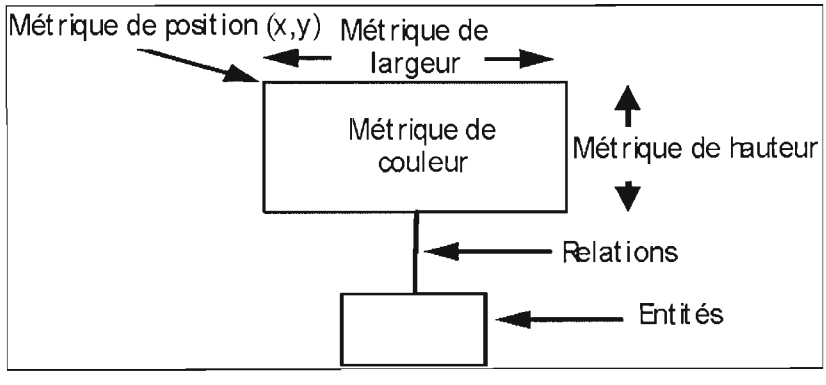

Figure 3.14 Représentation polymétrique (adaptée de Ducasse, Lanza et Bertuli, 2004)

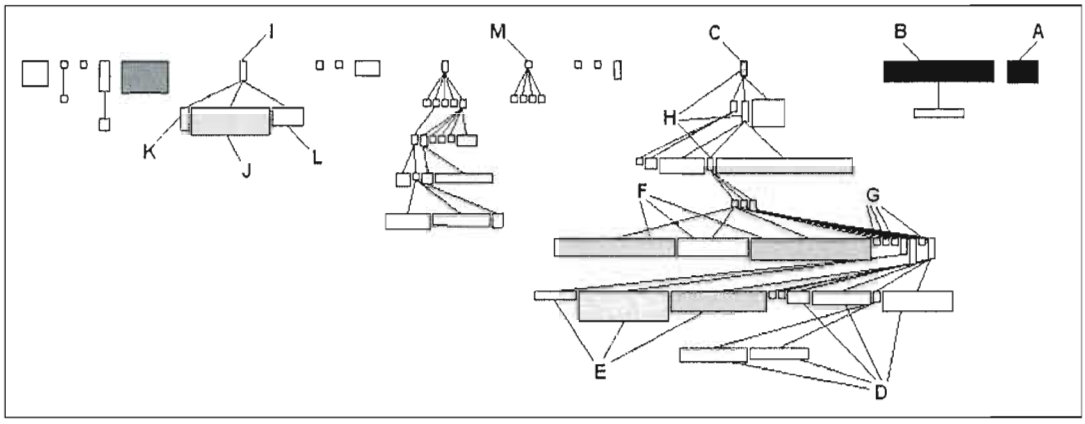

Figure 3.15 Affichage polymétrique d'usage de classes (tirée de Ducasse, Lanza et Bertuli, 2004)

3.3.2.8 Approche de Feild, Binkley et Lawrie.

Feild, Binkley et Lawrie (Feild, Binkley et Lawrie, 2006) s'intéressent à l'extraction des mots contenus dans les identificateurs. Ils testent deux approches de séparation des identificateurs, une approche gloutonne et une employant un réseau neuronal.

Dans leur algorithme, ils utilisent trois listes. Un dictionnaire de mots, un dictionnaire d'abréviations connues et une liste de mots à ne pas conserver. Ils appellent mots, les constituants d'un identificateur. Un mot représente une séquence de caractère avec un sens précis. Deux types de mots sont identifiés : dur (« hard ») et mou (« soft »). Le premier type représente les mots délimités par des séparateurs et le second les autres qui sont définis moins clairement.

La fonction employée par leur approche gloutonne analyse chacun des identificateurs. Pour chacun des mots durs d'un identificateur qui n'est pas dans une liste, elle cherche le plus long préfixe qui est dans la liste, et s'appelle récursivement pour le reste du mot. Elle cherche aussi le plus long suffixe qui est dans la liste, et s'appelle récursivement pour le reste du mot. Les résultats des recherches sont comparés. Le résultat ayant le plus grand ratio de mots moues est conservé. Si aucun résultat n'est

trouvé dans les deux recherches précédentes, ils enlèvent une lettre au mot et recommencent. Le résultat final est la séparation du mot en identificateurs.

L'approche par réseau de neurones se base sur l'entraînement. Ils commencent par fournir des listes contenant des identificateurs et leur séparation idéale. Elle servira à entraîner le logiciel et à le tester. Après cette phase d'entraînement, le réseau pourra de lui-même diviser les identificateurs en mots. Avec l'approche gloutonne, ils obtiennent un succès de 74 % à 80 %. Avec le réseau de neurones, le résultat varie selon les listes d'entraînements. Un réseau ayant été entraîné avec les informations de différents logiciels réussira moins bien, car il aura à concilier différents systèmes de nommage.

3.3.2.9 Approche de Hamou-Lhadj *et al.*

Hamou-Lhadj *et al.* (Hamou-Lhadj *et al.*, 2005) présentent une approche ayant pour objectif de filtrer les traces d'exécution d'un logiciel en retirant les composantes utilitaires et en ne conservant que celles qui implantent un concept de haut niveau.

Ils commencent par générer les traces pour certaines parties du logiciel. Ces traces sont filtrées en retirant les utilitaires qu'elles contiennent. Ils sont retirés en se basant sur le fait qu'un utilitaire est accédé de plusieurs endroits d'un logiciel. Ce qui se concrétise dans un graphe des dépendances par le fait que les classes utilitaires vont être des nœuds « puits ». Donc à partir du graphe des dépendances des classes, ils donnent une note comprise entre 0 et 1 à chaque module. Il s'agit de trouver le bon seuil afin d'isoler les classes utilitaires. À partir de cette évaluation, ils commencent par détecter les utilitaires globaux afin de faire une première partition du logiciel. Ensuite, ils isolent à l'intérieur de ceux-ci les utilitaires plus locaux. Le résultat est finalement affiché dans un diagramme montrant le flot d'exécution entre les différentes fonctions.

3.3.2.10 Approche de Hung et Zou

Hung et Zou (Hung et Zou, 2005) proposent une approche pour extraire la logique d'affaires d'un logiciel. Ils supposent un logiciel basé sur une architecture trois niveaux : interface, logique et base de données. L'approche se base sur l'identification des données d'affaires et les politiques d'affaires. Ils identifient ces informations en examinant le flux d'information entre la couche logique et la couche de base de données.

Les auteurs commencent par identifier les règles d'affaires contenues dans le logiciel. D'abord, ils identifient les données d'affaires en se basant sur les opérations de lecture et de mises à jour des données qu'effectue la couche logique sur la couche de base de données. Ensuite, ils identifient les directives qui conditionnent l'exécution de la logique d'affaires sur les données d'affaires. Pour se faire, ils cherchent à filtrer celles spécifiques à l'implantation et non à logique d'affaires. Finalement, chaque séquence d'exécution représentée par les directives est représentée comme une règle d'affaires. Ils terminent par l'identification du processus d'affaires qui représente les liens entre les règles d'affaires. Cela est créé par une analyse statique qu'ils ont présentée dans un autre travail (Zou *et al.*, 2004).

3.3.2.11 Approche de Mendelzon et Sametinger

Mendelzon et Sametinger (Mendelzon et Sametinger, 1995) démontrent l'intérêt d'utiliser des outils généralistes afin de faire de la rétro-ingénierie. Ils utilisent l'outil Hy+ (Mendelzon, 1993) qui permet d'afficher des objets et les relations issues d'une base de données. Cet outil est employé avec le langage Graphlog. Il s'agit d'un langage qui permet avec des requêtes de spécifier de nouvelles relations pouvant être affichées. Ils se servent d'une base de données comportant les principales caractéristiques du logiciel à analyser.

Graphlog permet de spécifier des expressions arithmétiques dans les requêtes. Il est donc possible de rechercher par exemple, les méthodes qui sont surchargées dans plus de 20 classes et d'afficher à l'écran le résultat. Il est aussi possible grâce à des requêtes d'analyser les composantes du logiciel et de vérifier si elles répondent à certaines contraintes. Comme Graphlog permet l'utilisation d'expressions régulières, il serait possible de trouver les classes ne respectant pas la contrainte que le nom d'une classe doit débuter par une majuscule. Ils démontrent aussi que nous pouvons utiliser l'approche pour la détection de patrons de conception. Par exemple selon les auteurs, la requête pour trouver une « factory » est plutôt simple, tout ce qu'il y a à faire est de rechercher des méthodes virtuelles contenant le mot « Make » ou « Create ».

3.3.2.12 Approche de Müller *et al.*

Müller *et al.* (Müller *et al.*, 1993) présentent une méthodologie utilisant l'outil Rigi afin de faire une décomposition descendante en employant une approche ascendante. C'est-à-dire qu'ils commencent par utiliser l'outil de rétro-ingénierie d'une manière « classique » afin d'identifier les sous-systèmes qui sont présents dans le code source du système, il s'agit de la partie ascendante. Une fois que le système est décomposé en sous-systèmes, il devient possible de regrouper et de filtrer les sous-systèmes afin de trouver et de représenter des concepts plus généraux, la partie descendante.

3.3.2.13 Approche de Ohba et Gondow

Ohba et Gondow (Ohba et Gondow, 2005) cherchent à isoler des mots-clefs représentant des concepts dans un logiciel. Ils classent les concepts reflétés par les mots en quatre catégories : les concepts, les rassembleurs (des préfixes ou suffixes qui regroupent des entités), les attributs et concepts moins importants et les verbes génériques (par exemple : « read » et « set »).

Leur approche est basée sur la fréquence des identificateurs dans un logiciel. Ils commencent par isoler la liste des identificateurs des fonctions et des attributs. Ensuite, ils les séparent en mots (ici, ils supposent des mots dont les séparateurs sont clairement définis). Ils donnent par la suite des pointages aux mots en calculant leur fréquence locale et leur fréquence globale. Un mot obtient un pointage plus fort en relation avec un autre s'il apparaît plus souvent dans un artefact et moins souvent dans un autre. Le tout est enregistré dans une base de données. Un navigateur permet par la suite de voir les occurrences des différents termes dans différents identificateurs. Nous voyons dans l'arborescence de la figure 3.16, les identificateurs qui apparaissent dans plus d'un fichier sont dans un carré, et ceux qui sont seulement dans un fichier le sont que de façon locale. Un lien entre deux nœuds indique que le patron se retrouve dans un même identificateur. Selon eux, un concept humain correspond souvent à un nœud non global avec plusieurs liens comme `dirent` ou `root`.

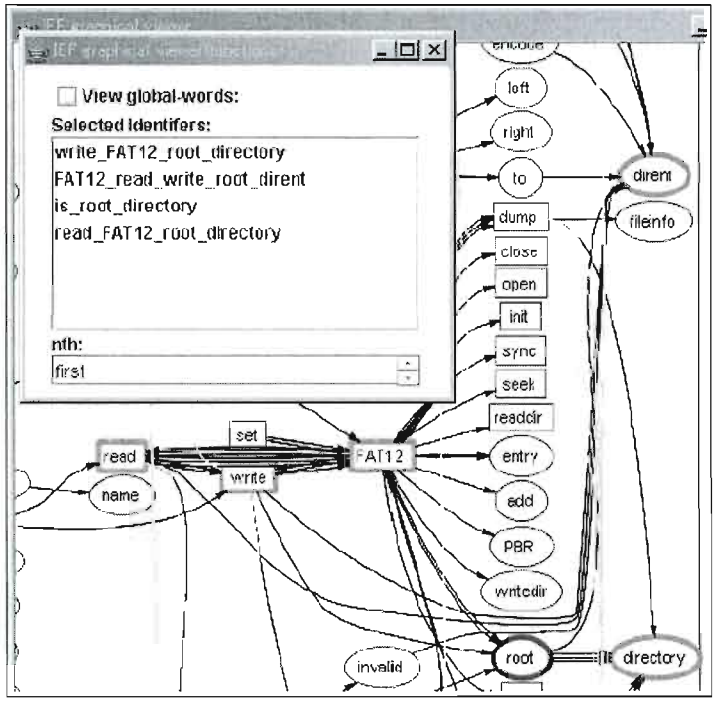

Figure 3.16 Navigateur de termes (tirée de Ohba et Gondow, 2005)

3.3.2.14 Approche de Richner et Ducasse

Richner et Ducasse (Richner et Ducasse, 1999) utilisent la programmation logique dans le but de créer des représentations à la demande. Ils commencent d'abord par emmagasiner des informations statiques et dynamiques sur le logiciel à analyser sous forme de faits.

Par exemple, la figure 3.17 montre qu'`EllipseFigure` provient de l'artefact `HotDraw-Figures`, en 2, qu'`EllipseFigure` hérite de `EllipseFigure`, en 3 que la définition des méthodes `self` de type `displaying` de `EllipseFigure`, en 5 que l'instance 39 de la classe `EllipseFigure` appelle la méthode `fillColor` de l'instance 39 de `ElipseFigure`. Les séquences 5 et 7 décrivent le niveau de la pile.

À partir de ces informations, des représentations sont créées à la demande à partir de requêtes. Par exemple, la requête : `createView(invokesClass, allInCategory)`, demande de créer une représentation avec pour nœud `allInCategory` et pour relations `invokesClass`. `invokesClass(Class1, Class2) :- invokesMethodClass(Class1, Class2)` indique qu'une classe en évoque une autre. Et `allInCategory(Category, ListOfClasses) :- setof(Class, class(Class, Category), ListOfClasses)`, indique dans quelle catégorie de nœuds classer une classe. Cela donne la représentation affichée à la figure 3.18.

```
1. class('EllipseFigure','HotDraw-Figures')
2. superclass('Figure','EllipseFigure')
3. method('EllipseFigure','displayFilledOn:',false,'displaying')
4. access('EllipseFigure','self','EllipseFigure','displayFilledOn
   :')
5. send(5,7,'EllipseFigure',39,'EllipseFigure',39,'fillColor')
6. send(6,8,'EllipseFigure',39,'Drawing',85,'fillColor')
```

Figure 3.17 Usage des instances d'une classe (tirée de Richner et Ducasse, 1999)

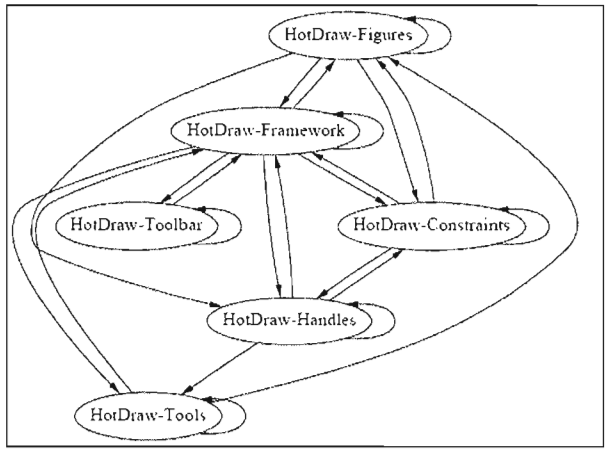

Figure 3.18 Exemple de représentation (tirée de Richner et Ducasse, 1999)

3.3.2.15 Approche de Systä

Systä (Systä, 1999) utilise les traces générées par le débogueur Java afin d'enrichir les informations extraites d'une façon statique. C'est-à-dire que l'approche commence par extraire les informations statiques sur un logiciel à l'aide de l'outil Rigi. Ensuite à l'aide du débogueur Java, elle utilise un outil nommé SCED qui crée des diagrammes de scénarios. Comme les modèles statiques et dynamiques contiennent des informations au sujet d'artefacts du logiciel et leurs relations, ceux-ci peuvent être utilisés comme une connexion permettant l'échange d'information entre les modèles et la navigation entre ceux-ci. Ce regroupement est effectué en recoupant les informations sur les artefacts et leurs relations contenues dans les modèles statiques et dynamiques. Nous voyons à la figure 3.19 un diagramme de scénario généré par leur approche. Dans ce diagramme, nous pouvons entre autres constater que la méthode `addToParam` de la classe `UMLMethod` est appelée par la classe `PEParameters`.

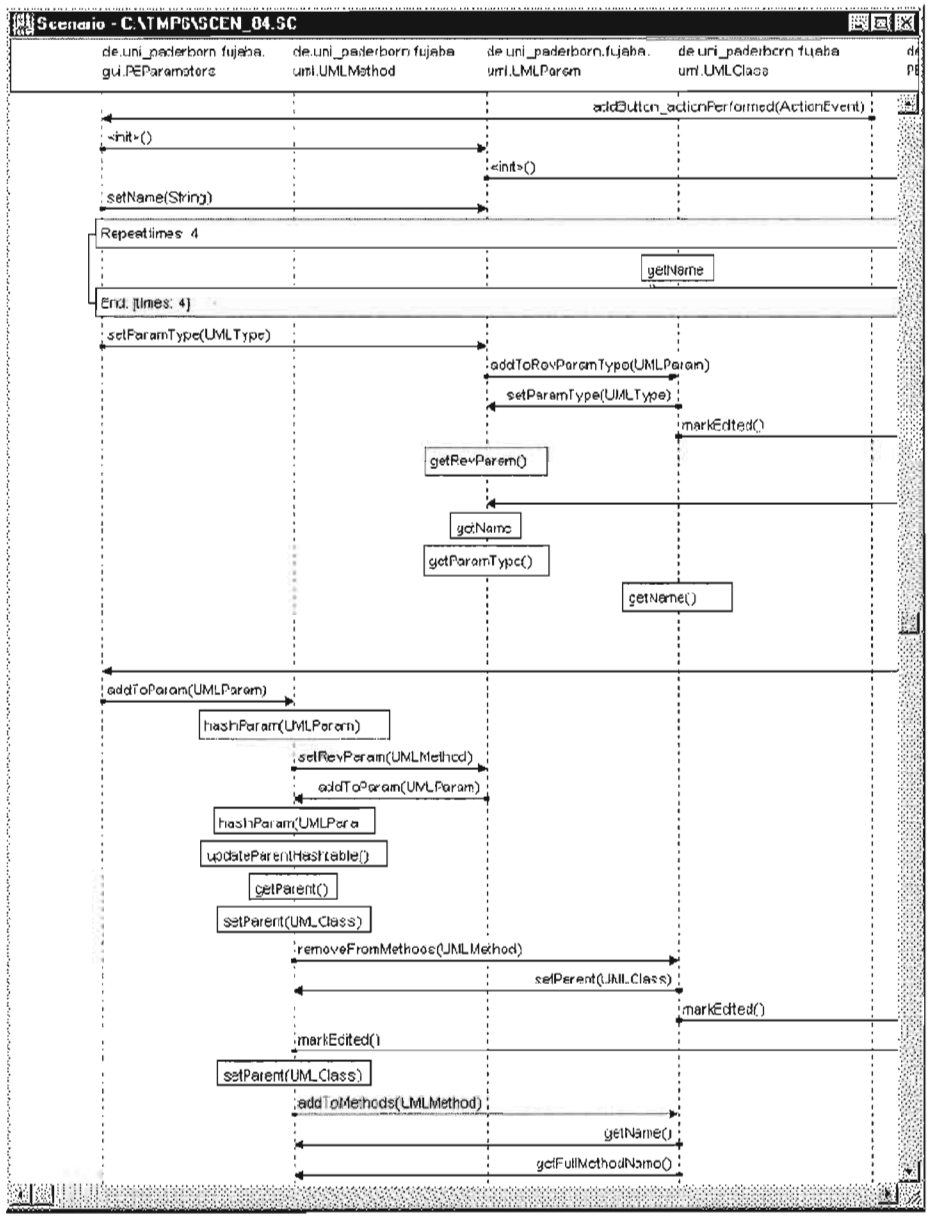

Figure 3.19 Diagramme de scénarios généré (tirée de Systä, 1999)

3.4 DISCUSSION

Nous venons de présenter dans la section précédente plusieurs approches de rétro-ingénierie. Nous résumons et classons ces approches dans les tableaux 3.1, 3.2 et 3.3

suivant notre cadre de classification. Dans ce qui suit, nous allons faire une discussion de ces approches suivant trois aspects à savoir : approches formelles vs approches informelles, validité des approches et intérêts des approches pour nos objectifs.

Tableau 3.1
Tableau récapitulatif des approches basées sur les méthodes formelles

	Entrées	Application	Langage intermédiaire	Objectif
Bowen, Breuer et Lano, 1993	Code source Fortran Code source Cobol	Association de concepts	Z++	Approche circulaire du génie logiciel Code source Fortran Code source Cobol
Ward. 2005	Code source assembleur	Analyse de flux de contrôle Analyse de flux de données	WSL	Traduction et restructurotion de code Code source C
Gannod et Cheng, 1996	Code source C	Plus fortes post-conditions	Z++	Spécifications formelles

Tableau 3.2
Tableau récapitulatif des approches basées sur la détection des clichés

	Entrées	Application	Langage intermédiaire	Objectif
Fiutem *et al.*, 1996	Code source C Makefiles Spécification de clichés	AST Analyse de flux	AST REFINE	Représentation hiérarchique de l'architecture
Heuzeroth *et al.*, 2003	Code source Java Traces d'exécution Spécification de patrons	Détection de patrons spécifiés sous forme d'algorithmes	AST	Détection de patrons de conceptions
Huang *et al.*, 2005	Code source Java Traces d'exécution Spécification de patrons	Logique des prédicats Logique des intervalles	Prolog	Liste des patrons de conception détectés sous forme de faits Prolog
Jahnke, Schäfer et Zundorf, 1997	Code source C Base de données SQL	Logique floue	Réseaux de Pétri	Conversion de base de données en objets
Kramer et Prechelt, 1996	Code source C Spécification de potrons	Requêtes	OMT Prolog	Détection de patrons de conception
Mitchell, Mancoridis et Traverso, 2002	Code source C++ Spécification de relations	Recherche : escalade, génétique et exhaustive	Graphe des dépendances	Partition du logiciel et restructuration
Moha et Guéhéneuc, 2005	Code source Définition d'anti-patrons et de défauts recherchés	Inspection du code		Détection de défouts
Niere, Wadsock et Z., 2001	Code source Java Base de connaissances	Logique floue	AST	Diagramme de collaboration
Paul et Prakash, 1994	Code source Description de clichés textuels	Recherche de patrons d'après description et automate	Arbre syntoxique abstrait	Recherche de patrons textuels
Philippow *et al.*, 2005	Code source C++ Spécification de patrons	Recherche de potrons d'après des critères positifs et négatifs	Représentation de Borland Together	Détection de potrons de conceptions
Pinali, 1999	Code source Java Spécification de patrons	Moteur d'inférence	Base de connaissances Jess	Détection de patrons de conceptions
Rich et Wills, 1990	Code source Lisp Bibliothèque de cliché	Analyse de graphes	« Plan calculus »	Détection de cliché en rapport textuel
Riva et al, 2006	Code source	Procédure manuelle	Diagrammes UML	Détection de patrons prédéfinis ou non
Vokác, 2006	Code source C et CVS	Requêtes SQL	BD SQL	Détection de patrons de conceptions
Wills, 1993	Code source Représentation des clichés	Analyse de graphes	Graphe de flux	Détection d'une bibliothèque de clichés

Tableau 3.3
Tableau récapitulatif des approches basées sur l'analyse syntaxique

	Entrées	Application	Langage intermédiaire	Objectif
Anquetil et Lethbridge, 1999	Code source	Analyse des identificateurs	Liste d'identificateurs	Partition du logiciel
Baxter *et al.*, 1998	Code source C	Table de hachage	arbre syntaxique abstrait	Identification de clones sous forme de rapport textuels et macro de substitution
Bowman, Wgodfrey et Holt, 1999	Code source Java Pseudo-code binaire Traces d'exécutions	Analyse statique Analyse dynamique Analyse du binaire	Tupple-attribute language	Architecture sous forme de diagramme de classe
Caprile et Tonello, 1999	Code source C	Analyse lexicale, syntaxique et grammaticale	Dictionnaires en Lisp	Partition du logiciel
Chen, Nishimata et Ramamoorthy, 1990	Code source C	Interrogation d'une base de données	Base de données	Partition du logiciel Enlèvement de code mort Décisians de conception
Ducasse, Rieger et Demeyer, 1999	Code source	Table de hachage	Matrice	Détection de clanes dans une matrice
Ducasse, Lanza et Bertuli, 2004	Code source Smalltalk Traces d'exécution	Analyse statique Analyse dynamique	CDIF, XMI FAMIX	Cycle de vie des objets saus une représentation polymétrique
Feild, Binkley et Lawrie, 2006	Code source C ,C++, Java et Fortran Liste d'entraînement	Approche gloutonne Réseou de neurones	Dictionnaires Listes de mots	Séparer les identificateurs en mats
Hamou-Lhadj *et al.*, 2005	Code source	Analyse du graphe des dépendances	Graphe des dépendances	Comportement sous forme de diagramme UCM
Hung et Zou, 2005	Traces d'exécutions	Analyse des traces d'exécution	Base de données	Logiques d'affaires sous farme XML
Mendelzon et Sametinger, 1995	Code source	Requêtes	Base de données	
Müller et al, 1993	Code source	Analyse de graphes	GRAS	Architectures Partition du logiciel
Ohba et G., 2005	Code source C	Fréquence des identificateurs	Matrice de cooccurrence	Explorateur du graphe de co occurrences
Richner et D., 1999	Code source Smalltalk	Analyse statique Analyse dynamique	Faits Prolog	Architecture saus forme de graphes
Systä, 1999	Exécutable Java Trace d'exécution Java	Regroupement des informatians statiques et dynamiques	Rigi Standard format Diagramme de scénario	Relation entre les composantes

3.4.1 Approches formelles vs approches informelles

Nous voyons que la plupart des approches présentées se retrouvent du côté informel. Ce fait reflète la réalité où les approches formelles sont moins exploitées et par un groupe restreint de chercheurs.

Aussi dans le domaine de la compréhension du logiciel, la traduction simple et moins intéressante. Ainsi dans ce domaine, l'approche de Gannod et Cheng (Gannod et Cheng, 1996) pour dériver des spécifications formelles de code source C est moins intéressante. Car, elle ne fournit évidemment pas d'abstraction. C'est pourquoi les approches transformationnelles sont plus intéressantes. Bowen, Breuer et Lano (Bowen, Breuer et Lano, 1993), Guo, Liao et Pamula (Guo, Liao et Pamula, 2006) se basent au moins en partie sur la traduction, mais font des abstractions pour ne pas se limiter à une simple traduction, ce qui est déjà plus intéressant. Pour Ward (Ward, 2004), il s'agit d'une étape intermédiaire où le code est finalement transformé en C. Ce qui contradictoirement serait plus utile à bien des programmeurs que des spécifications formelles.

3.4.2 Validité des approches

Il est généralement extrêmement difficile de juger la validité des approches proposées car la plupart de celles-ci n'abordent que superficiellement ce sujet. D'autres parts, les approches ne sont pas toujours présentées avec un détail qui nous permet de les juger. Dans ce qui suit, nous allons quand même montrer des limites à certaines des approches que nous avons pu identifier.

Nous avons vu que plusieurs travaux basés sur la détection de clichés sont disponibles dans la littérature. Beaucoup recherchent des patrons de conception. Dans ceux-ci, tous proposent un langage pour spécifier un patron de conception à rechercher. Certains s'intéressent seulement à la spécification statique d'un patron. D'autres s'intéressent aussi à la description dynamique des patrons de conception. Les travaux basés sur la détection des clichés présentent une limite majeure à savoir

qu'il est difficile sinon impossible de prévoir toutes les implantations possibles d'un patron de conception. Par exemple, Kim et Benner ont identifié que le patron Observer possède au moins douze implantations différentes (Kim Jung et Benner Kevin, 1996).

En ce qui concerne les approches basées sur les identificateurs (Anquetil et Lethbridge, 1997; Feild, Binkley et Lawrie, 2006). Nous pouvons inclure l'approche que nous présentons dans ce mémoire dans ces approches. Certaines de ces approches se basent sur l'existence des dictionnaires afin d'associer des mots de la langue anglaise à des abréviations possibles trouvées dans les identificateurs. Ainsi, la qualité de ces approches repose sur celle des dictionnaires. En outre, qu'en est-il d'autres langues comme le français ou du code qui utilisent plusieurs langues?

Finalement, plusieurs approches (Baxter *et al.*, 1998; Caprile et Tonella, 1999) supportent les systèmes non orientés objet. Il est intéressant de voir comment ces approches vont se comporter devant des systèmes orientés objet. Nous avons déjà indiqué dans le premier chapitre que les anciennes méthodes sont plus adaptées aux anciennes technologies et que les nouvelles technologies vont continuellement en nécessiter de nouvelles.

3.4.3 Intérêts des approches pour nos objectifs

Les approches que nous avons discutées, même celles possédant des limites, offrent une mine inestimable pour nous. En effet, il suffit d'étudier les techniques employées pour retenir plusieurs idées qui peuvent être utilisées pour nos objectifs actuels et futurs. Nous en présentons plusieurs dans cette section.

En outre, certaines approches fournissent, par leurs objectifs, une aide directe pour nous. Ainsi, les approches basées sur les identificateurs, même avec leurs limites, présentent un intérêt certain pour nous comme nous allons voir dans les prochains chapitres. Ainsi, des travaux sont actuellement en cours pour voir si ces approches peuvent être appliquées, comme traitement préalable à notre approche, afin que cette

dernière puisse supporter les applications où les développeurs n'ont pas suivi les bonnes règles de nommage.

Le travail présenté par Hamou-Lhadj *et al.* (Hamou-Lhadj *et al.*, 2005) présente aussi un intérêt puisqu'il permet de détecter des classes utilitaires; alors que le nôtre ne le permet pas toujours. Comme notre approche a de la difficulté à filtrer des classes utilitaires, nous allons dans le futur étudier celle-ci et d'autres approches afin de filtrer les classes utilitaires.

Nous pensons que la technique de l'analyse de flux employée par Hung et Zou (Hung et Zou, 2005) présente un intérêt certain. En effet, l'approche propose d'extraire la logique d'affaires d'un logiciel et par conséquent ils ont le même objectif que nous. Nous considérons que l'analyse de flux d'information peut donner de bons résultats pour nos objectifs. Évidemment, nous voulons être plus généraux que Hung et Zou.

Finalement, les approches ayant comme objectif de faire de la détection des clones présentent aussi un intérêt pour nous. En effet, nous allons voir que le code relié à l'implantation d'une architecture est souvent répétitif. Un enjeu de taille reste cependant du fait que l'architecture est souvent aussi semi-répétitive. Cette semi-répétition dépend des métadonnées que nous pouvons trouver dans un code source. Ainsi, le code des méthodes des *getters* et *setters* est plutôt semi-répétitif dans la mesure où il varie suivant le nom des attributs. Cette identification est complexe lorsque les règles de nommage ne sont pas uniformes partout dans le code.

3.5 CONCLUSION

Dans ce chapitre nous avons présenté diverses approches de rétro-ingénierie et ainsi montré un large éventail de cette pratique en regard de la compréhension du logiciel. Nous avons conclu ce chapitre par des tableaux les résumant et une discussion sur celles-ci. Au chapitre suivant, nous abordons la notion d'architecture dans les logiciels, car elle est centrale à l'approche que nous proposons dans le chapitre suivant.

CHAPITRE IV

NOTION D'ARCHITECTURE DANS LES LOGICIELS

4.1 INTRODUCTION

Dans le chapitre précédent, nous avons vu différentes approches de rétro-ingénierie. Dans ce chapitre, nous allons aborder la notion d'architecture dans l'optique du génie logiciel. La récupération et l'affichage de l'architecture sont des aspects importants de la rétro-ingénierie du logiciel. De plus, la notion d'architecture en elle-même peut guider la tâche de rétro-ingénierie. Par contre, il y a souvent confusion lorsqu'il est question du terme architecture. Pour certains, une architecture représente un ensemble de règles à suivre afin de bâtir un logiciel, alors que pour d'autres, l'architecture représente la structure d'un logiciel.

Nous commencerons ce chapitre par quelques définitions sur l'architecture. Suit une discussion des manières avec lesquelles nous pouvons représenter cette architecture. En particulier, nous allons expliquer comment nous voyons la relation qui existe entre l'architecture et des notions comme les stratégies et les mécanismes architecturaux, les patrons d'architecture et de conception et les cadres d'application. Notons que cette relation n'est pas toujours claire dans la littérature. Par la suite, nous allons expliquer ce que nous voulons dire par une bonne architecture. Nous allons conclure ce chapitre par une présentation de l'architecture J2EE comme exemple d'illustration.

4.2 DÉFINITIONS D'ARCHITECTURE

Nous retrouvons plusieurs définitions du terme d'architecture logicielle. En voici quelques-unes intéressantes, accompagnées d'idées qu'elles font ressortir.

Premièrement, certaines définitions font voir qu'une architecture est une structure organisée de composantes. L'IEEE la définit comme la structure organisationnelle d'un système (IEEE, 1990) ou plus précisément (IEEE, 2000) comme l'organisation fondamentale d'un logiciel, incorporée dans ses composantes, les rapports entre elles et leur environnement, et les principes régissant sa conception et son évolution. Selon Shaw et Garlan (Shaw et Garlan, 1996), l'architecture comprend une description des composantes contenues dans le logiciel, les interactions entre elles, les patrons qui guident leurs compositions et les contraintes sur ceux-ci.

Deuxièmement, certaines définitions de l'architecture mettent l'emphase sur le fait que l'architecture permet de faire des abstractions. Bass, Clements et Kazman (Bass, Clements et Kazman, 2003) décrivent l'architecture logicielle comme la structure ou les structures d'un logiciel comportant ses composantes, leurs propriétés visibles de l'extérieur et les relations entre elles. Selon Hayes-Roth (Hayes-Roth, 1994), l'architecture d'un logiciel est une description abstraite d'un système contenant d'abord une description fonctionnelle des composantes sous forme de la description de leur comportement et des interfaces et interrelations entre les composantes.

Troisièmement, l'architecture sert à apporter et à proposer des solutions à un problème plus général comme dans le cas des cadres de conception. Selon Bhansali et Nii (Bhansali et Nii, 1992), une architecture générique est définie comme l'organisation topologique d'un ensemble de modules paramétrables, ainsi que les rapports inter modulaires. Concevoir un logiciel en employant une architecture générique consiste donc à instancier les paramètres de chaque module paramétré par une valeur concrète tout en maintenant les contraintes inter modulaires. Selon Crispen et Stuckey (Crispen et Stuckey, 1994), une architecture se compose d'une stratégie de division et d'une autre de coordination. La stratégie de division amène à diviser le système entier en composantes « discrètes » qui ne se recoupent pas. La stratégie de coordination mène à définir les interfaces entre elles, la manière dont elles doivent communiquer entre elles. Selon Jackson et Boasson (Jackson et Boasson, 1995),

l'architecture comporte la définition d'un ensemble de composantes génériques ainsi qu'une description des propriétés de chaque type, les règles régissant la manière dont chaque type peut agir les uns avec les autres, le style des interactions permises entre les composantes et les règles régissant comment un système (ou le sous-système) peuvent se composer d'instances des composantes génériques.

Quatrièmement, selon certains, une architecture représente un style, une façon de faire les choses. Selon Hayes-Roth *et al.* (Hayes-Roth *et al.*, 1995), l'architecture d'un système logiciel complexe est son style et ses méthodes de conception et de construction. Par exemple : utilise-t-on une architecture objet ? Le traitement se fait-il de façon procédurale ? Utilise-t-on une architecture multi-couches ? Kruchten et Thompson affirment (Kruchten et Thompson, 1994) : L'architecture logicielle doit gérer les abstractions avec la décomposition et la composition avec style et esthétique.

Donc, nous voyons que l'architecture logicielle sert à décrire avec ou sans abstraction un logiciel, avec la possibilité de proposer des manières de faire afin de concevoir un logiciel. Ces manières peuvent être documentées dans une spécification appelée la spécification d'architecture.

4.3 DESCRIPTION D'UNE ARCHITECTURE

Nous avons vu dans la section précédente, qu'en plus de permettre la compréhension du logiciel, la notion d'architecture permet d'en guider la conception et la réalisation. Ce qui en retour, permet de faciliter encore une fois la compréhension en créant une architecture standard d'analyse plus facile. Mais une question reste encore sans réponse : quelle forme prendra cette spécification d'architecture ?

Généralement, l'architecture est documentée selon deux approches : une approche formelle et une approche informelle (Bass, Clements et Kazman, 2003; Shaw et Garlan, 1996). Dans une approche formelle, l'architecture est décrite à l'aide d'un

langage de description d'architecture (en anglais *Architecture Description Language*, ADL), Wright (Allen et Garlan, 1997), UniCon (Shaw *et al.*, 1995) et Darwin (Magee *et al.*, 1995) sont quelques exemples d'ADL. L'utilisation d'un langage formel pour la description des architectures a pour objectif de faciliter l'analyse et la simulation de ces architectures. Dans une approche informelle, nous pouvons utiliser un langage naturel (le français ou l'anglais par exemple) ou de plus en plus un langage graphique comme UML (Booch, Rumbaugh et Jacobson, 1999).

Cette description n'est pas suffisante comme le suggèrent Bass, Clements et Kazman (Bass, Clements et Kazman, 2003) qui proposent aussi d'expliquer pourquoi l'architecture est comme elle est. Cette description complémentaire doit donc contenir les décisions faites lors de l'architecture. En fait, ce diagnostic a aussi été fait en industrie. L'implantation d'architectures, comme J2EE, a été tellement problématique pour les organisations que plusieurs ont jugé que les documentations traditionnelles ne sont pas suffisantes pour bien les mettre en pratique. Il a même été nécessaire de rendre disponible le code de plusieurs exemples de systèmes afin de montrer comment bien mettre en œuvre ces architectures. Cette initiative n'a pas été suffisante pour remédier au problème. C'est pourquoi, une nouvelle tendance émerge, ces dernières années, qui consiste à documenter les architectures à l'aide des patrons de conception. Mais quelle relation existe-t-il entre l'architecture et les patrons d'architecture et encore plus avec les patrons de conception ? Et qu'en est-il donc des relations avec d'autres concepts comme les cadres d'application, les stratégies d'architecture et les mécanismes d'architecture ?

4.3.1 Stratégies architecturales

Clements *et al.* (Clements *et al.*, 2003) définissent une stratégie architecturale[1] comme un ensemble de décisions de conception pour supporter un besoin architectural particulier. Les besoins architecturaux sont (Withrow, 1990) généralement des attributs de qualité tels que la fiabilité, la performance ou l'adaptabilité. Par exemple, pour supporter le besoin de performance, un architecte peut adopter une stratégie d'architecture consistant en la réduction du flux de communication entre les composantes de l'architecture (Thiel, 2005).

4.3.2 Mécanismes architecturaux

Le concept de mécanisme architectural a été introduit par Booch (Booch, 1995). Un mécanisme architectural est la définition d'une structure par laquelle des objets collaborent entre eux afin d'accomplir une tâche pour satisfaire aux exigences d'un problème. Il s'agit d'une solution concrète de conception. Plus précisément, ils décrivent un ensemble de composantes et de connecteurs (et leurs responsabilités) pour faire cette tâche (Thiel, 2005). Ces mécanismes permettent d'implanter rapidement la solution désirée (Kruchten, 2000).

Un mécanisme architectural implante une ou même plusieurs stratégies architecturales. Par exemple, la réplication, qui est un mécanisme pour réaliser la fiabilité (Bass, Klein et Bachmann, 2000; Knoernschildn 2001), permet aussi de supporter la performance.

Comme les mécanismes architecturaux consistent en un nombre restreint de composantes et de connecteurs types, ils permettent aux concepteurs d'examiner et de documenter les rapports entre les modèles architecturaux de logiciel et leur qualité. Évidemment plus les décisions seront décrites précisément, plus il sera facile de

[1] Bass, Clements et Kazman (Bass, Clements et Kazman, 2003) utilisent plutôt le terme tactique, alors que Smith et Williams (Smith et Williams, 2003) utilisent le terme de principe de conception (Thiel, 2005).

mesurer leurs attributs. Ils leur permettent ainsi d'identifier les choix qu'ils doivent faire afin d'atteindre des objectifs de qualité. L'étude de ces mécanismes peut également aider à créer des bases de connaissances ayant comme objectif d'aider à la conception et à l'analyse (Bass, Klein et Kazman, 2001).

4.3.3 Patrons d'architecture et de conception

Le principe de patron dans les logiciels s'inspire de sa contrepartie architecturale énoncée par Alexander, Ishikawa et Silverstein (Alexander, Ishikawa et Silverstein, 1977). Ils le définissent comme la description d'un problème fréquent et la base de sa solution, de manière qu'elle puisse être employée à de multiples reprises, sans jamais le faire de la même manière. Dans le monde informatique, les solutions sont exprimées en terme d'objets et d'interfaces au lieu de portes et fenêtres, mais chacune est l'expression d'une solution à un problème dans un contexte donné (Gamma *et al.*, 1995).

Un patron est constitué de quatre composantes principales. D'abord son nom qui se doit de spécifier le problème de conception visé, ses solutions et ses conséquences de façon aussi concise que possible. Deuxièmement, il doit posséder une description du problème de conception visé et les symptômes décrivant la mauvaise conception ciblée. La description peut aussi spécifier une liste de conditions où le patron est utile. Troisièmement, nous retrouvons la solution proposée qui est une description abstraite d'un problème de conception et comment un arrangement général des composantes (des classes et des objets dans notre cas) le résout. Finalement, nous retrouvons une description des conséquences illustrant les impacts sur des points comme la flexibilité, l'extensibilité et la portabilité.

Buchmann *et al.* (Buschmann *et al.*, 1996) établissent une distinction entre les patrons d'architecture et les patrons de conception. La distinction entre les deux types se fonde sur le fait que le premier a une incidence sur la structure d'un système en entier alors que le deuxième a une conséquence plus locale. Un exemple de patron

d'architecture peut être le patron *Layer* (Buschmann *et al.*, 1996) qui permet d'organiser un système informatique sous forme de plusieurs couches comme dans l'architecture OSI pour les réseaux. Un exemple de patron de conception peut être le patron *Observer* qui permet de définir une interdépendance de type un à plusieurs, de telle façon que, quand un objet change d'état, tous ceux qui en dépendent en soient avisés et automatiquement mis à jour (Gamma *et al.*, 1995). D'autres exemples de patrons de conception seront vus à la section 4.4.

Selon Buchmann *et al.* (Buschmann *et al.*, 1996), un patron contient des mécanismes qui décrivent comment ce patron implante une solution. Par cela, nous pouvons supposer que nous avons plus de mécanismes que de patrons. La réalité en est une autre. En fait, souvent plusieurs patrons utilisent les mêmes mécanismes. Par exemple, la plupart des patrons du catalogue de Gamma *et al.* (Gamma *et al.*, 1995) utilisent principalement deux mécanismes architecturaux ; à savoir l'encapsulation et la délégation. C'est pourquoi les mécanismes sont plus utilisés pour analyser une architecture; alors que les patrons sont plus utilisés pour la documenter.

4.3.4 Cadre d'application (« Framework »)

Tout comme les définitions d'architecture, les définitions du terme cadre d'application (« Framework » ou parfois « Cadriciel ») varient souvent selon les personnes et les contextes. Les deux définitions suivantes sont les plus utilisées. Un cadre d'application est une conception réutilisable de l'ensemble ou d'une partie d'un système qui est représentée par un ensemble de classes abstraites et la manière dont leurs instances agissent l'une sur l'autre. Un cadre d'application est en fait le squelette d'une application qui peut être adapté aux besoins du client par un développeur (Johnson, 1997).

Un cadre d'application implante en fait plusieurs patrons de conception. D'ailleurs, la plupart des patrons de Gamma *et al.* sont extraits des cadres d'application que nous

retrouvons dans les premiers environnements de développement avec le langage Smalltalk.

4.3.5 Notion d'architecture revisitée

En analysant les différentes définitions vues précédemment, nous pouvons déduire qu'une architecture qui sert à décrire avec ou sans abstraction un logiciel, avec la possibilité de proposer des manières de faire afin de concevoir un logiciel, est le fruit d'une ou de plusieurs stratégies d'architecture. Ces dernières sont implantées par des mécanismes architecturaux qui permettent d'analyser cette architecture vis-à-vis des attributs de qualité.

Nous pouvons aussi conclure que les manières proposées par une architecture peuvent être documentées dans une spécification appelée la spécification d'architecture. Cette spécification peut aussi être décrite sous forme de patrons de conception. Par exemple, J2EE, qui est une architecture permettant de construire une application distribuée comme nous allons voir dans la prochaine section, peut être décrit par un ensemble de patrons de conception. Pour implanter une architecture, le concepteur utilise en général un cadre d'application qui donne un ensemble de classes abstraites prêtes à l'emploi. L'utilisation consiste à hériter de ces classes abstraites pour créer des classes réalisant le rôle décrit par ces classes abstraites dans un cadre concret. Pour cela, un développeur peut choisir une des implantations de J2EE disponibles sur le marché proposées par des compagnies de logiciels comme Oracle ou Sun Microsystems.

4.4 J2EE : UN EXEMPLE D'ARCHITECTURE

Nous verrons dans cette section un exemple de cadre d'application de J2EE. Nous présenterons d'abord les niveaux sur lesquels son architecture est fondée, pour ensuite voir les patrons de conception qu'elle préconise et un exemple de logiciel l'employant, le Java Pet Store.

L'objectif poursuivi par Sun avec son cadre d'application J2EE, est de fournir une plateforme pour le déploiement de logiciels d'entreprise côté serveur écrits en Java. Elle a notamment comme objectifs d'être indépendante de plateforme, portable, multi-usagers, sécurisée, et standardisée. J2EE est principalement un ensemble de spécifications fournies par SUN qui encourage divers fournisseurs à l'implanter.

La plateforme J2EE comme le montre la figure 4.1 est fondée sur une architecture multi-couches (et utilise donc le patron *Layer*) basée sur les composantes afin de faciliter la conception, le développement, l'assemblage et le déploiement d'applications d'entreprises.

La première couche est celle du client. Elle est l'interface par laquelle l'utilisateur interagit avec le logiciel. Elle s'exécute sur la machine du client. Cette couche peut prendre la forme d'une page Web interactive, d'un applet, d'un logiciel client autonome ou autre.

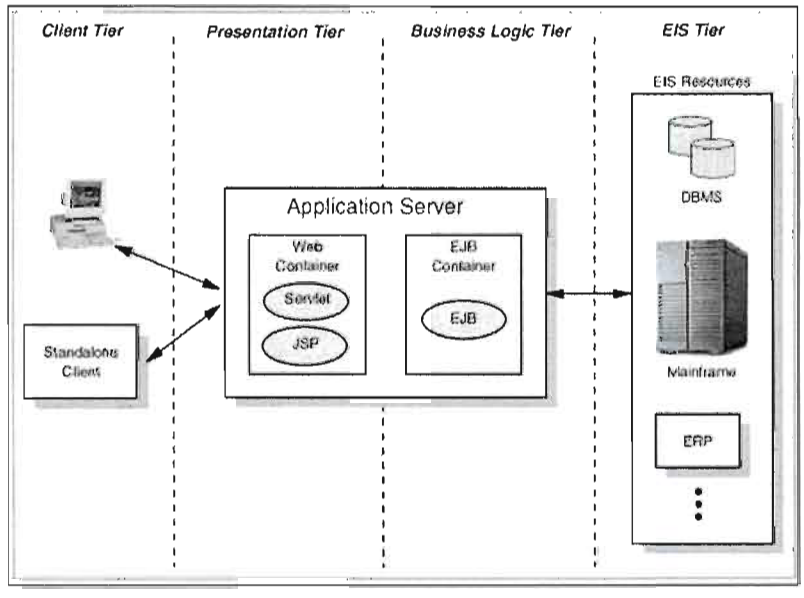

Figure 4.1 Les couches de l'architecture J2EE (tirée de Endrei *et al.*, 2002)

La seconde couche est celle de la présentation (ou Web). Elle est responsable de répondre aux requêtes de la première couche. Elle transforme les requêtes de cette couche en commandes qui seront exécutées par la logique d'affaires et ensuite elle formate la réponse qui sera envoyée au client. Les composantes de cette couche peuvent prendre la forme de « Servlets » ou de pages JSP.

La troisième couche représente la logique d'affaires. Ses composantes prennent la forme d'« Enterprise Java Beans (EJB) ». Ces EJB sont des composantes distribuées et standardisées. Le développement des EJB est simplifié et un EJB développé et déployé sur un serveur EJB d'un fournisseur spécifique peut être déployé facilement sur un serveur EJB d'un fournisseur différent. Nous élaborerons davantage à leur sujet à la section 4.4.1.

La dernière couche concerne les bases de données et les services d'information de l'entreprise. Une force de l'architecture J2EE est de prôner l'indépendance de la logique d'affaires vis-à-vis des sources de données, cela facilite la migration d'une base de données à une autre. Cette couche peut prendre notamment la forme d'une interface avec les services d'information de l'entreprise, un service de répertoire des objets du système ou de services pour employer les fichiers XML.

4.4.1 Les Entreprise Java Bean (EJB)

J2EE définit trois types d'EJB. Leurs tâches principales sont d'exécuter la logique d'affaires et d'accéder à des bases de données et à d'autres systèmes d'information d'une entreprise. Il y a trois types d'EJB, dont voici la description.

4.4.1.1 EntityBean

Le premier type d'EJB est l'EntityBean qui représente des données d'affaires persistantes. Il se rapproche un peu de la sérialisation, car la persistance peut être

gérée par le serveur (Container Management Persistence, CMP). Mais elle peut être aussi gérée par lui-même (Bean Management Persistence, BMP).

Un EntityBean implante l'interface *javax.ejb.EntityBean* qui possède les méthodes suivantes : Les méthodes nommées *ejbCreate* initialisent l'EJB. Il peut en avoir avec des arguments différents qui permettent différentes initialisations. Les méthodes *ejbPostCreate* correspondent aux méthodes *ejbCreate* avec les mêmes arguments. La méthode *ejbActivate* est appelée juste avant que l'EJB soit activé (par exemple, lors du transfert du disque parce qu'un client a besoin de l'EJB). La méthode *ejbPassivate* est appelée juste avant que l'EJB soit passivé (permuté sur le disque parce qu'il y a un trop grand nombre d'EJB instanciés). La méthode *ejbRemove* est appelée avant que l'EJB soit détruit. La méthode *setEntityContext* associe un EJB à un contexte d'environnement reçu en paramètre. La méthode *unsetEntityContext* enlève l'association entre l'EJB et son environnement. Les méthodes *ejbHome* sont des méthodes non spécifiques à des instances d'EJB. Elles peuvent par exemple, compter le nombre d'enregistrement dans une table.

Dans le cas des EntityBean dont la persistance est gérée par le serveur, nous retrouvons en plus les méthodes *ejbSelect* qui permettent au serveur (conteneurs) d'effectuer des requêtes internes, mais ne sont pas accessibles aux clients.

En plus dans le cas des EntityBean dont la persistance n'est pas gérée par le serveur, nous trouvons les méthodes suivantes. La méthode *ejbLoad* qui permet de charger la source de données dans l'EJB. Les méthodes *ejbFind* permettent de trouver des instances d'EntityBean selon différents critères. La méthode *ejbStore* est appelée pour enregistrer dans la source de données l'état de l'EJB.

4.4.1.2 SessionBean

Le second type d'EJB est SessionBean. Il représente une interaction avec un client et reflète les processus d'affaires. Un SessionBean conserve ou non un état. C'est-à-dire que d'une requête à l'autre il garde les informations sur le client qui y est associé.

Un SessionBean implante *javax.ejb.SessionBean* qui possède les méthodes suivantes communes avec EntityBean : *ejbCreate*, *ejbActivate*, *ejbPassivate* et *ejbRemove*, en plus de setSessionContext qui associe l'EJB à une session.

4.4.1.3 MessageDrivenBean

Le troisième type d'EJB est le MessageDrivenBean qui permet à une composante d'affaires de recevoir un message de façon asynchrone. Il s'approche des SessionBean, car il réalise des actions liées aux processus d'affaires.

Un MessageDrivenBean implante *javax.ejb.MessageDrivenBean* qui a les méthodes suivantes *ejbRemove* et *setMessageDrivenContext* qui associent l'EJB à un contexte.

4.4.1.4 L'architecture des EJB

Un EJB est assisté d'un ensemble de classes auxquelles l'équipe de BluePrint (Sun) propose des noms standards. Mais, ils ne sont pas toujours respectés. Les EJB peuvent être assistés de plusieurs classes comme nous allons voir ci-dessous.

La classe de base

D'abord, l'EJB en lui-même est représenté par une classe portant son nom suivi de *Bean*, qui implante sa logique d'affaires. Le patron suivant le nom de classe est aussi parfois *EJB* ou *MDB* (Message Driven Bean) ou même aucun.

La classe manufacturière

Il y a d'abord des classes manufacturières (« Factory classes »). Selon qu'elles seront accédées localement ou non, elles implanteront l'interface *javax.ejb.EJBHome* ou l'interface *javax.ejb.EJBLocalHome*. Ces classes contiennent les méthodes pour créer, trouver et supprimer les EJB. Elles peuvent servir à gérer un bassin d'EJB lorsque le contexte le permet. Le nom est en général celui de l'EJB suivi de *Home* ou de *RemoteHome* dans le cas d'un accès distant et *Home* ou *LocalHome* dans le cas local. Parfois, le *Remote* et le *Local* se trouvent devant le nom de l'EJB.

Ces classes dans le cas de l'accès distant implantent l'interface *javax.ejb.EJBHome* qui contient les méthodes suivantes. La méthode *getEJBMetaData* retourne des métadonnées sur l'entreprise, la méthode *getHomeHandle* retourne le « handle » à l'objet Home et la méthode *remove* détruit l'EJB indiqué en argument. Dans le cas de l'accès local, elle implante *javax.ejb.EJBLocalHome* qui possède aussi la méthode *remove*.

Les classes d'accès

Ensuite, il y a les classes qui permettent d'accéder aux méthodes de ces EJB. Elles peuvent encore être locales ou non. Elles implantent selon *javax.ejb.EJBLocalObject* ou *javax.ejb.EJBObject*. Leur nom est celui de l'EJB suivi de *Local* ou *Remote*. .

Dans le cas local, ces classes ont les méthodes suivantes : *getEJBLocalHome* qui obtient l'objet *LocalHome* de cet EJB, *getPrimaryKey* qui retourne la clé primaire si c'est une entité, *isIdentical* qui permet de savoir si l'EJB est identique à celui en paramètre et *remove* qui détruit cet EJB.

Dans le cas distant, il possède les méthodes semblables au cas local. Ces méthodes sont : *getPrimaryKey*, *isIdentical* et *remove*. En plus, nous retrouvons, les méthodes *getEJBHome,* qui retourne l'objet *Home* de cet EJB et *getHandle,* qui retourne un « handle » à cet EJB.

Autres classes

Les classes ou couches architecturales précédentes sont celles que nous retrouvons en général. D'autres classes plus ou moins standards peuvent aussi accompagner un EJB comme une classe implantant la clé primaire d'un EntityBean souvent avec PK comme nom de couche ou une classe implantant les exceptions.

4.4.2 Patron de conceptions dans J2EE

Sun prône dans son architecture un ensemble de patrons de conception à appliquer aux logiciels J2EE afin d'atteindre une architecture idéale. La figure 4.2 les représente. Nous verrons par la suite un énoncé concis décrivant chacun des patrons de conception présents. Par la suite, nous verrons l'application de ces patrons à un logiciel employé lors de notre validation, le Java Pet Store. Nous avons déjà vu que J2EE est basé sur une architecture multi-couches. Cette architecture est implicitement présente dans ce diagramme. Voici une brève présentation des patrons de ce diagramme.

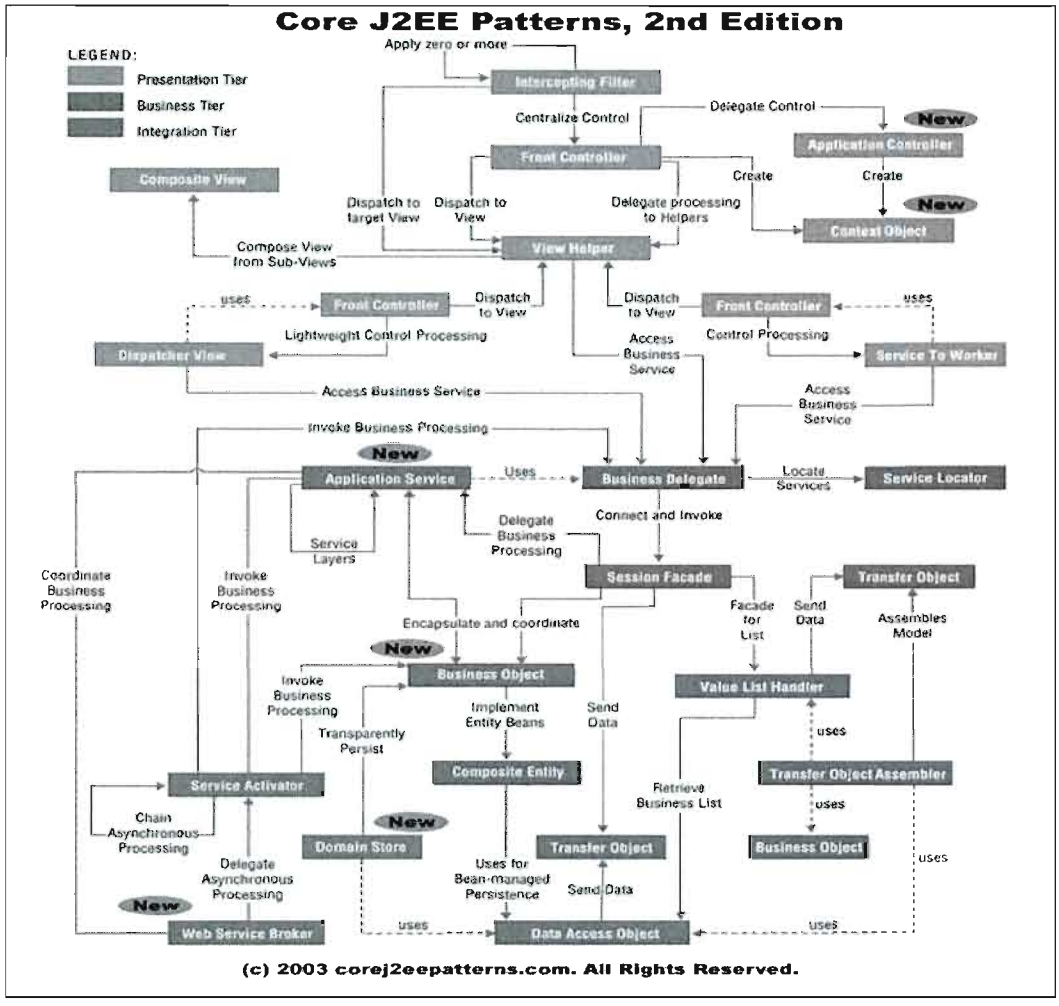

Figure 4.2 Patrons J2EE (tirée de Alur, Malks et Crupi, 2003)

4.4.2.1 La couche présentation

Intercepting Filter

Ce patron implante un filtre entre le client et le reste du système en permettant de vérifier la validité des requêtes et des réponses.

Front Controller

Le patron « Front Controller » définit une composante qui est responsable de traiter les requêtes. Il centralise les fonctions telles que les sélections de représentations et la sécurité de façon uniforme à toutes les pages ou les vues. En conséquence, quand le comportement de ces fonctions doit changer, seulement une petite partie du logiciel doit être changée, le contrôleur et ses classes aidantes.

Context Object

Ce patron permet d'éviter que les communications à l'intérieur du logiciel se fassent en dehors d'un protocole spécifique. Par exemple, une composante Web reçoit des requêtes http, mais pour continuer la communication avec les autres composantes, il est préférable d'adopter un autre protocole.

Application Controller

Ce patron permet de centraliser les actions et la gestion des vues dans la couche composante.

View Helper

Un « View Helper » est une classe qui fait l'extraction de données pour la représentation. Elle adapte une source de données à un API simple utilisable par des vues d'application. Le modèle « View Helper » découple des classes d'affaires et d'application les unes des autres et leur permet de changer à leurs propres rythmes. Le découplage également favorise la réutilisation, parce que chaque composante d'affaires ou de présentation a peu de dépendances. Ainsi, la représentation peut se concentrer sur le formatage et la logique de présentation, et laisser le « View Helper » manipuler le traitement et la récupération des données.

Composite View

Un « Composite view » est une représentation créée en utilisant d'autres vues secondaires réutilisables. Un changement simple à une représentation secondaire est

automatiquement reflété dans chaque représentation composée qui l'utilise. En outre, la représentation composée contrôle la disposition de ses vues secondaires et peut fournir un gabarit, faisant une présentation uniforme plus facile à réaliser et à modifier à travers toute l'application.

Service to Worker

Ce patron combine un « Front Controller » et un « View Helper » afin de manipuler des requêtes de client pour préparer une présentation dynamique comme réponse.

Dispatcher View

Ce patron est employé quand il y a peu ou pas de logique d'affaires à effectuer pour créer une vue dynamique.

4.4.2.2 Couche de logique d'affaires

Business Delegate

Dans des applications réparties, la consultation et la manipulation des composantes distantes d'affaires peuvent être complexes. Quand les applications emploient des composantes d'affaires directement, le code d'application doit changer pour refléter des changements à l'API des composantes d'affaires. Ces problèmes peuvent être résolus en présentant une classe intermédiaire appelée « Business Delegate » qui découple les composantes d'affaires du code qui les emploie. Le « Business Delegate » contrôle la complexité de la consultation des composantes distribuées et la manipulation des exceptions, et peut adapter l'interface de composante d'affaires à une interface plus simple pour les vues.

Service Locator

Le patron « Service Locator » centralise les consultations d'objets distribués, fournit un point centralisé de commande et peut agir en tant que cache qui élimine des

consultations redondantes. Il encapsule également tous les dispositifs du processus de consultation spécifiques à un fournisseur.

Session Facade

Le patron « Session Facade » définit une composante de haut niveau d'affaires qui contient et centralise les interactions complexes entre les composantes de plus bas niveau. Une « Session Facade » est mise en application comme un « session enterprise bean ». Elle fournit à ses clients une interface simple pour la fonctionnalité d'une application ou d'un sous-ensemble d'applications. Elle découple également les composantes de plus bas niveau d'affaires les unes des autres, rendant la conception plus flexible et plus compréhensible.

Application Service

Ce patron permet de centraliser la logique d'affaires dans quelques composantes et services d'affaires.

Business Object

Ce patron permet de séparer les données d'affaires de la logique d'affaires.

Composite Entity

Le patron « Composite Entity » offre une solution pour la modélisation de réseaux de composantes d'affaires reliées. L'interface « Composite Entity » est à grain grossier. Elle contrôle les interactions entre les objets à grain plus fin. Ce patron de conception est particulièrement utile pour contrôler efficacement les rapports avec les objets dépendants.

Transfer Object

Un « Transfer Object » (ou « Value Object ») est une classe sérialisable qui groupe des attributs reliés formant une valeur composée. Cette classe est employée comme

type de retour d'une méthode d'affaires à distance. Les clients reçoivent des instances de cette classe en appelant des méthodes d'affaires à grain grossier et puis en accédant localement aux valeurs à grain fin dans l'objet de transfert.

Transfer Object Assembler

Le patron « Transfer Object Assembler » construit un objet composé qui représente des données de différentes composantes d'affaires. Il transfert les données pour le modèle au client dans un appel unique de la méthode.

Value List Handler

Le patron de conception « Value List Handler » fournit une manière plus efficace d'itérer une grande liste de valeurs en lecture seule. Un « Value List Handler » fournit à un client un itérateur pour une liste virtuelle qui réside dans un autre tiers d'application. L'itérateur accède généralement à une collection locale d'objets représentant un sous-ensemble d'une grande liste. Un « Data Access Object » contrôle habituellement l'accès aux données de liste, et peut fournir une cache.

4.4.2.3 La couche d'intégration

Data Access Object

Le patron « Data Access Object » (ou DAO) sépare l'interface du client d'une source de données de ses mécanismes d'accès et adapte l'API d'une source spécifique de données à une interface générique.

Service Activator

Le « Service Activator » est utilisé pour recevoir les requêtes et les messages asynchrones de client. À la réception d'un message, le « Service Activator » localise et appelle les méthodes d'affaires nécessaires sur les composantes de service pour accomplir la demande asynchrone.

Domain Store

Ce patron permet de déterminer la persistance des objets.

Web Service Broker

Ce patron permet d'offrir et de regrouper un ou plusieurs services en utilisant XML et les protocoles Web.

4.4.3 Un exemple d'application des patrons de J2EE, le Pet Store

Nous allons voir dans l'exemple du Java Pet Store de Sun (Jaber, 2002; Nambiar, 2005) l'application des patrons de conception énoncés plus tôt. Il s'agit de l'exemple classique de Sun (Sun, 2006) d'une application utilisant J2EE. La figure 4.3 résume la présence de certains patrons de conception dans cette application. Il est à noter que d'autres patrons de conception se retrouvent dans cette application.

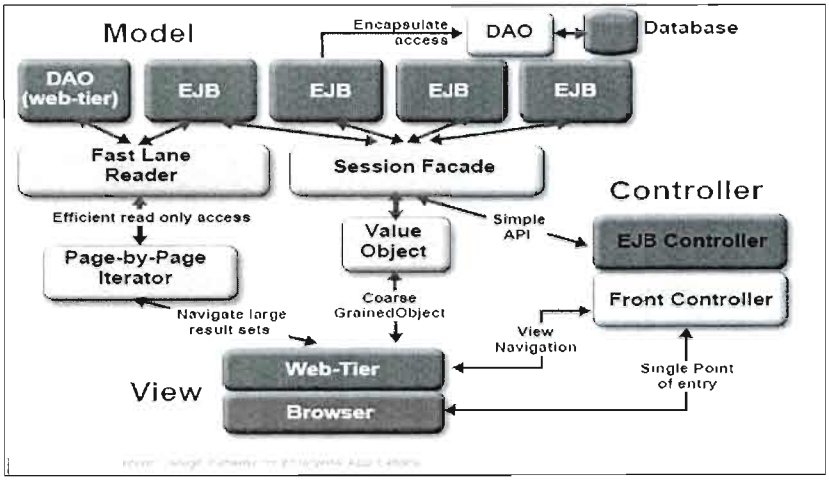

Figure 4.3 Les patrons de conception utilisés dans le Java Pet Store (tirée de Singh et Johnson, 2001)

4.4.3.1 Front Controller

Toutes les requêtes http des clients passent par le Front Controller (*MainServlet*) qui est un Servlet (voir figure 4.4). Ce contrôleur traite les requêtes et les convertit en événements pouvant être traités par le modèle d'affaires.

La requête est ensuite passée au contrôleur de requête EJB *RequestProcessor* qui le délègue à un autre correspondant (tels que *ShoppingClientController* ou *AdminClientController*), qui fait correspondre l'événement à la commande appropriée. Finalement, la commande appelle l'action appropriée via un Session Facade qui se charge d'exécuter la logique d'affaires.

Le Front Controller sélectionne aussi les vues à afficher via le *ScreenFlowManager*.

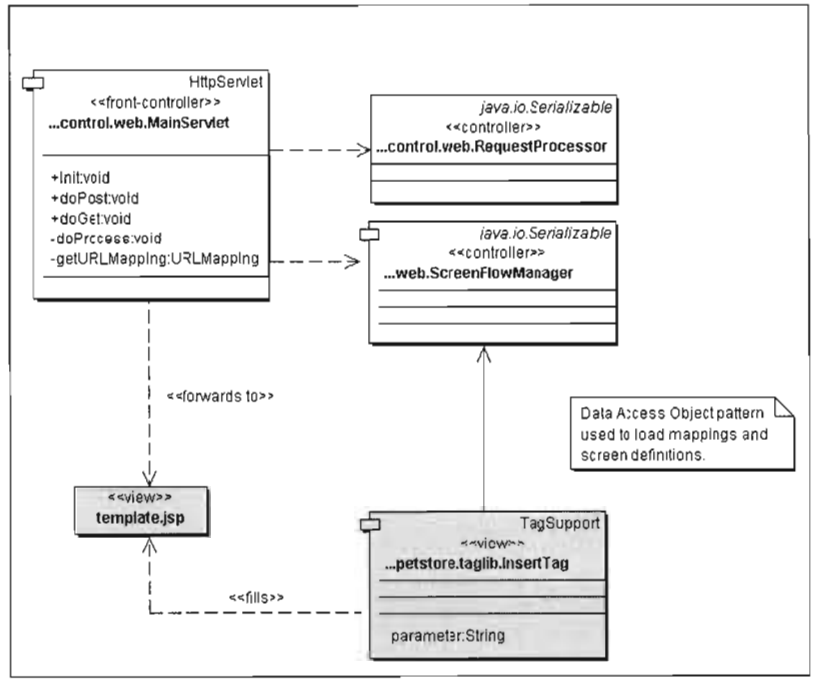

Figure 4.4 Front Controller (tirée de Gronback 2002)

4.4.3.2 Session Facade

L'application emploie un Session Facade entre les EJB afin d'unifier leur interface. Par exemple, nous voyons dans la figure 4.5 que l'application a ses tâches administratives regroupées au travers d'un objet de la classe *OPCAdminFacade*. Par son unique intermédiaire, le client peut effectuer différentes tâches de la logique d'affaires.

4.4.3.3 Transfer Object

Les Transfer Object sont employés afin de limiter le trafic réseau via une Session Facade qui ne serait pas sur une même machine. L'utilité de ce patron est par exemple, comme le montre la figure 4.6, dans le cas où nous voudrions afficher les détails d'une commande. Au lieu de recueillir chacun des items un par un lors de la construction de la vue, il est plus pratique de recueillir un objet qui contient tous ces détails. Dans ce cas c'est un *OrderTO*, dont le suffixe TO signifie « TransfertObject ».

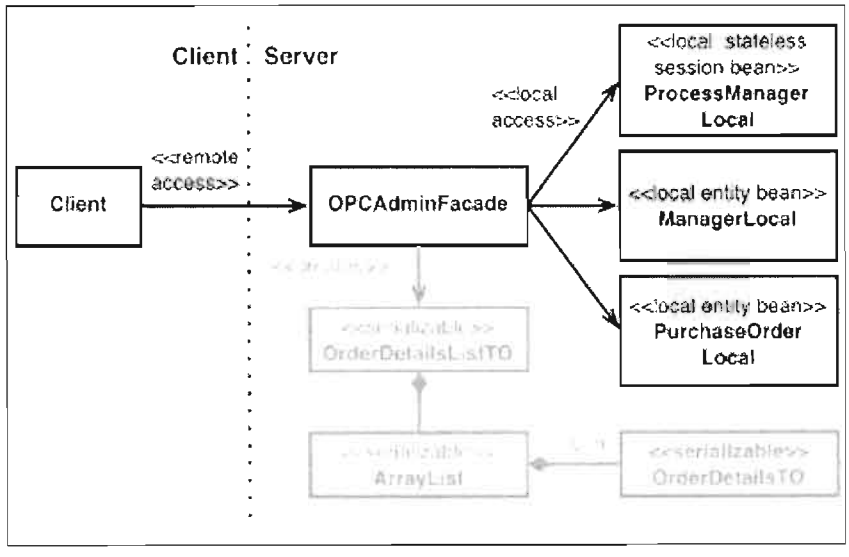

Figure 4.5 Session Facade (tirée de Microsystems, 2001)

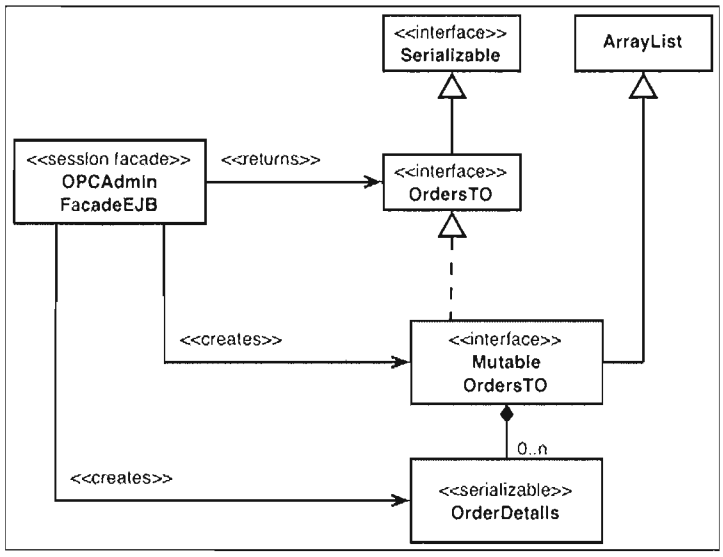

Figure 4.6 Transfer Object (tirée de Microsystems, 2001)

4.4.3.4 Data Access Object

Le Data Access Object permet de simplifier l'utilisation de sources de données comme le montre la figure 4.7. À l'exécution, le *CatalogHelper* emploie *CatalogDAOFactory* afin de créer un objet héritant de *CatalogDAO*. L'usine recherche le nom de la classe implantant DAO dans une variable d'environnement. Le *CatalogHelper* accède à la source de données en employant l'objet créé par l'usine. Dans notre exemple, la classe *CloudscapeCatalogDAO* est employée. Cette classe permet l'accès à la base de données *Cloudscape*.

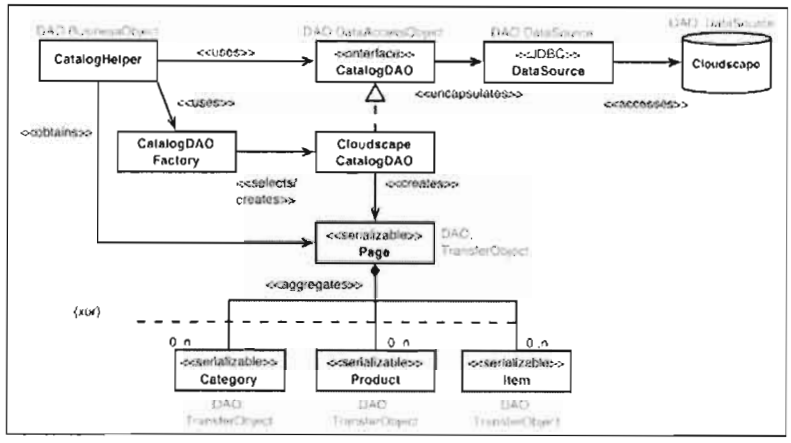

Figure 4.7 Data Access Object (tirée de Microsystems, 2001)

Cette approche est plus souple que celle utilisant une classe codée en dure où l'accès à la source de données est configuré dans le code source et ne peut être modifié sans recompiler le logiciel. Pour ajouter une nouvelle source de données, un développeur a simplement à créer une classe qui met en application *CatalogDAO*, et à indiquer le nom dans un fichier de configuration. L'usine créerait une instance de la nouvelle classe et l'application emploierait la nouvelle source de données.

4.4.3.5 Value List Handler (Fast-Lane Reader)

Le patron de conception Value List Handler permet d'obtenir rapidement l'accès en lecture seule à des données sans avoir à passer par la logique d'affaires ce qui accélère le tout. Nous voyons, par exemple comme le montre la figure 4.8, que cela peut être utile pour parcourir un catalogue et en afficher les pages dans une page JSP. Un objet de la classe *CatalogHelper* qui est un ViewHelper qui agit aussi comme un Value List Handler, permet à la page JSP au lieu d'accéder à la logique d'affaires pour obtenir une liste, d'accéder directement au *CatalogDAO* qui peut lui renvoyer un objet de type Transfer Object contenant cette liste.

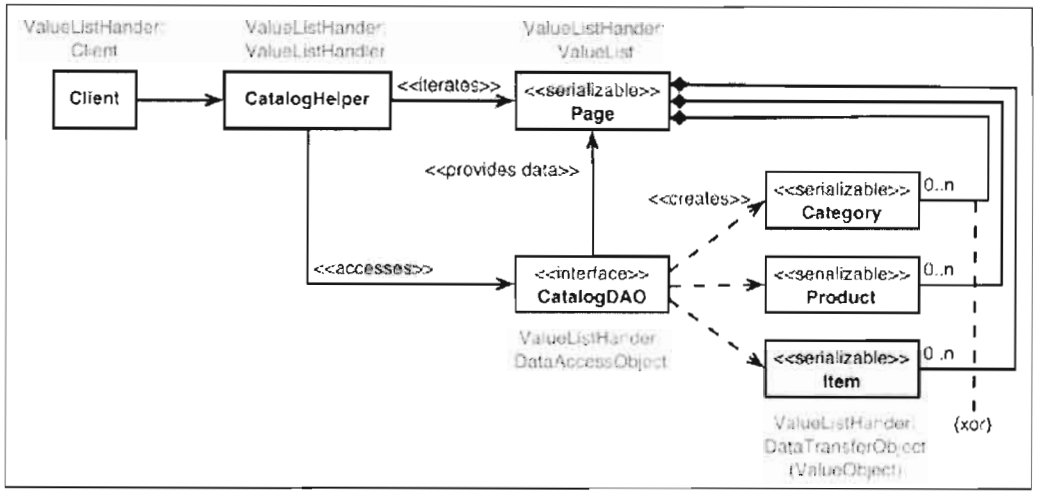

Figure 4.8 Value List Handler et Page by Page Iterator (tirée de Microsystems, 2001)

4.4.3.6 Page-by-Page Iterator

Nous voyons aussi dans la figure 4.8 qu'en plus d'accéder directement au DAO, un objet de la classe *CatalogHelper* récupère une liste d'objets au lieu d'y aller un par un. Ceci accélère d'autant plus sa tâche.

4.5 NOTION DE BONNE ARCHITECTURE

Nous avons précédemment employé la notion de « bonne architecture ». Tout comme la notion d'architecture, elle varie. Certains affirment qu'il n'y a pas de bonne ou de mauvaise architecture, mais des architectures qui répondent plus ou moins aux besoins et aux exigences auxquelles elle doit répondre (Kazman, Clements et Bass, 2003). Hofmeister, Nord et Soni (Hofmeister, Nord et Soni, 2000) définissent une bonne architecture comme une qui, quand le système est implanté en la suivant, satisfait les spécifications et les ressources économiques. Rybin *et al.* (Rybin *et al.*, 1996) disent qu'une bonne architecture montre comment bâtir le système selon les exigences des usagers, mais aussi selon « les besoins » moins tangibles. Lenz et

Wienands (Lenz et Wienands, 2006) ajoutent qu'elle doit fournir une approche standardisée, claire, concise et consistante de développement. Ils ajoutent aussi que le tout doit être bien documenté. Hunt (Hunt, 2003) énonce aussi ces qualités comme souhaitable : résilience au changement, simplicité, clarté de présentation, séparation des problématiques bien définie et balancée.

Dans ce mémoire, notre définition de bonne architecture rejoint celle de Lenz et Wienands. En effet, notre approche, comme nous allons le voir dans le prochain chapitre, repose sur la détection du code implantant des mécanismes architecturaux. Or cette détection nécessite que l'architecture offre une approche standardisée et consistante pour leur implantation.

4.6 CONCLUSION

Nous avons vu dans ce chapitre des définitions de l'architecture en rapport avec le logiciel et des manières avec lesquelles nous pouvons représenter cette architecture en particulier, les stratégies et les mécanismes architecturaux, les patrons d'architecture et de conception et les cadres d'application ont été présentés. Nous avons aussi présenté le cadre d'application J2EE, ses patrons de conception et son application dans le Java Pet Store. Nous avons conclu ce chapitre par une explication de ce que nous voulons dire par une bonne architecture. Dans le chapitre suivant, nous allons présenter l'approche de rétro-ingénierie que nous avons développée et un exemple de son application.

CHAPITRE V

L'APPROCHE PROPOSÉE

5.1 INTRODUCTION

L'objectif de notre projet est d'extraire les règles d'affaires d'une application orientée objet. Pour cela, nous proposons une nouvelle approche de rétro-ingénierie qui permet d'analyser son code source afin d'en extraire un modèle abstrait ne décrivant que les règles d'affaires de cette application. Ce modèle prend la forme d'un diagramme de classes UML présentant les classes d'affaires de cette application ainsi que les relations entre ces classes. La définition de ce modèle passe donc en premier lieu par l'identification des classes d'affaires que cette application contient. Pour chaque classe d'affaires trouvée, nous voulons aussi identifier ses attributs et ses opérations. C'est-à-dire que nous voulons regrouper dans une seule classe d'affaires les attributs et opérations de celle-ci qui pourraient être répartis entre plusieurs classes de l'application analysée.

Notre approche consiste donc à pouvoir distinguer les classes d'affaires des autres classes ajoutées par les besoins de l'infrastructure utilisée pour implanter cette application. Cette infrastructure est le résultat de l'adoption d'une architecture et le choix de la plateforme d'implantation. Dans le chapitre précédent, nous avons étudié la notion d'architecture en rapport avec le domaine du logiciel. Nous avons aussi présenté la plateforme J2EE comme exemple d'architecture. Nous verrons dans ce chapitre, comment ces connaissances peuvent être utilisées afin d'isoler les classes d'affaires d'une application. Ce chapitre débutera par la présentation d'un exemple d'une petite application qui sera utilisée pour illustrer notre approche. Cette application implante une partie de l'architecture J2EE. Nous présenterons par la suite

quelques définitions utiles pour la compréhension de cette approche. Finalement, nous expliquerons en détail ses étapes.

5.2 EXEMPLE D'ILLUSTRATION

Afin de bien faire saisir le but de notre approche, nous débutons ce chapitre par une illustration graphique de cet objectif. À la figure 5.1 et à la figure 5.2, nous voyons des diagrammes de classes UML représentant un exemple d'une application « jouet » sur laquelle nous désirons appliquer notre approche et le résultat escompté. Le code source de cette application se retrouve à l'appendice A.

Nous voyons donc à la figure 5.1 un diagramme de classes UML qui décrit une vue graphique de cette application. Malgré sa simplicité, il n'est pas facile de reconnaître au premier coup d'œil ce qu'elle fait exactement. Les fonctionnalités des classes d'affaires sont reparties parmi tous les détails de l'implantation de son architecture. En fait, cette application contient vingt-huit classes (et/ou interfaces) qui sont listées au tableau 5.1. Les noms des classes nous informent que nous sommes devant une application bancaire implantant une partie de l'architecture J2EE.

Tableau 5.1
Nom des classes

Account	CustomerHome
AccountBean	CustomerWebFacade
AccountHome	Deposit
AccountWebFacade	DepositBean
BankTransaction	DepositHome
BankTransactionBean	DepositWebFacade
BankTransactionHome	SavingAccount
BankTransactionWebFacade	SavingAccountBean
CheckingAccount	SavingAccountHome
CheckingAccountBean	SavingAccountWebFacade
CheckingAccountHome	Withdrawal
CheckingAccountWebFacade	WithdrawalBean
Customer	WithdrawalHome
CustomerBean	WithdrawalWebFacade

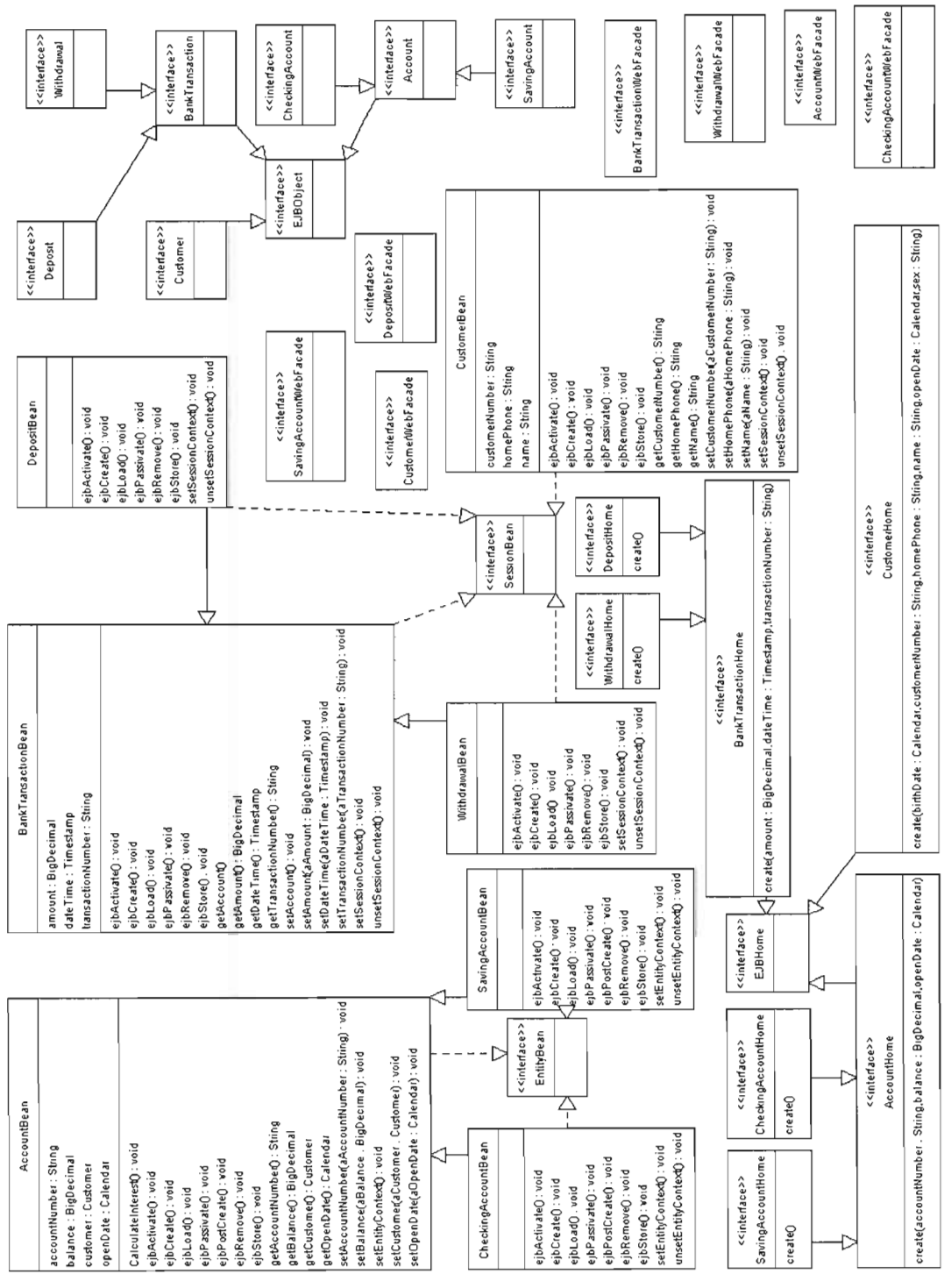

Figure 5.1 Modèle original de l'application.

Par contre à la figure 5.2, nous voyons les objets d'affaires qu'il contient et les propriétés qui leur sont associées. Nous voyons donc ce que nous cherchons à faire, c'est à dire faire ressortir les classes d'affaires et leurs relations. Ainsi, nous voyons instantanément les classes d'affaires, sans avoir à faire une lecture exigeante du modèle. En outre, il est plus facile de connaître quelques règles d'affaires de cette application. Par exemple, qu'il existe deux types de compte, ceux de chèque et ceux d'épargne.

5.3 DÉFINITIONS

Afin de bien comprendre les notions définissant notre approche, nous allons d'abord nous entendre sur un ensemble de définitions données ci-dessous.

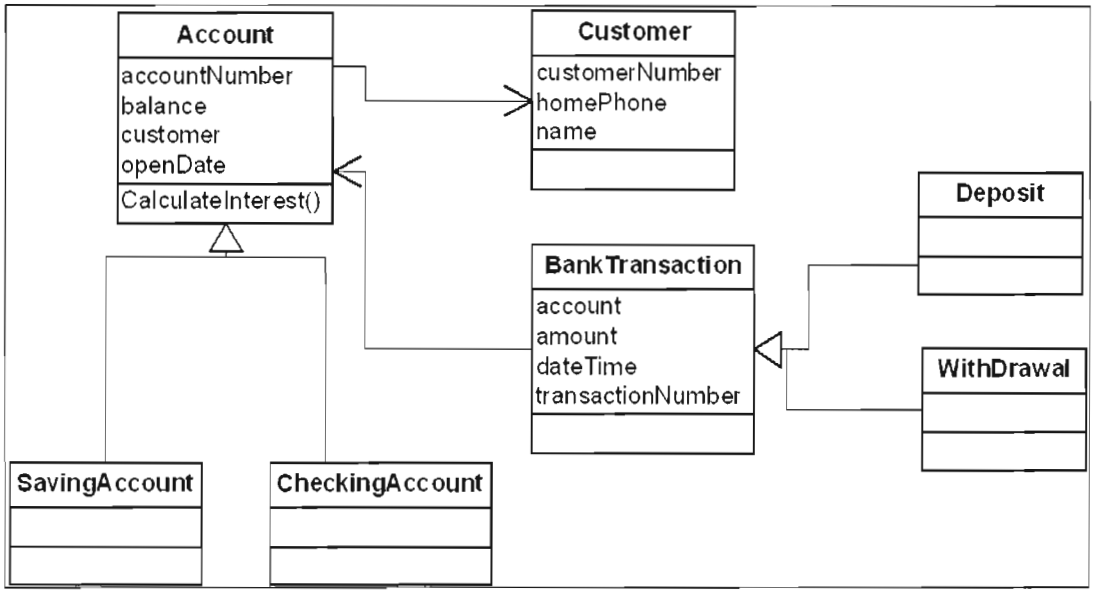

Figure 5.2 Abstraction du modèle original désiré

5.3.1 Vocabulaire

Le vocabulaire d'un logiciel représente les mots retrouvés dans le code source de celui-ci. Il est constitué de l'union de trois vocabulaires distincts. D'abord, il y a le vocabulaire propre aux langages de programmation (par exemple, pour le langage Java: `for`, `int` ou `case`). Il y a ensuite le vocabulaire introduit par la logique d'affaires qui représente les concepts de celui-ci (par exemple, pour une application bancaire : `Account`, `Check` ou `Client`). Finalement, nous avons le vocabulaire qui représente l'implantation de la logique d'affaires dans une architecture (par exemple, pour une application EJB nous pourrions avoir `Home`, `Remote` ou `LocalHome`).

5.3.2 Mot architectural

Un mot architectural est un mot appartenant au vocabulaire architectural. Par exemple, pour une application EJB, nous pourrions avoir `Home`, `Remote` ou `LocalHome`.

5.3.3 Nom

Le nom d'une classe (ou attribut ou opération) est une chaîne de caractères composée de sous-chaînes séparées par des délimiteurs (par exemple, `Account`, `AccountHome`, `AccountBean`).

5.3.4 Patron d'un nom

Un patron d'un nom est une sous-chaîne ou une composition de sous-chaînes. Par exemple, l'identificateur `AccountBean` contient trois patrons : `Account`, `Bean` et `AccountBean`.

5.3.5 Groupe relié à un patron

Un groupe relié à un patron est un regroupement de noms, parmi une liste de noms basés sur ce patron. Par exemple, le groupe {`Deposit`, `DepositBean`,

`DepositHome, DepositWebFacade}` est le groupe relié au patron `Deposit`. Évidemment, un nom peut se retrouver dans plusieurs groupes.

5.3.6 Sous-groupes reliés à un patron suivant un critère

Les sous-groupes reliés à un patron sont obtenus en partitionnant, suivant un critère, le groupe relié à un patron. Par exemple, nous pouvons obtenir les sous-groupes d'un patron suivant l'implantation d'une interface ou l'héritage d'une classe abstraite. Ainsi, nous pouvons répartir les classes d'un groupe dans différents sous-groupes suivant qu'une classe implante certaines interfaces ou hérite d'une certaine classe abstraite. Évidemment, une classe peut se retrouver dans différents sous-groupes d'un groupe relié à un patron. Par exemple, les classes `AccountBean` et `TransactionBean` qui appartiennent entre autres au groupe du patron `Bean` vont se retrouver dans des sous-groupes différents suivant le critère d'implantation d'interface. En effet, `AccountBean` implante l'interface `javax.ejb.EntityBean`, alors que `TransactionBean` implante l'interface `javax.ejb.SessionBean`.

5.3.7 Méthode principale

Une méthode (ou opération) principale d'une classe est une méthode dont le nom ne contient pas de patron et qui n'est ni contenu dans le nom de la classe (comme dans un constructeur) ni dans ses attributs (comme dans les getters et les setters). Par exemple, pour la classe `TransactionBean`, qui possède l'attribut `sessionContext`, sa méthode `setSessionContext` n'est pas une méthode principale, alors que sa méthode `ejbCreate` en est une.

5.3.8 Méthode auxiliaire

Une méthode (ou opération) auxiliaire d'une classe est une méthode qui n'est pas considérée comme une méthode principale.

5.4 DESCRIPTION DE L'APPROCHE

D'après les définitions précédentes, nous savons que le vocabulaire d'une application est constitué de l'union du vocabulaire du langage, de celui de la logique d'affaires et de celui de l'architecture. Nous connaissons le vocabulaire introduit par les langages de programmation. Il est facile de le mettre de côté en utilisant un analyseur syntaxique qui peut enlever facilement les mots reliés au langage de programmation. Si nous réussissons à isoler le vocabulaire d'architecture, nous pourrons isoler le vocabulaire d'affaires en enlevant du vocabulaire de l'application à la fois celui introduit par les langages de programmation et celui d'architecture. Nous avons vu dans le quatrième chapitre que le code relié à l'architecture est en général répétitif ou semi-répétitif. Son vocabulaire doit l'être lui aussi si nous nous limitons à des systèmes ayant une architecture implantée uniformément et ayant été développés suivant les bonnes règles de programmation. C'est sur cette constatation que se base notre approche.

Comme il est illustré dans la figure 5.3, pour effectuer cette tâche, nous avons divisé l'algorithme supportant notre approche en trois grandes étapes (voir le pseudo-code de l'algorithme dans l'Appendice B). D'abord, nous extrayons un arbre syntaxique abstrait du code source à analyser à l'aide d'un analyseur syntaxique généré avec l'outil SableCC (Gagnon et Hendren, 1998). Ensuite, nous effectuons l'extraction des classes d'affaires en nous basant sur les identificateurs des classes, des méthodes et des attributs et des relations entre ces classes retrouvées dans l'arbre syntaxique abstrait, cette étape est expliquée plus en détail à la figure 5.4. Finalement, le tout est exporté dans un document XML avec une classe que nous avons développée pour cette tâche. Ce document XML suit la norme XMI (pour XML Model Interchange) (Grose, Doney et Brodsky, 2002). Le but ici est de permettre d'utiliser n'importe quel outil de modélisation UML supportant la norme XMI afin d'importer le modèle dans l'outil et ainsi de permettre une visualisation graphique du modèle d'affaires extrait.

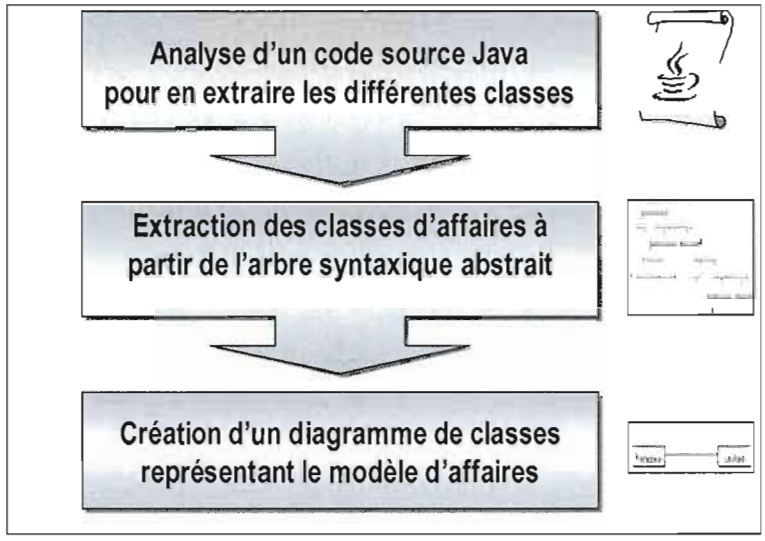

Figure 5.3 Étapes de l'algorithme

5.5 DESCRIPTION DE L'ÉTAPE D'EXTRACTION DES RÈGLES D'AFFAIRES

Nous allons nous concentrer sur l'étape d'extraction des classes d'affaires. Cette étape est sous-divisée en six étapes telles qu'illustré à la figure 5.4.

Figure 5.4 Les sous-étapes du processus d'extraction des règles d'affaires

Les sous-étapes du processus d'extraction des règles d'affaires sont basées sur un ensemble de règles. Vous pouvez retrouver la description formelle de ces règles dans (Khriss et Chénard, 2007).

5.5.1 Extraire les patrons

Lorsque nous analysons un système, nous commençons par décomposer l'ensemble des noms des classes pour en extraire l'ensemble des patrons le composant. Dans notre approche, ce sont les majuscules et les espaces qui servent à délimiter les patrons. Nous sommes conscients que ce choix n'est pas universel et que la seule séparation des patrons composants un identificateur et en soi une problématique comme nous avons expliqué dans le troisième chapitre. Ensuite, l'ensemble des combinaisons de patrons est placé dans une liste. Comme cela peut faire rapidement une grande liste, nous ne conservons que les combinaisons effectivement retrouvées dans le nom du logiciel. Ainsi, `BankTransactionBean` donnera les patrons : `Bank`, `BankTransaction`, `BankTransactionBean`, `Transaction`, `TransactionBean` et `Bean`. Et par exemple, `BeanTransaction` ne sera pas dans la liste, car il n'existe aucune classe contenant ce patron.

L'exemple de la figure 5.1 contient donc les combinaisons suivantes de patrons :

`Account`, `AccountBean`, `AccountHome`, `AccountWeb`, `AccountWebFacade`, `Bank`, `BankTransaction`, `BankTransactionBean`, `BankTransactionHome`, `BankTransactionWeb`, `BankTransactionWebFacade`, `Bean`, `Checking`, `CheckingAccount`, `CheckingAccountBean`, `CheckingAccountHome`, `CheckingAccountWeb`, `CheckingAccountWebFacade`, `Customer`, `CustomerBean`, `CustomerHome`, `CustomerWeb`, `CustomerWebFacade`, `Deposit`, `DepositBean`, `DepositHome`, `DepositWeb`, `DepositWebFacade`, `Façade`, `Home`, `Saving`, `SavingAccount`, `SavingAccountBean`, `SavingAccountHome`, `SavingAccountWeb`, `SavingAccountWebFacade`, `Transaction`, `TransactionBean`, `TransactionHome`,

TransactionWeb, BankTransactionWebFacade, Web, WebFacade, Withdrawal, WithdrawalBean, WithdrawalHome, WithdrawalWeb et WithdrawalWebFacade.

5.5.2 Extraire les mots architecturaux

La seconde sous-étape de l'extraction des règles d'affaires consiste à extraire les mots architecturaux contenus dans le nom des classes du modèle original. Cette étape est fondée sur cinq règles, les règles 1, 2, 3a, 3b et 3c.

Règle 1 : La première règle stipule que si un patron isolé n'est pas le nom d'une classe alors ce patron est un mot architectural candidat. Cette liste servira à délimiter l'espace de recherche pour les quatre autres règles recherchant les mots architecturaux. Elle se base sur l'intuition qu'un mot représentant un détail de l'implantation d'une architecture a peu de chance d'être le nom d'une classe si le développeur utilise une façon logique pour nommer des identificateurs des classes.

Pour l'exemple de la figure 5.1 les patrons qui ne sont pas le nom d'une classe sont : AccountWeb, Bank, BankTransactionWeb, Bean, Checking, CheckingAccountWeb, CustomerWeb, DepositWeb, Façade, Home, Saving, SavingAccountWeb, Transaction, TransactionBean, TransactionHome, TransactionWeb, TransactionWebFacade, Web, WebFacade et WithdrawalWeb.

Règle 2 : Notre seconde règle stipule qu'un mot architectural candidat est sélectionné pour faire partie du vocabulaire d'architecture si au moins deux classes font partie exclusivement de son groupe (c.-à-d. n'appartenant pas à d'autres groupes). Cette règle se base sur le fait que, dans les cas les plus simples à détecter, un mot architectural serait implanté par des classes d'affaires qui se retrouvent dans le modèle original. Donc en accolant le mot architectural à leur nom, nous devrons retrouver le nom d'une autre classe du modèle. Car nous supposons qu'un mot architectural doit se retrouver au moins deux fois dans le code source. Cela vient du fait que nous considérons que nous sommes face à du code répétitif ou semi-répétitif.

Règle 3a : Un mot architectural candidat est sélectionné pour faire partie du vocabulaire d'architecture, si son groupe satisfait les trois conditions suivantes :

1. il contient au moins deux classes
2. une de ses classes possède au moins une méthode principale
3. toutes ses classes ont les mêmes méthodes principales

Cette règle se base sur le fait que le mécanisme architectural implanté par un mot architectural est censé offrir les mêmes services pour toutes ses instances et qu'en conséquence, il doit offrir les mêmes méthodes principales.

Règle 3b : Un mot architectural candidat est sélectionné pour faire partie du vocabulaire d'architecture si un de ses sous-groupes, suivant les critères d'implantation d'interfaces ou d'héritage de classes abstraites, satisfait les trois conditions suivantes :

1. il contient au moins deux classes
2. une de ses classes possède au moins une méthode principale
3. toutes ses classes ont les mêmes méthodes principales

Cette règle se base sur la constatation qu'un mot architectural peut implanter plusieurs mécanismes architecturaux. Chaque mécanisme offre toujours les mêmes services pour toutes ses instances. Nous supposons que cette variation peut être identifiée par la variation des interfaces implantées (ou des classes abstraites héritées) par les classes contenant ce mot architectural. Comme dans le cas de l'exemple de la figure 5.1, le mot architectural `Bean` implante parfois un `EntityBean` et parfois un `SessionBean`.

Règle 3c : Un mot architectural candidat est sélectionné pour faire partie du vocabulaire d'architecture si un de ses sous-groupes, suivant le critère d'appartenance à un espace de noms, satisfait les conditions suivantes :

1. il contient au moins deux classes
2. toutes ses classes possèdent au moins une méthode principale

3. toutes ses classes ont les mêmes méthodes principales

Cette règle se base sur le fait que des classes d'un groupe relié à un mot architectural, peuvent parfois selon les dossiers ou selon les paquetages où elles se retrouvent, varier dans les services qu'elles offrent.

L'application des règles 2, 3a, 3b et 3c sur les mots architecturaux candidats obtenus lors de la sous-étape précédente a permis de ne laisser que trois mots architecturaux qui sont : `Bean`, `Home` et `WebFacade`. Une seule règle acceptante est suffisante pour que le mot soit accepté comme l'illustre le tableau 5.2. Les mots candidats non listés, ne sont pas acceptés et un X signifie une acceptation par la règle.

5.5.3 Extraire les classes d'affaires

La troisième sous-étape de l'extraction des règles d'affaires se concentre sur l'extraction des classes d'affaires, en se basant sur le nom des mots architecturaux isolés à l'étape précédente et se résume à la règle 4.

Règle 4 : Pour chaque classe du code source, le nom obtenu en enlevant toutes les occurrences des mots architecturaux le contenant, devient le nom d'une classe d'affaires.

Pour l'exemple de la figure 5.1, la règle 4 trouve les classes d'affaires suivantes : `Account`, `BankTransaction`, `CheckingAccount`, `Customer`, `Deposit`, `SavingAccount` et `Withdrawal`.

Tableau 5.2
Application des règles 2, 3a, 3b et 3c.

	Règle 2	Règle 3a	Règle 3b	Règle 3c
`Bean`	X		X	
`Home`	X	X	X	X
`WebFacade`	X			

5.5.4 Extraire les attributs des classes

La quatrième sous-étape vise à déterminer les attributs des classes d'affaires. Elle prend la forme des règles 5a, 5b et 5c. Ces règles se basent sur le fait qu'un attribut propre à une classe d'un groupe relié à un mot architectural est sans doute relié à la logique d'affaires de la classe d'affaires qui y est reliée et y est donc associée. Les trois règles servent simplement à s'assurer que nous vérifions-le tout pour les mots architecturaux n'implantant pas le même mécanisme architectural ou étant dans des paquetages différents comme pour la détection des mots architecturaux.

Règle 5a : Un attribut appartenant à une classe d'un groupe relié à une classe d'affaires ne sera pas considéré comme appartenant à cette classe d'affaires, s'il appartient à toutes les classes d'un groupe relié à un mot architectural contenant au moins deux classes.

Règle 5b : Un attribut appartenant à une classe d'un groupe relié à une classe d'affaires ne sera pas considéré comme appartenant à cette classe d'affaires s'il appartient à toutes les classes d'un des sous-groupes, suivant les critères d'implantation d'interfaces ou d'héritage de classes abstraites ou d'un mot architectural. Le sous-groupe en question doit contenir au moins deux classes.

Règle 5c : Un attribut appartenant à une classe d'un groupe relié à une classe d'affaires ne sera pas considéré comme attribut de cette classe d'affaires si cet attribut appartient à toutes les classes appartenant à un des sous-groupes d'un mot architectural suivant le critère d'appartenance à un espace de noms. Le sous-groupe en question doit contenir au moins deux classes.

Pour l'exemple de la figure 5.1, l'application des règles 5a, 5b et 5c donne le résultat tel qu'illustré dans le tableau 5.3. Il faut qu'un attribut soit accepté par les trois règles pour être ajouté. Ainsi, les attributs de la classe d'affaires `Account` sont : `accountNumber`, `balance`, `customer` et `openDate`. L'attribut `entityContext` sera plutôt considéré comme un attribut architectural.

Tableau 5.3
Application des règles 5a, 5b et 5c sur notre exemple d'illustration

Classe d'affaires	Attribut des classes reliées	Règle 6a	Règle 6a	Règle 6b
Account				
	accountNumber	X	X	X
	balance	X	X	X
	customer	X	X	X
	entityContext	X		X
	openDate	X	X	X
BankTransaction				
	account	X	X	X
	amount	X	X	X
	dateTime	X	X	X
	sessionContext	X		X
	transactionNumber	X	X	X
CheckingAccount				
	entityContext	X		X
Customer		X	X	X
	customerNumber	X	X	X
	homePhone	X	X	X
	name	X	X	X
	sessionContext	X		X
Deposit				
	sessionContext	X		X
SavingAccount				
	entityContext	X		X
Withdrawal				
	sessionContext	X		X

5.5.5 Extraire les opérations des classes

La cinquième sous-étape est la suite logique de la précédente et consiste à extraire les opérations pouvant être reliées aux classes d'affaires et ne reflétant pas l'architecture. Cette sous-étape se base sur les règles 6a, 6b, 6c et 7. Le raisonnement est le même pour les trois premières règles que celui que nous avons discuté lors de la sous-étape

précédente. La règle 7 permet d'identifier les opérations dont le nom est relié à celui d'un attribut d'une classe d'affaires. Ces opérations ne seront pas conservées, car elles font partie du détail de l'implantation. Un exemple fréquent de ces opérations est les « getters » et les « setters » des attributs d'une classe.

Règle 6a : Une méthode appartenant à une classe d'un groupe relié à une classe d'affaires ne sera pas considérée comme une opération de cette classe d'affaires si cette méthode appartient à toutes les classes appartenant à un groupe, contenant au moins deux classes, d'un mot architectural.

Règle 6b : Une méthode appartenant à une classe d'un groupe relié à une classe d'affaires ne sera pas considérée comme une opération de cette classe d'affaires si cette méthode appartient à toutes les classes appartenant à un des sous-groupes d'un mot architectural suivant les critères d'implantation d'interfaces ou d'héritage de classes abstraites. Le sous-groupe en question doit contenir au moins deux classes.

Règle 6c : Une méthode appartenant à une classe d'un groupe relié à une classe d'affaires ne sera pas considérée comme une opération de cette classe d'affaires si cette méthode appartient à toutes les classes appartenant à un des sous-groupes d'un mot architectural suivant le critère d'appartenance à un espace de noms. Le sous-groupe en question doit contenir au moins deux classes.

Règle 7 : Une méthode appartenant à une classe d'un groupe relié à une classe d'affaires ne sera pas considérée comme une opération de cette classe d'affaires si le nom de cette méthode contient comme patron le nom d'une classe ou le nom d'un attribut d'une classe de ce groupe.

Pour l'exemple de la figure 5.1, l'application de la règle 7 permet d'éliminer un ensemble de méthodes comme il est illustré dans le tableau 5.4. Le nom de ces opérations contient le nom d'un attribut d'une des classes du groupe correspondant à la classe d'affaires.

Tableau 5.4
Application de la règle 7 sur notre exemple d'illustration.

Classe d'affaires	Méthode	Attribut relié
Account		
	get(set)AccountNumber	accountNumber
	get(set)Balance	balance
	get(set)Customer	customer
	get(set)OpenDate	openDate
	set(unset)EntityContext	entityContext
BankTransaction		
	get(set)Account	account
	get(set)Amount	amount
	get(set)DateTime	dateTime
	get(set)TransactionNumber	transactionNumber
	set(unset)SessionContext	sessionContext
CheckingAccount		
	set(unset)EntityContext	entityContext
Customer		
	get(set)CustomerNumber	customer
	get(set)HomePhone	homePhone
	get(set)Name	name
	set(unset)SessionContext	sessionContext
Deposit		
	set(unset)SessionContext	sessionContext
SavingAccount		
	set(unset)EntityContext	entityContext
Withdrawal		
	set(unset)SessionContext	sessionContext

L'application des règles 6a, 6b et 6c permet d'éliminer un autre ensemble de méthodes comme il est illustré dans le tableau 5.5. Il faut qu'une méthode soit acceptée par les trois règles pour qu'elle ne soit pas éliminée. Ainsi, seule l'opération `calculateInterest` est considérée comme une opération d'affaires et le sera pour la classe d'affaires `Account`.

Tableau 5.5
Application des règles 6a, 6b et 6c sur notre exemple d'illustration (un X indique que la règle s'applique).

Classe d'affaires	Méthode	Règle 6a	Règle 6a	Règle 6b
Account				
	calculateInterest	X	X	X
	ejbActivate			
	ejbCreate			
	ejbLoad			
	ejbPassivate			
	ejbPostCreate	X		X
	ejbRemove			
	ejbStore			
BankTransaction				
	Create			
	ejbActivate			
	ejbCreate			
	ejbLoad			
	ejbPassivate			
	ejbRemove			
	ejbStore			
	Create			
CheckingAccount				
	ejbActivate			
	ejbCreate			
	ejbLoad			
	ejbPassivate			

	ejbPostCreate	X		X
	ejbRemove			
	ejbStore			
	create			
Customer				
	ejbActivate			
	ejbCreate			
	ejbLoad			
	ejbPassivate			
	ejbRemove			
	ejbStore			
	create			
Deposit				
	ejbActivate			
	ejbCreate			
	ejbLoad			
	ejbPassivate			
	ejbRemove			
	ejbStore			
	create			
SavingAccount				
	ejbActivate			
	ejbCreate			
	ejbLoad			
	ejbPassivate			
	ejbPostCreate	X		X
	ejbRemove			
	ejbStore			
	create			
Withdrawal				
	ejbActivate			
	ejbCreate			
	ejbLoad			
	ejbPassivate			

	ejbRemove			
	ejbStore			
	create			

5.5.6 Extraire les relations entre les classes

Une fois les classes d'affaires, leurs attributs et opérations identifiés, nous tentons de trouver les liens entre ces classes en employant les règles 8,9 et 10.

Règle 8 : Si le type d'un attribut d'une classe d'affaires `C1` est une classe d'affaires `C2` alors, cet attribut ne devient plus un attribut de la classe `C1` mais plutôt une association de la classe `C1` vers la classe `C2`. Le nom de l'extrémité `C2` de cette association portera le nom de cet attribut.

Le tableau 5.6 résume les substitutions effectuées pour l'exemple de la figure 5.1.

Règle 9 : Si dans le code source, une classe d'affaires `C1` (ou une classe de son groupe) hérite directement d'une classe d'affaires `C2` (ou une classe de son groupe), alors une relation d'héritage est créée entre les classes `C1` et `C2` dans le modèle d'affaires résultant.

Règle 10 : Si dans le code source, une interface d'un groupe relié à une classe d'affaires `C1` hérite directement d'une interface d'un groupe d'une classe d'affaires `C2`, alors une relation d'héritage est créée entre les classes `C1` et `C2` dans le modèle d'affaires résultant.

Pour l'exemple de la figure 5.1, la règle 9 n'est pas utilisée; alors que l'application de la règle 10 crée les héritages comme illustré dans le tableau 5.7. Par exemple, nous trouvons dans le code source l'interface `CheckingAccount` qui appartient au groupe relié à la classe d'affaires du même nom.

Tableau 5.6
Application de la règle 8 sur notre exemple d'illustration.

Classe	Attribut	Type de l'attribut
Account	customer	Customer
BankTransaction	account	Account

Tableau 5.7
Application de la règle 10 sur notre exemple d'illustration.

Classe d'affaires	Interface reliée	Interface héritée	Classe d'affaires reliée à l'interface héritée
CheckingAccount	CheckingAccount	Account	Account
Deposit	Deposit	BankTransaction	BankTransaction
SavingAccount	SavingAccount	Account	Account
Withdrawal	Withdrawal	BankTransaction	BankTransaction

Cette interface hérite de l'interface `Account`. Par conséquent, une relation d'héritage est créée entre leurs classes d'affaires respectives, c'est-à-dire entre la classe d'affaires `CheckingAccount` et la classe d'affaires `Account`.

5.6 CONCLUSION

Dans ce chapitre, nous avons vu les objectifs de notre approche et nous avons expliqué son fonctionnement et les règles qui la constituent. Dans le chapitre suivant, nous allons parler du processus de validation de notre approche que nous avons effectué pour l'évaluer.

CHAPITRE VI

VALIDATION DE NOTRE APPROCHE

6.1 INTRODUCTION

Dans le chapitre précédent, nous avons présenté notre approche qui permet d'extraire les règles d'affaires d'une application orientée objet. Dans ce chapitre, nous allons présenter les résultats du processus de validation que nous avons effectué pour évaluer notre approche. Nous allons parler dans la section 6.2 du prototype d'outil implantant notre approche. Par la suite, dans la section 6.3, nous allons parler du protocole que nous avons suivi lors de notre processus de validation. Puis, nous allons donner, dans la section 6.4, les résultats que nous avons obtenus sur un jeu d'essai composé de plusieurs applications de différentes tailles. Nous allons conclure ce chapitre par des conclusions tirées de notre processus de validation.

6.2 DESCRIPTION DE L'OUTIL

L'outil dont il est question a été développé avec le langage de programmation Java et permet actuellement d'extraire la logique d'affaires des applications écrites avec le langage Java. Il est constitué de trois principaux modules, comme l'illustre la figure 6.1.

Le premier a pour tâche d'analyser le code source d'un système qui lui a été fourni. L'outil utilise un analyseur syntaxique (« parser ») créé avec l'outil SableCC (Gagnon et Hendren, 1998). À partir de l'arbre syntaxique obtenu, le module remplit une structure d'objets commune aux trois modules permettant la manipulation et des opérations sur ceux-ci. Les informations emmagasinées pour les classes et les

interfaces sont : leur nom complet avec leur paquetage, leurs modificateurs, les interfaces implantées ou héritées, leurs parents, leurs attributs et types et leurs méthodes, types et arguments.

Le second module a pour tâche de récupérer la logique et est une implantation de nos règles en Java. Afin de vérifier le fonctionnement des différentes règles, l'outil produit des fichiers permettant d'en mesurer le fonctionnement. Pour cela, il crée un fichier HTML permettant de retracer les différentes étapes du fonctionnement des règles.

Le troisième module crée un document XML décrivant les classes d'affaires isolées. Ce document XML suit la norme XMI (pour XML Model Interchange) (Grose, Doney et Brodsky, 2002). Le but ici, comme nous l'avons déjà mentionné dans le chapitre précédent, est de permettre d'utiliser n'importe quel outil de modélisation UML supportant la norme XMI afin d'importer le modèle dans l'outil et ainsi permettre une visualisation graphique du modèle d'affaires extrait.

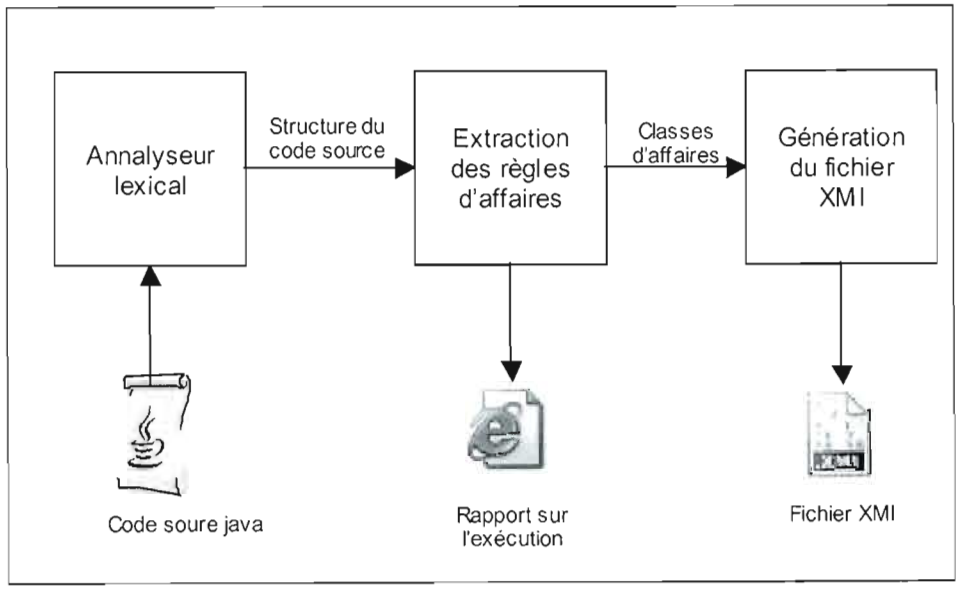

Figure 6.1 Architecture de l'outil

6.3 CADRE DE VALIDATION

Afin de vérifier la pertinence de notre approche, nous avons donc appliqué ce prototype à différentes applications et analysé les fichiers résultants pour en tirer des statistiques et des enseignements. L'appendice D présente un sommaire des résultats obtenus.

La recherche d'éléments par la rétro-ingénierie peut mener à trois résultats possibles (Philippow *et al.*, 2005). Un élément identifié par les résultats de la recherche est un « vrai positif » s'il fait bien partie des résultats attendus. Il est un « faux positif » s'il ne fait pas parti des résultats attendus. Enfin, un « faux négatif », est un élément qui devrait s'y retrouver, mais qui ne l'est pas. Basé sur ces résultats, il est possible de développer des métriques. Dans notre cas, nous employons les deux métriques suivantes : la précision et la couverture.

Dans ce qui va suivre, nous allons définir ces deux propriétés, puis nous allons parler des éléments que nous voulons mesurer. Nous allons aussi mentionner le protocole que nous avons suivi pour obtenir ces mesures.

6.3.1 Précision

Une première mesure intéressante à extraire est la précision. Elle cherche à évaluer si les éléments identifiés comme composantes d'affaires en sont vraiment. Par exemple si nous avons identifié 50 classes d'affaires et que sur ces 50, 5 n'en sont pas vraiment nous avons donc une précision de 90 % (50-5/50). Par exemple, les classes identifiées par le prototype comme classes d'affaires qui n'en sont pas vraiment sont des « faux positifs ».

6.3.2 Couverture (Taux de rappel)

Une seconde mesure intéressante à extraire est la couverture (ou taux de rappel). Il s'agit d'évaluer quel pourcentage des éléments recherchés est identifié. Ainsi, si nous

savons (ou estimons) que dans le modèle original il y a 50 classes d'affaires et que le prototype en ignore 5, nous pouvons dire que nous avons une couverture de 90 % (50 - 5/50).

6.3.3 Éléments mesurés et protocole suivi

Nous avons essayé d'obtenir des mesures sur tous les éléments importants de notre approche. Ces éléments sont : les mots architecturaux, les classes d'affaires, les opérations d'affaires, les attributs d'affaires et les relations. Deux types de relations nous intéressent principalement : l'héritage et les associations.

Le protocole que nous avons suivi pour obtenir les mesures se résume dans les points suivants :

1. Les résultats obtenus par notre prototype sont comparés avec la documentation du système analysé s'il en possède. La documentation que nous recherchons est un document contenant un diagramme de classes représentant les classes d'affaires du système ou tout document permettant d'extraire ce genre d'informations.
2. Dans le cas où cette comparaison montrerait que la documentation est peut-être non à jour, une analyse manuelle du code source est faite afin de vérifier notre hypothèse.
3. Si la documentation n'est pas présente (ce qui est fréquent malheureusement), les résultats obtenus par le prototype sont comparés avec les résultats obtenus manuellement suite à une analyse manuelle du code source.
4. L'analyse manuelle du code source est faite quand la taille du système analysé le permet.
5. Dans le cas où la taille du système ne permettrait pas une analyse manuelle, cette analyse sera plutôt faite sur un échantillon du système analysé. Dans le cas de l'échantillonnage, nous avons sélectionné en analysant le code source un ensemble de classes d'affaires et de couches architecturales présentes dans le système pour ensuite vérifier leur détection par l'outil.

6.4 LES LOGICIELS ANALYSÉS

Nous avons validé notre approche sur plusieurs systèmes de différentes tailles. Comme le prototype d'outil que nous avons développé pour supporter notre approche accepte seulement le code écrit en langage de programmation Java, tous les systèmes analysés sont donc des systèmes développés dans ce langage. Certains systèmes implantent l'architecture J2EE, d'autres utilisent d'autres types d'architecture. L'éventail de ces systèmes est assez large et couvre plusieurs domaines d'application. À l'appendice D il y a un tableau contenant les résultats pour tous les systèmes et en plus le nombre de faux positifs et de faux négatifs pour les différentes données. Dans ce qui va suivre, nous allons discuter des résultats pour cinq systèmes. Les validations d'autres systèmes seront discutées à l'appendice C. Les autres systèmes sont donc :

- Jedit : Un éditeur de texte (Pestov, Gellene et Ezust).
- ArgoUML : Un outil de modélisation UML (Ramirez *et al.*, 2006) très utilisé surtout au sein de la communauté universitaire.
- ComPiere : Un progiciel de gestion de la relation client pour les PME dans la distribution et le service (ComPiere, 2006) .
- NetBean : Un environnement de développement de la compagnie Sun Microsystems (Pestov, Gellene et Ezust).
- Eclipse : Un environnement de développement (D'Anjou *et al.*, 2004) de la compagnie IBM.

6.4.1 Premier Test

Cette première mise en œuvre de notre algorithme s'effectuait sur un exemple assez facile qui est l'application bancaire présentée lors de l'illustration de notre approche au chapitre précédent. En fait, elle a été conçue spécialement pour mettre en évidence

les propriétés de notre approche. Cela explique le résultat parfait réalisé (voir le tableau 6.1). Cet exemple implante une partie de l'architecture J2EE.

6.4.2 Code généré par SourceCafe

Nous avons soumis à notre outil un exemple de code généré avec un générateur de code nommé « SourceCafe » de la compagnie EJD Technologies (EJD, 2006). Ce qui est intéressant avec cet exemple, c'est qu'un diagramme représentant les classes d'affaires se trouvant dans cette implantation est disponible (voir figure 6.2).

Tableau 6.1
Résultats obtenus pour le premier test

Nombre de lignes de code	158	Nombre de classes en entrée	28
Nombre de méthodes	88	Nombre de classes conservées	7
Nombre d'attributs	45		
Nombre de mots architecturaux détectés	3	Nombre d'attributs d'affaires estimés	9
Nombre de mots architecturaux estimés	3	Nombre d'attributs d'affaires détectés	9
Précision en %	100,00	Précision en %	100,00
Couverture en %	100,00	Couverture en %	100,00
Nombre de classes d'affaires estimées	7	Nombre d'héritages estimés	4
Nombre de classes d'affaires détectées	7	Nombre d'héritages détectés	4
Précision en %	100,00	Précision en %	100,00
Couverture en %	100,00	Couverture en %	100,00
Nombre d'opérations d'affaires estimées	1	Nombre d'associations estimées	2
Nombre d'opérations d'affaires détectées	1	Nombre d'associations détectées	2
Précision en %	0,00	Précision en %	100,00
Couverture en %	0,00	Couverture en %	100,00

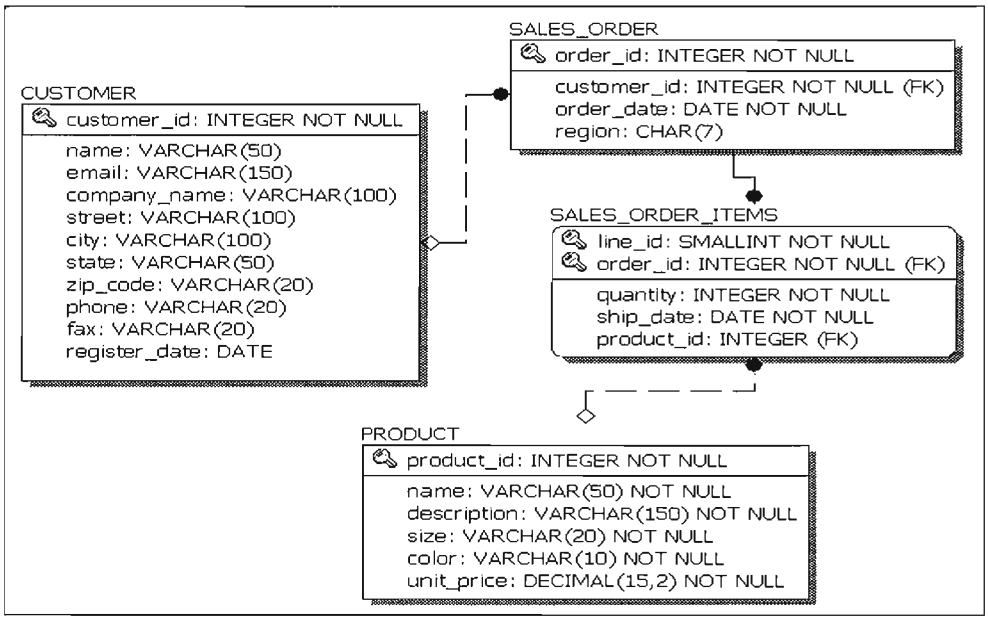

Figure 6.2 Diagramme des classes d'affaires du système généré par SourceCafe (tirée de EJD, 2006)

Le résultat est affiché à la figure 6.3. Notre outil a permis de détecter pratiquement tous les mots architecturaux (couverture de 83.33%, voir tableau 6.2). Cependant, la précision de nos règles pour cet aspect n'a été que de 45.45%. L'approche a identifié de faux mots architecturaux (par exemple, le mot SALES_ORDER) à cause de la règle 3b. La raison de ceci est que toutes les classes reliées à un faux mot architectural font partie du groupe d'un vrai mot architectural; c'est pourquoi la règle 3b se trouve satisfaite. Il suffit donc d'ajouter une condition supplémentaire faisant cette vérification pour que le problème se corrige.

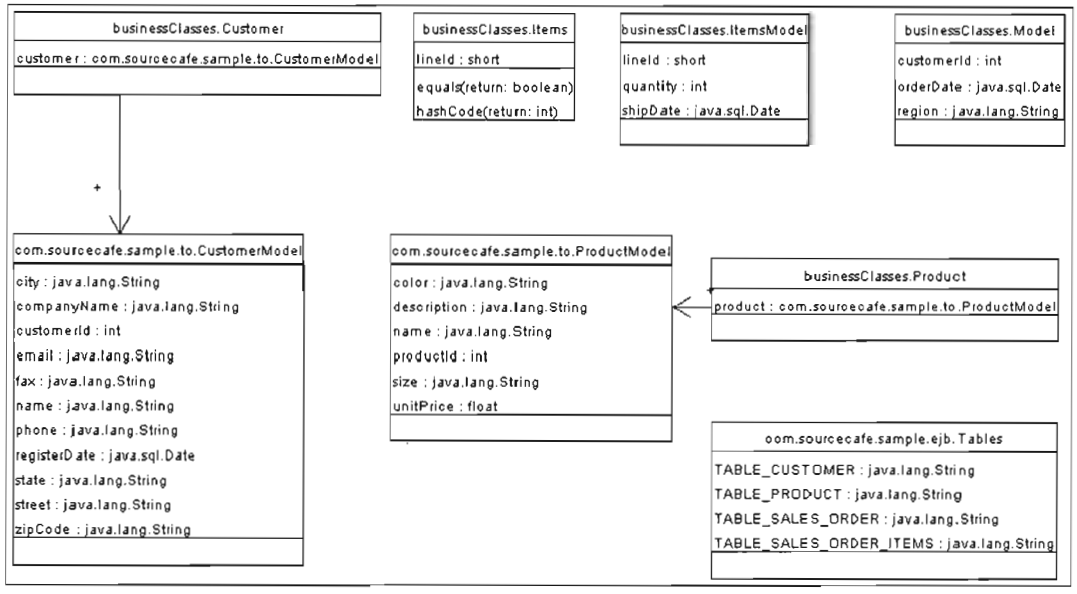

Figure 6.3 Modèle obtenu pour le système généré par SourceCafe

Notre outil a trouvé la plupart des classes d'affaires (couverture de 75 %); quant à la précision, elle a été de 50 %. Un seul mot architectural non identifié a été responsable de la majorité des erreurs dans l'identification des classes d'affaires. En effet, le système utilise des noms de classes qui finissent par le mot `Model`. Or, nos règles n'ont pas permis d'identifier ce mot comme faisant partie de l'architecture. En outre, le système contient des classes utilitaires que l'outil a considérées comme des classes d'affaires. Par conséquent, il nous faut trouver une autre technique pour bien identifier les classes utilitaires. Peut-être en nous inspirant de celle de Hamou-Lhadj *et al.* (Hamou-Lhadj *et al.*, 2005).

Tableau 6.2
Résultats obtenus pour le système généré par SourceCafe

Nombre de lignes de code	1091	Nombre de classes en entrée	29
Nombre de méthodes	207	Nombre de classes conservées	8
Nombre d'attributs	53		
Nombre de mots architecturaux détectés	11	Nombre d'attributs d'affaires estimés	26
Nombre de mots architecturaux estimés	6	Nombre d'attributs d'affaires détectés	30
Précision en %	45,45	Précision en %	80,00
Couverture en %	83,33	Couverture en %	92,31
Nombre de classes d'affaires estimées	4	Nombre d'héritages estimés	0
Nombre de classes d'affaires détectées	8	Nombre d'héritages détectés	0
Précision en %	50,00	Précision en %	100,00
Couverture en %	100,00	Couverture en %	100,00
Nombre d'opérations d'affaires estimées	0	Nombre d'associations estimées	3
Nombre d'opérations d'affaires détectées	2	Nombre d'associations détectées	2
Précision en %	0,00	Précision en %	0,00
Couverture en %	100,00	Couverture en %	0,00

Pour les opérations d'affaires, il n'y en a pas, car le code original ne fait que représenter le modèle dans l'architecture EJB. Les deux opérations qui ont été trouvées viennent des faux mots architecturaux détectés. Pour les attributs d'affaires la majorité ont été détectés sauf product_id car toutes les classes implantant `java.io.Serializable` le possèdent.

L'évaluation de la détection des relations d'héritage et d'association est plus complexe. Une des classes d'affaires, ayant 2 des 3 associations, n'a pas été détectée. Il devait y avoir une relation entre `CUSTOMER` et `SALES_ORDER`, et entre `SALES_ORDER` et `SALES_ORDER_ITEMS`, mais `SALES_ORDER` est détectée comme mot architectural. Dans le tableau, nous avons été sévères : les relations détectées bien que bonnes avec les classes détectées ne représentent pas ce que nous cherchions. Donc précision et couverture de 0%. Aucun héritage n'était recherché.

6.4.3 Java Pet Store

Java Pet Store est un exemple classique, fourni par Sun, illustrant l'application de la technologie J2EE (Singh, Stearns et Johnson, 2002).

Notre approche donne une précision et une couverture de respectivement 17.39 % et 22.22 % pour l'identification des mots architecturaux (voir tableau 6.3). La raison principale de ceci est le fait qu'un mécanisme architectural représentant le mot architectural n'est pas toujours implanté de façon uniforme partout dans le code. Ainsi, les classes implantant ce mécanisme ne possèdent pas les mêmes méthodes principales (voir les règles 3a et 3b). Une façon pour remédier à la situation serait de voir comment les pondérer. La figure 6.4 représente les résultats d'un autre test réalisé avec ce système afin de vérifier l'intérêt de donner une certaine tolérance dans le pourcentage de méthodes principales partagées entre les classes d'un groupe. En passant de 0 à 100 % (0 à 10 dans l'image). Les noms des mots architecturaux en rouge sont des faux positifs, en jaune, des mots architecturaux dont la pertinence est discutable et en vert des mots architecturaux qui sont pertinents. Nous voyons qu'à 60 % (6) nous trouvons le maximum de mots architecturaux.

Tableau 6.3
Résultats obtenus pour le système Java Pet Store

Nombre de lignes de code	6150	Nombre de classes en entrée	188
Nombre de méthodes	1253	Nombre de classes conservées	134
Nombre d'attributs	628		
Nombre de mots architecturaux détectés	23	Nombre d'attributs d'affaires estimés	177
Nombre de mots architecturaux estimés	18	Nombre d'attributs d'affaires détectés pour les classes d'affaires correctement détectées	176
Précision en %	17,39	Précision en %	91,48
Couverture en %	22,22	Couverture en %	90,96
Nombre de classes d'affaires estimées	45	Nombre d'héritages estimés	4
Nombre de classes d'affaires détectées	134	Nombre d'héritages détectés	4
Précision en %	20,15	Précision en %	0,00
Couverture en %	60,00	Couverture en %	0,00
Nombre d'opérations d'affaires estimées	72	Nombre d'associations estimées	7
Nombre d'opérations d'affaires détectées pour les classes d'affaires correctement détectées	68	Nombre d'associations détectées pour les classes d'affaires correctement détectées	32
Précision en %	73,53	Précision en %	0,00
Couverture en %	69,44	Couverture en %	0,00

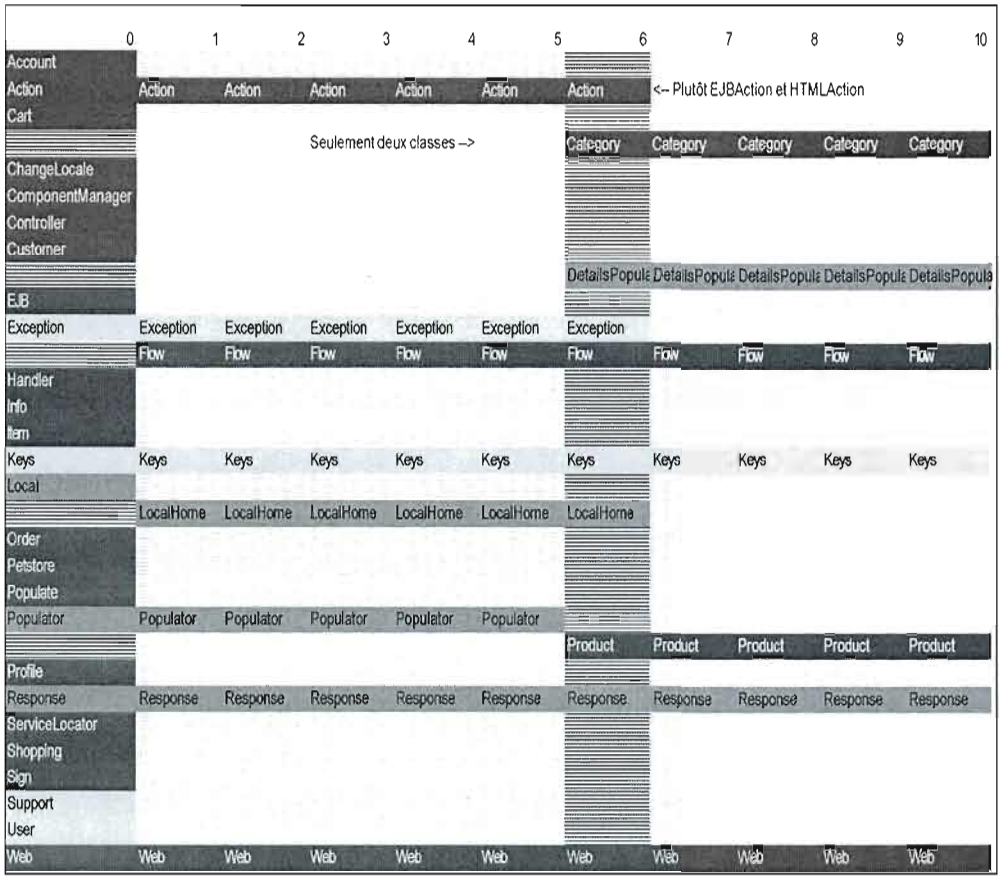

Figure 6.4 Nombre de mots architecturaux détectés et méthodes principales

Pour la détection des classes d'affaires, l'outil donne une précision et une couverture respectivement 20.15 % et 60.00 %. Comme l'identification des mots architecturaux a une incidence sur l'extraction des classes d'affaires, augmenter les résultats sur le premier contribuera sûrement à augmenter les résultats du second. Au total, 28 classes d'affaires ont été correctement détectées.

Comme il y a plusieurs mots architecturaux et classes d'affaires détectées faussement et plusieurs autres qui ne le sont pas, il est difficile de mesurer avec fiabilité la détection des attributs et des méthodes. La méthode choisie a consisté – pour chacune des classes d'affaires estimées et effectivement détectées – à déterminer leurs attributs et méthodes de ces classes. Les résultats pour la précision et la couverture

des attributs et des classes d'affaires détectées sont relativement bons, car en examinant les classes d'affaires nous constatons que la majorité de ces éléments se retrouvent dans la classe de base (celle qui n'implante pas l'architecture) et qui est celle généralement détectée. Les résultats sont un peu moins bons pour les méthodes, car nous constatons que les méthodes d'affaires sont un peu plus souvent réparties dans les classes implantant un mot architectural et aussi qu'une méthode en lien avec un attribut d'affaires non détecté ne va pas être conservée. Une fois les mots architecturaux correctement détectés, il est évident que la précision augmentera.

En ce qui concerne les associations et les héritages, nous avons décidé d'y aller sévèrement comme avec le système précédent. Nous avons comparé directement ce qui a été trouvé avec ce que nous estimions devoir y être. Pour ce faire, nous nous sommes basés sur les classes d'affaires estimées. Les résultats ne sont pas très convaincants. Il y a beaucoup trop d'associations détectées, car il y a trop de classes d'affaires trouvées, ce qui implique la possibilité de retrouver plus de liens entre elles. Les héritages, quant à eux, sont aussi mal détectés, car il s'agit d'héritages entre classes d'affaires non détectées pour au moins l'une des deux. Encore ici, une meilleure détection des classes d'affaires augmenterait cette performance.

6.4.4 OTN Financial Brokerage Service

OTN Financial Brokerage Service (OFBS) (Oracle, 2006) est un logiciel qui simule un service de courtage en ligne où les utilisateurs peuvent échanger des actions, lire des nouvelles financières, et contrôler leur portefeuille.

Notre approche donne une précision (respectivement couverture) de 31.58 % (respectivement 27.27 %) dans l'identification des mots architecturaux (voir le tableau 6.4). Pour les classes d'affaires, l'outil donne une précision (respectivement couverture) de 41.38 % (respectivement 72.73 %).

Tableau 6.4
Résultats obtenus pour le système OFBS

Nombre de lignes de code	3082	Nombre de classes en entrée	75
Nombre de méthodes	665	Nombre de classes conservées	58
Nombre d'attributs	132		
Nombre de mots architecturaux détectés	19	Nombre d'attributs d'affaires estimés	62
Nombre de mots architecturaux estimés	22	Nombre d'attributs d'affaires détectés	62
Précision en %	31,58	Précision en %	96,77
Couverture en %	27,27	Couverture en %	96,77
Nombre de classes d'affaires estimées	33	Nombre d'héritages estimés	2
Nombre de classes d'affaires détectées	58	Nombre d'héritages détectés	3
Précision en %	41,38	Précision en %	0,00
Couverture en %	72,73	Couverture en %	0,00
Nombre d'opérations d'affaires estimées	53	Nombre d'associations estimées	0
Nombre d'opérations d'affaires détectées	52	Nombre d'associations détectées	1
Précision en %	100,00	Précision en %	0
Couverture en %	98,11	Couverture en %	100,00

Les explications données pour le système précédent au sujet des résultats obtenus lors de l'identification des mots architecturaux restent aussi valables pour cet exemple. La seule différence est que cette identification n'a pas eu beaucoup d'incidences sur les résultats de l'identification des classes d'affaires. Les remarques pour les attributs, les méthodes, les héritages et les relations sont aussi semblables.

6.4.5 Virtual Shopping Mall (VSM)

VSM est une application implantant un centre commercial virtuel (Oracle, 2006). L'analyse de cette application a été facilitée par le fait qu'Oracle fournit sur son site un diagramme de classes, soit la logique d'affaires de l'application (voir figure 6.5). Toutefois, nous avons remarqué que l'implantation ne représente pas tout à fait le modèle et que donc notre approche isole bien entendu ce qui a été implanté et non conçu.

Notre approche donne une précision (respectivement couverture) de 100.00 % (respectivement 58.33 %) dans l'identification des mots architecturaux (voir tableau

6.5). Il y a deux catégories de mots architecturaux qui ne sont pas identifiées. La raison de la non-identification de la première catégorie est la même dont nous avons déjà parlé, à savoir la non-uniformisation de l'implantation de ces mécanismes.

Pour les mots architecturaux de la deuxième catégorie, la raison est ailleurs. En effet, nous avons trouvé que chacun de ces mécanismes a été implanté seulement une fois (dans une seule classe). Or, nos règles supposent qu'un mot architectural doit se retrouver au moins dans deux endroits. Par exemple, les mots architecturaux `Entry`, `PK`, `Session` et `Response` ne sont pas trouvés, car elles n'ont qu'une seule occurrence.

Pour les classes d'affaires, l'outil donne une précision de 22.22 % et une couverture de 95.83 %. Les classes utilitaires (le système en contient plusieurs) sont responsables en grande partie de la baisse de précision. Les remarques pour les attributs, les méthodes, les héritages et les relations sont aussi semblables aux deux systèmes précédents.

Tableau 6.5
Résultats obtenus pour le système VSM

Nombre de lignes de code	3198
Nombre de méthodes	511
Nombre d'attributs	149
Nombre de mots architecturaux détectés	7
Nombre de mots architecturaux estimés	12
Précision en %	100,00
Couverture en %	58,33
Nombre de classes d'affaires estimées	49
Nombre de classes d'affaires détectées	63
Précision en %	22,22
Couverture en %	82,35
Nombre d'opérations d'affaires estimées	19
Nombre d'opérations d'affaires détectées	13
Précision en %	23,08
Couverture en %	15,79

Nombre de classes en entrée	90
Nombre de classes conservées	63
Nombre d'attributs d'affaires estimés	29
Nombre d'attributs d'affaires détectés	27
Précision en %	100,00
Couverture en %	93,10
Nombre d'héritages estimés	3
Nombre d'héritages détectés	3
Précision en %	0
Couverture en %	0
Nombre d'associations estimées	5
Nombre d'associations détectées	7
Précision en %	14,29
Couverture en %	20,00

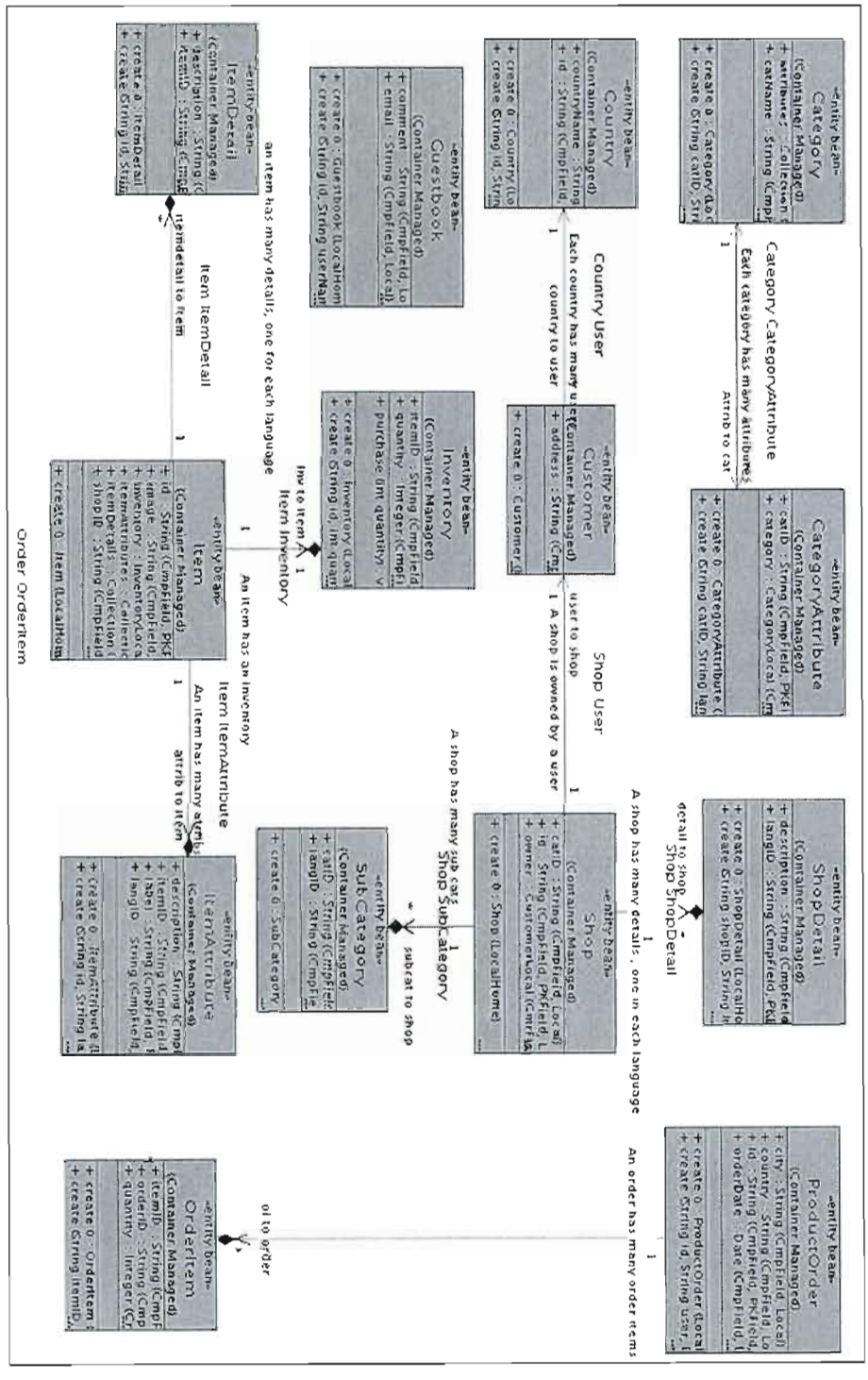

Figure 6.5 Diagramme modélisant les classes d'affaires du système VSM (figure adaptée de Oracle, 2006)

6.5 DISCUSSION

Les différents essais, que nous venons de présenter, nous aident à esquisser des réflexions et des constatations sur notre approche. Certaines nous font voir les causes des limites de notre approche alors que d'autres sont une piste de réflexion afin d'y remédier. Ces réflexions sont classées selon différents thèmes dans le texte qui suit.

6.5.1 Les identificateurs et les patrons

6.5.1.1 Flexibilité dans l'identification des patrons

Comme notre approche se fonde pour beaucoup sur les identificateurs et leurs sous-patrons, voici quelques remarques à ce sujet. Couramment dans les codes analysés, les normes que nous avons établies pour la formulation des identificateurs ne sont pas respectées ou sont différentes selon les systèmes. Par exemple, il y a des patrons partiellement ou totalement en majuscule (par exemple dans Eclipse : `BITMAP`, `BITMAPINFOHEADER`, `GC`, `GCData`, `GCRESULTS`, `GRADIEN` ou `GRADIENRECT`). Dans certains cas, c'est la séparation des identificateurs qui est en cause. Souvent, les séparateurs ne sont pas toujours employés d'une manière uniforme à l'intérieur d'un même logiciel. Les mots sont séparés par des « _ », par des majuscules, par les deux précédents à la fois, par d'autres ou même par aucun. Ainsi dans Compiere nous retrouvons `GLJournal` et `GL_Journal`. Il y a des identificateurs de classe dont le premier patron commence par un « _ » ou une minuscule (par exemple dans Eclipse : `_t` ou `utsname`). Il y a aussi des mots architecturaux qui ne diffèrent que par un nombre (par exemple dans Eclipse : `Extension2` et `Extension3`). Dans Eclipse, ce nombre peut indiquer qu'une telle classe sert à lire un fichier de la version x ou qu'il s'agit d'un numéro représentant une version différente d'une même classe.

Donc, nous constatons qu'il est important d'avoir une certaine flexibilité dans la détection des patrons constituant un mot. Les approches basées sur les identificateurs présentées au troisième chapitre montrent une voie à suivre.

6.5.1.2 La sémantique des patrons

Les identificateurs et leurs patrons peuvent signifier plusieurs choses comme des actions, des noms ou des états. Cet aspect est difficile à analyser, mais il pourrait aider à distinguer les objets architecturaux et d'affaires. Selon le contexte, un mot n'a pas toujours la même fonction. Dans Eclipse, le mot `Classe` détecté comme architectural est selon la classe, le contenant, un mot architectural ou une classe d'affaires. Il y a des classes d'affaires détectées dont le nom n'est qu'un verbe d'action ce qui nous éloigne du concept d'objet (par exemple dans NetBeans : `Add` ou `Change`). Dans ArgoUML, une fois que nous enlevons les verbes des noms de classes, nous voyons apparaître plus clairement certains objets d'affaires. Des mots architecturaux identifient un style ou un regroupement (par exemple dans Eclipse : `WithTaggs` ou `List`). D'autres mots architecturaux indiquent des mécanismes comme le patron `Msg`. Parmi les mots architecturaux, certains ne sont que le même mot écrit de façon différente, par exemple : `Adapter` et `Adaptor` ou sont les dérivés d'un même mot `Export`, `Exported` et `Exporter`. Parfois, il s'agit simplement d'une mauvaise orthographe.

Donc, si nous pouvions dans une certaine mesure détecter le type et l'utilité des patrons il serait possible de renforcer notre approche. Ce n'est pas une voie facile, mais pas impossible comme l'ont démontré Caprile et Tonella (Caprile et Tonella, 1999) avec leur approche présentée au troisième chapitre qui s'intéresse à l'aspect lexical, syntaxique et grammatical des identificateurs.

6.5.1.3 Les homonymes

Toujours dans le domaine des identificateurs, nous constatons que souvent dans les logiciels analysés, se retrouvent des classes ayant le même nom. Notamment dans Eclipse et NetBeans. Plusieurs classes peuvent avoir le même nom, mais ne pas représenter la même classe d'affaires et conséquemment, elles n'auraient pas les mêmes classes reliées. Il est donc important d'identifier un moyen pour distinguer ces classes et leurs classes reliées respectives. Peut-être, en retraçant les classes qui ont des liens entre elles ou des attributs du type d'un autre et nous pourrions aussi vérifier si une hiérarchie existe entre elles pour trouver la classe de base. Les classes homonymes pourraient aussi servir à détecter un mot architectural non intéressant. Dans Eclipse, le mot architectural `CollapseAll` apparaît seulement dans quatre classes d'affaires ayant un nom identique. C'est un signe que ce mot architectural n'en est possiblement pas un.

Donc, il est important de pouvoir bien distinguer les classes qui sont reliées entre elles.

6.5.1.4 La quantité de classes reliées

Une fois les classes d'affaires détectées, nous remarquons deux cas intéressants en rapport avec leurs classes reliées. D'abord, il y a des classes d'affaires avec une multitude de classes reliées. Par exemple, dans le cas d'Eclipse la classe d'affaires `action` possède 230 classes reliées. Au contraire, il y a des classes d'affaires n'ayant qu'une classe reliée, car elles ont été créées en retirant les mots architecturaux trouvés dans leur nom d'une classe du modèle de base.

Donc, ces observations permettent de constater qu'une piste pour améliorer les règles serait de supposer que quand un grand nombre de classes reliées pour une classe d'affaires existe, alors elle serait en partie un mot architectural. Ainsi pour Eclipse, même si `Event` existe en tant que classe d'affaires, il est souvent un mot architectural.

Une autre possibilité serait aussi d'éviter la création de classes d'affaires possédant une seule classe reliée. Peut-être ne devrions-nous garder que les classes d'affaires ayant au moins deux classes reliées. Sinon, la classe resterait dans le modèle final, mais en conservant dans son nom son mot architectural.

6.5.1.5 La présence des mots architecturaux

En nous intéressant aux mots architecturaux contenus dans les classes de base, nous pouvons découvrir des cas intéressants. Il y a des classes d'affaires qui recèlent plusieurs mots architecturaux. Par exemple : **`Add`**`Ejb`**`Reference`**`Form` (les mots architecturaux sont en gras) en possède trois. Dans le cas d'Eclipse, si nous comptons les différentes combinaisons de mots architecturaux présentes dans les classes du modèle de base nous en trouvons 3654, dont 1241 uniques. En outre, il arrive qu'une classe d'affaires soit coupée en plusieurs pièces comme la précédente. Il y a aussi des classes constituées seulement de mots architecturaux. Par exemple pour Eclipse, 757 classes disparaissent, car elles ne sont formées que de mots architecturaux.

Donc, nous pourrions supposer que les combinaisons de mots architecturaux apparaissant dans le modèle original peuvent aider à détecter ou à confirmer les mots architecturaux faits de plusieurs patrons et à éliminer ceux qui en sont de sous-patrons. Lors de la détection, nous pourrions ne pas permettre ou limiter les mots architecturaux se retrouvant dans le nom d'une même classe. Si un mot architectural coupe une classe en deux, peut-être ne pas la conserver ou explorer une autre option.

En poursuivant avec les mots architecturaux, il y en a qui ne se rencontrent qu'un nombre de fois vraiment limité parmi des classes reliées. Nous pourrions peut-être laisser tomber les mots architecturaux qui ne sont présents que pour un nombre limité de classes reliées à une classe d'affaires. Même s'il faut les retrouver plus tard comme mots architecturaux orphelins. Et si pour un mot architectural, il n'y a pas un

nombre minimal de classes ayant ce mot architectural, il pourrait être rejeté, toujours quitte à être détecté comme mot architectural orphelin pour une classe particulière.

6.5.1.6 La présence de la classe de base

En suivant les traces de l'algorithme et ses résultats, nous constatons que quand la classe de base d'une classe d'affaires n'existe pas, les méthodes des classes reliées ayant le nom de celle-ci ne sont pas filtrées. Ainsi, dans VSM, la classe d'affaires `Inventory` possède des méthodes contenant ce patron. La règle 8 ne filtre que les méthodes ayant le nom de classes existantes et dans ce cas il n'y a pas de classe d'`Inventory` dans le modèle original. Nous constatons souvent avec Compiere que le nom du logiciel se trouve dans des mots architecturaux non pertinents. Aussi la règle sur le nom des attributs fait que même pour une classe n'ayant pas de classe reliée, ses attributs et ses méthodes sont filtrés. Par exemple, la classe d'affaires de `Jedit` `org.gjt.sp.jedit.Buffer`, voit ses méthodes renfermant le patron `Buffer` disparaître.

Donc, il faudrait ajouter dans la règle sur les attributs une condition sur le nom de la classe d'affaires de base. Une autre idée de règle serait qu'un mot architectural ne puisse pas contenir le nom du logiciel. Il serait aussi utile quand une classe d'affaires n'a pas de classe reliée avec un mot architectural, de ne pas lui appliquer les règles à ce sujet. Et un peu comme pour les méthodes, nous devrions éviter de conserver les attributs en liens avec le nom de la classe d'affaires.

6.5.2 Répéter les itérations

Afin de raffiner la détection, l'algorithme pourrait fonctionner en plusieurs étapes. Nous savons, que les classes d'affaires sont détectées en ôtant du nom des classes du modèle original les mots architecturaux détectés. Une fois les classes d'affaires détectées, il serait intéressant de refaire les règles 2 et 3 en ajoutant les classes d'affaires isolées à la liste des classes d'affaires et donc en les retirant des mots

architecturaux candidats. De même, une fois que les classes d'affaires sont reconnues, il serait imaginable de distinguer les classes qui y sont reliées et en extraire d'autres mots architecturaux. Ensuite, il serait possible d'extrapoler des classes d'affaires non découvertes avec ces nouveaux mots architecturaux.

6.5.3 Assurer une traçabilité

Tout comme l'approche de Fiutem *et al.* (Fiutem *et al.*, 1996), vue au troisième chapitre, qui permet une navigation hiérarchique entre différents niveaux d'abstraction, il serait utile de pouvoir naviguer directement entre le modèle d'affaires et le modèle résultant. Ce qui assurerait une traçabilité à notre approche. En examinant l'approche de Richner et Ducasse (Richner et Ducasse, 1999), aussi présentée dans le même chapitre et qui autorise la confection de vues à la demande, nous nous rendons compte qu'une certaine interactivité dans la création du modèle serait profitable. Car nous notons que certaines règles sont plus souhaitables pour certains logiciels que d'autres. De plus avec cette forme d'interactivité il serait concevable de pouvoir régler le logiciel avec un seuil de sensibilité comme Hamou-Lhadj *et al.* (Hamou-Lhadj *et al.*, 2005) le font. Par exemple, chercher un nombre maximal de mots architecturaux et laisser tomber les moins prometteurs. Ensuite, dans le cas où la détection serait insatisfaisante, rendre les critères de sélection plus ou moins souple. L'approche de Hamou-Lhadj *et al.* (Hamou-Lhadj *et al.*, 2005) pourrait aussi nous inspirer pour le filtrage des classes utilitaires. En effet, nous constatons que dans la plupart des cas le modèle final est encombré de plusieurs classes de ce type.

6.5.4 Assurer la flexibilité

L'utilisation de la logique floue comme dans l'approche de Jahnke, Schäfer et Zundorf (Jahnke, Schäfer et Zundorf, 1997) présentée au troisième chapitre serait peut-être intéressante, car dans notre cas tout est rarement tout blanc ou noir et nous

avons donc couramment à faire avec des nuances de gris. Par exemple, si nous observons les relations d'héritage et d'implantation, nous découvrons que souvent un groupe qui implante ou hérite d'une même classe forme les membres d'un mot architectural. Nous pourrions dans le cas où nous adopterions une approche avec incertitude renforcer la probabilité de la présence d'un mot architectural quand toutes ses instances implantent ou héritent d'une même classe. Dans le cas d'une approche avec incertitude, nous pourrions aussi pénaliser les mots architecturaux qui ont moins d'instances. Ainsi, plus il y aurait d'instances pour un mot architectural candidat, plus il serait toléré un grand nombre de différences dans ses méthodes propres. Aussi, nous voyons aussi que les classes implantant `java.io.Serializable` ont tendance à représenter des classes de bases. Nous pourrions donner une probabilité plus forte à ces classes d'être des classes d'affaires.

6.5.5 Diviser pour régner

D'une manière plus générale, nous pouvons imaginer qu'idéalement le nombre de mots architecturaux ne devrait pas augmenter de façon explosive avec le nombre de classes dans un système. Quand nous voyons la taille d'Eclipse et les résultats, la conclusion vraisemblablement est qu'il est illusoire de vouloir tout analyser d'un seul coup. L'idéal serait peut-être de trouver un moyen de décomposer le logiciel en morceau avant l'analyse. Peut être en effectuant une partition comme proposée par Mitchell, Mancoridis et Traverso (Mitchell, Mancoridis et Traverso, 2002).

6.5.6 Comparaison de notre approche avec quelques approches concurrentes

Il est délicat de comparer notre approche à d'autres, car la forme de logique d'affaires résultante varie beaucoup d'une à l'autre. Nous la comparons ici à quelques autres approches ayant pour but de montrer la logique d'affaires d'une application. Nous montrons des forces et des faiblesses de la nôtre face à elles. De même, que des pistes que nous pouvons en tirer afin de l'améliorer.

Une approche près de la nôtre par une des formes d'affichage de son résultat est celle de Di Lucca *et al* .(Di Lucca *et al.*, 2003). Elle crée des diagrammes de classes semblables aux nôtres. Par contre, elle vise des applications Web ce qui pose certaines problématiques comme d'isoler des objets dont les propriétés peuvent être réparties dans différents langages souvent non objets et l'implémentation et aussi bien sur le client que le serveur. Ce qui est une approche qui pourrait améliorer la nôtre, car souvent des applications comportent une partie Web, comme celles créées avec J2EE. Une autre différence majeure est que la procédure n'est pas totalement automatisée ce qui implique des risques d'erreurs humaines et les coûts humains. Mais le facteur humain donne sans doute plus de souplesse. Ce pourrait être intéressant d'amener une certaine interactivité dans notre approche où l'utilisateur pourrait indiquer au système des points que ce dernier ne voit pas.

L'approche de Hung et Zou (Hung et Zou, 2005) présentée à la section 3.3.2.10, s'intéresse, elle aussi, à l'expression de la logique d'affaires. Elle examine le flux d'information entre la couche logique et la couche de base de données. Contrairement à la nôtre, elle est limitée à une architecture trois tiers. Le résultat final est donné sur la forme de fichiers XML identifiant les données d'affaires et les politiques d'affaires. Le résultat est moins intuitif que nos diagrammes UML. Par contre, l'approche permet de cerner l'interaction entre les différents objets d'affaires. Cela serait une voie intéressante à explorer dans notre approche afin de mieux expliquer le comportement d'une application.

Earls, Embury et Turner (Earls, Embury et Turner, 2002) proposent une méthodologie « manuelle » pour extraire la logique d'affaires. Cette méthodologie est une analyse du code source d'une application dans un traitement de texte qui permet en analysant les différentes structures et notamment les conditions d'erreurs, d'énoncer des règles d'affaires. On retrouve ici aussi les limites et les forces des approches humaines. Par contre, le résultat est des règles d'affaires décrites en langage naturel. Ce qui peut être un point positif malgré l'imprécision de ce langage.

Surtout si les règles doivent être présentées à des non-informaticiens. Ce serait aussi une voie intéressante de tenter de fournir en plus une description du modèle résultant à ce niveau d'abstraction.

L'approche de Huang *et al.*(Huang *et al.*, 1996) est basée sur le tranchage de code. Elle commence par rechercher les variables d'entrées/sorties des fonctions et du système qui seront considérées comme données d'affaires. Pour chacune de ces variables, les portions de code où elles sont impliquées sont identifiées. Les règles d'affaires associées à une donnée d'affaires peuvent être alors présentées sous trois formes. D'abord, les règles peuvent s'exprimer comme une « tranche de code » reliée à une donnée d'affaires. Deux abstractions d'un niveau légèrement plus élevé représentent la formulation de la tranche sous forme algébrique ou d'un diagramme représentant les liens entre les données en entrées et celles en sortie. Contrairement à notre approche, le résultat reste étroitement associé au code source de l'implémentation. L'idée d'associer des aspects d'affaires à des sections de code peut être intéressante surtout, si on s'intéresse à des approches d'ingénierie circulaire.

6.5.7 Simplifier la tâche de rétro-ingénierie.

Nous avons constaté qu'une des difficultés de la rétro-ingénierie vient du fait que chaque système à analyser comporte ses particularités. Dans ce contexte il est donc difficile de faire une approche qui puisse s'adapter à la majorité des cas. Il serait donc intéressant que dans le futur les développeurs utilisent des stratégies de développement favorisant la rétro-ingénierie. D'abord, l'utilisation d'outils générateurs de code pourrait faciliter la rétro-ingénierie, car ils créent du code semi-répétitif. Comme le démontre l'exemple de code créé par le générateur de code « SourceCafe » de la compagnie EJD Technologies (EJD, 2006). De même, l'adoption et le respect de normes comme J2EE peuvent aider les approches de rétro-ingénierie. Car ils favorisent l'adoption d'une architecture plus facile à isolée. De plus si le développeur adopte une architecture qui permet d'isoler la logique d'affaires de

l'architecture, celle-ci sera plus facile à isoler du reste. L'approche trois tiers en est un bon exemple (Hung et Zou, 2005).

6.5.8 Les bases de connaissances

Comme indiqué à la section 2.2.2, certains algorithmes de rétro-ingénierie sont fondés sur des bases de connaissances. Nous avons vu des approches comme celle de Heuzeroth *et al.* (Heuzeroth *et al.*, 2003) et de Huang *et al.* (Huang *et al.*, 2005) qui permettent de décrire des informations sur les patrons à rechercher. Il est vrai qu'en connaissant ce qu'il y a recherché la tâche peut être simplifiée. Par contre, le fait de spécifier ce qu'il y a à rechercher, risque de mettre des « œillères » à l'approche et de lui faire ignorer des situations inattendues. Afin de garder une souplesse à notre approche tout en la renforçant, il serait peut-être intéressant de la faire collaborer avec une base de connaissance, mais en ne se reposant pas totalement sur elle. Une idée de base de connaissances à employer serait un dictionnaire de verbes qui aiderait à distinguer le nom des objets d'affaires, car en général le nom d'un objet d'affaires n'est pas un verbe.

6.6 CONCLUSION

Nous avons présenté dans ce chapitre l'outil que nous avons développé pour implanter notre approche, le protocole que nous avons suivi lors de notre processus de validation, les résultats de notre processus de validation. Nous avons conclu ce chapitre par une discussion sur notre processus de validation et des pistes d'améliorations à suivre.

CONCLUSION

Au cours de leur vie, les logiciels sont amenés à évoluer. Cette évolution peut être notamment rendue nécessaire afin de corriger des erreurs ou pour s'adapter à de nouvelles exigences ou de nouvelles technologies. Pour mettre en œuvre cette évolution, il faut souvent comprendre des logiciels mal documentés. En l'absence d'une documentation adéquate, la compréhension des logiciels devient difficile et accapare une grande partie des ressources d'une équipe de développement. Simplifier le processus de compréhension du logiciel devient alors nécessaire. Une bonne manière d'y arriver est de permettre l'extraction d'une représentation abstraite du code source d'un logiciel.

Une des abstractions intéressantes pouvant simplifier la compréhension du logiciel représente ses règles d'affaires. Dans ce mémoire, nous avons présenté une nouvelle approche qui permet d'analyser le code source d'une application orientée objet et de créer un modèle représentant ses règles d'affaires. Ce modèle prend la forme d'un diagramme de classes UML présentant les classes d'affaires de cette application. Un prototype d'outil supportant notre approche a été développé et accepte présentement les systèmes écrits en Java. Le prototype a été validé sur différents systèmes de tailles variées avec des résultats variables. Nous avons obtenu des résultats convaincants pour ceux possédant une architecture implantée uniformément. Étant basée sur les identificateurs trouvés sur le code source, notre approche donne de bons résultats pour les systèmes qui ont été développés en suivant les bonnes règles de nommage lors de l'étape de programmation.

Le processus de validation nous a aussi permis d'identifier des pistes qui peuvent nous aider à améliorer notre approche. Par exemple, nous avons remarqué que beaucoup de modèles obtenus sont pollués de classes utilitaires. Leur détection est donc l'un des enjeux importants. Aussi nous avons vu que l'ajustement de certaines

règles, par l'ajout de conditions supplémentaires ou l'ouverture à une certaine tolérance, peut améliorer les résultats obtenus.

Le processus d'extraction des règles d'affaires est un processus complexe qui va demander beaucoup d'énergie pour atteindre des résultats optimaux et supporter les systèmes ayant une architecture moins uniforme et n'ayant pas été développés suivant les bonnes règles de l'art. Malheureusement, c'est le genre du système que nous trouvons le plus fréquemment. Par exemple, comme notre approche se base sur une bonne stratégie de nommage, les règles peuvent être bien adaptées; mais si nous ne pouvons distinguer les patrons d'un mot, elles deviennent inutiles. Donc, la séparation des patrons des identificateurs demeure un enjeu. C'est pourquoi nous comptons explorer d'autres techniques utilisées dans la rétro-ingénierie autre que l'utilisation des identificateurs, telles que l'analyse des flux de données ou celles utilisées pour la détection des clones, afin d'étudier leur efficacité pour atteindre nos objectifs.

Finalement, afin d'étendre l'utilité de notre approche, nous comptons extraire le code architectural sous forme de patrons de génération afin de pouvoir régénérer un code équivalent et ainsi permettre l'approche circulaire du génie logiciel. Le fait de supporter d'autres langages de programmation serait aussi un apport appréciable, car ce n'est qu'une partie des systèmes qui sont écrits en Java. Ils poseront certainement d'autres problématiques tout en nous faisant voir de nouvelles pistes de solutions.

APPENDICE A

LE CODE POUR L'EXEMPLE D'ILLUSTRATION DE L'APPROCHE

Nous voyons dans cette section le code de 28 classes analysées dans l'exemple d'illustration de l'approche. Comme expliqué au chapitre 5 ce qui nous intéresse d'abord c'est le nom des classes. Dans code on peut voir qu'il y a les classes d'affaires : `Account`, `BankTransaction`, `CheckingAccount`, `Customer`, `Deposit`, `SavingAccount` et `Withdrawal`. Qui se retrouvent implantées dans ce code par des classes reliées dont une contient seulement le nom et dont les autres contiennent le nom de l'objet d'affaires accolé à un des mots architecturaux suivants : `Bean`, `Home` et `WebFacade`. Les déclarations de variables ou de méthode suivie du commentaire « // D'affaires » représentent ceux conservés dans le modèle final.

ACCOUNT.JAVA

```
package Test;

import javax.ejb.EJBObject;

public interface Account extends EJBObject
{
}
```

ACCOUNTBEAN.JAVA

```
package Test;

import java.math.BigDecimal;
import java.util.Calendar;

import javax.ejb.CreateException;
import javax.ejb.EntityBean;
import javax.ejb.EntityContext;

public class AccountBean implements javax.ejb.EntityBean
{
    private String accountNumber; // D'affaires
    private BigDecimal balance; // D'affaires
    private Customer customer; // D'affaires
    private Calendar openDate; // D'affaires
    transient protected EntityContext entityContext;

    public void CalculateInterest() // D'affaires
    {
    }

    public void ejbActivate()
    {
    }

    public void ejbCreate()
    {
    }
```

```
    public void ejbLoad()
    {
    }

    public void ejbPassivate()
    {
    }

    public void ejbPostCreate() throws javax.ejb.CreateException
    {
    }

    public void ejbRemove()
    {
    }
    public void ejbStore()
    {
    }

    public String getAccountNumber()
    {
        return accountNumber;
    }

    public BigDecimal getBalance()
    {
        return balance;
    }

    public Customer getCustomer()
    {
        return customer;
    }

    public Calendar getOpenDate()
    {
        return openDate;
    }

    public void setAccountNumber( String aAccountNumber )
    {
        this.accountNumber = aAccountNumber;
    }

    public void setBalance( BigDecimal aBalance )
    {
        this.balance = aBalance;
    }

    public void setEntityContext( javax.ejb.EntityContext entityContext )
    {
        this.entityContext = entityContext;
    }

    public void setCustomer( Customer aCustomer )
    {
        this.customer = aCustomer;
    }

    public void setOpenDate( Calendar aOpenDate )
    {
        this.openDate = aOpenDate;
    }
}
```

ACCOUNTHOME.JAVA

```
package Test;

import java.math.BigDecimal;
import java.util.Calendar;
import javax.ejb.EJBHome;
import javax.ejb.EJBObject;

public interface AccountHome extends EJBHome
{
    EJBObject create( String accountNumber, BigDecimal balance, Calendar openDate );
}
```

ACCOUNTWEBFACADE

```
package Test;

import javax.ejb.EJBObject;

public interface AccountWebFacade
{
}
```

BANKTRANSACTION.JAVA

```
package Test;

import javax.ejb.EJBObject;

public interface BankTransaction extends EJBObject
{
}
```

BANKTRANSACTIONBEAN.JAVA

```
package Test;

import java.math.BigDecimal;
import java.sql.Timestamp;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

public class BankTransactionBean implements javax.ejb.SessionBean
{
    private Account account; // D'affaires
    private BigDecimal amount; // D'affaires
    private Timestamp dateTime; // D'affaires
    private String transactionNumber; // D'affaires
    transient protected SessionContext sessionContext;

    public void ejbActivate()
    {
    }

    public void ejbCreate()
    {
    }

    public void ejbLoad()
    {
```

```
    }

    public void ejbPassivate()
    {
    }

    public void ejbRemove()
    {
    }
    public void ejbStore()
    {
    }

    public Account getAccount()
    {
        return account;
    }

    public BigDecimal getAmount()
    {
        return amount;
    }

    public Timestamp getDateTime()
    {
        return dateTime;
    }

    public String getTransactionNumber()
    {
        return transactionNumber;
    }

    public void setAccount( Account aAccount )
    {
        account = aAccount;
    }

    public void setAmount( BigDecimal aAmount )
    {
        amount = aAmount;
    }

    public void setDateTime( Timestamp aDateTime )
    {
        dateTime = aDateTime;
    }

    public void setTransactionNumber( String aTransactionNumber )
    {
        transactionNumber = aTransactionNumber;
    }

    public void setSessionContext( javax.ejb.SessionContext sessionContext )
    {
        this.sessionContext = sessionContext;
    }

    public void unsetSessionContext()
    {
        this.sessionContext = null;
    }
}
```

BANKTRANSACTIONHOME.JAVA

```
package Test;

import java.math.BigDecimal;
import java.sql.Timestamp;
import javax.ejb.EJBHome;
import javax.ejb.EJBObject;

public interface BankTransactionHome extends EJBHome
{
    EJBObject create( BigDecimal amount, Timestamp dateTime, String transactionNumber
);
}
```

BANKTRANSACTIONWEBFACADE

```
package Test;

import javax.ejb.EJBObject;

public interface BankTransactionWebFacade
{
}
```

CHECKINGACCOUNT.JAVA

```
package Test;

public interface CheckingAccount extends Account
{
}
```

CHECKINGACCOUNTBEAN.JAVA

```
package Test;

import javax.ejb.CreateException;
import javax.ejb.EntityBean;
import javax.ejb.EntityContext;

public class CheckingAccountBean extends AccountBean implements javax.ejb.EntityBean
{
    transient protected EntityContext entityContext;

    public void ejbActivate()
    {
    }

    public void ejbCreate()
    {
    }

    public void ejbLoad()
    {
    }

    public void ejbPassivate()
    {
    }

    public void ejbPostCreate() throws javax.ejb.CreateExcpetion
```

```
    {
    }

    public void ejbRemove()
    {
    }
    public void ejbStore()
    {
    }

    public void setEntityContext( javax.ejb.EntityContext entityContext )
    {
        this.entityContext = entityContext;
    }

    public void unsetEntityContext()
    {
        this.entityContext = null;
    }
}
```

CHECKINGACCOUNTHOME.JAVA

```
package Test;

import javax.ejb.EJBObject;

public interface CheckingAccountHome extends AccountHome
{
    EJBObject create();
}
```

CHECKINGACCOUNTWEBFACADE

```
package Test;

import javax.ejb.EJBObject;

public interface CheckingAccountWebFacade
{
}
```

CUSTOMER.JAVA

```
package Test;

import javax.ejb.EJBObject;

public interface Customer extends EJBObject
{
}
```

CUSTOMERBEAN.JAVA

```
package Test;

import java.util.Calendar;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

public class CustomerBean implements javax.ejb.SessionBean
```

```
{
    private String customerNumber; // D'affaires
    private String homePhone; // D'affaires
    private String name; // D'affaires
    transient protected SessionContext sessionContext;

    public void ejbActivate()
    {
    }

    public void ejbCreate()
    {
    }

    public void ejbLoad()
    {
    }

    public void ejbPassivate()
    {
    }

    public void ejbRemove()
    {
    }
    public void ejbStore()
    {
    }

    public String getCustomerNumber()
    {
        return customerNumber;
    }

    public String getHomePhone()
    {
        return homePhone;
    }

    public String getName()
    {
        return name;
    }

    public void setCustomerNumber( String aCustomerNumber )
    {
        customerNumber = aCustomerNumber;
    }

    public void setHomePhone( String aHomePhone )
    {
        homePhone = aHomePhone;
    }


    public void setName( String aName )
    {
        name = aName;
    }

    public void setSessionContext( javax.ejb.SessionContext sessionContext )
    {
        this.sessionContext = sessionContext;
    }
```

```
    public void unsetSessionContext()
    {
        this.sessionContext = null;
    }
}
```

CUSTOMERHOME.JAVA

```
package Test;

import java.util.Calendar;
import javax.ejb.EJBHome;
import javax.ejb.EJBObject;

public interface CustomerHome extends EJBHome
{
    EJBObject create( Calendar birthDate, String customerNumber,
        String homePhone, String name, Calendar openDate, String sex);
}
```

CUSTOMERWEBFACADE

```
package Test;

import javax.ejb.EJBObject;

public interface CustomerWebFacade
{
}
```

DEPOSIT.JAVA

```
package Test;

public interface Deposit extends BankTransaction
{
}
```

DEPOSITBEAN.JAVA

```
package Test;

import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

public class DepositBean extends BankTransactionBean implements javax.ejb.SessionBean
{
    transient protected SessionContext sessionContext;

    public void ejbActivate()
    {
    }

    public void ejbCreate()
    {
    }

    public void ejbLoad()
    {
    }
```

```
    public void ejbPassivate()
    {
    }

    public void ejbRemove()
    {
    }
    public void ejbStore()
    {
    }

    public void setSessionContext( javax.ejb.SessionContext sessionContext )
    {
        this.sessionContext = sessionContext;
    }

    public void unsetSessionContext()
    {
        this.sessionContext = null;
    }
}
```

DEPOSITHOME.JAVA

```
package Test;

import javax.ejb.EJBObject;

public interface DepositHome extends BankTransactionHome
{
    EJBObject create();
}
```

DEPOSITWEBFACADE

```
package Test;

public interface DepositWebFacade
{
}
```

SAVINGACCOUNT.JAVA

```
package Test;

public interface SavingAccount extends Account
{
}
```

SAVINGACCOUNTBEAN.JAVA

```
package Test;

import javax.ejb.CreateException;
import javax.ejb.EntityBean;
import javax.ejb.EntityContext;

public class SavingAccountBean extends AccountBean implements javax.ejb.EntityBean
{
    transient protected EntityContext entityContext;
```

```
    public void ejbActivate()
    {
    }

    public void ejbCreate()
    {
    }

    public void ejbLoad()
    {
    }

    public void ejbPassivate()
    {
    }

    public void ejbPostCreate() throws javax.ejb.CreateExcpetion
    {
    }

    public void ejbRemove()
    {
    }
    public void ejbStore()
    {
    }

    public void setEntityContext( javax.ejb.EntityContext entityContext )
    {
        this.entityContext = entityContext;
    }

    public void unsetEntityContext()
    {
        this.entityContext = null;
    }
}
```

SAVINGACCOUNTHOME.JAVA

```
package Test;

import javax.ejb.EJBObject;

public interface SavingAccountHome extends AccountHome
{
    EJBObject create();
}
```

SAVINGACCOUNTWEBFACADE

```
package Test;

public interface SavingAccountWebFacade
{
}
```

WITHDRAWAL.JAVA

```
package Test;

public interface Withdrawal extends BankTransaction
```

```
{
}
```

WITHDRAWALBEAN.JAVA

```
package Test;

import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

public class WithdrawalBean extends BankTransactionBean implements
javax.ejb.SessionBean
{
    transient protected SessionContext sessionContext;

    public void ejbActivate()
    {
    }

    public void ejbCreate()
    {
    }

    public void ejbLoad()
    {
    }

    public void ejbPassivate()
    {
    }

    public void ejbRemove()
    {
    }
    public void ejbStore()
    {
    }

    public void setSessionContext( javax.ejb.SessionContext sessionContext )
    {
        this.sessionContext = sessionContext;
    }

    public void unsetSessionContext()
    {
        this.sessionContext = null;
    }
}
```

WITHDRAWALHOME.JAVA

```
package Test;

import javax.ejb.EJBObject;

public interface WithdrawalHome extends BankTransactionHome
{
    EJBObject create();
}
```

WITHDRAWALWEBFACADE

```
package Test;

public interface WithdrawalWebFacade
{
}
```

APPENDICE B

PSEUDO-CODE

Dans ce qui suit, nous allons présenter une description plus détaillée de l'application des règles en pseudo-code, en nous inspirant de l'application de l'algorithme que nous avons implanté en Java.

ÉTAPES GLOBALES DE L'ALGORITHME

Nous voyons d'abord que la structure globale de l'algorithme suit les étapes énoncées à la section 5.5. Elle reçoit en paramètre une liste de classes extraites de l'arbre syntaxique et en retourne une autre contenant les classes d'affaires identifiées. Donc, le code commence par extraire la liste des combinaisons de patrons présentes dans le nom des classes d'affaires. Ensuite, il extrait les mots architecturaux. Il se sert de ces mots pour extraire les classes d'affaires, classes pour lesquelles il va finalement extraire les attributs, les méthodes, leurs relations et leurs héritages.

```
extractBusinessLogic(IN classList, OUT businessLogic)

 patternList           := extractPatterns(classList)

 architecturalWordList := extractArchitecturalWords(patternList,
  classList)

 businessClassList     :=
  retrieveBusinessClasses(architecturalWordList, classList)

 retrieveAttributesForBusinessClasses(architecturalWordList,
  classList, businessClassList)

 retrieveMethodsForBusinessClasses(architecturalWordList, classList,
  businessClassList)

 retrieveRelationsForBusinessClasses(classList, businessClassList)

END extractBusinessLogic
```

EXTRAIRE LES PATRONS

La première étape effectuée consiste donc à découper les patrons contenus dans le nom des identificateurs des classes. La fonction `extractPattern` analyse chacune des classes qui lui sont passées en argument, pour en séparer les patrons que leur nom contient. Les différents patrons présents sont délimités par des majuscules et le

caractère « _ ». Un patron doit avoir au moins une longueur de deux caractères et nous ajoutons toutes les combinaisons de patrons présentes dans la classe en cours.

```
extractPatterns(IN classList)

 patternList := empty Vector
 aPattern    := empty String
 /* Chaque nom de classe est analysé lettre par lettre pour identifier les patrons qui le
  composent */
 FOR each className in classList DO
   FOR i:=1 to length(className) DO
     IF   className.at(i) is the last          or
          className.at(i)= '_'                 or
          (('A'<=className.at(i)<='Z')         and
          (not('A'<=className.at(i-1)<='Z')    or
           not('A'<=className.at(i+1)<='Z'))) THEN
       IF i = length (className) THEN
         aPattern += className.at(i)
       ENDIF

       IF aPattern.length>1 THEN
         add aPattern to patternList if it does not exist
       ENDIF

       /* Ajouter les combinaisons de patron présentes dans le nom de la classe à la
        liste des patrons */
       FOR each bPattern previously found in className DO
         IF bPattern is ended by previousPattern THEN
           add bPattern + aPattern to patternList
         ENDIF
       ENDFOR

 previousPattern = aPattern
       aPattern := empty String
     ENDIF
     aPattern += className.at(i)
   ENDFOR
 ENDFOR
 RETURN patternList
 END extractPatterns
```

EXTRAIRE LES MOTS ARCHITECTURAUX

La seconde étape consiste à trouver les mots architecturaux présents dans le code. Elle consiste à appliquer les règles 1, 2, 3a, 3b et 3c. Elle reçoit en paramètre la liste de classes du modèle original et une liste de patrons et retourne une liste de mots architecturaux.

```
extractArchitecturalWords(IN patternList, IN classList)
 /* Règle 1 : chaque patron isolé qui n'est pas un nom de classe est placé dans une
  liste de patrons candidats (seulement une fois incluant la forme singulière ou
  plurielle.) */
 FOR each pattern in patternList DO
  relatedPattern := get plural or singular of pattern
  IF pattern and relatedPattern does not exist in classList
  THEN
   add pattern in candidatArchitecturalWordList
  ENDIF
 ENDFOR

 /* Nous examinons chaque mot architectural candidat dans le but de déterminer
  ceux qui seront conservés */
 FOR each architecturalWord in candidatArchitecturalWordList DO
   relatedClassList := get all classes in classList that has
    architecturalWord as a pattern

   /* Règle 2 : nous identifions pour chaque mot architectural candidat une liste de
    classes dont le nom le contient. Si cette liste contient au moins deux classes dont
    le nom ne contient pas d'autres mots architecturaux candidat alors le mot
    architectural candidat est conservé. */
   FOR each relatedClass in relatedClassList DO
    IF relatedClass has as pattern another architecturalWord in
     candidatArchitecturalWordList
    THEN
     remove relatedClass from relatedClassList
    ENDIF
   ENDFOR
   IF relatedClassList.size >= 2 THEN
    add architecturalWord in architecturalWordList
   ENDIF
   relatedClassList := get all classes in classList that has
    architecturalWord as a pattern

   /* Règle 3a : Si un mot architectural candidat possède un groupe relié avec au
    moins 2 classes qui ont une ou plusieurs méthodes principales qui sont les
    mêmes alors le mot architectural candidat est ajouté. */
   IF relatedClassList.size >= 2                                    and
      allClassesHavePrincipalMethods     (relatedClassList)  and
      allClassesHaveSamePrincipalMethods(relatedClassList)  and
   THEN
     add architecturalWord in architecturalWordList
   ENDIF
```

```
    /* Règle 3b : Si un mot architectural candidat possède un groupe relié, suivant les
     critères d'implantation d'interfaces ou d'héritage de classes abstraites avec au
     moins 2 classes qui ont une ou plusieurs méthodes principales qui sont les
     mêmes alors le mot architectural candidat est ajouté. */
    parentClassList := get all classes in classList implemented or
     inherited by a class or an interface in relatedClassList
    FOR each parentClass in parentClassList
     IF a class in relatedClassList implement or inherit parentClass
     THEN
      parentRelatedClassList := get all classes in relatedClassList
       which implement or inherit parentClass
      IF parentRelatedClassList.size >= 2
       and
         allClassesHavePrincipalMethods(parentRelatedClassList)
       and
         allClassesHaveSamePrincipalMethods(parentRelatedClassList)
       and
      THEN
       add architecturalWord in architecturalWordList
      ENDIF
     ENDIF
    ENDFOR

    /* Règle 3c : Si un mot architectural candidat, possède un groupe relié, suivant le
     critère d'appartenance à un espace de noms avec au moins 2 classes qui ont une
     ou plusieurs méthodes principales qui sont les  mêmes alors le mot architectural
     candidat est ajouté. */
    pkgList := get all packages in application
    FOR each pkg in pkgList
     IF a class in relatedClassList is in pkg
     THEN
      pkgRelatedClassList := get all classes in relatedClassList in
       pkg
      IF pkgRelatedClassList.size >= 2                              and
         allClassesHavePrincipalMethods(pkgRelatedClassList)        and
         allClassesHaveSamePrincipalMethods(pkgRelatedClassList)
      THEN
        add architecturalWord in architecturalWordList
      ENDIF
     ENDIF
    ENDFOR

  ENDFOR
  RETURN architecturalWordList
END extractArchitecturalWords.
```

EXTRAIRE LES CLASSES D'AFFAIRES

La troisième étape consiste à déterminer les classes d'affaires que contient l'application. Le résultat est obtenu en enlevant du nom de chaque classe du modèle original les mots architecturaux déterminés à l'étape précédente. La fonction reçoit donc en entrée la liste des mots architecturaux et la liste des classes du modèle original et retourne la liste des classes d'affaires.

```
retrieveBusinessClasses(IN architecturalWordList, IN classList)

 /* Règle4 : Chaque nom des classes du modèle original est examiné pour en retirer
   les mots architecturaux qu'il pourrait contenir. Ce nom servira ensuite à désigner
   une classe d'affaires qui sera ajoutée à la liste des classes d'affaires. */
 FOR each class in classList DO
  FOR each architecturalWord in architecturalWordList DO
   IF class name contains architecturalWord THEN
    remove architecturalWord from class name
   ENDIF
  ENDFOR
  add className to classList
 ENDFOR

 RETURN classList
END retrieveBusinessClasses
```

EXTRAIRE LES ATTRIBUTS DES CLASSES

Après avoir extrait et identifié les classes d'affaires, l'étape suivante consiste à identifier leurs attributs. La fonction reçoit donc en entrée la liste des mots architecturaux, la liste des classes originales et la liste des classes du modèle d'affaires auquel elle rajoute des attributs.

```
retrieveAttributesForBusinessClasses(IN   architecturalWordList,   IN
 classList , IN/OUT businessClassList)
BEGIN
  /* Chaque attribut de chaque classe reliée à une classe d'affaires est ajouté à celle-
   ci dans un premier temps. */
 FOR each businessClass in businessClassList DO
  relatedClassList := get all classes in classList that has
   businessClass as a pattern
  FOR each relatedClass in relatedClassList DO
    candidateAttributeList = attributes of relatedClass
    FOR each candidateAttribute of candidateAttributeList DO
     add candidateAttribute in businessClass attributeList
```

```
      /* Règle 5a : Si nous trouvons un groupe relié à un mot architectural dont les
       classes contiennent toutes cet attribut, il est enlevé de la classe d'affaires. */
      FOR each architecturalWord in architecturalWordList
       architecturalWordListRelatedClassList := get all classes in
        classList that has architecturalWord as a pattern
       IF architecturalWordListRelatedClassList.size >= 2 and

         allClassesHaveAttribute(architecturalWordListRelatedClassList
         , candidateAttribute)
       THEN
        remove candidateAttribute from businessClass attributeList
       ENDIF
      ENDFOR

      /* Règle 5b : Si nous trouvons un groupe relié suivant les critères
       d'implantation d'interfaces ou d'héritage de classes abstraites dont les classes
       contiennent toutes cet attribut, il est enlevé de la classe d'affaires. */
      parentClassList := get all classes in classList implemented or
       inherited by a class or an interface in relatedClassList
      FOR each parentClass in parentClassList
       parentRelatedClassList := get all classes in classList which
        implement or inherit parentClass
       IF parentRelatedClassList.size >= 2 and
        allClassesHaveAttribute(parentRelatedClassList,
        candidateAttribute)
       THEN
         remove candidateAttribute from businessClass attributeList
       ENDIF
      ENDFOR

      /* Règle 5c : Si nous trouvons un groupe relié suivant le critère d'appartenance
       à un espace de noms dont les classes contiennent toutes cet attribut, il est
       enlevé de la classe d'affaires. */
      pkgList := get all package in application
      FOR each pkg in pkgList
        pkgRelatedClassList := get all classes in classList in pkg
       IF pkgRelatedClassList.size >= 2 and
        allClassesHaveAttribute(pkgRelatedClassList,
        candidateAttribute)
       THEN
         remove candidateAttribute from businessClass attributeList
       ENDIF
      ENDFOR
     ENDFOR
   ENDFOR
END retrieveAttributesForBusinessClasses.
```

EXTRAIRE LES OPÉRATIONS DES CLASSES

Après avoir extrait et identifié les attributs des classes d'affaires, l'étape suivante consiste à identifier les méthodes. La fonction reçoit donc en entrée la liste des mots architecturaux, la liste des classes originales et la liste des classes du modèle d'affaires auquel elle rajoute des méthodes.

```
retrieveMethodsForBusinessClasses(IN architecturalWordList, IN
 classList , IN/OUT businessClassList)
BEGIN
 /* Chaque méthode de chaque classe reliée à une classe d'affaires est ajoutée à celle-
   ci dans un premier temps. */
 FOR each businessClass in businessClassList DO
 relatedClassList := get all classes in classList that has
  businessClass as a pattern
  FOR each relatedClass in relatedClassList DO
    candidateMethodList = Methods of relatedClass
    FOR each candidateMethod of candidateMethodList DO
     add candidateMethod in businessClass methodList

     /* Règle 6a : Si nous trouvons un groupe relié à un mot architectural dont les
      classes contiennent cette méthode, elle est enlevée de la classe d'affaires. */
     FOR each architecturalWord in architecturalWordList
       architecturalWordListRelatedClassList := get all classes in
        classList that has architecturalWord as a pattern
      IF architecturalWordListRelatedClassList.size >= 2 and
       allClassesHaveMethod(architecturalWordListRelatedClassList,
        candidateMethod)
      THEN
        remove candidateMethod from businessClass methodList
      ENDIF
     ENDFOR

     /* Règle 5b : Si nous trouvons un groupe relié suivant les critères
      d'implantation d'interfaces ou d'héritage de classes abstraites dont les classes
      contiennent toutes cette méthode, elle est enlevée de la classe d'affaires. */
     parentClassList := get all classes in classList implemented or
      inherited by a class or an interface in relatedClassList
     FOR each parentClass in parentClassList
       parentRelatedClassList := get all classes in classList which
        implement or inherits parentClass
      IF parentRelatedClassList.size >= 2 and
       allClassesHaveMethod(parentRelatedClassList, candidateMethod)
      THEN
        remove candidateMethod from businessClass methodList
      ENDIF
    ENDIF
```

```
    ENDFOR

    /* Règle 6c : Si nous trouvons un groupe relié suivant le critère d'appartenance à
      un espace de noms dont les classes contiennent toute cette méthode, elle est
      enlevée de la classe d'affaires. */
    pkgList := get all package in application
    FOR each pkg in pkgList
      pkgRelatedClassList := get all classes in classList which is in
       pkg
      IF pkgRelatedClassList.size >= 2 and
       allClassesHaveMethod(pkgRelatedClassList, candidateMethod)
      THEN
        remove candidateMethod from businessClass methodList
      ENDIF
    ENDFOR

    /* Règle 7, si la méthode contient le nom d'un attribut ou d'une classe reliée à la
      classe d'affaires, la méthode est retirée. */
    IF candidateMethod has as a pattern an attribute of businessClass
     or the name of a relatedClass
    THEN
     remove candidateMethod from businessClass methodList
    ENDIF
  ENDFOR
END retrieveMethodsForBusinessClasses
```

EXTRAIRE LES RELATIONS ENTRE LES CLASSES

La dernière étape consiste à extraire les relations entre les classes d'affaires découvertes. La fonction reçoit donc en entrée la liste des classes originales et la liste des classes du modèle d'affaires auquel elle rajoute des relations.

```
retrieveRelationsForBusinessClasses (IN classList , IN/OUT
 businessClassList)
 /* Pour chaque classe d'affaires. */
 FOR each businessClass in businessClassList DO
  /* Règle 8 : Si une classe d'affaires possède des attributs qui est du type d'une
    autre classe d'affaires nous la remplaçons par une relation. */
  IF businessClass has an attribute which is an other businessClass
  THEN
    remove attribute from businessClass attributeList
    add association between the businessClasses
  ENDIF

  /* Règle 9 : Si une classe d'affaires possède une classe reliée qui hérite d'une
    classe d'un groupe reliée à une autre classe d'affaire, la relation est ajoutée entre
    les classes d'affaires. */
```

```
  IF businessClass has a related class which inherit from an other
   businessClass related class
  THEN
    add inheritance association between the businessClasses
  ENDIF

  /* Règle 10 : Si une classe d'affaires a une interface reliée qui hérite d'une autre
   interface reliée à une autre classe d'affaires, la relation est ajoutée entre les classes
   d'affaires.
  IF businessClass has an related interface which inherit an other
   businessClass related interface
  THEN
    add inheritance association between the businessClasses
  ENDIF
 ENDFOR
END retrieveMethodsForBusinessClasses
```

APPENDICE C

AUTRES SYSTÈMES ANALYSÉS

JEDIT

Jedit (Pestov, Gellene et Ezust) est un éditeur de texte pour programmeur écrit en Java.

Résultats

Tableau C.1
Résultats obtenus pour l'analyse du système Jedit

Nombre de lignes de code	26562
Nombre de méthodes	2666
Nombre d'attributs	211
Nombre de classes en entrée	229
Nombre de classes conservées	211
Nombre de mots architecturaux détectés	9
Nombre de mots architecturaux estimés	79
Précision en %	55,56
Couverture en %	6,33
Nombre de classes d'affaires estimées	145
Nombre de classes d'affaires détectées	218
Précision en %	14,22
Couverture en %	21,38

Analyse

Nous voyons d'abord dans le tableau C.1 que pour la détection des mots architecturaux, une couverture de 6,33%. En effet, nous ne trouvons que 9 mots architecturaux alors que plusieurs dizaines sont évaluées. La cause est que plusieurs mots architecturaux sont aussi le nom de classes du modèle original, certains sont le nom de classes et de mots architecturaux et d'autres ne sont présents qu'une fois. La précision est de 55,56 % pour les mots architecturaux, car 4 des neuf mots architecturaux sont de faux négatifs. `BufferChangeAdapter` et `BufferChangeListener` entraînent la détection de `BufferChange` car ils ont les mêmes méthodes propres. `Fold` est détecté comme mot architectural, car `org.gjt.sp.jedit.buffer.FoldHandler` est implanté par `DummyFoldHandler`

`ExplicitFoldHandleret` et `IndentFoldHandler` qui ont la même méthode pure. `Recent` est détecté comme mot architectural, car `org.gjt.sp.jedit.menu` est implanté par : `RecentDirectoriesProvider` et `RecentFilesProvider`.

Étant donné qu'il nous manque autant de mots architecturaux, c'est normal, que la couverture et la précision pour les classes d'affaires soient limitées. La couverture est aussi limitée par le fait que certains mots architecturaux sont détectés comme mots architecturaux et inversement. La précision est limitée, notamment car des classes reliées à une classe d'affaires deviennent aussi une classe d'affaires.

ARGOUML

ArgoUML (Ramirez *et al.*, 2006) est un outil de modélisation UML très utilisé surtout au sein de la communauté universitaire. Le code source testé est celui de la version 0.18.1.

Résultats

Tableau C.2
Résultats obtenus pour le système ArgoUML

Nombre de lignes de code	57192
Nombre de méthodes	8966
Nombre d'attributs	2897
Nombre de classes en entrée	1210
Nombre de classes conservées	744
Nombre de mots architecturaux détectés	415
Nombre d'échantillons de mots architecturaux	14
Couverture en %	35,71

Analyse

Comme il y a un grand nombre de classes, nous avons sélectionné un sous-ensemble de mots architecturaux présents dans l'application et avons vérifié s'ils sont détectés. Pour les 14 mots du sous-échantillon, seulement 5 ont été détectés. Ce qui nous donne une couverture de 35,71% comme nous pouvons le constater au tableau C.2.

Les raisons pour lesquelles ces neuf mots architecturaux n'ont pas été détectés même s'ils se retrouvent relativement souvent, sont nombreuses. D'abord, ils n'ont pas deux classes de base les implantant. Parfois, ils sont une classe de base. Parfois, ils sont inclus dans le nom d'un autre mot architectural ou ils sont le nom d'une classe du modèle original. Les classes dont ils sont un mot architectural n'ont pas les mêmes méthodes propres ou même aucune. Celles implantant ou héritant d'une même classe ou interface qui y sont reliées sont uniques ou n'ont pas les mêmes méthodes propres. Elles sont donc indétectables avec nos règles.

COMPIERE

Résultats

Tableau C.3
Résultats obtenus pour le système Compiere

Nombre de lignes de code	86013
Nombre de méthodes	15615
Nombre d'attributs	682
Nombre de classes en entrée	1078
Nombre de classes conservées	657
Nombre de mots architecturaux détectés	194
Nombre d'échantillons de mots architecturaux	20
Couverture en %	55,00

Compiere (ComPiere, 2006) est un progiciel de gestion intégrée et de gestion de la relation client pour les PME dans la distribution et le service.

Analyse

Comme il y a un grand nombre de classes, nous avons sélectionné encore une fois un sous-ensemble de mot architectural présent dans l'application et avons vérifié s'ils sont détectés. Comme illustré au tableau C.3, 11 des 20 mots architecturaux ont été

trouvés, pour une couverture de 55 %. Les raisons pour les mots architecturaux non trouvés sont semblables.

NETBEANS

NetBean est un environnement de développement de la compagnie Sun Microsystems (Boudreau, Glick et Spurlin, 2002).

Résultats

Tableau C.4
Résultats obtenus pour le système NetBeans

Nombre de lignes de code	812801
Nombre de méthodes	128856
Nombre d'attributs	44503
Nombre de classes en entrée	13262
Nombre de classes conservées	6062
Nombre de mots architecturaux détectés	2384
Nombre d'échantillons de mots architecturaux	24
Couverture en %	25,00

Analyse

Encore une fois, comme il y a un grand nombre de classes, nous avons sélectionné un sous-ensemble de mots architecturaux présents dans l'application et avons vérifié s'ils sont détectés. Ce qui nous a donné une couverture de 55 % comme le montre le tableau C.4. S'il y a moins de classes en sorties, c'est que beaucoup de classes au même nom sont regroupées.

ECLIPSE

Eclipse (D'Anjou *et al.*, 2004) est un environnement de développement de la compagnie IBM.

Résultats

Tableau C.5
Résultats obtenus pour le système Eclipse

Nombre de lignes de code	694792
Nombre de méthodes	116568
Nombre d'attributs	49229
Nombre de classes en entrée	11049
Nombre de classes conservées	4979
Nombre de mots architecturaux détectés	1704
Nombre d'échantillons de mots architecturaux	43
Couverture en %	58,14

Analyse

Comme il y a un grand nombre de classes, nous avons aussi sélectionné un sous-ensemble de mots architecturaux présents dans l'application et avons vérifié s'ils sont détectés. Ceci nous donne une couverture de 58.14 % comme illustré dans le tableau C.5.

APPENDICE D

SOMMAIRE DES RÉSULTATS

Tableau D.1
Sommaire des résultats

	Test1	Generated	PetShop	OFBS	VSM	Jedit	ArgoUML	Compiere	NetBean	Eclipse
Nombre de classes en entrée	28	29	188	75	90	229	1210	1078	13262	11049
Nombre de classes conservées	7	8	134	58	63	211	744	657	6062	4979
Nombre de lignes de code	158	1091	6150	3082	3198	26562	57192	86013	812801	694792
Nombre de méthodes	88	207	1253	665	511	2666	8966	15615	128856	116568
Nombre d'attributs	45	53	628	132	149	211	2897	682	44503	49229
Nombre de mots architecturaux	3	6	*18	*22	*12	*79				
Nombre de mots architecturaux détectés	3	11	23	19	7	9	415	194	2384	1704
Nombre d'échantillon de mots arch.							14	20	24	43
Nombre de faux positifs	0	6	19	13	0	4				
Nombre de faux négatifs	0	1	14	16	5	74	9	9	18	18
Précision en %	100,00	45,45	17,39	31,58	100,00	55,56				
Couverture en %	100,00	83,33	22,22	27,27	58,33	6,33	35,71	55,00	25,00	58,14
Nombre de classes d'affaires	7	4	*45	*33	*17	*145				
Nombre de classes d'affaires détectées	7	8	134	58	63	218				
Nombre de faux positifs	0	4	107	34	49	187				
Nombre de faux négatifs	0	0	18	9	3	114				
Précision en %	100,00	50,00	20,15	41,38	22,22	14,22				
Couverture en %	100,00	100,00	60,00	72,73	82,35	21,38				
Nombre d'opérations d'affaires	1	0	*72	*53	*19					
Nombre d'opérations d'affaires détectées	1	2	68	52	13					
Nombre de faux positifs	1	2	18	0	10					
Nombre de faux négatifs	1	0	22	1	16					
Précision en %	0,00	0,00	73,53	100,00	*23,08					
Couverture en %	0,00	100,00	69,44	98,11	15,79					
Nombre d'attributs d'affaires	9	26	*177	*62	*29					
Nombre d'attributs d'affaires détectés	9	30	176	62	27					
Nombre de faux positifs	0	6	15	2	0					
Nombre de faux négatifs	0	2	16	2	2					
Précision en %	100,00	80,00	91,48	96,77	100,00					
Couverture en %	100,00	92,31	90,96	96,77	93,10					
Nombre d'héritages	4	0	*4	*2	*3					
Nombre d'héritages détectés	4	0	4	3	3					
Nombre de faux positifs	0	0	4	3	3					
Nombre de faux négatifs	0	0	4	2	3					
Précision en %	100,00	100,00	0,00	0,00	0,00					
Couverture en %	100,00	100,00	0,00	0,00	0,00					
Nombre d'associations	2	3	*7	*0	*5					
Nombre d'associations détectées	2	2	32	1	7					
Nombre de faux positifs	0	2	32	1	6					
Nombre de faux négatifs	0	3	7	0	4					
Précision en %	100,00	0,00	0,00	0,00	14,29					
Couverture en %	100,00	0,00	0,00	100,00	20,00					

BIBLIOGRAPHIE

Abd-El-Hafiz, S. K., et V. R. Basili. 1993. «Documenting Programs Using a Library of Tree Structured Plans». In *Proceedings of the Conference on Software Maintenance* (septembre): IEEE Computer Society Press.

Abd-El-Hafiz, S. K., et V. R. Basili. 1996. «A Knowledge-Based Approach to the Analysis of Loops». *IEEE Transactions on Software Engineering*. vol. 22, no 5, p. 339-360.

Aiken, P., A. Muntz et R. Richards. 1994. «DoD legacy systems: reverse engineering data requirements». *Communications of the ACM*. vol. 37, no 5, p. 26-41.

Albrecht. , A. . 1979. «Measuring application development productivity.». In *Proceedings of the IBM Applications Development Symposium* (octobre): IBM Press.

Alexander, C., S. Ishikawa et M. Silverstein. 1977. *A Pattern Language: Towns, Buildings, and Construction*: New York: Oxford University Press.

Allen, Robert, et David Garlan. 1994. «Beyond denition/use: Architectural interconnection». In *Proceedings of the ACM Interface Definition Language Workshop* (août): SIGPLAN Notice.

Allen, Robert, et David Garlan. 1997. «A formal basis for architectural connection». *ACM Transactions on Software Engineering and Methodology*. vol. 6, no 3, p. 213-249.

Anquetil, N., et T. C. Lethbridge. 1997. «File clustering using naming conventions for legacy systems». In *Proceedings of the 1997 Conference of the Centre For Advanced Studies on Collaborative Research* (novembre): IBM Press.

Anquetil, N., et T. C. Lethbridge. 1998. «Extracting concepts from file names: a new file clustering criterion». In *Proceedings of the 20th international conference on Software engineering* (avril): IEEE Computer Society Press.

Anquetil, N., et T. C. Lethbridge. 1999. «Recovering software architecture from the names of source files». *Journal of Software Maintenance: Research And Practice*. vol. 11, no 3, p. 201-221.

Antoniol, Giuliano, Gerardo Canfora, Gerardo Casazza et A. DeLucia. 2002. «Recovering Traceability Links between Code and Documentation». *IEEE Transactions on Software Engineering*. vol. 28, no 10, p. 970-983.

April, A., E. Merlo et A. Abran. 1997. «A Reverse Engineering Approach to Evaluate Function Point Rules». In *Proceedings of the Fourth Working Conference on Reverse Engineering* (octobre): IEEE Computer Society Press.

Archer, Tom, et Andrew Whitechapel. 2002. *Inside C#,Second Edition*: Microsoft Press, 848 p.

Balmas, F. 1997. «Toward a framework for conceptual and formal outlines of programs». In *Proceedings of the Fourth Working Conference on Reverse Engineering* (octobre): IEEE Computer Society Press.

Bass, L., M. Klein et F. Bachmann (2000). Quality Attribute Design Primitives. Pittsburgh, PA, USA, Carnegie Mellon University

Bass, Len , Mark Klein et Rick Kazman. 2001. «Architecture Mechanisms». En ligne. <http://www.sei.cmu.edu/news-at-sei/columns/the_architect/2001/1q01/architect-1q01.htm> Consulté le 16 août 2007.

Bass, Len, Paul Clements et Rick Kazman. 2003. *Software architecture in practice : second edition*: Addison-Wesley Longman Publishing Co., Inc., 480 p.

Baxter, I. D., A. Yahin, L. Moura, M. Sant'Anna et L. Bier. 1998. «Clone Detection Using Abstract Syntax Trees». In *Proceedings of the International Conference on Software Maintenance* (novembre): IEEE Computer Society Press.

Bhansali, S., et H.P. Nii. 1992. «Software design by reusing architectures». In *Proceedings of the Seventh Knowledge-Based Software Engineering Conference* (septembre): IEEE Computer Society Press.

Biggerstaff, T. J. 1989. «Design recovery for maintenance and reuse». *Computer*. vol. 22, no 7, p. 36-49.

Biggerstaff, T. J., B. G. Mitbander et D. E. Webster. 1993. «The concept assignment problem in program understanding». In *Proceedings of the 15th international conference on Software Engineering* (mai): IEEE Computer Society Press.

Booch, G., J. Rumbaugh et I. Jacobson. 1999. *The Unified Modeling Language User Guide*: Addison Wesley Longman Publishing Co., Inc., 482 p.

Booch, Grady. 1995. *Object solutions: managing the object-oriented project*: Addison Wesley Longman Publishing Co., Inc., 323 p.

Booch, Grady. 2004. *Object-Oriented Analysis and Design with Applications*. Redwood City: The Benjamin/Cummings Publ. Comp., 704 p.

Borland, Software Corporation. 2005. *User Guide for Borland Together Architect 1.1*: Borland Software Corporation. En ligne. <http://www.borland.com/together>. Consulté le 8 janvier 2007.

Boudreau, Tim, Jesse Glick et Vaughn Spurlin. 2002. *NetBeans: The Definitive Guide*: O'Reilly & Associates, Inc., 672 p.

Bowen, J. P., P. T. Breuer et K. C. Lano. 1993. «Formal specifications in software maintenance: From code to Z++ and back again». *Information and Software Technology*. vol. 35, no 11/12, p. 679-690.

Bowman, I., M. Wgodfrey et R. C. Holt. 1999. «Extracting Source Models from Java Programs: Parse, Disassemble, or Profile?». En ligne. <http://plg.uwaterloo.ca/~migod/papers> Consulté le 15 août 2007.

Brown, A. W., et K. C. Wallnau. 1996. «A Framework for Evaluating Software Technology». *IEEE Software*. vol. 13, no 5, p. 39–49.

Brown, W. J., R. C. Malveau et H. W. McCormick. 1998. *AntiPatterns: Refactoring Software, Architectures and Projects in Crisis*: John Wiley & Sons, 336 p.

Bruce, Tate, Clark Mike et Linskey Patrick. 2003. *Bitter EJB*: Manning Publications Co., 440 p.

Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad et Michael Stal. 1996. *Pattern-oriented software architecture: a system of patterns*: John Wiley & Sons, Inc., 457 p.

Buss, E., R. De Mori, M. Gentleman, J. Henshaw, H. A. Johnson, K. Kontogiannis, E. Merlo, H. A. Müller, J. Mylopoulos, S. Paul, A. Prakash, M. Stanley, S. R. Tilley, J. Troster et K. Wong. 1994. «Investigating Reverse Engineering Technologies for the CAS Program Understanding Project». *IBM Systems Journal*. vol. 33, no 3, p. 477-500.

Caprile, B., et P. Tonella. 1999. «Nomen Est Omen: Analyzing the Language of Function Identifiers». In *Proceedings of the Sixth Working Conference on Reverse Engineering* (octobre): IEEE Computer Society Press.

Caprile, B., et P. Tonella. 2000. «Restructuring program identifier names». In *Proceedings of the international Conference on Software Maintenance (Icsm'00)* (octobre): IEEE Computer Society Press.

Chen, Yih-Farn , M. Y. Nishimoto et C. V. Ramamoorthy. 1990. «The C Information Abstraction System». *IEEE Transactions on Software Engineering*. vol. 16, no 3, p. 325-334.

Chikofsky, Elliot J. 1990. «Reverse engineering and design recovery: A taxonomy». *IEEE Software*. vol. 7, no 1, p. 13-17.

Claudio, R., et J. V. Rodriguez. 2002. «Combining Static and Dynamic Views for Architecture Reconstruction». In *Proceedings of the Sixth European Conference on Conference on Software Maintenance and Reengineering* (mars): IEEE Computer Society Press.

Clements, Paul, David Garlan, Reed Little, Robert Nord et Judith Stafford. 2003. «Documenting software architectures: views and beyond». In *Proceedings of the 25th International Conference on Software Engineering* (mai): IEEE Computer Society Press.

Cohen, W. W. 1995. «Inductive specification recovery: Understanding software by learning from example behaviors». *Automated Software Engineering*. vol. 2, no 2, p. 107-129.

ComPiere, Inc. 2006. «ComPiere». ComPiere, Inc. En ligne. <http://www.compiere.org/> Consulté le 15 août 2007.

Cooper, David, Benjamin Khoo, Brian R. von Konsky et Michael Robey. 2004. «Java implementation verification using reverse engineering». In *Proceedings of the 27th Australasian conference on Computer science - Volume 26* (janvier): Australian Computer Society, Inc.

Crispen, Robert G., et Lynn D. Stuckey, Jr. 1994. «Structural model: architecture for software designers». In *Proceedings of the conference on TRI-Ada '94* (novembre): ACM Press.

D'Ambros, Marco, et Michele Lanza. 2006. «Reverse Engineering with Logical Coupling». In *Proceedings of the 13th Working Conference on Reverse Engineering* (octobre): IEEE Computer Society Press.

D'Anjou, Jim, Scott Fairbrother, Dan Kehn, John Kellerman et Pat McCarthy. 2004. *Java Developer's Guide to Eclipse, The (2nd Edition)*: Addison-Wesley Professional.

Desclaux, C., et M. Ribault. 1991. «Macs: Maintenance Assistance Capability for Software Maintenance». In *Proceedings. Conference on Software Maintenance* (octobre): IEEE Computer Society Press.

Di Lucca, G. A, A. R. Fasolino, P. Tramontana et U. De Carlini. 2003. «Abstracting business level UML diagrams from Web applications». In *Proceedings of the 5th IEEE International Workshop on Web Site Evolution* (septembre).

Dijkstra, Edsger W., et Carel S. Scholten. 1990. *Predicate calculus and program semantics*: Springer-Verlag New York, Inc., 220 p.

Doval, D., S. Mancoridis et B. S. Mitchell. 1999. «Automatic Clustering of Software Systems Using a Genetic Algorithm». In *Proceedings of Software Technology and Engineering Practice* (août - septembre): IEEE Computer Society Press.

Ducasse, S. 1999. «Evaluating message passing control techniques in Smalltalk». *Journal of Object-Oriented Programming*. vol. 12, no 16, p. 39-50.

Ducasse, S., et M. Lanza. 2005. «The class blueprint: Visually supporting the understanding of classes». *IEEE Transactions on Software Engineering*. vol. 31, no 1, p. 75-90.

Ducasse, S., M. Lanza et R. Bertuli. 2004. «High-Level Polymetric Views of Condensed Run-time Information». In *Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering* (mars): IEEE Computer Society Press.

Ducasse, S., M. Rieger et S. Demeyer. 1999. «A Language Independent Approach for Detecting Duplicated Code». In *Proceedings of the IEEE International Conference on Software Maintenance* (août - septembre): IEEE Computer Society Press.

Earls, A. B., S. M. Embury et N. H. Turner. 2002. «A method for the manual extraction of business rules from legacy source code». *BT Technology Journal*. vol. 20, no 4, p. 127-145.

Eilam, E. 2005. *Reversing: Secrets of Reverse Engineering*: John Wiley & Sons, 617 p.

EJD, Technologies. 2006. «SourceCafe». EJD Technologies En ligne. <http://www.sourcecafe.com> Consulté le 15 août 2007.

Feild, H., D. Binkley et D. Lawrie. 2006. «Identifier Splitting: A Study of Two Techniques». In *Proceedings of the 2006 Mid-Atlantic Student Workshop on Programming Languages and Systems* (avril).

Fiutem, R., P. Tonella, Giuliano Antoniol et E. Merlo. 1996. «A Cliché-Based Environment to Support Architectural Reverse Engineering». In *Proceedings of the Third Working Conference on Software Maintenance* (octobre): IEEE Computer Society Press.

Fowler, M. 1999. *Refactoring: Improving the Design of Existing Programs*: Addison-Wesley.

Gagnon, Etienne M., et Laurie J. Hendren. 1998. «SableCC, an Object-Oriented Compiler Framework». In *Proceedings of the Technology of Object-Oriented Languages and Systems* (août): IEEE Computer Society Press.

Gamma, Erich, Richard Helm, Ralph Johnson et John Vlissides. 1995. *Design patterns: elements of reusable object-oriented software*: Addison-Wesley Longman Publishing Co., Inc., 395 p.

Gannod, G. C., et B. H. C. Cheng. 1996. «Strongest postcondition semantics as the formal basis for reverse engineering». *Journal of Automated Software Engineering*. vol. 3, no 38749, p. 139–164.

Gannod, G. C., et B. H. C. Cheng. 1999. «A framework for classifying and comparing software reverse engineering and design recovery techniques». In *Proceedings of the Sixth Working Conference on Reverse Engineering* (octobre): IEEE Computer Society Press.

Grose, Timothy J. , Gary C. Doney et Stephen A. Brodsky. 2002. *Mastering XMI: Java Programming with XMI, XML, and UML*: Wiley, 480 p.

Guo, Jiang, Yuehong Liao et Raj Pamula. 2006. «Static Analysis Based Software Architecture Recovery». *Lectures notes in computer science*. vol. 3982, no 974-983 p. 974-983.

Hajitaheri, A., A. Gaffar et R. Kline. 2003. «Data Reverse Engineering: Overview, Recovery of Business Rules, and Representation of Application-Independent Data with XML». Concordia University. En ligne. <http://www.cs.concordia.ca/~comp691l/Data_Reverse_Engineering__Gaffar__Kline__et_al._.pdf> Consulté le 8 janvier 2007.

Hamou-Lhadj, A., E. Braun, D. Amyot et T. C. Lethbridge. 2005. «Recovering Behavioral Design Models from Execution Traces». In *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering* (mars): IEEE Computer Society Press.

Harris, D. R., H. B. Reubenstein et A. S. Yeh. 1995. «Recognizers for extracting architectural features from source code». In *Proceedings of the Second Working Conference on Reverse Engineering*: IEEE Computer Society.

Hayes-Roth, Barbara, Karl Pfleger, Philippe Lalanda, Philippe Morignot et Marko Balabanovic. 1995. «A Domain-Specific Software Architecture for Adaptive Intelligent Systems». *IEEE Trans. Softw. Eng.* vol. 21, no 4, p. 288-301.

Hayes-Roth, R. (1994). Architecture-based acquisition and development of software: Guidelines and recommendations from the ARPA domain-specific software architecture program. . Palo Alto, CA., Techknowledge Federal Systems

Heuzeroth, D., T. Holl, G. Högström et W. Löwe. 2003. «Automatic Design Pattern Detection». In *Proceedings of the 11th IEEE International Workshop on Program Comprehension* (mai): IEEE Computer Society Press.

Hoare, C. A. R. 1969. «An axiomatic basis for computer programming». *Commun. ACM*. vol. 12, no 10, p. 576-580.

Hofmeister, Christine, Robert Nord et Dilip Soni. 2000. *Applied software architecture*: Addison-Wesley Longman Publishing Co., Inc., 397 p.

Huang, H. , W. T. Tsai, S. Bhattacharya, X. P. Chen, Y. Wang et J. Sun. 1996. «Business Rule Extraction from Legacy Code». In *Proceedings of the 20th Conference on Computer Software and Applications*: IEEE Computer Society Press.

Huang, H., S. Zhang, J. Cao et Y. Duan. 2005. «A practical pattern recovery approach based on both structural and behavioral analysis». *Journal of Systems and Software*. vol. 75, no 1/2, p. 69-87.

Hung, Maokeng, et Ying Zou. 2005. «Extracting Business Processes from Three-Tier Architecture Systems». In *Proceedings of International Workshop on Reverse Engineering To Requirements* (novembre).

Hunt, John. 2003. *Guide to the Unified Process Featuring Uml, Java and Design Patterns*: Springer, 440 p.

IEEE (1990). IEEE Std 610.12-1990, Standard Glossary of Software Engineering Terminology, IEEE. IEEE Std 610.12-1990,

IEEE (2000). ANSI/IEEE Std 1471-2000 Recommended Practice for Architectural Description of Software-Intensive Systems

Jaber, Sami. 2002. «Le PetShop .NET : un anti-pattern d'architecture». En ligne. <http://www.dotnetguru.org/articles/Reflexion/PetShopArchitecture/PetShop Architecture.pdf#search=%22%20%20%20%22%20Le%20PetShop%20.NET %20%3A%20un%20anti-pattern%20d'architecture%20%22%22> Consulté le 12 octobre 2006.

Jackson, K., et M. Boasson. 1995. «Systems Engineering of Computer Based Systems». In *Proceedings of the 1995 International Symposium and Workshop on Publication* (mars): IEEE Computer Society Press.

Jahnke, J. H., W. Schäfer et A. Zundorf. 1997. «Generic fuzzy reasoning nets as a basis for reverse engineering relational database applications». In *Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering* (septembre): Springer.

Johnson, Ralph E. 1997. «Frameworks = (components + patterns)». *Communications of the ACM*. vol. 40, no 10, p. 39-42.

Kazman, R., et S. J. Carrière. 1999. «Playing Detective: Reconstructing Software Architecture from Available Evidence». *Automated Software Engineering*. vol. 6, no 2, p. 107-138.

Kazman, Rick, Paul Clements et Len Bass. 2003. «Software Architecture in Practice». p. 560 Addison-Wesley Professional.

Khriss, I., et Gino Chénard (2007). Extracting business rules from an object oriented application, Université du Québec à Rimouski

Kim Jung, J., et M. Benner Kevin. 1996. «Implementation patterns for the observer pattern». In *Pattern languages of program design,* p. 75-86: Addison-Wesley Longman Publishing Co. Inc.

Knoernschildn , Kirk 2001. *Java Design: Objects, UML, and Process*: Addison Wesley Professional, 304 p.

Kollman, R., P. Selonen, E. Stroulia, T. Systä et A. Zundorf. 2002. «A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering». In *Proceedings of the Ninth Working Conference on Reverse Engineering* (octobre - novembre): IEEE Computer Society Press.

Kontogiannis, K., R. Di Mori, M. Bernstein et E. Merlo. 1994. «Localization of Design Concepts in Legacy Systems». In *Proceedings of the International Conference onSoftware Maintenance* (septembre): IEEE Computer Society Press.

Kramer, C., et L. Prechelt. 1996. «Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software». In *Proceedings of the Third Working Conference on Reverse Engineering* (novembre): IEEE Computer Society Press.

Kruchten, P. 2000. *The Rational Unified Process -- An Introduction, 2nd ed*: Addison-Wesley-Longman, Reading, 320 p.

Kruchten, Phillipe, et Christopher J. Thompson. 1994. «An object-oriented, distributed architecture for large-scale Ada systems». In *Proceedings of the conference on TRI-Ada '94* (novembre): ACM Press.

Lanza, M., et S. Ducasse. 2003. «Polymetric Views - A Lightweight Visual Approach to Reverse Engineering». *IEEE Transactions on Software Engineering*. vol. 29, no 9, p. 782–795.

Layzell, P. J., M. J. Freeman et P. Benedusi. 1995. «Improving Reverse Engineering through the Use of Multiple Knowledge Sources». *Journal of Software Maintenance: Research And Practice*. vol. 7, no 4, p. 279-299.

Lenz, Gunther, et Christoph Wienands. 2006. *Practical Software Factories in .NET*: Apress.

Li, Y., H. Yang et W. Chu. 2000. «Clarity guided belief revision for domain knowledge recovery in legacy systems». In *Proceedings of 12th International*

Conference on Software Engineering and Knowledge Engineering (juillet): Knowledge Systems Institute.

Liu, K., A. Alderson et Z. Qureshi. 1999. «Requirements Recovery from Legacy Systems by Analysing and Modelling Behaviour». In *Proceedings. IEEE International Conference on Software Maintenance, 1999. (ICSM '99)* (août - septembre): IEEE Computer Society Press.

Lu, Chih-Wei, William C. Chu, Chih-Hung Chang, Yeh-Ching Chung, Xiaodong Liu et Hongji Yang. 2002. «Reverse Engineering». In *Software Engineering and Knowledge Engineering Hanbook,Volume 2.*

Magdalena, B., E. Merlo, M. Dagenais, B. Lague et K. Kontogiannis. 1999. «Partial Redesign of Java Software Systems Based on Clone Analysis». In *Proceedings of the Ninth Working Conference on Reverse Engineering (Wcre'02)* (octobre): IEEE Computer Society Press.

Magdalena, B., E. Merlo, M. Dagenais, B. Lague et K. Kontogiannis. 2000. «Advanced clone analysis to support object-oriented system refactoring». In *Proceedings of Seventh Working Conference on Reverse Engineering* (novembre): IEEE Computer Society Press.

Magee, Jeff, Naranker Dulay, Susan Eisenbach et Jeff Kramer. 1995. «Specifying Distributed Software Architectures». In *Proceedings of the 5th European Software Engineering Conference* (septembre): Springer-Verlag.

Mendelzon, A. O. (1993). Declarative Database Visualization: Recent Papers from the Hy+/Graphlog Project, Computer Systems Research Institute, University of Toronto

Mendelzon, A. O., et J. Sametinger. 1995. «Reverse engineering by visualizing and querying». *Software Concepts and Tools*. vol. 16, no 4, p. 170-182.

Mens, T., et T. Tourwé. 2004. «A Survey of Software Refactoring». *IEEE Transactions on Software Engineering*. vol. 30, no 2, p. 126-139.

Mitchell, B. S., S. Mancoridis et M. Traverso. 2002. «Search based reverse engineering». In *Proceedings of the 14th international conference on Software engineering and knowledge engineering* (juillet): ACM Press.

Moha, Naouel , et Yann-Gaël Guéhéneuc. 2005. «On the Automatic Detection and Correction of Software Architectural Defects in Object-Oriented Designs». In

Proceedings of the 6th ECOOP Workshop on Object-Oriented Reengineering (juillet).

Müller, H. A., J. H. Jahnke, D. B. Smith, M.-A.D. Storey, S. R. Tilley et K. Wong. 2000. «Reverse engineering: A roadmap». In *Proceedings of the 22nd International Conference on Software Engineering (ICSE'2000). Future of Software Engineering* (juin): ACM Press.

Müller, H. A., M. A. Orgun, S. R. Tilley et J. S. Uhl. 1993. «A reverse engineering approach to subsystem structure identification». *Journal of Software Maintenance: Research And Practice*. vol. 5, no 4, p. 181–204.

Nambiar, Rohit 2005. «Java Petstore a case study». Manhattan, Kansas, Department of Computing and Information Sciences, Kansas State University.

Neapolitan, R. E., et K. Naimipour. 2004. *Foundations of Algorithms Using C++ Pseudocode (3rd Ed.)*: Jones and Bartlett Publishers, Inc., 523 p.

Niere, J., J. P. Wadsack et A. Zündorf. 2001. «Recovering UML Diagrams from Java Code using Patterns». In *Proceedings of the 2nd Workshop on Software Computing Applied to Software Engineering* (février): Springer Verlag.

Ohba, M., et K. Gondow. 2005. «Toward mining "concept keywords" from identifiers in large software projects». In *Proceedings of the 2005 international workshop on Mining software repositories* (mai): ACM Press.

Oracle. 2006. «OTN Financial Brokerage Service 10g». Oracle. En ligne. <www.oracle.com/technology/sample_code/tech/java/j2ee/fbs10g/index.html > Consulté le 15 août 2007.

Oracle. 2006. «Virtual Shopping Mall 1.3 ». Oracle. En ligne. <http://www.oracle.com/technology/sample_code/tutorials/vsm1.3/over/design.htm> Consulté le 15 août 2007.

Paul, S., et A. Prakash. 1994. «A Framework for Source Code Search Using Program Patterns». *IEEE Transactions on Software Engineering*. vol. 20, no 6, p. 463-475.

Paul, S., et A. Prakash. 1994. «Supporting queries on source code: A formal framework.». *International Journal of Software Engineering and Knowledge Engineering*. vol. 4, no 3, p. 325-348.

Paul, S., A. Prakash, E. Buss et J. Henshaw. 1991. «Theories and techniques of program understanding». In *Proceedings of the 1991 conference of the Centre for Advanced Studies on Collaborative research* (octobre): IBM Press.

Pestov, Slava , John Gellene et Alan Ezust. 2006. «jEdit 4.3 User's Guide». En ligne. <http://www.jedit.org> Consulté le 15 août 2007.

Philippow, I., D. Streitferdt, M. Riebisch et S. Naumann. 2005. «An approach for reverse engineering of design patterns». *Software and Systems Modeling*. vol. 4, no 1, p. 55-70.

Pinali, D. O. 1999. «Design Pattern Extraction for Software Documentation». Vrije, Belgium, Computer Science Department, Universiteit Brussel.

Platt, David S. 2002. *Introducing Microsoft .Net, Second Edition*: Microsoft Press, 250 p.

Pressman, R. S. 2001. *Software Engineering: A Practitioner's Approach, (5th ed.)*: McGraw-Hill, 915 p.

Rajlich, V. 1997. «Incremental Redocumentation with Hypertext». In *First Euromicro Conference on Software Maintenance and Reengineering* (mars): IEEE Computer Society Press.

Ramirez, Alejandro, Philippe Vanpeperstraete, Andreas Rueckert, Kunle Odutola, Jeremy Bennett et Linus Tolke. 2006. «ArgoUML User Manual». En ligne. <http://argouml-stats.tigris.org/documentation/defaulthtml/cookbook/> Consulté le 15 août 2007.

Rich, Charles, et Linda M. Wills. 1990. «Recognizing a Program's Design: A Graph-Parsing Approach». *IEEE Software*. vol. 7, no 1, p. 82-89.

Richner, T., et S. Ducasse. 1999. «Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information». In *Proceedings of the IEEE International Conference on Software Maintenance* (août - septembre): IEEE Computer Society Press.

Riel, A.J. . 1996. *Object-Oriented Design Heuristics*: Addison Wesley.

Riva, Oriana, Cristiano Di Flora, Stefano Russo et Kimmo Raatikainen. 2006. «Unearthing Design Patterns to Support Context-Awareness». In *Proceedings of the Fourth Annual IEEE International Conference on Pervasive Computing and Communications Workshops* (mars): IEEE Computer Society Press.

Rugaber, S., K. Stirewalt et L. M. Wills. 1995. «The Interleaving Problem in Program understanding». In *Proceedings of 2nd Working Conference on Reverse Engineering* (juillet): IEEE Computer Society Press.

Rybin, Sergey, Alfred Strohmeier, Alexey Kuchumov et Vasiliy Fofanov. 1996. «ASIS for GNAT: From the Prototype to the Full Implementation». In *Proceedings of the 1996 Ada-Europe International Conference on Reliable Software Technologies* (juin): Springer-Verlag.

Sartipi, K. 2003. «Software Architecture Recovery based on Pattern Matching». School of Computer Science, University of Waterloo.

Shaw, M., et D. Garlan. 1996. *Software Architecture: Perspectives on an Emerging Discipline*: Prentice-Hall.

Shaw, Mary, Robert DeLine, V. Klein Daniel, L. Theodore, M. Young Ross David et Gregory Zelesnik. 1995. «Abstractions for Software Architecture and Tools to Support Them». *IEEE Trans. Softw. Eng.* vol. 21, no 4, p. 314-335.

Singh, Inderjeet, Beth Stearns et Mark Johnson. 2002. *Designing enterprise applications with the J2EE platform*: Addison-Wesley Longman Publishing Co., Inc., 417 p.

Smith, C. U. , et Lloyd G. Williams. 2003. «More New Software Antipatterns: Even More Ways to Shoot Yourself in the Foot». *CMG -CONFERENCE-*. vol. 2, p. 717-726.

Sneed, Harry M. , et Katalin Erdos. 1996. «Extracting Business Rules from Source Code». In *Proceedings of the 4th International Workshop on Program Comprehension (WPC '96)* (mars): IEEE Computer Society Press.

Storey, M.-A.D. 2005. «Theories, Methods and Tools Program Comprehension: Past, Present and Future». In *13th International Workshop on Program Comprehension* (mai): IEEE Computer Society.

Sun. «BluePrint». En ligne. <http://java.sun.com/blueprints/> Consulté le 15 août 2007.

Sun. 2006. «Java Petstore ». En ligne. <http://java.sun.com/j2ee//1.4/download.html#samples> Consulté le 9 janvier 2007.

Sward, R. E., et A. Chamillard. 2004. «AdaSlicer: an Ada program slicer». In *Proceedings of the 2003 annual ACM SIGAda international conference on Ada: the engineering of correct and reliable software for real-time and distributed systems using ada and related technologies* (décembre): ACM Press.

Systä, T. 1999. «On the Relationships between Static and Dynamic Models in Reverse Engineering Java Software». In *Proceedings of Sixth Working Conference on Reverse Engineering* (octobre): IEEE Computer Society Press.

Thiel, Steffen. 2005. «A framework to improve the architecture quality of software intensive systems». Duisburg, Essen, Univ. Duisburg-Essen, Wirtschaftswissenschaften.

Tilley, S. R. 2000. «The Canonical Activities of Reverse Engineering». *Annals of Software Engineering*. vol. 9, no 1-4, p. 249-271.

Tip, F. 1995. «A survey of program slicing techniques». *Journal of programming languages*. vol. 3, no 3, p. 121-189.

Tonella, P., et A. Potrich. 2005. *Reverse Engineering of Object Oriented Code*. Coll. «Monographs in Computer Science»: Springer.

Ward, M. P. 2004. «Pigs from sausages? Reengineering from assembler to C via FermaT transformations». *Sci. Comput. Program.* vol. 52, no 1-3, p. 213-255.

Ward, M. P., et K. H. Bennett. 1995. «Formal methods for legacy systems». *Journal of Software Maintenance: Research And Practice*. vol. 7, no 3, p. 203-219.

Wills, L. M. 1993. «Flexible Control for Program Recognition». In *Proceedings of Working Conference on Reverse Engineering, 1993.* (mai): IEEE Computer Society Press.

Withrow, C. 1990. «Error density and size in Ada software». *IEEE Software*. vol. 7, no 1, p. 26-30.

Wuyts, R. 1998. «Declarative Reasoning about the Structure of Object-Oriented Systems». In *Proceedings of the Technology of Object-Oriented Languages and Systems* (août): IEEE Computer Society Press.

Xinyu, Wang, Sun Jianling, Yang Xiaohu, He Zhijun et S. Maddineni. 2004. «Application of information-flow relations algorithm on extracting business

rules from legacy code». In *Fifth World Congress on Intelligent Control and Automation, 2004. WCICA 2004.* (juin).

Xu, B., J. Qian, X. Zhang, Z. Wu et L. Chen. 2005. «A brief survey of program slicing». *ACM SIGSOFT Software Engineering Notes.* vol. 30, no 2, p. 1-36.

Zou, Ying, Terence C. Lau, Kostas Kontogiannis, Tack Tong et Ross McKegney. 2004. «Model-Driven Business Process Recovery». In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04) - Volume 00* (novembre): IEEE Computer Society Press.