Université de Montréal

Détection des patrons de conception dans les systèmes orientés objet

par

Bouazza Bachar

Département d'informatique et de recherche opérationnelle

Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures

en vue de l'obtention du grade de

Maîtrise ès sciences (M.Sc.)

en informatique

Août 1998

Université de Montréal

Faculté des études supérieures

Ce mémoire intitulé

Détection des patrons de conception dans les systèmes orientés objet

présenté par

Bouazza Bachar

a été évalué par un jury composé des personnes suivantes :

François Lustman	:	Président
Claude Frasson	:	Membre
Rudolf K. Keller	:	Directeur de recherche

Mémoire accepté le : 99.02.01

Sommaire

La compréhension des programmes joue un rôle très important dans les activités de maintenance, de réutilisation et d'évolution des logiciels. Par exemple, la réutilisation, facilitée et supportée par la programmation orientée objet, est une activité presque quotidienne et indispensable. Les nouvelles techniques de réutilisation comme les cadres d'application et les patrons de conception ne font qu'accroître cette nécessité. L'unité de réutilisation n'est plus simplement la classe mais toute une structure bien définie ayant une sémantique et un comportement d'ensemble précis. Dans les systèmes d'envergure la présence de ces structures est éparpillée et dispersée partout dans le code. Des outils d'aide à la reconnaissance et à la détection de composantes ayant certaines propriétés deviennent de plus en plus une nécessité, pour assister le développeur et l'usager du système durant le développement et la maintenance. La détection des patrons est très utile, entre autres, pour la bonne conduite des activités de compréhension, de réutilisation, et de maintenance des programmes.

Ce travail de recherche apporte une solution à la problématique de la détection des patrons de conception dans des systèmes orientés objet. Ainsi, nous avons conçu et développé des détecteurs pour douze patrons de conception.

Contrairement aux approches classiques purement structurelles, les heuristiques que nous avons développées intègrent les trois aspects : structurel, sémantique et stylistique. En effet, une analyse sémantique exploite l'interaction et le couplage entre les composantes du patron. De plus, lorsque l'élément sémantique est difficile à détecter on se sert de l'approche stylistique pour prendre une décision. Ainsi, notre approche s'appuie sur la "troïka" : comportement/sémantique/style.

Ceci nous a permis de concevoir des heuristiques plus robustes pour la détection dont les efforts de validation sont plus réduits. Ces détecteurs ont été testés et validés sur des systèmes tels que *ACE*, *ET++*, *LEDA* et *xForms*.

Remerciements

Je tiens à présenter mes remerciements à,

Monsieur Rudolf K. Keller, professeur à l'Université de Montréal, pour m'avoir fait l'honneur de diriger ce mémoire au sein du laboratoire du Génie logiciel. Qu'il trouve ici l'expression de ma très grande gratitude pour la confiance qu'il m'a sans cesse témoigné durant ces deux années.

Monsieur Reinhard Schauer, Analyste à l'Université de Montréal, pour ses différentes suggestions et critiques qui m'étaient d'un grand intérêt.

Bouazza Bachar, le 04 Juillet 1998

Table des matières

1 Introduction **1**

1.1 Problématique 1

1.2 État de l'art 3

1.3 Contributions majeures 4

1.4 Projet SPOOL 5

1.5 Organisation du mémoire 6

2 Le paradigme orienté objet **7**

2.1 Historique 8

2.1.1 Début de la programmation 8

2.1.2 Structures de données 9

2.1.3 Programmation par objets 10

2.2 Concepts orientés objet 10

2.2.1 Encapsulation 11

2.2.2 Héritage 11

2.2.3 Agrégation et association 12

2.2.4 Polymorphisme et liaison dynamique 13

2.2.5 Délégation et composition 14

2.2.6 Interface 15

2.2.7 Notation graphique 16

2.2.8 Un exemple de langage orienté objet 20
2.3 Les types de logiciel 21
2.3.1 Application 21
2.3.2 Bibliothèque de fonctions 21
2.3.3 Bibliothèque de classes 23
2.3.4 Cadre d'application 24

3 Patrons de conception **26**
3.1 Définition 27
3.2 Exemple de patron de conception 29
3.3 Description des patrons de conception 32
3.4 Classification 34
3.5 Patron de conception et cadre d'application 37
3.6 Langages de patrons 39

4 Qualité et complexité des logiciels **40**
4.1 Qualité 40
4.2 Complexité 41
4.3 Cohésion et Couplage 44
4.3.1 Cohésion 44
4.3.2 Couplage 45
4.4 Évaluation de la qualité 45
4.5 Patrons de conception et qualité des logiciels 47

5 Détection des patrons **50**
5.1 État de l'art 50
5.2 Principes de détection 52

5.2.1 Généralités 52
5.2.2 Nécessité des heuristiques 54
5.3 Approches de détection des patrons 55
5.3.1 Approche basée sur l'analyse structurelle 55
5.3.2 Approche basée sur des styles 57
5.3.3 Notre approche 58

6 Techniques de détection développées pour SPOOL 71
6.1 L'outil d'analyse *Gen++* 71
6.1.1 Aperçu de *GENOA/GENII* 72
6.1.2 Utilisation de *Gen++* 73
6.2 Implantation des algorithmes 76
6.2.1 Implémentation du détecteur de *Factory Method* 77
6.2.2 Taille des analyseurs 83

7 Expérimentation et discussion 85
7.1 Systèmes de tests 85
7.1.1 *ET++* 86
7.1.2 ACE 87
7.1.3 LEDA 88
7.1.4 xForms 88
7.2 Analyse et interprétation des expériences 89
7.2.1 Mesures de performances des heuristiques 89
7.2.2 Résultats des expériences 91
7.3 Problèmes rencontrés 95

8 Conclusion 98

Bibliographie **101**

Table des figures

2.1 Classe abstraite et concrète 17
2.2 relation entre classes 18
2.3 Notation des diagrammes objets 18
2.4 Notation des diagrammes d'interaction 19
2.5 Une application 22
2.6 Une bibliothèque de fonctions 22
2.7 Une bibliothèque de classes 23
2.8 Un cadre d'application 25

3.1 Exemple du modèle MVC (extrait du livre de Gamma et al [GHJV94]) . . . 30
3.2 Le patron de conception Observer 31
3.3 Un diagramme d'interaction 32

5.1 Le patron de conception Abstract Factory 60
5.2 Le patron de conception Adapter 61
5.3 Le patron de conception Bridge 62
5.4 Le patron de conception Chain Of Responsability 63
5.5 Le patron de conception Composite 64
5.6 Le patron de conception Decorator 65
5.7 Le patron de conception Factory Method 66
5.8 Le patron de conception Mediator 66

5.9 Le patron de conception Observer 67
5.10 Le patron de conception Proxy 68
5.11 Le patron de conception Singleton 69
5.12 Le patron de conception Template Method 69

Liste des tableaux

3.1 Critères de classification des patrons de conception. 35

6.1 Taille des analyseurs (LOC dans le langage *Gen++*) 84

7.1 Taille des systèmes de tests 86

7.2 Résultats des expériences 91

7.3 Comparaison avec *PAT* 95

Chapitre 1

Introduction

1.1 Problématique

Les compagnies utilisatrices des systèmes de grande taille, comme les contrôleurs des commutateurs, les contrôleurs du trafic aérien, imposent à leurs constructeurs de logiciels des normes de qualité souvent très sévères. En effet, dans un environnement très changeant ces compagnies désirent savoir si leurs fournisseurs sont capables de répondre à leur demande dans des délais parfois limités. En d'autres termes, on veut s'assurer que les constructeurs de logiciels utilisent de bonnes techniques de développement et qu'ils nous livrent des systèmes flexibles, maintenables, simples, etc. Actuellement, on dispose d'une panoplie d'outils et de métriques pour vérifier et valider des systèmes d'envergure [KTLD97, LH93, Nav87, BBM96, HCN97, HS96, LH93, MLM96, FMvW97]. Ces outils et ces métriques ont été développés au cours de l'évolution de l'informatique et des paradigmes de programmation.

Le paradigme orienté objet est de plus en plus utilisé pour l'analyse, la conception et l'implantation des systèmes logiciels de grande taille [JCJO92, JEJ94, BCC+96]. Certes, l'approche orientée objet semble apporter des solutions aux problèmes de complexité, de maintenance, de réutilisation etc., qui sont engendrés intrinsèquement par la taille des systèmes d'envergure [Boo94]. Mais l'utilisation de l'approche orientée objet comme fondation

conceptuelle ne permet pas spontanément la résolution des problèmes de la qualité. En d'autres termes, la qualité des systèmes est une activité d'ingénierie qu'il faut maîtriser et soutenir et non une conséquence d'utilisation d'un paradigme en particulier. L'intervention de l'intelligence humaine soutenue par des outils d'aide à la conception, à la maintenance et à la compréhension restent très importants sinon indispensables.

Récemment, la technique de conception par patron a été proposée pour alléger le fardeau de la conception et réduire le temps de cette phase qui constitue le subconscient du système futur [BMR+96, TK96, GHJV94, AIS+77, And94, BCC+96, Coa92, CS95, SS95a, GHJV93, Gam92]. Elle consiste à réutiliser et à encapsuler l'expertise des concepteurs rodés dans le domaine. Cette technique de conception et d'implantation constitue une progression en avant vers une standardisation et une normalisation de l'architecture des systèmes, via des composantes connues, testées et approuvées. Celles-ci promettent de faciliter, au moins, la maintenance, la compréhension et l'utilisation des systèmes. Cependant, dans un système d'envergure, la répartition des patrons de conception est éparse dans le code. On veut reconnaître la présence de ces patrons et localiser leur position pour les manipuler. Cela nous permet, entre autres, de :

- comprendre le système en étudiant les patrons qui le composent et ainsi réduire la complexité apparente du système. De plus, les patrons de conception sont mieux compris et enseignés en regardant des exemples d'implantation dans des systèmes réels. Généralement, la détection des patrons de conception dans un code est un bon indicateur de la maturité du développeur du système.
- maintenir le système. Les patrons architecturaux, par exemple, nous renseignent sur la structure globale du système. Cela nous aide à localiser les modules à maintenir. En effet, même les composantes d'un patron peuvent être éparpillées dans le système et la détection du patron nous informe sur tous les lieux où il faut apporter un changement

sans oubli.

- utiliser le système et ré-utiliser ses composantes. La plupart des systèmes actuels sont documentés par des patrons. La reconnaissance des patrons qui forment un système facilite son utilisation.

1.2 État de l'art

Le problème de la détection et de la reconnaissance de structures a été bien établi dans le domaine de la réingénierie il y a bien longtemps. Cette problématique a été élaborée par le besoin de synthétiser une conception à partir d'une implantation. Avec l'introduction des patrons de conception dans le domaine du génie logiciel, une problématique semblable se pose : celle de la détection de ces patrons. Très peu de travaux concluants ont été conduits dans ce domaine; nous voulons contribuer par la recherche de moyens automatiques de reconnaissance et de détection des patrons de conception dans un code orienté objet.

À l'état de l'art actuel, il n'y a eu que quelques tentatives pour concevoir et réaliser des algorithmes pour détecter des patrons de conception. Alberto Mendezon et Johannes Sametinger [MS95] ont développé des heuristiques basées sur des conventions de nom et des styles. Christian Krämer et Lutz Prechelt [KP96] se sont concentrés sur la détection des patrons du type structurel, selon la classification de Gamma et al. Ils ont développé des heuristiques pour détecter les patrons *Adapter, Bridge, Composite, Decorator, et Proxy.* Dans sa thèse, Kyle Brown [Bro97] a donné des heuristiques pour détecter des patrons dans du code écrit en Smalltalk. Brown n'est pas allé plus loin que Krämer et al. par rapport à la technique de la détection. Son approche se base aussi sur une analyse purement structurelle. Il a pu détecter seulement quelques patrons. Brenda S. Baker [Bak95] donne une technique pour la détection des duplications de code dans un système. Cette technique peut être utilisée

pour développer des heuristiques de détection des patrons. Cependant, les heuristiques qui en découlent restent purement structurelles, et on revient aux mêmes lacunes que celles qu'on trouve chez Krämer et al. et chez Brown. Néanmoins, sa technique de détection des duplications est d'une grande valeur par rapport à celles qui se basent sur des styles et des conventions.

1.3 Contributions majeures

Les contributions majeures de ce travail résident dans :

- La conception des heuristiques pour la détection des douze patrons de conception : *Abstract Factory, Adapter, Bridge, Chain of Responsability, Composite, Decorator, Factory Method, Mediator, Observer, Proxy, Singleton*, et *Template Method.*
- Implantation des heuristiques conçues en forme de "détecteurs" supportant le langage *C++* en utilisant le système *Gen++* [Dev92, DE94].
- La validation et les tests de nos détecteurs sur des systèmes de taille moyenne : *ACE, ET++, LEDA*, et *xForms.*

Contrairement aux approches purement structurelles mentionnées en haut, les heuristiques que nous avons développées intègrent les trois aspects : structurel, sémantique et stylistique. Ceci nous a permis de concevoir des heuristiques plus robustes pour la détection dont les efforts de validation sont réduits.

1.4 Projet SPOOL

Notre travail de recherche s'insère dans le cadre du projet "SPOOL"[1]. Ce projet a été défini par le groupe "GELO" (Génie logiciel) de l'Université de Montréal et est supporté par l'organisme CSER avec le partenaire industriel Bell Canada. Le projet SPOOL est organisé en quatre sous projets (A,B,C,D) :

A : Décomposition de systèmes basée sur le couplage et la cohésion,

B : Relation entre la conception de systèmes et leurs évolutions,

C : Intégration de la conception orientée objet de haut niveau et de bas niveau,

D : Patrons de conception spécifiques aux domaines.

Mon implication dans le projet *SPOOL* se situe au niveau du sous-projet (D). Dans ce projet, nous avons conçu et développé un outil graphique de détection et de présentation des patrons de conception. L'architecture de l'outil comporte des modules de base de données, une interface usager graphique et un module de détection des patrons de conception. Ce dernier module étant ma contribution personnelle. L'interface graphique est écrite avec le cadre d'application "JKit/Go", à son tour implémenté dans le langage Java. Pour le stockage des patrons de conception, une base de donnée orientée objet a été utilisée. Cette base de donnée contient des informations qui caractérisent les patrons de conception ainsi que les instances représentées par l'interface usager. Les détecteurs des patrons sont écrits dans le langage de spécification *GENOA*, qui fait partie du système *Gen++* [Dev92, DE94].

[1]SPOOL est l'acronyme de Spreading Desirable Properties into the Design of Object-Oriented Large-Scale Software.

1.5 Organisation du mémoire

Dans le prochain chapitre, nous allons résumer le paradigme orienté objet, la fondation conceptuelle sur laquelle se basent les chapitres subséquents. Dans le troisième chapitre, nous introduirons la notion des patrons de conception. Le quatrième chapitre sera consacré à la discussion de la qualité des logiciels et son rapport avec la conception orientée objet. Le cinquième chapitre présente la problématique de la détection des patrons de conception et discute l'ensemble de principes et d'approches sous-jacent à nos techniques de détection. Dans le sixième chapitre, nous parlerons de l'implantation des algorithmes et des heuristiques de détection que nous avons développés pour le projet SPOOL. Dans le chapitre sept, nous parlerons des systèmes de test que nous avons utilisés pour valider nos travaux, et nous discuterons les performances de nos détecteurs par rapport à d'autres détecteurs décrits dans la littérature. Finalement, une conclusion et discussion des travaux futurs seront présentés.

Chapitre 2

Le paradigme orienté objet

L'approche orientée objet est une technique d'analyse, de conception et d'implantation de systèmes. Cette approche propose une nouvelle façon de penser les problèmes en utilisant des modèles organisés autour des concepts du monde réel [RBP+91]. Dans ce paradigme [1], le concept fondamental est l'objet, qui combine en lui à la fois les données et les opérations, contrairement à la programmation conventionnelle où les données et les opérations sont indépendantes. De plus, l'objet constitue l'unité minimale de décomposition, qui est souvent la même que l'unité de perception du réel, contrairement aux approches conventionnelles où l'unité de décomposition est la procédure.

Cependant, l'approche orientée objet est une évolution, et non une révolution, manifestée par la maturation du domaine de l'informatique. Elle est proposée essentiellement pour remédier aux problèmes rencontrés avec l'utilisation des approches classiques. On cherche une approche permettant la maîtrise de la complexité des systèmes et la construction de systèmes évolutifs, faciles à maintenir et réutilisables. L'approche orientée objet paraît prometteuse pour promouvoir ces attributs de la qualité [Boo94, RBP+91, JCJO92].

[1] Selon Thomas Kuhn [Kuh86] un paradigme désigne un ensemble de convictions partagées par une communauté professionnelle. Il contient un nombre limité d'assertions qui, par leur application, ordonnent et légitiment les résultats ultérieurs.

Ce chapitre présente les préceptes de base du monde orienté objet. Il introduit également les distinctions entre une application, une bibliothèque de classe (toolkit) et un cadre d'application (framework).

2.1 Historique

Le paradigme orienté objet n'est pas apparu subitement de nulle part. Il est en fait le prolongement logique des développements qui se sont faits dans le monde de l'informatique au niveau des techniques de programmation. Dans cette section nous allons parler des origines et de l'historique du paradigme orienté objet.

2.1.1 Début de la programmation

Dans les débuts de l'informatique, les techniques de programmation ne consistaient guère plus qu'en une suite d'instructions qui étaient envoyées directement au processeur. On parlait alors de programmation en langage machine ou assembleur. Très rapidement, on s'est rendu compte que certaines suites revenaient très souvent. On les a donc regroupées dans une entité qui est devenue la procédure, également appelée une fonction. Plutôt que de reproduire la série d'instructions lorsqu'on en a besoin, on place un appel à la procédure. Ceci a donné lieu à l'apparition des langages procéduraux tel que Fortran.

Cette transformation des méthodes de programmation comporte trois avantages sérieux : une diminution de la taille des programmes, une meilleure gestion du code et la réutilisation du code. Le remplacement d'une série complexe d'instructions par un simple appel de procédure, reproduit à plusieurs reprises dans un programme, en diminue grandement la taille et la complexité. De plus, des erreurs peuvent survenir lors de la copie d'une séquence. Avec le mécanisme de procédure, comme tous les appels font référence à la même définition, il ne

reste plus qu'à s'assurer que celle-ci est correcte. La compréhension du programme est grandement améliorée car plusieurs détails du traitement peuvent être cachés dans la procédure. Finalement, il devient facile de réutiliser cette section de code. Il suffit de copier la procédure dans un nouveau programme et de faire des appels à la procédure.

La réutilisation fréquente de certaines procédures a amené la création des bibliothèques de procédures. Une telle bibliothèque peut regrouper un grand nombre de procédures, le plus souvent adaptées à un domaine particulier, comme le traitement mathématique ou les entrées/sorties.

2.1.2 Structures de données

L'idée de l'agrégation des instructions en procédures a ensuite été appliquée aux données du programme. On a regroupé les données qui étaient apparentées en structures indivisibles. Dès lors, une donnée n'est plus significative uniquement par elle-même, mais fait partie d'un contexte déterminé par les autres données qui lui sont associées. Si la procédure est un regroupement d'instructions liées entre elles, on parlera d'une structure de données pour décrire un ensemble de données liées entre elles. Le concepteur d'un programme doit alors non seulement porter attention aux liens entre les étapes du traitement des données, mais également aux liens entre les données elles-mêmes.

La définition d'une structure de données introduit un nouveau type de données pour le programme. Ce mécanisme, en plus de bien définir la complexité du programme, aide encore une fois à diminuer la taille de celui-ci et à mieux en gérer le code. La centralisation de la définition de la structure évite les erreurs de duplication, en plus de faciliter la modification de la structure.

À cela s'est ajouté une meilleure définition des procédures, avec l'accent mis sur le contrôle du flot d'exécution grâce à l'utilisation d'énoncés conditionnels et de boucles. Les procédures

sont alors plus linéaires et ont des points d'entrée et de sortie bien définis. Ceci a donné lieu à l'apparition de langages dits structurés, comme Pascal.

2.1.3 Programmation par objets

L'étape suivante consiste à regrouper les structures de données avec les procédures qui les concernent. On parle alors d'un objet, qui consiste en une structure de données et des procédures qui peuvent lui être appliquées, formant ainsi un tout indivisible. Les champs de l'objet sont souvent appelés attributs et ses procédures en sont les méthodes. Les objets sont caractérisés à l'aide du concept de classe, qui représente à toutes fins pratiques le type de l'objet. On dit souvent d'un objet qu'il est une instance de sa classe. Le premier langage à introduire le concept d'objets et de classes a été Simula.

Comme pour les procédures, on peut concevoir des bibliothèques de classes. Celles-ci permettent la réutilisation de classes entre les programmes.

2.2 Concepts orientés objet

Les concepts de classe et d'objet ne suffisent cependant pas à décrire le paradigme orienté objet. Dans les langages dits traditionnels, la prépondérance est normalement donnée aux procédures. Un programme consiste en un agencement de procédures qui se passent des données les unes aux autres. Dans le monde orienté objet, les rôles sont inversés. Le programme est vu comme un agencement d'objets, de structures, sur lequel un certain nombre d'opérations peuvent êtres effectuées. Ce changement introduit une demande pour certains autres concepts qui vont êtres introduits en bas. Ensuite, une notation graphique sera présentée, et les concepts seront illustrés dans un langage orienté objet.

2.2.1 Encapsulation

Un objet est similaire à une valeur dans un type de donné abstrait. Il renferme des données et des opérations. L'encapsulation est le mécanisme par lequel le développeur cache une partie de l'information pour préserver l'intégrité de l'objet. Ainsi, il établit une séparation entre une interface et une implémentation. Pour ce faire, le paradigme objet fournit des outils de support de l'encapsulation, par exemple des clauses de visibilité permettant de protéger un attribut.

Ce mécanisme permet, d'une part, d'utiliser un objet à l'aide de son interface, et d'autre part de modifier l'objet sans affecter ses utilisateurs. L'objet est alors vu comme une boîte noire dont le fonctionnement est caché à l'usager. Grâce à l'encapsulation, l'utilisateur d'un objet n'a pas à se préoccuper des détails de l'implantation. De plus, le programmeur est libre quant à son choix d'implantation.

2.2.2 Héritage

La relation d'héritage constitue la deuxième propriété qui distingue les langages orientés objet des autres langages. C'est le procédé par lequel une classe, dite sous-classe, reçoit une partie de sa définition d'une autre classe, dite super-classe.

L'héritage est un mécanisme d'abstraction permettant d'organiser les classes dans des hiérarchies de spécialisation [Boo94, RBP+91]. De plus, il permet la réutilisation des classes en définissant une nouvelle classe à partir d'une classe déjà existante. On dit de la nouvelle classe qu'elle est un enfant de l'autre classe, qui est de son côté son parent. De ce fait, l'enfant hérite de toutes les propriétés du parent, y-inclus ses attributs et ses méthodes. L'enfant peut modifier les méthodes ainsi obtenues et également ajouter ses propres attributs et méthodes. La relation d'héritage supporte un style de programmation appelé programmation

par différence, où le programmeur définit une nouvelle classe en héritant d'une autre classe déjà existante et ensuite décrit la différence entre l'ancienne et la nouvelle classe.

La relation d'héritage permet de définir un système de façon incrémentielle, chaque génération de classes ajoutant une nouvelle série de fonctionnalités à la précédente. Elle facilite également l'encapsulation en mettant l'interface dans une classe parent et en laissant à ses enfants la responsabilité de l'implantation. Elle permet, aussi, de localiser les changements apportés à une classe.

La relation d'héritage peut être soit simple, où chaque enfant possède un seul parent, soit multiple, où chaque enfant peut avoir plus d'un parent et ainsi combiner les propriétés de plusieurs classes.

2.2.3 Agrégation et association

Selon James Rumbaugh [RBP+91] le concept de lien est une connexion physique ou conceptuelle entre deux objets. Un objet collabore avec d'autres objets à travers ces liens. Une association est alors une abstraction d'un ensemble de liens entre des objets de la même classe. Une association est une relation entre deux classes. Souvent, elle est implantée par des pointeurs.

Une relation d'agrégation se définit comme une relation de composé à composant. Cette relation est, en effet, un cas particulier de la relation d'association. Tandis qu'une association, ou lien, dénote une relation client-client ou client-serveur, une agrégation dénote une hiérarchie composé/composant, avec la possibilité de naviguer du composé au composant.

Contrairement à la relation d'héritage, qui est uniquement définie entre classes, la relation d'agrégation est définie entre deux classes, entre une classe et un objet, ou entre deux objets.

La plupart des langages de programmation, entre autres *C++* et *Java*, ne supportent pas directement cette relation, comme ils le font pour la relation d'héritage. En fait cette

relation est seulement modélisée, ou simulée, par d'autres constructeurs du langage.

2.2.4 Polymorphisme et liaison dynamique

Le polymorphisme est un concept fondamental dans le paradigme orienté objet. Il vient du fait qu'un objet combine, en lui, le type de tous ses ancêtres. Il peut donc être utilisé partout où un objet d'une classe parent est attendu. L'objet peut se faire passer pour une instance d'une autre classe, en autant que cette autre classe fasse partie de la liste de ses ancêtres.

Parallèlement au polymorphisme, on retrouve la liaison dynamique. Elle est le mécanisme par lequel le choix des méthodes à activer s'effectue non pas statiquement lors de la compilation, mais dynamiquement lors de l'exécution du programme. Celle-ci consiste à vérifier le type d'un objet de façon dynamique, lors de l'exécution, et d'activer les méthodes en fonction du type identifié.

La liaison dynamique a comme effet que le lancement d'une requête n'impose pas d'avoir à se conformer à une implémentation particulière avant l'exécution. Il est par conséquent possible d'écrire des programmes qui escomptent un objet à l'interface particulière, sachant que tout objet doté de l'interface adéquate acceptera la requête. Mieux encore, lors de l'exécution, la liaison dynamique permet de substituer des objets, les uns aux autres, pourvu que leurs interfaces soient identiques. Elle permet à un objet client de faire sur un autre objet peu d'hypothèses au-delà du service d'une interface particulière. Le polymorphisme simplifie la définition des clients, il découple les objets les uns des autres et leur permet de faire varier leurs relations les uns aux autres lors même de l'exécution.

Le polymorphisme et la liaison dynamique permettent de renforcer les conditions de l'encapsulation en cachant le fonctionnement de l'objet à son utilisateur. Celui-ci n'utilise qu'une classe parent qui sert à définir l'interface d'utilisation. L'implantation est reléguée aux

classes enfants de cette dernière. À l'exécution, une instance d'une classe enfant est passée à l'utilisateur par polymorphisme et est utilisée à travers les liens dynamiques. La séparation de l'interface et de l'implantation est complète.

2.2.5 Délégation et composition

Les deux techniques les plus courantes pour la réutilisation des fonctionnalités dans les systèmes orientés objet sont l'héritage de classes et la composition d'objets. Ainsi, comme nous l'avons expliqué, l'héritage de classes permet de définir l'implémentation d'une classe à partir de l'implémentation d'autres classes. Ce type de réutilisation est appelé dans la littérature réutilisation boîte blanche. Le terme boîte blanche fait référence à la visibilité : avec l'héritage, le contenu des classes parentes est généralement visible aux sous-classes.

La composition d'objets est une solution alternative à l'héritage de classe. Dans ce cas, on obtient de nouvelles fonctionnalités par assemblage ou composition d'objets pour en construire de plus complexes. La composition d'objets impose que les objets entrant dans la composition aient des interfaces bien définies. Ce style de réutilisation est appelé réutilisation boîte noire, du fait qu'aucun détail interne des objets n'est visible. Les objets apparaissent simplement comme des boîtes noires.

L'héritage et la composition ont chacun leurs avantages et leurs inconvénients. L'héritage de classe est défini de manière statique à la compilation et son utilisation est immédiate, du fait que le langage de programmation en assure directement le support. De plus, l'héritage de classe permet de modifier facilement le code réutilisé.

Mais l'héritage de classe a aussi des inconvénients. D'abord, il n'est pas possible de modifier pendant l'exécution le code hérité des parents, puisque l'héritage est défini à la compilation. Ensuite, et plus grave, les classes parentes définissent souvent au moins une partie de la représentation physique de leurs sous-classes. Du fait que l'héritage assujettit une

sous-classe à des détails de l'implémentation de ses parents, on dit fréquemment que l'héritage rompt l'encapsulation. Le code d'une sous-classe devient à ce point lié au développement de ses parents, de façon que tout changement de l'implémentation de la classe parente impose une modification de la sous-classe.

La composition d'objets est définie dynamiquement, à l'exécution, par l'introduction dans des objets des références à d'autres objets. La composition nécessite que les objets respectent les uns les autres leurs interfaces, ce qui, par conséquent, impose une conception soigneuse des interfaces. Du fait que l'accès aux objets ne peut se faire que par l'intermédiaire de leurs interfaces, il n'y a pas rupture de l'encapsulation. Chaque objet peut être remplacé par un autre pendant l'exécution, pourvu que celui-ci ait le même type.

La délégation est le processus qui fait de la composition un outil de réutilisation aussi puissant que l'héritage. Dans la délégation, deux objets sont impliqués dans le traitement d'une requête : un objet récepteur qui délègue les opérations à son délégué. C'est un comportement analogue à celui des sous-classes, qui défèrent les requêtes à leurs classes parentes. Mais dans l'héritage, une opération héritée peut toujours faire référence à l'objet reçu, par l'intermédiaire de la variable membre *this* en C++ et Java et *self* en Smalltalk. Pour obtenir le même effet avec la délégation, le récepteur passe sa propre référence à son délégué, afin que l'opération déléguée puisse s'adresser à lui directement.

2.2.6 Interface

Chaque opération déclarée par une classe spécifie le nom de l'opération, les objets paramètres, et le type de la valeur de retour de l'opération. C'est ce qui est connu sous le nom de signature de l'opération. L'ensemble de toutes les signatures définies pour les opérations d'un objet s'appelle l'interface de l'objet [2]. L'interface d'un objet spécifie donc le jeu complet

[2]Par abus de langage on parle d'objet et de classe interchangeablement

des requêtes qui peuvent être adressées à l'objet. Toute requête conforme à une signature de l'interface d'un objet, peut être envoyée à ce dernier.

Un type est un nom utilisé pour caractériser une interface particulière. On dira d'un objet qu'il a le type "Fenêtre", s'il accepte toutes les requêtes concernant les opérations définies dans l'interface nommée "Fenêtre". Un objet peut avoir plusieurs types et des objets très différents peuvent partager un même type. Dans l'interface d'un objet, une partie peut avoir les caractères d'un type et les autres parties celles d'autres types. Deux objets du même type ne doivent avoir en commun qu'une partie seulement de leur interface. Des interfaces peuvent contenir d'autres interfaces à titre de sous-ensembles. On dit qu'un type est sous-type d'un autre type, si son interface contient l'interface de l'autre, qui devient un super-type. On dit souvent qu'un sous-type hérite de l'interface de son super-type.

Les interfaces sont le fondement des systèmes orientés objet. Les objets ne sont connus qu'à travers leurs interfaces. Il n'est pas possible de connaître quoique ce soit d'un objet ou de demander à celui-ci d'effectuer quelque chose que ce soit sans recourir à son interface. L'interface d'un objet ne révèle rien de sa réalisation. Ceci veut dire que deux objets, dotés d'implémentations complètement différentes, peuvent avoir des interfaces identiques.

Dans C++ on utilise les classes abstraites pour implanter les interfaces. Par contre, le langage Java offre des mécanismes séparés pour définir les interfaces, au sens propre du terme, et les classes [KA96].

2.2.7 Notation graphique

Pour le reste de ce mémoire, nous allons utiliser comme notation une variante de celle de *OMT* [RBP+91]. Dans la littérature nous trouvons d'autres notations, telles que celle de Booch, Ivar Jacobson, *UML*, etc. Nous avons choisi une variante de la notation de *OMT* pour sa simplicité, et aussi parce que la structure des patrons de conception, que nous allons

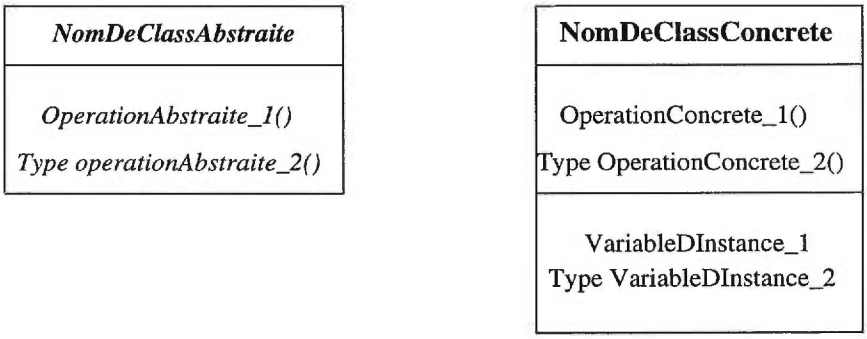

FIG. 2.1: Classe abstraite et concrète

voir plus loin, est exprimée dans les catalogues en utilisant une variante de la notation de *OMT* [GHJV94].

Diagramme de classe

La figure 2.1 montre la notation OMT pour les classes abstraites et concrètes. Une classe est représentée par une boîte avec à son sommet le nom de classe en caractères gras. Les opérations de base figurent en dessus du nom.

Toutes les variables d'instance figurent en dessous des opérations, et les informations sur leurs types sont optionnelles. Nous utilisons la convention C++ qui place le nom du type de la valeur de retour devant le nom de l'opération. De la même façon, le nom d'une variable d'instance est précédé du nom de son type. Les caractères italiques indiquent des classes ou des opérations abstraites.

La figure 2.2 montre les différents types de relation entre classes. La notation OMT pour l'héritage de classes est un triangle reliant une sous-classe à sa classe parente. La référence à un objet correspondant à une relation d'association ou d'agrégation est représentée par une flêche. La relation d'agrégation est représentée par une flêche dotée d'un losange à sa base. La flêche pointe sur la classe agrégative. Une flêche sans losange à sa base décrit une

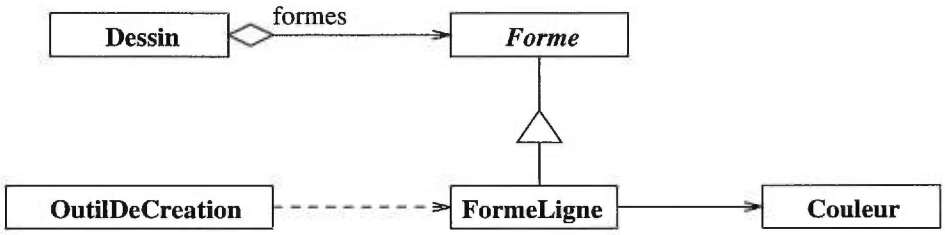

FIG. 2.2: relation entre classes

relation d'association. Un nom pour la référence peut figurer près de la base de la flèche pour la distinguer des autres références. Une flèche en "pointillé" représente une relation de création. La classe figurant à la base de la flèche crée un objet de la classe pointée par la flèche. Ce type de constructeur n'est pas défini par OMT.

Diagramme d'objet

Un diagramme d'objet montre exclusivement des instances. Le symbole pour un objet est une boîte à coins arrondis, avec une ligne qui sépare le nom de l'objet de toutes références d'objets. Les flèches indiquent les objets référencés. La figure 2.3 en montre un exemple.

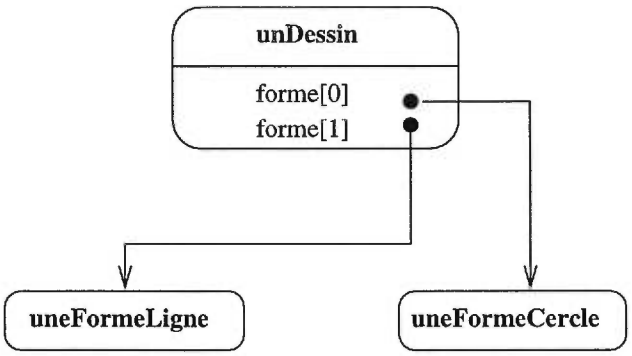

FIG. 2.3: Notation des diagrammes objets

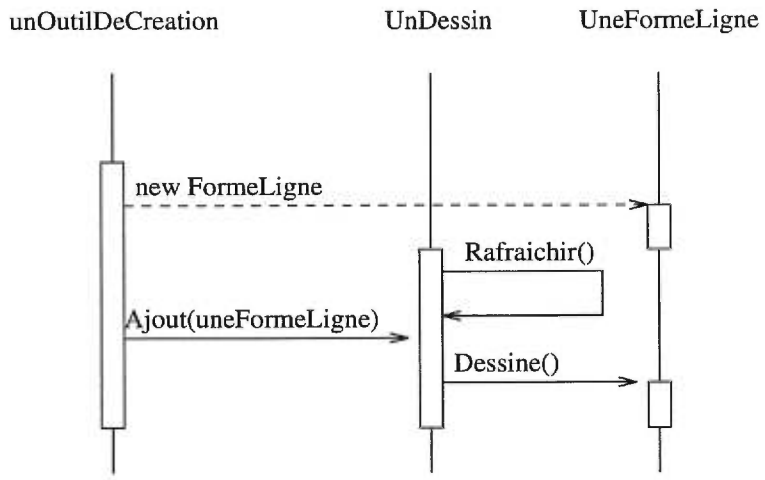

FIG. 2.4: Notation des diagrammes d'interaction

Diagramme d'interaction

Un diagramme d'interaction montre dans quel ordre les requêtes entre les objets sont exécutées. La figure 2.4 représente un diagramme d'interaction, qui montre comment une forme est ajoutée à un dessin.

Dans un diagramme d'interaction, le temps s'écoule selon la verticale du haut vers le bas. Une ligne verticale indique la durée de vie d'un objet particulier. Les conventions de nomination des objets sont les mêmes que pour les diagrammes d'objets.

Dernièrement, on a senti le besoin d'un standard pour les notations graphiques. Les "trois amigos", i.e. Booch [Boo94], Rumbaugh [RBP+91] et Jacobson [JCJO92] se sont unis pour définir un langage de modélisation graphique appelé *UML* (Unified Modeling Language) [BRJ96, BRJ97]. Ce langage a été soumi et approuvé par le *OMG* (Object Management Group), ce qui constitue un grand pas vers un langage de modélisation standardisé.

2.2.8 Un exemple de langage orienté objet

Le langage C++ est un langage orienté objet qui descend du langage C. Il en hérite les structures et les mécanismes de procédures. Il y ajoute les classes et les mécanisme du paradigme orienté objet qui viennent d'être décrits. Il ajoute également quelques autres mécanismes plus particuliers. C++ a été conçu comme un sur-ensemble de C. Les programmes écrits en C fonctionnent toujours en C++. Plusieurs compilateurs C++ génèrent du code intermédiaire en C qui est passé à un compilateur C performant. C++ peut ainsi tirer part du grand déploiement de C.

Les classes C++ possèdent trois parties : une privée, une protégée et une publique. La partie privée n'est accessible que par la classe elle-même. Même ses descendants ne peuvent y accéder. La partie protégée est accessible à la classe ainsi qu'à ses descendants. La section publique est accessible à tous. Ce mécanisme aide l'encapsulation en restreignant la partie de l'implantation qui est offerte à l'usager.

Les classes possèdent deux méthodes particulières : le constructeur et le destructeur. Le constructeur est appelé automatiquement lors de la création d'un objet et le destructeur lors de sa destruction. Le constructeur sert à initialiser l'objet et peut être surchargé. Le destructeur sert à faire le ménage avant la disparition de l'objet et ne peut pas être surchargé puisqu'il ne peut pas prendre de paramètre.

Pour des raisons d'efficacité, C++ ne fait pas de liaison dynamique par défaut. Il faut préciser, pour chaque classe, les méthodes qui seront liées dynamiquement. Ces méthodes sont dites virtuelles. Il est même possible de spécifier une méthode virtuelle sans en donner une implantation. La classe est alors dite abstraite et elle ne peut pas être instantiée. Elle ne peut que servir de parent à d'autres classes à qui il revient de fournir une implantation pour ces méthodes.

Inversement, les classes peuvent avoir des composantes qui sont partagées entre toutes les

instances de cette classe. Ces composantes sont dites statiques et peuvent servir à contenir des ressources communes.

2.3 Les types de logiciel

Il existe plusieurs taxonomies du logiciel et chaque auteur a ses propres préférences. Ce document utilise les types de logiciels qui sont définis dans [Gamm95]. Cette description est assez complète pour les besoins de ce texte tout en demeurant assez simple pour être aisément compréhensible. Les diverses catégories de logiciels sont décrites dans les sections qui suivent. Le point de vue adopté pour ces descriptions est celui de quelqu'un qui voudrait développer un tel logiciel.

2.3.1 Application

Une application consiste en un programme qui, une fois compilé, remplit une fonction bien déterminée. L'application est normalement complète en elle-même, et son comportement est entièrement défini par elle. Cela ne veut pas dire qu'elle est monolithique, elle peut être composée de plusieurs modules qui interagissent entre eux. La figure 2.5 montre le schéma d'une petite application composée de quelques modules et qui interagit directement avec le système d'exploitation.

2.3.2 Bibliothèque de fonctions

La bibliothèque de fonctions est simplement un regroupement de procédures utiles. Le comportement des composantes de la bibliothèque est également clos, c'est-à-dire qu'il ne dépend pas du contexte d'utilisation. Le comportement global est cependant sous le contrôle

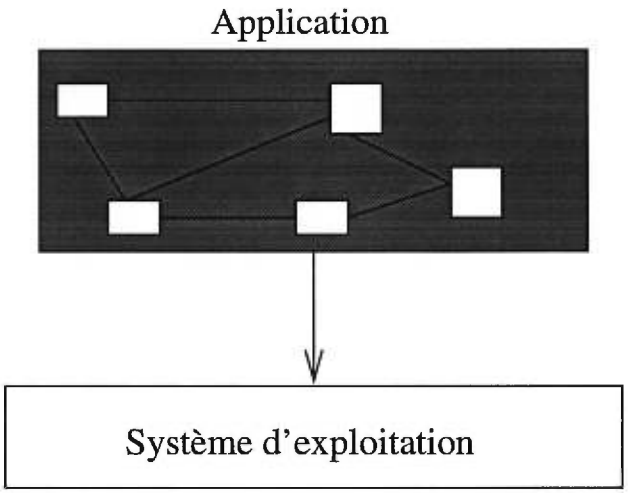

FIG. 2.5: Une application

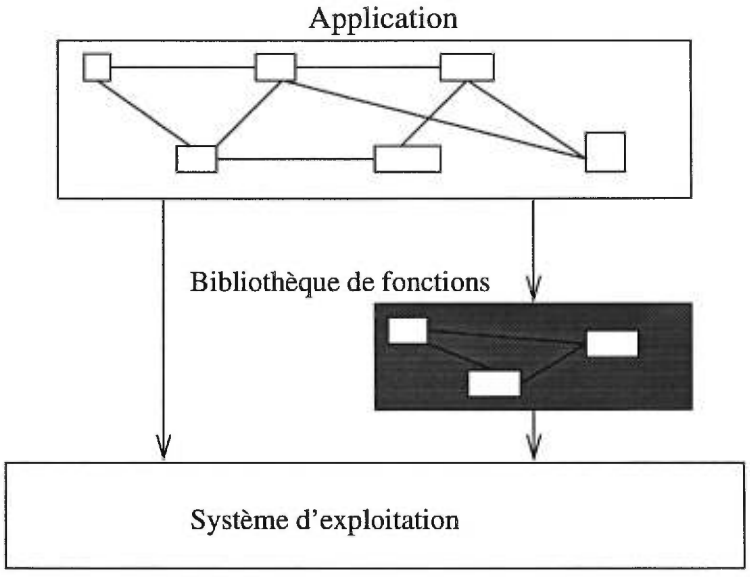

FIG. 2.6: Une bibliothèque de fonctions

de l'application. La figure 2.6 montre le schéma d'une application qui utilise une bibliothèque de fonctions pour une partie de son traitement. La bibliothèque utilise à son tour les ressources du système d'exploitation pour remplir son rôle.

2.3.3 Bibliothèque de classes

La bibliothèque de classes ressemble beaucoup à la bibliothèque de fonctions, sauf qu'elle contient des définitions de classes. Cependant, à cause des liens dynamiques et du polymorphisme, le comportement de la bibliothèque n'est pas uniquement défini par ses composantes. Ce comportement dépend également des définitions qui sont faites par l'usager de la bibliothèque. Le comportement final est cependant toujours sous le contrôle absolu de l'application qui ne fait que déléguer certaines tâches à la bibliothèque.

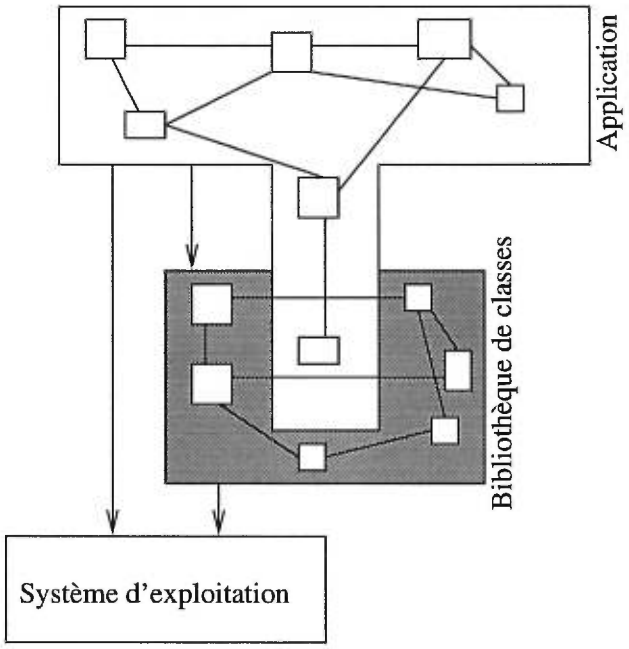

FIG. 2.7: Une bibliothèque de classes

La figure 2.7 montre le même schéma que la Figure 2.6, mais l'application utilise ici une bibliothèque de classes. On voit bien comment la bibliothèque utilise des composantes qui ont été définies par l'application et que, par conséquent, le code de l'usager est exécuté à l'intérieur de la bibliothèque.

2.3.4 Cadre d'application

Un cadre d'application est un ensemble de classes plus un modèle d'interaction. C'est une forme évoluée de la bibliothèque de classes où les rôles avec l'application sont inversés. Un cadre fournit et impose une certaine architecture conceptuelle à l'application utilisatrice. Il définit sa structure globale, le partitionnement en classes et en objets ; il en déduit les responsabilités essentielles, la façon de collaborer des classes et des objets, et la tâche de contrôle. Ainsi, le cadre pré-définit les paramètres de la conception, et par conséquent il débarrasse le concepteur ou le développeur de la tâche ardue de la conception pour qu'il puisse se concentrer essentiellement sur l'application elle-même. En effet, le cadre d'application implémente les décisions de conception courantes dans son domaine d'application. Il donne la priorité à la réutilisation de conception par rapport à la réutilisation de code. Habituellement le cadre contient des classes concrètes prêtes à utilisation immédiate.

A ce niveau, l'approche de la réutilisation conduit à une attribution inversée du contrôle entre l'application et le logiciel sur lequel elle est fondée. Lorsqu'on utilise une bibliothèque de classes, on écrit le tronc principal de l'application et on appelle le code que l'on souhaite réutiliser. Par contre, lorsqu'on utilise un framework, on réutilise le tronc principal et on écrit le code que celui-ci appelle. On aura à coder des opérations avec des noms et des conventions d'appels particuliers, mais l'approche réduit le nombre de décision à prendre pour la conception. Les figures 2.7 et 2.8 schématisent ce que nous venons de dire. En fin de compte, non seulement, on parvient à réaliser plus rapidement les applications, mais celles-ci

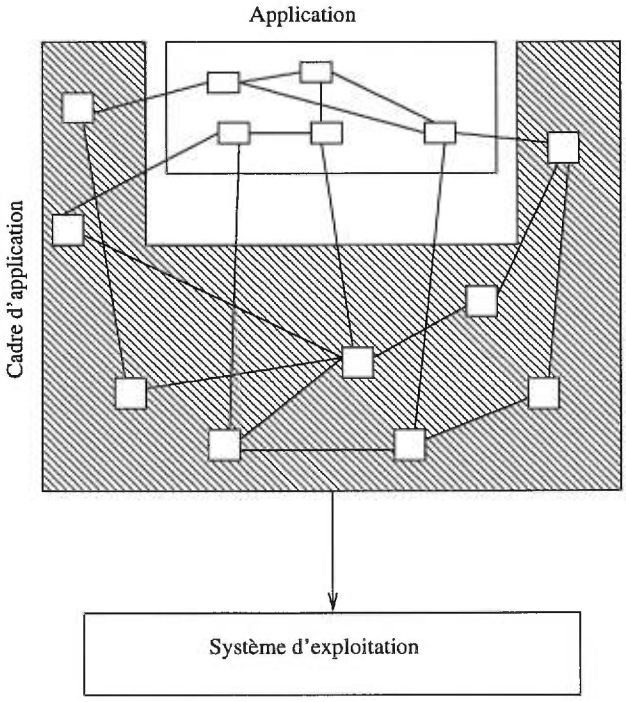

FIG. 2.8: Un cadre d'application

présentent des structures semblables. Leur maintenance est plus facile, et elles paraissent plus cohérentes aux utilisateurs. En revanche, on aura perdu une certaine liberté de conception, puisque un nombre important de décisions dans ce domaine auront déjà été prises pour nous.

Chapitre 3

Patrons de conception

La conception d'un logiciel orienté objet est une tâche difficile, surtout lorsqu'on exige du logiciel une certaine qualité, telle que : réutilisabilité, portabilité et maintenabilité. Une conception doit être spécifique au problème à résoudre, mais doit aussi être assez générale et flexible pour répondre aux problèmes et aux exigences futures. Les concepteurs expérimentés réutilisent des conceptions déjà vues et connues d'eux. Ils savent qu'il ne faut pas chercher à résoudre un problème en partant de mécanismes de base ; qu'il vaut mieux réutiliser des solutions qui ont déjà fait leurs preuves. Quand ils détiennent une bonne solution, ils la réutilisent systématiquement. C'est cette expérience qui contribue à faire d'eux des experts.

Les patrons de conception facilitent la réutilisation de solutions de conception et d'architecture efficaces. La représentation par des patrons de conception de techniques standards rend celles-ci plus facilement accessibles aux développeurs de nouveaux systèmes. Les patrons de conception aident à choisir, parmi les alternatives de conception, celles qui favorisent la réutilisation, et à éviter celles qui la compromettent. En clair, les patrons de conception aident un concepteur à obtenir plus rapidement une conception "juste".

Ce chapitre est consacré au sujet des patrons de conception. Après avoir défini le concept du patron, dans la première section, nous donnerons par la suite un exemple pour illustrer le concept dans la deuxième section. La troisième section traite de la description des patrons de

conception dans un format unique et standard. Dans la quatrième section nous discuterons de la classification et du catalogage des patrons. Une comparaison des patrons de conception et des cadres d'application sera donnée dans la cinquième section, et finalement nous donnerons une brève introduction aux langages de patrons.

3.1 Définition

Une définition du terme patron pourrait être : une solution à un problème dans un certain contexte. Cette définition qui peut paraître simpliste représente bien l'idée qui se trouve derrière les patrons.

Les patrons viennent des travaux et des écrits de l'architecte Christopher Alexander. Dans les années 60, les architectes cherchaient des moyens pour automatiser et informatiser l'élaboration de plans de construction d'immeubles. Ils étaient particulièrement intéressés par des règles et des algorithmes qui auraient pour effet de transformer des besoins en un assemblage de modules de construction (building modules). Ce mouvement est devenu ce que nous appelons aujourd'hui la construction modulaire. Alexander a donné le nom de patrons aux thèmes récurrents en architecture [AIS+77]. Ensuite, il les a présentés sous forme de descriptions et d'instructions. Il en a répertorié 253 dans le livre "The timeless way of building" [Ale79]. Ces patrons peuvent être utilisés pour construire une chambre ou un immeuble, etc.

Les informaticiens ont découvert des analogies entre les patrons d'Alexander et les patrons qui ont pu être repérés dans des architectures logicielles. Un grand nombre de chercheurs effectuent des recherches sur les patrons en orienté objet. On peut citer Booch [Boo94], Buschmann [BMR+96], Coplien [CS95], Gamma [Gam92, GHJV94] et Johnson [JF88]. L'un des véritables aboutissements de ces travaux est sans doute la parution du livre "Design

Patterns. Elements of Reusable Object-Oriented Software" de Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides [GHJV94]. Ce livre contient le premier catalogue de patrons de conception à être publié.

Dans le monde informatique, un patron de conception est aussi une description d'un problème fréquent avec le coeur d'une solution pour le résoudre. Celle-ci peut ensuite être ajustée au contexte d'une application donnée. Il est important de comprendre que le patron n'est pas la simple description d'une solution au problème. Il décrit, de façon générale, une méthode pour le résoudre. Cette méthode n'est pas nécessairement unique, ni optimale, mais elle peut être appliquée à toutes les formes sous lesquelles le problème se manifeste.

Les patrons deviennent un outil puissant pour la documentation de systèmes existants. On peut citer l'exemple des frameworks dont la compréhension est facilitée quand les patrons sous-jacents sont présentés plutôt que seulement le code. Plus spécifiquement, les patrons permettent de mieux comprendre le fonctionnement de certains frameworks. En effet, en étudiant les patrons qui sont à la base de ceux-ci, on dispose d'un niveau d'abstraction plus élevé que le code du framework qui codifie la connaissance pour un domaine particulier. La connaissance des patrons sous-jacents à un framework nous permet de mieux comprendre les interactions entre les classes de ce framework.

Leur apport peut aussi se concrétiser dans le monde de l'enseignement et en particulier celui de l'informatique. Ils fournissent un cadre particulièrement opportun pour l'explication de solutions. Leur principale force est de ne pas se contenter de présenter la solution mais aussi d'obliger l'écrivain du patron à poser le problème, replacer le contexte ainsi que les forces en présence. De plus, et encore mieux, les patrons permettent une réutilisation des connaissances en apportant une certaine formalisation concernant les explications pour une solution à un problème. Cela permet donc une transmission de la connaissance facilitée. En effet, les solutions préconisées par les patrons ont été testées, utilisées et validées par un

grand nombre de personnes.
Cela permet donc de ne pas avoir à rechercher soi-même une solution à un problème qui a déjà été résolu plusieurs fois. De plus, en développant une solution personnelle sans tenir compte du patron, on risque d'avoir un certain nombre de 'bugs' avant d'obtenir une solution satisfaisante.

3.2 Exemple de patron de conception

La "troïka" de classes Model/View/Controller (MVC) [KP88] est utilisée pour construire les interfaces utilisateurs en Smalltalk-80. La recherche de patrons de conception dans l'environnement MVC devrait aider à comprendre ce que nous entendons par le terme "patron".

MVC est composé de trois types d'objets. Le modèle est l'objet application, la vue est sa représentation visuelle, et le contrôleur définit la nature des réactions de l'interface usager aux entrées et sorties de données. Avant que MVC n'existe, l'interface usager consistait en un simple amalgame de ces objets. De fait, MVC les a découplés pour accroître la flexibilité et la réutilisation.

MVC découple vues et modèle en instaurant entre eux un protocole souscrit/notifie. Une vue doit faire en sorte que son aspect traduise l'état du modèle. Chaque fois que les données du modèle changent, celui-ci le notifie aux vues qui dépendent de lui. Par conséquence, chaque vue bénéficie d'opportunités de mises à jour. Cette approche permet d'attacher plusieurs vues à un modèle pour en donner différentes représentations. On peut créer de nouvelles vues d'un modèle sans avoir à réécrire ce dernier.

Le diagramme de la figure 3.1 montre un modèle et trois vues (le contrôleur n'y figure pas pour simplifier le diagramme). Le modèle contient les valeurs de quelques données, et des vues représentant une grille de tableur, un histogramme, et un diagramme circulaire

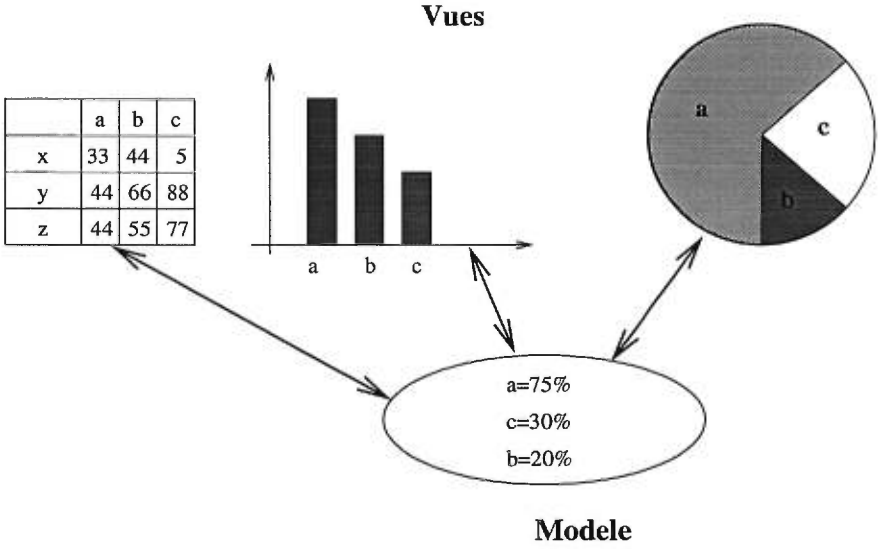

FIG. 3.1: Exemple du modèle MVC (extrait du livre de Gamma et al [GHJV94])

(camembert), affichant les données de diverses façons. Le modèle communique avec ses vues quand les valeurs de ses données changent, et les vues communiquent avec le modèle pour accéder aux valeurs.

A première vue, cet exemple représente une conception qui dissocie les vues des modèles. Mais le concept peut s'exprimer en termes plus généraux : découpler des objets de sorte que les modifications de l'un d'entre eux puissent affecter un nombre quelconque des autres, sans qu'il soit nécessaire pour l'objet modifié de connaître les détails des autres. Ce concept plus général est décrit par le patron de conception *Observer* dont la structure est illustrée par la figure 3.2.

Pour mieux décrire un patron de conception on a besoin de spécifier comment les objets constituants collaborent-t-ils pour assumer leurs responsabilités. Dans le cas du patron de conception *Observer* les règles d'interactions sont données par :

- *ConcreteSubject* notifie les *Observer* de tout changement qui pourra rendre leurs états

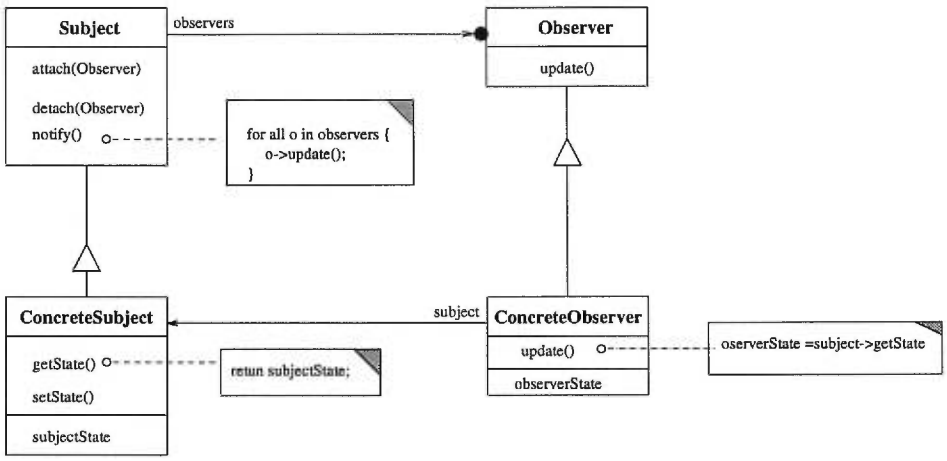

FIG. 3.2: Le patron de conception Observer

inconsistants avec le sien.

- Après avoir été informé d'un changement d'état du *ConcreteSubject*, un *ConcreteObserver* peut demander le sujet des informations concernant ce changement. Un *ConcreteObserver* utilise ces informations pour la mise à jour de son état.

La figure 3.3 représente un diagramme d'interaction. Il illustre comment un sujet (aConcreteSubject) et deux observateurs (aConcreteObserver, anotherConcreteObserver) collaborent-t-ils pour assumer leurs responsabilités.

MVC utilise d'autres patrons de conception tel que *Builder*, qui spécifie les classes *Controller* par défaut d'une vue, et *Decorator* pour ajouter la propriété de défilement à une vue. Mais, les relations les plus importantes de MVC, sont assurées par le patron de conception *Observer*.

Dans le contexte de notre travail de recherche, on s'intéresse aux patrons de conception tels qu'ils sont définis par Gamma.

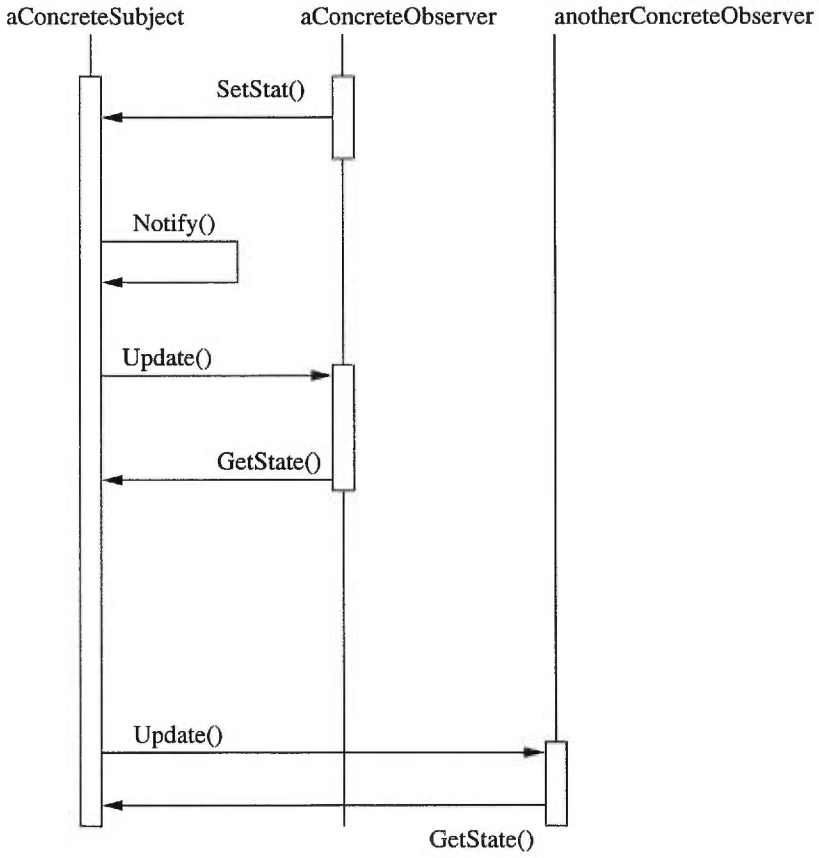

FIG. 3.3: Un diagramme d'interaction

3.3 Description des patrons de conception

Nous décrirons les patrons de conception en utilisant un format unique. Chaque patron est présenté en sections conformément au standard, de-facto, suivant [GHJV94] :

Noms du patron et classification

Le nom du patron contient succinctement son principe. Un bon nom est vital, car il va faire partie du vocabulaire du concepteur.

Intention

C'est une courte déclaration qui répond aux questions suivantes : que fait effectivement le patron de conception ? Quel est sa raison d'être et son but ? Quel cas ou quel problème particulier de conception concerne-t-il ?

Alias

Quelques autres noms reconnus du patron, s'ils en existent.

Motivation

C'est un scénario qui illustre un cas vécu de conception, et qui montre comment la structure de classe et d'objets du patron contribuent à la solution de ce cas.

Indication d'utilisation

Quels sont les cas qui justifient l'utilisation du patron de conception ? Quelles situations de conception peu satisfaisantes peuvent tirer avantage de l'utilisation du patron ? Comment reconnaître ces situations ?

Structure C'est une représentation graphique des classes du patron, qui utilise une notation issue de la méthode *OMT*.

Constituants Les classes et les objets qui interviennent dans le patron de conception avec leurs responsabilités.

Collaborations

Comment les constituants collaborent-t-ils pour assumer leurs responsabilités ?

Conséquences

Comment le patron de conception assume-t-il ses objectifs ? Quels sont les compromis qu'implique son utilisation et quel est l'impact de son utilisation ? Quelles parties de la structure d'un système permet-il de modifier indépendamment ?

Implémentation

De quels pièges, astuces ou techniques faut-il être averti lors de l'implémentation du patron ? Y a-t-il des solutions typiques du langage utilisé ?

Exemple de code

Ce sont des extraits de programmes qui illustrent la façon qu'il convient d'employer pour développer le patron dans des langages tels que C++, Smalltalk ou Java.

Utilisations remarquables

Exemples de patrons appartenant à des systèmes existants.

Patrons apparentés

Quels patrons de conception sont en relation étroite avec celui traité ? Quelles sont les différences importantes ? Avec quels autres patrons peut-il être employé ?

3.4 Classification

Les patrons de conception se différencient par la granularité et le niveau d'abstraction. Du fait qu'il existe de nombreux patrons de conception il nous faut trouver un moyen de les organiser. La classification aide à apprendre plus rapidement les patrons du catalogue, et elle permet également d'orienter le travail de recherche de nouveaux patrons.

Gamma et al [GHJV94] ont défini deux critères pour classifier et cataloguer les patrons de conception :

- **Rôle** : ce premier critère traduit ce que fait le patron. Les patrons peuvent avoir un rôle soit créateur, soit structurel, soit comportemental. Les patrons créateurs concernent le processus de création d'objets. Les patrons structuraux s'occupent de la composition de classes ou d'objets. Et finalement, les patrons de comportement spécifient les façons d'interaction des classes et des objets et définissent la répartition des responsabilités.

Critère	Type	Description
rôle	créationnel	Le patron traite principalement de la création des objets.
	structurel	Le patron traite principalement de la composition des objets ou des classes.
	comportemental	Le patron traite principalement des interactions entre les objets ou les classes.
domaine	classes	Le patron traite des relations entre les classes et leurs sous-classes selon la relation d'héritage.
	objets	Le patron traite des relations entre les objets selon les relations d'utilisation et de composition.

TAB. 3.1: Critères de classification des patrons de conception.

- **Domaine** : ce critère précise si le patron s'applique, en principe, à des classes, à des objets ou aux deux. Les patrons de classes traitent des relations entre les classes et leurs sous-classes. Ces relations sont établies par héritage, elles sont donc statiques. Par contre, les patrons d'objets traitent des relations entre objets, qui peuvent être modifiées pendant l'exécution.

La table 3.1 donne un résumé de la signification de chaque critère.

Toutefois, la classification introduite par [GHJV94] ne couvre pas tous les aspects d'un patron de conception. En effet, Buschmann et al. [BMR+96] classent les patrons de conception selon leurs portées :

- **Les patrons architecturaux** : décrivent la structure et l'organisation fondamentale

d'un système. Ils fournissent un ensemble pré-défini de sous-systèmes en spécifiant leurs responsabilités et les règles d'organisation des relations entre eux. Le choix d'un patron architectural est une décision de conception fondamentale lors de la conception du système. Le choix des patrons architecturaux se fait généralement pendant la phase de conception du système. Il constitue une décision stratégique. La "troïka" de classes Model/View/Controller (MVC) [KP88, BMR+96] utilisée pour construire les interfaces utilisateurs est un exemple de patron architectural (qui peur être décomposé en plusieurs patrons plus détaillés tels que *Observer*). Il fournit une structure pour les systèmes interactifs.

- **Les patrons de conception** : fournissent des schémas pour raffiner les sous-systèmes d'une application et les relations entre eux. Ils sont indépendants d'un langage de programmation ou d'un paradigme de programmation. Leur application n'a pas d'effet sur la structure fondamentale d'un système logiciel, mais a une influence importante sur l'architecture des sous-systèmes. Le choix d'un patron de conception constitue une décision tactique. Son impact sur le système est généralement réduit et limité. Par exemple, les patrons *Abstract Factory* et *Prototype* sont considérés, par Buschmann, comme des patrons de conception.
- **Les idiomes** : sont des patrons de bas niveau spécifiques à un langage de programmation [Cop92]. Il est évident que ce qui est considéré comme idiome dans un langage de programmation peut ne pas l'être dans un autre. Par exemple la panoplie d'idiomes et de styles conçus pour le problème de la ramasse-miette (garbage collection) dans le langage C++ n'ont pas une grande valeur dans le langage Java. Ce mécanisme de ramasse-miette est déjà incorporé dans le noyau du langage, et par conséquent le programmeur est presque débarrassé de cette tâche ardue. Le livre de James O. Coplien est entièrement dédié au sujet des idiomes. À titre d'exemple, mentionnons le

Handle-Body, *Reference Counting*, et *Pointer Counting*. Notons que parmi les patrons de conception dans le catalogue de Gamma et al. il y en a quelques-uns qui sont, en fait, une simple généralisation des idiomes qu'on trouve dans le livre de Coplien.

Il faut noter que ces différentes classifications sont complémentaires et ne sont pas contradictoires. En effet, Buschmann et al. [BMR+96] classifient les patrons de conception par rapport à leurs rôles et leurs importances dans la conception du système, tandis que Gamma et al. [GHJV94] classent les patrons par rapport à leurs structure intrinsèque et leurs sémantique.

Il existe d'autres moyens pour classer les patrons de conception. Disposer de multiples axes de réflexion à propos des patrons permet d'affiner la connaissance intime de ce qu'ils sont, des façons de les comparer et des situations où les appliquer. Walter Zimmer dans [CS95] propose une classification des patrons de conception basée sur des relations de similitude, d'utilisation et de variantes. Cette classification constitue un pas en avant vers la formalisation des catalogues. Dans la suite de ce travail, nous aurons l'occasion de revenir sur le sujet de la classification.

3.5 Patron de conception et cadre d'application

Dans cette section, nous allons discuter des similitudes entre les patrons et les frameworks. Tout d'abord, les deux concepts ont été proposés pour remédier au problème de la réutilisation. Les cadres d'application se situent au niveau de l'implémentation, par contre les patrons de conception sont au niveau conceptuel. En effet, les cadres peuvent être incorporés au code, alors que seulement des instances de patrons peuvent être incorporés au code. Un point fort des frameworks tient à ce qu'ils peuvent être concrétisés dans un langage de programmation, et pas seulement étudiés, mais exécutés et réutilisés directement. Par

contre, les patrons doivent êtres implémentés chaque fois qu'ils sont utilisés.

De plus, les patrons de conception sont moins spécialisés que les frameworks. Les frameworks correspondent toujours à un domaine d'application particulier. Par exemple, un framework éditeur graphique pourra être utilisé dans une simulation d'usine, mais il ne sera pas confondu avec un autre framework de simulation. Au contraire, les patrons de conception peuvent être utilisés dans à peu près n'importe quel type d'application. Cependant, il est possible d'avoir des patrons de conception plus spécialisés que ceux du catalogue de Gamma et al. [GHJV94]. Par exemple, Douglas C. Schmidt [Sch95b, Sch95a, SS95b, Sch95c, SS95a] a conçu un cadre d'application pour les systèmes distribués documentés par des patrons de conception spécifiques au domaine des télécommunications. Les patrons utilisés dans ce système sont moins généraux que ceux trouvés dans le catalogue de Gamma et al.

Cependant, il y a toujours une relation entre les deux concepts. Quelques patrons de conception, une fois implantés, peuvent donner naissance à un cadre d'application. Par exemple, l'implémentation du patron de conception MVC [BMR+96] dans le langage de programmation *Smalltalk* donne lieu à un cadre d'application connu aussi du nom MVC. Généralement, un patron architectural, selon la classification de Buschmann et al ; est en fait l'abstraction ou la généralisation d'un cadre d'application. De plus, si l'utilisation d'un cadre d'application est assez fréquente alors on l'exprime sous forme de patrons de conception et de systèmes de patrons.

De plus, les patrons de conception sont de bons candidats pour documenter les frameworks. En effet, Johnson [WBVC+90] propose d'utiliser un ensemble de patrons pour cette fin. Dans ce contexte, chaque tâche est vue comme un problème à régler. Chaque tâche donne ainsi lieu à un patron qui la décrit et indique comment l'accomplir. On obtient un ensemble de patrons interreliés qui décrit le cadre d'application et qui peut donc lui servir de documentation.

Chaque patron est vu comme un exemple générique d'une tâche à effectuer pour utiliser le cadre d'application adéquatement. Il peut être illustré à l'aide d'exemples qui démontrent comment le patron est utilisé. Ces exemples peuvent montrer comment les classes et les objets de l'usager interagissent avec ceux du cadre d'application.

3.6 Langages de patrons

Les patrons n'existent pas en isolation. Ils sont interreliés entre eux dans des systèmes de patrons. Un système de patrons décrit comment les patrons qui le constituent sont interreliés entre eux et comment ils se complètent mutuellement.

En effet, un système de patrons peut être comparé à un langage. Les patrons forment le vocabulaire du système tandis que les règles pour leurs implantations et combinaisons jouent le rôle de la grammaire. Cependant, un langage doit vérifier une propriété importante, celle de la complétude, alors qu'un système n'est pas concerné par cette propriété. De plus, la plupart des systèmes de patrons ne couvrent que des aspects très restreints des systèmes, sauf dans des cas très spécialisés. C'est pour cela que nous parlerons, tout au long de ce travail, de système de patrons et non pas de langage, bien que dans la littérature les deux termes soient souvent utilisés d'une façon interchangeable.

Chapitre 4

Qualité et complexité des logiciels

Les temps sont révolus où tout ce qu'on exigeait d'un logiciel était son bon fonctionnement. Actuellement, certaines normes de qualité sont imposées aux constructeurs de systèmes informatiques. En effet, un logiciel doit être conçu pour fonctionner dans des environnements différents et évolutifs. On doit donc prévoir qu'il sera modifié par plusieurs personnes, pour des buts différents, pendant des périodes de temps parfois très longues.

Dans ce chapitre, nous allons définir le concept de qualité d'un logiciel et les liens avec sa complexité. Aussi, il sera question de discuter de l'apport des patrons de conception en terme de qualité.

4.1 Qualité

Pressman [Pre97] a défini la qualité d'un logiciel comme étant la conformité à des besoins fonctionnels et de performances explicitées, à des standards professionnels de développement explicitement documentés et à des caractéristiques implicites attendues de tout logiciel développé de manière professionnelle.

L'évaluation de la qualité d'un logiciel se fait par l'étude des attributs suivants : compréhensibilité, fiabilité, commodité, portabilité, flexibilité, testabilité et efficacité. Cette liste

d'attributs n'est pas exhaustive et rien ne nous empêche d'y rajouter d'autres attributs, selon le contexte, qui semblent pertinents pour la définition d'un logiciel. En effet, dans l'état actuel du génie logiciel, on est dans l'étape de la détermination des attributs de base nécessaire à l'évaluation de la qualité. Par la suite, on passe à l'élimination de l'information redondante. Ceci se fait dans la plupart des cas par des techniques statistiques basées sur l'étude des corrélations entre les attributs. Ces attributs sont mesurables par l'intermédiaire des caractéristiques, possédant des points vérifiables. Cependant, la plupart de ces métriques sont subjectives, vu l'état actuel de la science informatique.

4.2 Complexité

Dans son livre intitulé "Object Oriented Analysis and Design with Applications" [Boo94], Grady Booch a consacré tout un chapitre à la complexité des systèmes. Booch postule que les systèmes d'envergures sont intrinsèquement complexes et que cette complexité est une propriété essentielle et non accidentelle. Notre défi est, alors, de maîtriser cette complexité et non pas de la faire disparaître. La complexité inhérente est due à quatre éléments : la complexité de l'espace du problème, la difficulté de gestion du processus de développement, la flexibilité possible à travers le logiciel et le problème de caractériser le comportement des systèmes discrets.

La complexité est la propriété d'une entité composée de plusieurs éléments reliés entre eux. Elle connote deux choses : la structure et la sémantique (le sens). Une chose est complexe lorsqu'on n'arrive pas à démêler les relations entre ses composants. Une chose est complexe aussi quand son sens et sa sémantique sont difficiles à percevoir et à appréhender. Cette division entre structure et sémantique reproduit celle entre couplage et cohérence, que nous allons voir dans des sections à venir. En effet, nous parlons de dépendances structurelles

quand deux entités se relient et s'échangent de l'information (soit des données, soit des commandes). Il y a une dépendance structurelle dés lors qu'une classe appelle ou déclare une autre classe. Nous parlons de dépendances sémantiques quand une entité dépend logiquement d'une autre pour se réaliser, ou qu'elle ne fait sens qu'en coopérant avec une autre. La dépendance sémantique est irréductible, incontournable et doit être recherchée dans l'univers du discours.

L'évaluation de la complexité d'un logiciel repose sur l'étude de cinq caractéristiques quantitatives :

- la structure du système ;
- la structure du flux de contrôle ;
- la structure du flux de données ;
- l'envergure du programme ;
- l'efficacité des algorithmes.

La plupart des métriques [1] de complexité s'attaquent à la complexité inhérente au traitement (i.e. complexité de la logique du programme). Cependant, ce qui importe avant c'est de pouvoir comprendre les logiciels, i.e. la complexité psychologique (aussi appelée complexité apparente).

Par exemple, McCabe a décrit une mesure de complexité basée sur la théorie des graphes et a montré comment elle peut être utilisée dans la gestion et le contrôle de la complexité d'un programme. Il a mis en relief plusieurs propriétés de la complexité théorique des graphes et en déduit, par exemple, que la complexité est indépendante de la taille physique et qu'elle dépend uniquement de la structure décisionnelle d'un programme [CSM+79, McC76, HS96,

[1] Par abus de langage, on confond métrique et mesure. La différence entre les deux est qu'une métrique prend deux arguments et la mesure en prend juste un.

Nav87]. De son côte, Halstead [Hal77] a proposé des métriques simples, qui sont extraits directement du code du logiciel analysé. Les mesures de Halstead ont été beaucoup critiquées dans la littérature, du fait qu'elles supposent la disponibilité du code source.

La complexité et la qualité, d'un logiciel, sont deux notions intimement liées. La complexité représente l'aspect le plus objectif et, aussi, le plus avancé dans l'investigation des différents paramètres de l'évaluation du logiciel. De ce fait, nous pouvons dire que la complexité n'est qu'une composante ou attribut de la qualité, qui donne une vue interne du logiciel, alors que la qualité en sera une vue plutôt externe et globale. Mais faute de données statistiques sur les logiciels et d'études comparatives sur la qualité et la complexité, il est difficile d'établir une corrélation nette entre ces deux notions, et même entre les attributs eux-mêmes.

L'étude de la qualité est une méthode plus complète que celle de la complexité, car elle offre une vue plus globale en désignant tous les éléments nécessaires qui devraient entrer en jeu lors de l'évaluation d'un logiciel. Elle comporte toutefois certaines faiblesses qui résident dans les faits suivants :

- les relations identifiées entre attributs et caractéristiques sont purement empiriques ;
- les métriques des caractéristiques sont subjectives, bien que quelques- unes d'entre elles puissent être objectives.

Les notions de qualité et de complexité sont traitées sous différents angles qui se recoupent. Un survol de la documentation scientifique sur ce sujet [CSM+79, McC76, HS96, Nav87, Hal77] nous conduit aux constatations suivantes :

- La qualité et la complexité ne sont pas clairement définies. La plupart des définitions sont fragmentaires ou confuses ;
- Les études menées sur la complexité ont abouti à des mesures plus formelles que celles portant sur la qualité ;

- L'objectif ultime de ces deux orientations est le même : systématiser la procédure de développement et réduire le fardeau de la maintenance des logiciels.

4.3 Cohésion et Couplage

Un bon moyen pour évaluer les méthodes ou les langages consiste à en chercher le critère de décomposition. C'est une évidence que la taille et la complexité des logiciels imposent la décomposition du problème en sous-problèmes, jusqu'à ce qu'on atteigne des unités plus facilement maîtrisables. Bien évidemment, pour la démarche orientée objet le critère de décomposition est l'objet. Cette unité de décomposition du logiciel est la même que l'unité de perception du réel. C'est une représentation d'une portion cohérente de la réalité. La cohérence est liée à l'adéquation de l'objet soit à une réalité soit à un concept. Bien que ce critère de décomposition soit une condition nécessaire, elle n'est pas suffisante pour appréhender un système. Cette limite essentielle aux systèmes nous amène à faire appel à l'approche systémique[2] pour la structuration des systèmes en sous-systèmes. Mais, cette décomposition doit prendre en compte certaines propriétés de conception, notamment la cohésion et le couplage. Dans les sous-sections suivantes, nous allons brièvement présenter les notions de couplage et de cohésion dans le contexte orienté objet.

4.3.1 Cohésion

La cohésion exprime le degré d'unité fonctionnelle d'un module ou d'un système [Sch96, EKS95]. Elle mesure la force des interrelations des éléments internes d'un module. La cohésion peut se définir en deux formules [Boo94] :

- formule faible : tout ce qui se trouve sous une entité se rapporte à un même concept ;

[2]un système est un ensemble d'objets organisé en fonction d'un but

- formule forte : tout ce qui se rapporte à un même concept se range sous une même entité.

Le paradigme classique parvient à appliquer la formule faible, mais échoue à réaliser la formule forte. La programmation orientée objet résout le problème de la cohésion grâce à l'encapsulation. Par une bonne encapsulation et en application du principe de localité, le développeur a la ressource de préserver la cohésion dans ces deux formules, faible et forte. La cohésion des classes s'évalue en considérant la liste des propriétés, mais aussi le contenu des méthodes. Une méthode ne doit pas, en principe, réaliser des opérations sur d'autres objets que celui auquel elle appartient. Elle peut seulement, par envoi de messages, activer des méthodes sur d'autres objets.

4.3.2 Couplage

Le couplage entre deux objets est une mesure du degré de leur connexion [Sch96, EKS95]. Comme dans la définition traditionnelle du couplage, nous cherchons à concevoir des objets à faible couplage. Les objets qui possèdent un degré faible de couplage sont plus réutilisables, maintenables et compréhensibles. La programmation orientée objet aide à maîtriser le couplage par l'encapsulation et par la spécification des interfaces. Le couplage est nécessaire bien sûre. Le principe de coopération le prouve. Cependant, il doit être limité au strict minimum. Ce qui nous oblige, parfois, à redessiner les classes.

4.4 Évaluation de la qualité

Dans les sections précédentes, nous avons introduit les concepts de qualité et de complexité des logiciels, ainsi que la relation entre les deux notions. Dans cette section, nous allons parler des techniques de mesure de la qualité des logiciels. L'emphase est mise sur les

systèmes orientés objet.

La panoplie des métriques développées pour les systèmes classiques, i.e. non orientés objet, ne sont, malheureusement, pas directement applicables aux systèmes orientés objet. Ces métriques ne tiennent pas en considération les effets des nouveaux mécanismes introduits, tels que l'héritage, la délégation, etc. Par exemple, les deux métriques, celles de Halstead et McCabe, lors qu'elles sont appliquées à un système orienté objet, ne donnent pas, vraiment, une idée sur la complexité, parce qu'elles ne tiennent pas en compte la structure de classes du système. Cependant, il est utile de calculer la métrique cyclomatique pour chaque classe. Elle nous donne des indications sur la complexité relative des classes isolées, et par la suite nous conduit à inspecter les classes soupçonnées de complexité.

Chidamber et Kemmer [CK94] en ont suggéré quelques métriques directement applicables aux systèmes orientés objet :

- WMC (Weighted Methods per Class) : Somme pondérée de toutes les méthodes de la classe,
- DIT (Depth of Inheritence Tree) : longueur du chemin le plus long, de la classe à la racine de l'arbre d'héritage,
- NOC (Number of Children) : nombre de classes immédiatement sous la classe dans l'arbre d'héritage,
- CBO (Coupling Between Object Classes) : nombre de classes auquelles la classe considérée est couplée. Deux classes sont couplées si les méthodes de l'une utilisent des méthodes ou des variables d'instance de l'autre,
- RFC (Response for a Class) : nombre de méthodes qui peuvent être éxécutées en réponse à un message reçu par la classe, pour tous les messages qu'un objet de la classe peut recevoir,

- LCOM (Lack of Cohesion of a Class) : différence entre le nombre de paires de méthodes ne partageant aucune variable d'instance et le nombre de paires de methodes partageant une ou plusieurs variables d'instance.

Le nombre moyen de méthodes par classe nous donne une mesure relative de la complexité de chaque classe ; si on suppose que toutes les méthodes ont la même complexité, elle devient une mesure du nombre de méthodes par classe.

La profondeur de l'arbre d'héritage et le nombre d'enfants sont des mesures de la forme et de la taille de la structure du système. Empiriquement, on suggère des architectures organisées sous forme de forêts, plutôt que des arbres profonds.

Le couplage entre les objets est une mesure de leurs inter-connections. Comme dans les mesures traditionnelles, nous cherchons à concevoir des objets à faibles couplage. Ces objets sont potentiellement réutilisables.

La réponse d'une classe est une mesure des méthodes que ces instances peuvent appeler.

La cohésion dans une méthode est une mesure de l'abstraction de la classe. Une classe avec un degré de cohésion faible doit être refaite pour refléter la bonne abstraction.

4.5 Patrons de conception et qualité des logiciels

Les patrons de conception ont été proposés pour faciliter la réutilisation de solutions de conception et d'architecture efficace. Les patrons de conception aident le concepteur à choisir, parmi les alternatives, celles qui favorisent la réutilisation, la flexibilité, la maintenance, etc., et à éviter celles qui les compromisent. Les patrons de conceptions peuvent même améliorer la documentation et la maintenance des systèmes existants en donnant explicitement la spécification des relations entre les classes et les objets ainsi que leurs mobiles sous-jacents.

Dans cette section, nous allons discuter de l'apport, des patrons de conception, en terme

de qualité du logiciel.

Essentiellement, nous allons voir comment cette technique de réutilisation nous aide à construire des systèmes de bonne qualité.

Les attributs de la qualité que le patron adresse sont évoqués dans les entrées du catalogue *Motivation* et/ou *Conséquences*. Par définition, un patron de conception est défini par une paire problème/ solution. Le problème est exprimé par des exemples typiques ou des contextes dans l'entrée réservée à la motivation. La solution proposée, sous forme du patron en question, est discutée dans l'entrée réservée aux conséquences de l'utilisation du patron.

Les patrons de conception spécialisés dans la création expriment le principe du processus d'instanciation. Tous les patrons de cette catégorie encapsulent la connaissance des classes concrètes que le système utilise. Par exemple, le patron de conception *Abstract Factory* a pour tâche de fournir une interface pour la création de familles d'objets apparentées ou interdépendantes. Parmi les conséquences de l'utilisation du patron on cite :

- favorise le maintien de la cohésion entre les objets : Si les objets produits d'une famille sont conçus pour travailler ensemble, il est important qu'une application utilise ceux d'une même famille à la fois.
- facilite la substitution de famille de produits : Une application utilise une *Abstract Factory*. La classe de *Concrete Factory* réellement utilisée est instanciée une seule fois. Ceci permet la reconfiguration de l'application pendant l'exécution ainsi que la portabilité du système.

Nous avons vu, par un exemple, comment cette catégorie de patron nous aide à construire des composantes qui améliorent des attributs de la qualité.

Les patrons structuraux étudient la façon de composer des classes et des objets pour réaliser des structures plus importantes. Eux aussi améliorent la portabilité du système, la flexibilité, le couplage, etc. Par exemple, le patron, façade, fournit une interface unifiée, à

l'ensemble des interfaces de plus haut niveau. Parmi les conséquences de l'utilisation de ce patron on note :

- favorise le couplage faible entre le sous-système et ses clients. Il est utile pour la structuration en couche d'un système et facilite les relations entre objets. Il peut, aussi, éliminer les interdépendances complexes ou circulaires.
- facilite l'utilisation du système au client en masquant des composantes dont celui-ci doit tenir compte.

La plupart des patrons du type comportemental ont pour but la réduction du couplage et l'augmentation de la cohésion des systèmes. Ceci est exemplifié par les patrons de conception : *Observer, Mediator, Chain of Responsability*, etc.

En général, les patrons de conception améliorent les attributs de la qualité suivants : réutilisation, flexibilité, portabilité, maintenabilité, compréhensibilité, utilisation.

L'étude de l'impact des patrons de conception sur la qualité des logiciels est un domaine de recherche très actif. Notamment, cette problématique est étudiée dans d'autres sous-projets du projet "SPOOL".

Chapitre 5

Détection des patrons

Ce chapitre sera consacré au problème de la détection des patrons de conception dans des systèmes orientés objet. Après une revue de la littérature, dans la première section, nous parlerons des principes de la détection des patrons. Dans la troisième section, nous introduirons les différentes approches de la détection et en particulier notre approche. De plus, dans cette section, nous discuterons la conception des détecteurs de patrons que nous avons considérés dans ce travail de recherche.

5.1 État de l'art

Dans la littérature, nous trouvons une panoplie de travaux qui portent sur la détection et la reconnaissance. Dans le domaine des télécommunications nous parlons de la détection et la reconnaissance de l'image et de la voix. Nous parlons aussi de la reconnaissance et la traduction dans le domaine linguistique et cognitif. Dans tous ces domaines, on s'intéresse à identifier et reconnaître, automatiquement, une forme pertinente dans ce domaine. En génie logiciel on s'intéresse plutôt à identifier des composantes qui possèdent des propriétés, bonnes ou mauvaises, bien définies. Par exemple, chercher des duplications d'un code, ce type de détection est très important pour l'estimation et la validation des coûts des systèmes.

Avec l'introduction des patrons de conception en génie logiciel deux questions immédiates se posent : 1- Comment reconnaître de nouveaux patrons dans un système ? 2- Comment détecter des patrons dans un code ? Ces deux questions, bien qu'elles semblent semblables, sont orthogonales. La première porte essentiellement sur la découverte de nouveaux patrons et de leur cataloguage. La deuxième, et c'est elle qui nous intéresse dans ce travail, consiste à chercher des implémentations et des occurrences de patrons de conception dans un code.

Plusieurs travaux ont été orientés dans cette direction, vue son importance et son impact sur la productivité. Citons les plus importants et les plus récents que nous ayons pus trouver :

- Alberto Mendezon et Johannes Sametinger [MS95] ont développé des heuristiques basés sur des conventions de nom et des styles.
- Christian Krämer et Lutz Prechelt [KP96] se sont concentrés sur la détection des patrons du type structurels, selon la classification de Gamma et al. Ils ont développé des heuristiques pour détecter les patrons *Adapter, Bridge, Composite, Decorator et Proxy.* Leurs heuristiques se basent sur le langage de programmation *Prolog.* Les détecteurs ainsi développés sont aveugles à la composante sémantique des patrons de conception. Par conséquent, le nombre de candidats fournis par leurs détecteurs est assez grand. Ce qui rend le travail de validation des candidats difficile.
- Dans sa thèse, Kyle Brown [Bro97] a donné des heuristiques pour détecter des patrons dans un code écrit en Smalltalk. Brown n'est pas allé plus loin que Krämer et al. dans le sens de la détection. Ses heuristiques se basent aussi sur une analyse purement structurelle. Il a pu détecter seulement quelques patrons.
- Brenda S. Baker [Bak95] donne une technique pour la détection des duplications de code dans un système. Cette technique peut être utilisée pour développer des heuristiques de détection des patrons. Cependant, les heuristiques qui en découlent restent purement structurelles, et on revient aux mêmes lacunes que celle qu'on trouve chez

Prechelt et Brown. Néanmoins, sa technique de détection des duplications est d'une grande valeur par rapport à celles qui se basent sur des styles et des conventions.

Dans le cadre de ce travail de recherche nous avons utilisé une approche qui intègre à la fois l'aspect structurel, l'aspect sémantique, et l'aspect stylistique dans le processus de la détection. Ceci est, bien entendu, rendu possible grâce à l'outil d'analyse *Gen++*, qui permet un accès et une manipulation programmatique de l'arbre syntaxique abstrait, qui représente le code du système en main. Dans le chapitre suivant, nous allons introduire cet outil et montrer comment il est utilisé pour spécifier des détecteurs de patrons de conception.

5.2 Principes de détection

Dans cette section, nous allons définir le problème de la détection des patrons de conception et montrer informellement que les heuristiques sont indispensables pour la détection.

5.2.1 Généralités

Les catalogues de patrons commencent par une motivation et une exposition du problème à l'origine du patron, ainsi qu'une solution pour l'implémentation du patron dans un langage particulier. La conception et l'implémentation de la solution qui est proposée ne sont pas uniques pour la résolution du problème, bien que ces solutions aient le même but et une sémantique d'ensemble identique. Dans certains cas même les structures des solutions peuvent être complètement différentes. Par exemple, comme implémentation du patron *Factory Method*, on peut utiliser l'héritage ou les templates, qui sont deux solutions différentes.

Dans la suite nous parlerons alors de la détection d'un schéma de solution d'un patron et non pas de la détection du patron. Nous pouvons parler de la détection d'un patron lorsqu'il

possède une seule solution pour le réaliser ou quand on est capable d'énumérer toutes ces solutions et de les détecter.

Les techniques de détection dépendent des éléments suivants :

- Le type du patron : créationnel, comportemental ou structurel, selon la classification de Gamma et al. [GHJV93].
- La structure du patron : la structure qu'on veut détecter comporte l'ensemble des liens d'héritage, d'associations, d'agrégations, de créations etc.
- La sémantique du patron : définie par l'ensemble des interactions entre les éléments qui constituent le patron.
- Le contexte du patron : par exemple, la détection du patron de conception *Observer* repose sur l'identification d'un cycle fermé composé des quatre classes *(ConcreteSubject, ConcreteObserver, Observer, Subject)*. Ce patron est du type architectural selon la classification de Buschmann et al. [BMR+96]. Or, sa sémantique est définie par les deux classes *Observer* et *Subject* ainsi que les mécanismes d'interactions entre elles, donc la recherche de ce patron nécessitera juste la présence des deux classes précédantes.

Le processus de détection est souvent fait en deux étapes :

- détection de la structure : lors de cette étape, on détecte la structure du patron. Pour ce faire, on exploite les liens structuraux entre les composantes du patron. Ces liens sont généralement des relations d'héritage, d'associations, d'agrégations et de créations. Cette phase constitue la condition nécessaire, comme elle a été définie dans Kyle Brown [Bro97].
- détection comme telle : durant cette étape on procède à l'analyse sémantique de la structure détectée. Cette phase constitue la condition suffisante.

Ces deux phases constituent les conditions nécessaires et suffisantes pour détecter un

patron de conception. Elles sont généralement effectuées en parallèle : au fur et à mesure que l'analyse structurelle se déroule, les éléments sémantiques sont vérifiés.

5.2.2 Nécessité des heuristiques

La question qui se pose immédiatement est : Peut-on concevoir des algorithmes de détection ? Vue la complexité des systèmes informatiques il est clair que la complexité des algorithmes de détection sera à son tour complexe à concevoir. En effet, pour concevoir un algorithme de détection d'un patron de conception il nous faut un modèle qui décrit toute sa sémantique en plus de sa structure. Cependant, la sémantique est unique au patron mais la structure ne l'est pas dans la majorité des cas. De plus, les patrons sont définis au niveau conceptuel et leur détection se fait au niveau implantation. Or, une conception donnée peut avoir plusieurs implantations. Donc, les algorithmes de détection doivent tenir en compte toutes les implantations possibles du patron. Ceci entraînera des algorithmes complexes. Cette complexité se traduira par des besoins d'espace importants et par des performances médiocre de ces algorithmes. Par exemple, comme nous allons le voir prochainement, la complexité d'un algorithme pour détecter un patron dépend à la fois de la structure du patron en question et aussi du système où il est encodé. C'est pour cette raison que nous cherchons, plutôt, des heuristiques simples et efficaces.

La plupart des patrons de conception qui sont actuellement catalogués sont exprimés en terme de relations d'héritage, d'agrégations, d'associations et de création. Cependant, seules les relations d'héritage et de création sont supportées par la plupart des langages dits orientés objets. Les autres relations comme l'association et l'agrégation ne le sont pas, elles sont plutôt modélisées par des constructions convenables. De plus, la distinction entre une relation d'association, plus générale, et une agrégation n'est pas facile.

5.3 Approches de détection des patrons

Dans cette section, nous allons discuter les différentes approches utilisées pour la détection des patrons de conception. Nous parlerons essentiellement de l'approche basée sur l'analyse structurelle et de l'approche qui exploite des styles et des conventions de programmation. Finalement, nous présenterons l'approche développée dans le cadre de ce travail de recherche. De plus, nous discuterons de la conception de détecteurs pour les douze patrons de conception que nous sommes présentement en mesure de détecter.

5.3.1 Approche basée sur l'analyse structurelle

Comme nous l'avons vu dans la section précédente, la phase d'analyse structurelle exploite les relations et les liens entre les composantes du patron. La technique de détection est souvent dirigée par ces liens. Cette phase s'avère cruciale dans le processus de détection. En effet, cette analyse est souvent suffisante pour la détection de certains patrons. Les patrons qui sont détectables par la simple inspection de leur structure sont non ambigus. Par exemple, le patron de conception "Composite" est dans cette catégorie de patrons. La structure de sa solution est unique et non ambiguë, dans le sens où une structure pareille ne peut être que celle d'un "Composite".

De plus, les patrons de conception ayant un élément sémantique faible i.e. une faible collaboration et communication entre ses composantes se prêtent bien à la détection automatique. Par exemple, le patron de conception "Template Method", représenté dans la figure 5.12, possède un élément sémantique faible. La détection de ce patron se résume dans une classification des méthodes d'une classe.

Par contre, les patrons de conception ayant un élément sémantique très fort sont difficiles à détecter automatiquement. Pour ce faire, nous devons utiliser des analyseurs de codes

puissants. Souvent, leur détection passe par l'utilisation d'heuristiques.

Christian Krämer et Lutz Prechelt ont utilisé le langage Prolog pour décrire la structure des patrons et ainsi les détecter. Le langage Prolog se prête très bien à la représention des structures définies par des liens d'héritage, d'association, et d'aggrégation. Ainsi, ils décrivent la structure d'un patron de conception par des règles et des faits, dans le jargon de la programmation logique. Par exemple, le patron de conception *Adapter* est décrit dans le catalogue de Gamma et al. par la structure représentée par la figure 5.2. Le modèle objet décrit par cette figure est converti dans le programme *Prolog* suivant :

```
adapter(Target,Adapter,Adaptee):-
  class(-,Target),
  class(concrete,Adapter),
  class(concrete,Adaptee),
  operation(-,-,Target,Request,-,-,-),
  operation(-,-,Adapter,Request,-,-,-),
  operation(-,-,Adaptee,SpecificRequest,-,-,-),
  inheritence(Target,Adapter),
  association(Adapter,Adaptee).
```

Ces règles décrivent les propriétés nécessaires (mais non-suffisantes) des trois classes *Target, Adapter, et Adpatee*, pour former une instance du patron de conception *Adapter*. Par exemple, ce patron nécessite qu'il y ait une délégation de la méthode *Adapter : :Request* à la méthode *Adaptee : :SpecificRequest*. Cette condition demande une analyse sémantique des interactions, qu'on ne peut pas extraire par *Prolog*. Nous alons voir que notre approche permet d'examiner la sémantique des patrons avec l'aide de l'outil *Gen++*.

5.3.2 Approche basée sur des styles

Cette approche repose sur des styles et des conventions de programmation. En effet, des styles et des conventions de programmation et de nommage des identificateurs, dans un langage de programmation particulier et même à travers les langages, deviennent de plus en plus utilisés par la communauté informatique. Parfois, des outils s'attendent à des styles pour se diriger et fonctionner. Les styles peuvent faciliter la tâche de maintenance et de compréhension du code. En outre, les styles définissent une culture d'entreprise et parfois para-entreprises, et ils aident à la communication. Par exemple, il est recommandé que les noms des classes reflètent les abstractions qu'elles modélisent et leurs noms commencent par une lettre en majuscule. Les noms des méthodes et des opérations donnent une idée du service offert. Les sélecteurs et les mutateurs commencent par "get" et "set" respectivement, suivis par le nom de la variable sur laquelle porte l'opération. L'approche stylistique repose sur des styles pour la détection des patrons de conception. Elle suppose que les développeurs suivent plus au moins ces styles. Ainsi, pour détecter le patron de conception "Factory Method", on cherche une classe, dans le code, qui déclare ou définit une méthode dont le nom commence par "make" ou "create", par exemple. Cette approche souhaite que les développeurs futurs, au moins, utilisent des noms et des styles d'implémentation des patrons de conception conformément aux styles et conventions qui sont utilisés dans les catalogues. Ces hypothèses restent difficiles à défendre dans la plupart des cas. Toutefois, cette approche peut être utilisée conjointement avec d'autres techniques de détection pour former des heuristiques qui intègrent les deux approches.

5.3.3 Notre approche

L'approche basée sur une analyse purement structurelle pour détecter un patron ne prend pas en considération l'élément sémantique qui, en fait, le distingue. Par exemple, les deux patrons "Strategy" et "State" ont la même structure de classes. Or, leurs sémantiques sont très différentes. Ainsi, l'approche structurelle est aveugle aux nuances qu'il y a entre les patrons.

Par contre, une analyse sémantique exploite l'interaction et le couplage entre les composantes du patron. Notre approche incorpore les éléments sémantiques et structurels pour la détection d'un patron. De plus, lorsque l'élément sémantique est difficile à détecter on se sert de l'approche stylistique pour prendre une décision. Ainsi, notre approche s'appuie sur la "troïka" : comportement/sémantique/style. L'utilisation d'une combinaison de ces trois éléments dépend du patron qu'on cherche à détecter. En général, on fait appel à l'analyse stylistique lorsque l'élément sémantique est assez important et difficile à détecter.

L'approche stylistique peut aussi être utilisée dans une deuxième passe pour valider les résultats de la détection. Par exemple, dans le catalogue de Gamma et al. on suggère, comme style, que le nom de la méthode de fabrication du patron "Factory Method", commence par "make" ou "create". Dans la première passe on exécute le détecteur, et dans la seconde on vérifie les méthodes de fabrication si elles suivent les conventions et les styles des catalogues.

Dans cette section, nous allons discuter de la conception des heuristiques de détection des patrons de conception retenus pour ce travail. Les patrons que nous avons considérés sont parmi ceux catalogués dans le livre de Gamma [GHJV94]. L'implémentation des heuristiques dans le langage *Gen++* sera donnée dans le chapitre suivant. Chaque sous-section sera consacrée à la conception d'un patron de conception.

Abstract Factory

Le patron de conception "Abstract Factory" a pour objectif de fournir une interface pour la création de familles d'objets apparentés ou interdépendants, sans qu'il soit nécessaire de spécifier leurs classes concrètes. La figure 5.1 donne une représentation graphique du patron. Le nom attribué à ce patron est celui d'une classe abstraite qui sert d'interface pour créer les différents produits. Cette classe abstraite contient la déclaration ou la définition d'une ou plusieurs méthodes qui manufacturent les différents objets. Souvent, l'implémentation de ce patron utilise l'un des patrons de conception : "Factory Method" ou "Prototype". Lorsque ce patron utilise le patron "Factory Method", la classe abstraite, "Abstract Factory", est un ensemble de "Factory Method". Ainsi, l'heuristique de détection du patron "Factory Method" exploite celui du "Factory Method". Cet heuristique détecte essentiellement une classe abstraite, dans le système, qui possède au moins deux "Factory Method". Il faut noter que l'heuristique ne vérifie pas que les différentes méthodes de la classe abstraite créent des objets ayant une relation entre eux. Cet élément sémantique qui caractérise le patron de conception "Factory Method" est difficile à détecter. De plus, nous ne considérons pas le cas où le patron de conception utilise le patron "Prototype", que nous n'avons pas étudié dans notre travail.

Adapter

Ce patron de conception sert à convertir l'interface d'une classe en une autre interface, conforme à l'attente du client. Il permet à des classes de collaborer, qui n'auraient pu le faire du fait d'interfaces incompatibles. La figure 5.2 donne la représentation graphique d'une variante de ce patron. Il y a deux variantes de ce patron de conception : une basée sur l'héritage (privé) et une autre basée sur le mécanisme de la délégation et de la composition. Dans ce travail, nous nous sommes intéressés uniquement à celle qui se base sur la composition, c'est

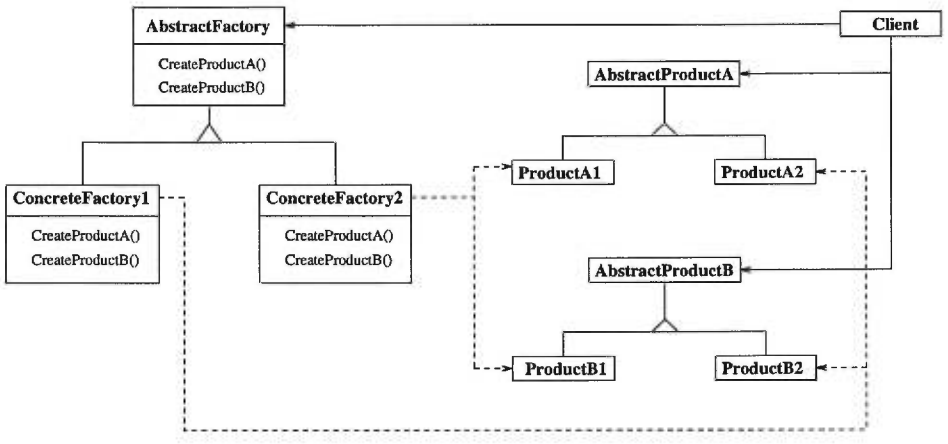

FIG. 5.1: Le patron de conception Abstract Factory

la technique qui est la plus utilisée et recommandée.

L'heuristique de détection de ce patron consiste à chercher dans le système une classe, "Adapter", qui dérive d'une autre classe, "Target". Lorsqu'une telle structure est détectée, l'heuristique cherche une association entre la sous-classe "Adapter" et une autre classe "Adaptee". La détection de cette structure complète la détection de la structure du patron.

L'analyse sémantique consiste à vérifier que la structure ainsi détectée possède les propriétés suivantes :

- la classe "Adapter" redéfinit une méthode virtuelle, "request", déclarée dans la superclasse "Target".
- dans la définition de la méthode "request" on fait appel, via le mécanisme d'association "adaptee", à une méthode, "specificRequest", définie dans la classe "Adaptee".

Le processus de détection de ce patron de conception suit de très près la représentation graphique du patron (fig. 5.2).

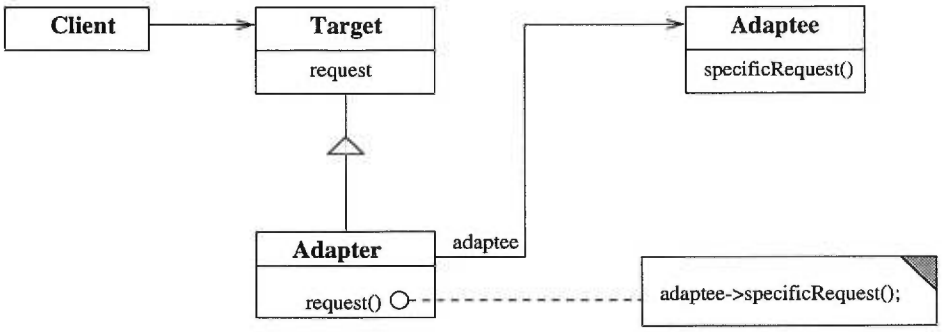

FIG. 5.2: Le patron de conception Adapter

Bridge

Le patron de conception "Bridge" sert à découpler une abstraction de son implémentation afin que les deux éléments puissent être modifiés indépendamment l'un de l'autre. Il est connu aussi du nom "Handle/Body" ou "Envelop/Letter", dans le livre de James O. Coplien [Cop92]. La figure 5.3 donne la représentation du patron.

L'heuristique commence en cherchant une structure composée de quatre classes reliées par des liens d'héritages et d'associations. Essentiellement, l'heuristique cherche une classe "RefinedAbstraction" qui dérive d'une classe "Abstraction". Ensuite, elle vérifie s'il y a une association entre la classe "Abstraction" et une classe "Implementation" qui possède à son tour au moins une sous-classe "ConcreteImplementation". Ceci constitue la détection de la structure du patron.

Dans la phase de l'analyse sémantique, on vérifie que :

- la classe "Implementor" déclare une méthode virtuelle "OperationImp" et que cette méthode est redéfinie dans les sous-classes "ConcreteImplementor".
- la classe "Abstraction" définit une méthode "Operation" qui délègue ces services à la méthode "OperationImp", via l'association "imp".

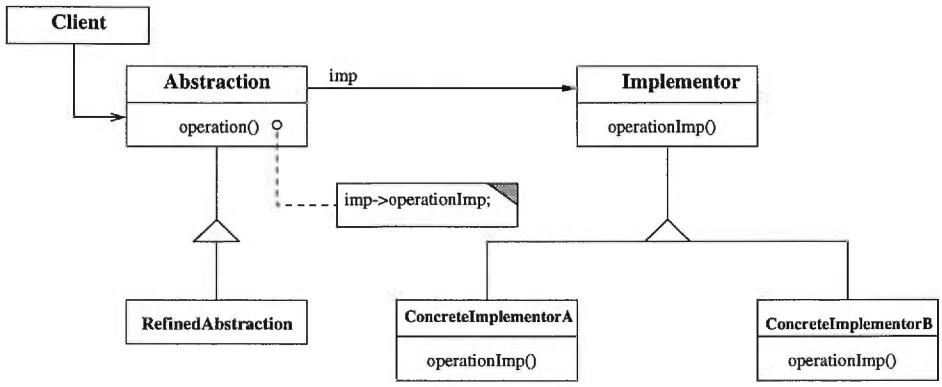

FIG. 5.3: Le patron de conception Bridge

Le processus de la détection de ce patron est reflété dans la représentation graphique du patron (fig. 5.3).

Chain of Responsability

Le but de ce patron est d'éviter le couplage de l'émetteur d'une requête à ses récepteurs, en donnant à plus d'un objet la possibilité d'entreprendre la requête. On place les objets récepteurs dans une liste, puis on déroule la liste pour identifier l'objet qui traitera la requête. La figure 5.4 en donne la représentation graphique.

L'heuristique de détection de ce patron commence par chercher, dans le code du système, une structure de classe composée d'une ou plusieurs classes "ConcreteHandler" qui dérivent d'une autre classe "Handler". De plus la classe "Handler" doit être associée à elle-même, via "successor". Cette étape constitue la détection de la structure du patron.

Composite

Ce patron sert à composer des objets en des structures arborescentes pour représenter des hiérarchies composant/composé. Il permet au client de traiter de la même façon les

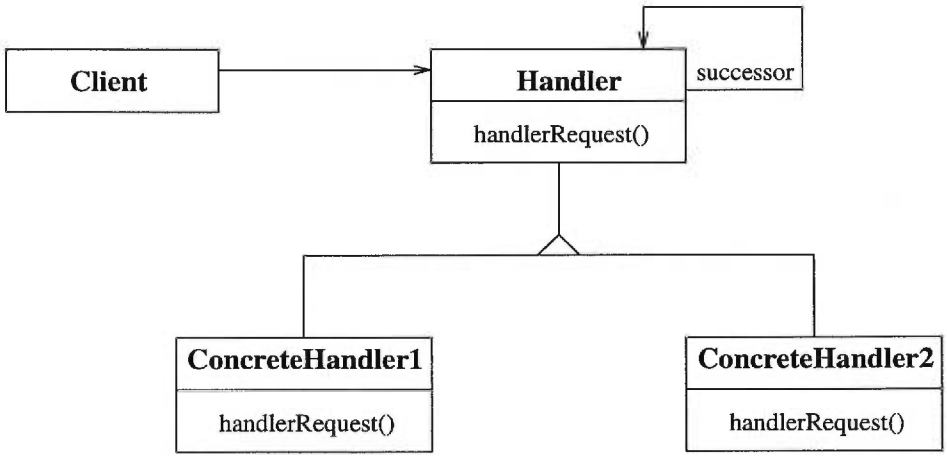

FIG. 5.4: Le patron de conception Chain Of Responsability

objets individuels et les combinaisons de ceux-ci. La figure 5.5 en donne une représentation graphique.

L'heuristique de détection de ce patron repose sur la recherche d'un cycle entre des classes qui est formé par des liens d'héritage et d'agrégation. Plus précisément, l'heuristique essaie d'identifier une classe *Composite* qui dérive d'une classe *Component*. De plus, la classe *Composite* doit être une agrégation de *Component*.

Decorator

Ce patron de conception sert à attacher dynamiquement des responsabilités supplémentaires à un objet. Les décorateurs fournissent une alternative souple à la dérivation pour étendre les fonctionnalités. La figure 5.6 en donne la représentation graphique.

L'heuristique de détection de ce patron est très similaire à celui du *Composite*, sauf qu'avec ce patron on n'a pas le problème d'agrégation. En effet, ces patrons ont des structures très semblables, et dans les deux cas on cherche à identifier un cycle composé de trois classes.

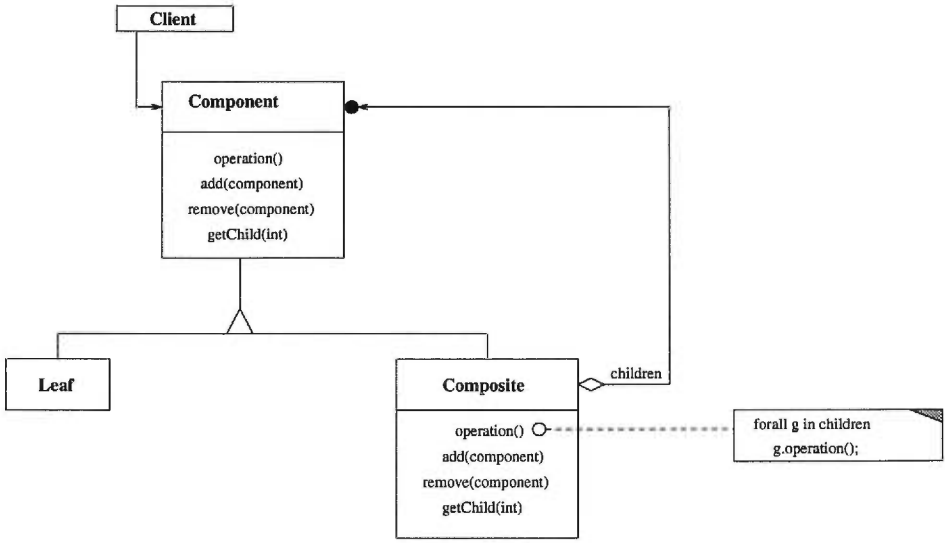

FIG. 5.5: Le patron de conception Composite

Factory Method

Ce patron définit une interface pour la création d'un objet tout en laissant à des sous-classes le choix des classes à instancier. Il permet à des classes de déléguer l'instantiation à des sous-classes. La figure 5.7 donne la représentation graphique de ce patron.

Comme les autres patrons, la détection de ce patron passe premièrement par la détection de sa structure, qui constitue la condition nécessaire. Ensuite on cherche à remplir les conditions suffisantes par l'interprétation de la sémantique du patron. Le nom attribué à ce patron est celui de la méthode qui sert à manufacturer, ou à créer des objets. Ceci nous amène immédiatement à utiliser ce processus comme artifice de détection. Plus précisément, après avoir détecté la structure du patron, i.e. une structure de classes formée d'une classe *ConcreteCreator* qui dérive d'une classe *Creator*, on cherche une méthode virtuelle implantée dans la classe *ConcreteCreator* qui retourne un nouvel objet d'une classe *Product*. Dans le langage $C++$, on cherche une méthode virtuelle dont le code contient l'instruction : *return*

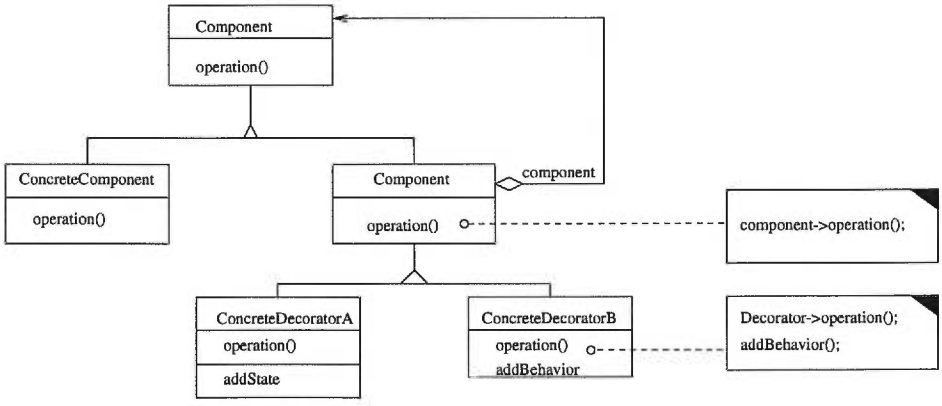

FIG. 5.6: Le patron de conception Decorator

new Product;.

Mediator

Ce patron définit un objet qui encapsule les modalités d'interaction d'un certain ensemble d'objets. Il favorise le couplage faible en dispensant les objets de se faire explicitement référence. Il permet donc de faire varier indépendamment les relations d'interaction. La structure de ce patron est donnée par la figure 5.8.

L'heuristique de détection de ce patron est très similaire à celle de l'*Observer*. On essaie d'identifier un cycle composé de quatre classes. Ces classes sont reliées par l'héritage et des associations. Ce qui simplifie l'heuristique est qu'on n'a pas d'agrégation dans ce cas.

Observer

Il définit une interdépendance de type un à plusieurs, de façon que, quand un objet change d'état, tous ceux qui en dépendent en soient notifiés et automatiquement mis à jour. La structure de ce patron est donnée par la figure 5.9.

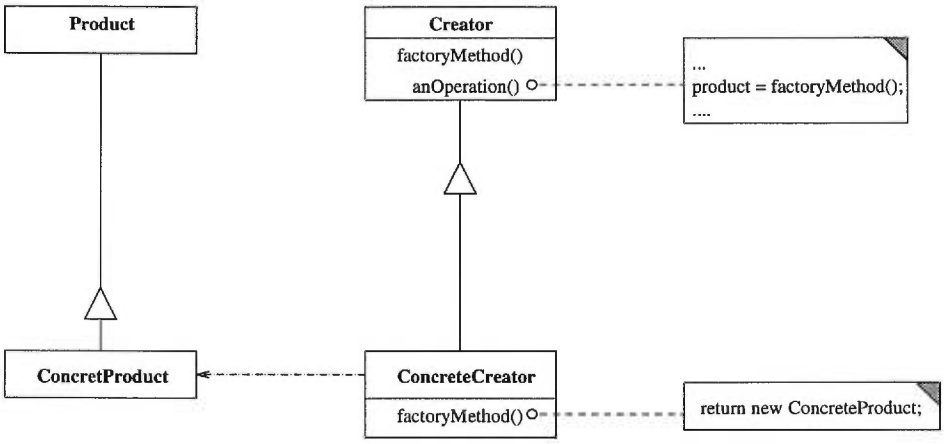

FIG. 5.7: Le patron de conception Factory Method

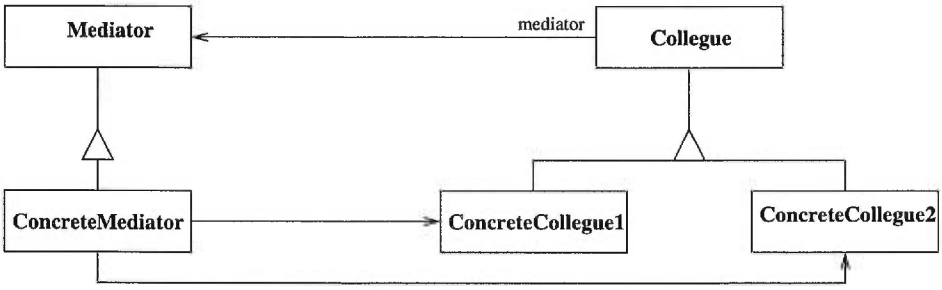

FIG. 5.8: Le patron de conception Mediator

Pour détecter ce patron, on cherche une structure de classe composée de quatre classes reliées par des liens d'héritages, d'associations et d'agrégations. D'abord, on cherche dans le système la paire de classes *ConcreteObserver* et *Observer*, ensuite une association, *subject*, entre la classe *ConcreteObserver* et la classe *ConcreteSubject*, en vérifiant que la classe *ConcreteSubject* dérive d'une classe *Subject*, et finalement une association *observers* entre la classe *Subject* et la classe *Observer*. Une fois la boucle est fermée on décide qu'il s'agit du patron de conception *Observer*. Cependant, il faut mentionner et préciser que les liens d'agrégation sont difficiles à détecter et à vérifier. Pour ce faire, il faut vérifier la sémantique

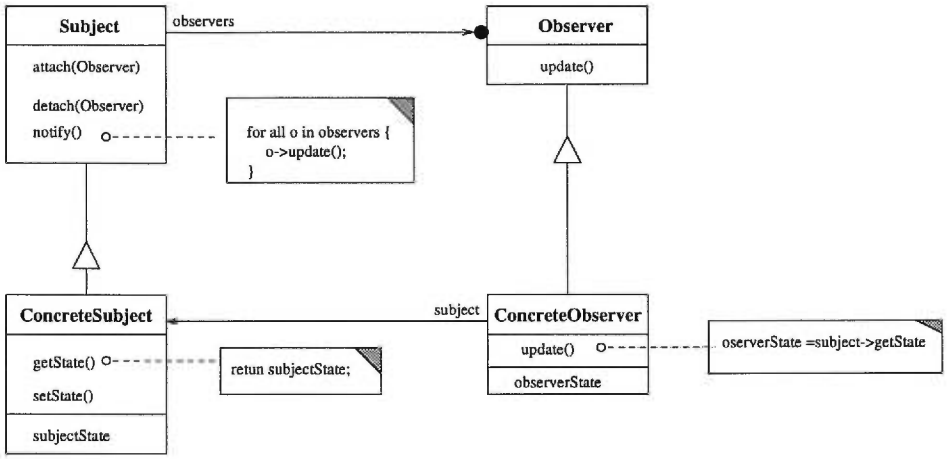

FIG. 5.9: Le patron de conception Observer

qui la différencie d'une simple association, à savoir : la fermeture transitive et la propagation des opérations.

Proxy

Ce patron fournit à un tiers objet un mandataire ou un remplaçant, pour contrôler l'accès à cet objet. La figure 5.10 en donne une représentation graphique.

L'heuristique de détection de ce patron commence par chercher, dans le code du système, une structure de classes composée d'une classe *Proxy*, associée à une classe *RealSubject* via *realSubject*, toutes les deux dérivant d'une classe *Subject*. Cette première étape constitue la détection de la structure de patron, illustrée par la figure 5.10.

La phase d'analyse sémantique consiste à vérifier que les services offerts par la classe *Proxy* sont bien délégués à la classe *RealSubject*. Plus précisément, on vérifie que la classe *Subject* déclare une méthode virtuelle, *Request*, qui est ensuite définie dans la classe *Proxy*.

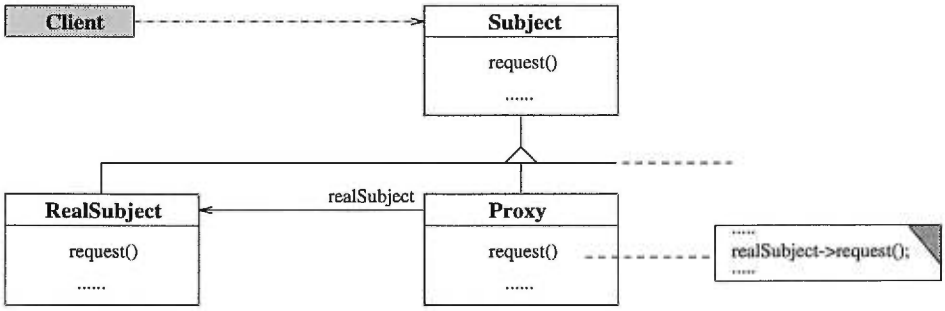

FIG. 5.10: Le patron de conception Proxy

De plus, cette méthode doit appeler au moins une méthode de la classe *RealSubject*, via le mécanisme d'association *realSubject*.

Singleton

Ce patron de conception sert à limiter le nombre d'instances créées par une classe, et fournit un point global pour y accéder. La figure 5.11 donne la représentation graphique du patron.

Parmi tous les patrons que nous avons considérés dans ce travail de recherche ce patron est celui qui possède la structure minimale, du point de vue nombre de classes. Dans ce cas toutes les classes sont candidates potentielles. L'heuristique cherche toutes les classes du système et vérifie si elles possèdent une méthode statique qui retourne une donnée statique de la classe, et ceci est fait après vérification que cette donnée n'est pas nulle.

Template Method

Le but de ce patron est de définir le squelette d'un algorithme dans une opération, tandis que les étapes de l'algorithme sont déférées aux sous-classes. Ce patron permet aux sous-classes de redéfinir certaines étapes d'un algorithme sans changer la structure de l'algorithme.

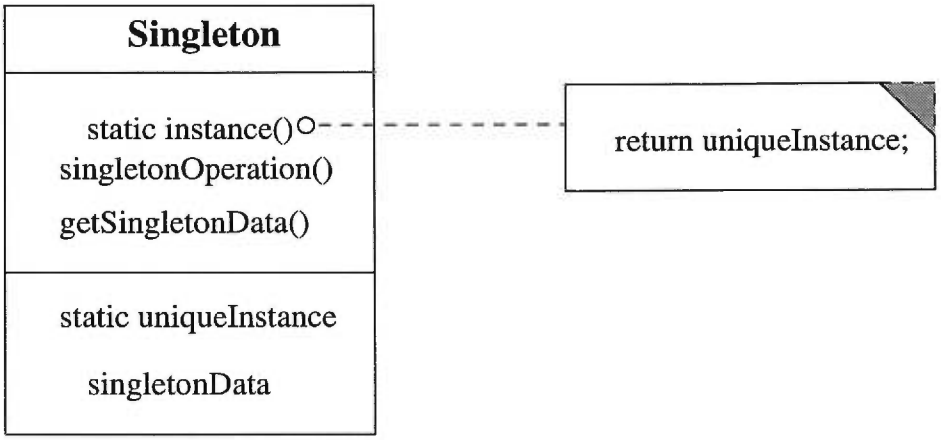

FIG. 5.11: Le patron de conception Singleton

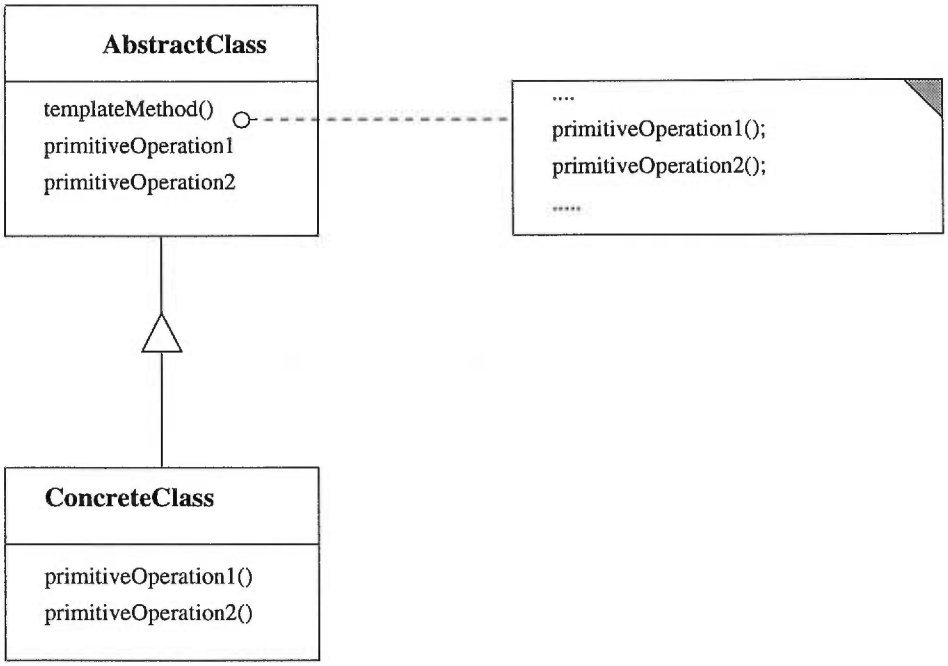

FIG. 5.12: Le patron de conception Template Method

La figure 5.12 donne la représentation graphique du patron. Ce patron, comme le patron *Factory Method*, se prête assez bien à la détection.

L'heuristique commence par chercher une paire de classes, "AbstractClass" et "ConcreteClass", liées par héritage. Cette étape constitue l'analyse structurelle du patron.

Par la suite, l'heuristique doit trouver dans la classe "AbstractClass" une méthode non virtuelle qui fait appel à au moins une méthode virtuelle de la classe "AbstractClass". Finalement il vérifie que ces méthodes virtuelles sont implantées dans la classe "ConcreteClass".

Chapitre 6

Techniques de détection développées pour SPOOL

Dans le chapitre précédant, nous avons parlé du problème de la détection des patrons en général. Ce chapitre sera consacré à la discussion sur les heuristiques développés dans le cadre du projet SPOOL. Ces heuristiques basent leurs décisions sur une analyse structurelle et sémantique. Pour ce faire, nous avons utilisé l'outil $\widehat{\text{Gen}}++$ pour l'analyse des programmes *C++*.

Nous allons commencer par l'introduction et l'illustration de l'outil *Gen++*. Ensuite, nous parlerons de la conception et de l'implémentation de nos détecteurs. Nous procéderons finalement à une discussion et une comparaison.

6.1 L'outil d'analyse *Gen++*

Dans cette section nous allons introduire l'outil *Gen++*. *Gen++* offre une belle interface d'accès et de parcours de l'arbre syntaxique et sémantique d'un programme écrit dans le langage *C++*. L'outil est basé sur deux autres langages : *GENOA* et *GENII*. Le premier sert comme langage de spécification des requêtes et des opérations sur les arbres abstraits,

tandis que l'autre sert principalement à spécifier les interfaces d'accès aux arbres. C'est grâce à *GENII* qu'on peut porter *GENOA* d'un langage à un autre [Dev92, DE94].

Nous allons commencer dans la première sous-section par introduire les outils *GENOA* et *GENII*. Dans la seconde sous-section nous présenterons l'outil *Gen++*. Finalement nous en donnerons un exemple d'illustration.

6.1.1 Aperçu de *GENOA/GENII*

L'exécution de *Gen++* se fait en deux phases. Dans la phase de traduction, le code source est analysé et vérifié pour produire une représentation intermédiaire. Cette représentation intermédiaire est appelée un arbre syntaxique abstrait. Cet arbre est décoré par des informations sémantiques pour faciliter son parcours et son analyse. Pendant la phase d'analyse, l'arbre syntaxique décoré est traversé, et les opérations spécifiées par l'usager sont exécutées.

Gen++ est basé sur le générateur d'analyseurs *GENOA/GENII* [Dev92, DE94] qui est portable et indépendant des langages. Les analyseurs sont spécifiés dans le langage *GENOA*. Le langag *GENOA* possède des opérateurs de parcours des arbres syntaxique abstraits, en profondeur et en largeur, et un opérateur d'itération. Les spécifications écrites dans le langage *GENOA* sont indépendantes du langage source. Pour utiliser le langage *GENOA* avec un nouveau langage, il suffit d'écrire une interface dans *GENII*. Une spécification *GENII* définit les types de données abstraits de l'arbre syntaxique abstrait, i.e. les types de noeuds, la hiérarchie des types, les champs de ces types, ainsi qu'une implémentation des structures de données abstraites en terme des structures de l'analyseur du langage cible. Cette spécification est compilée pour produire des fonctions de transformation et d'accès aux tables. Ces fonctions sont utilisées par l'analyseur généré par le système *GENOA*.

6.1.2 Utilisation de *Gen++*

La spécification d'un outil dans le système *Gen++* est composée d'un ensemble de requêtes *GENOA*. Ces requêtes se basent sur une hiérarchie de types qui caractérise les noeuds du graphe syntaxique. Cette hiérarchie de types est définie dans le langage de spécification *GENII*. Nous avons besoin de la spécification complète de la hiérarchie des types supportés par la version en main pour orienter et guider le fouineur du graphe syntaxique.

Pour illustrer l'utilisation de *Gen++*, nous avons implémentés un outil qui identifie les références aux variables et leur utilisation. La spécification de cet outil est donnée par le programme suivant :

```
ROOTPROC VarUseDef
PROC VarUseDef ROOT CFile;
  {
    [
      (?IdName
        (IF (AND (TYPEOF $parent Assignment) (EQUAL $slot "be_lhs"))
          (THEN
            (PRINT stdout "At %s assignment to  %s\n" $location $token)
          )
          (ELSE
            (PRINT stdout "At %s access of %s\n" $location $token)
          )
        )
      )
    ]
  }
```

Une spécification en *GENOA* est formée de plusieurs procédures, dont certaines ont un statut spécial, celles-ci sont appelées des procédures racines. Les procédures racines sont appelées automatiquement après la phase de transformation. Le noeud courant à partir duquel le processus de la traversée commence est spécifié dans la liste des paramètres de la procédure. Dans le cas de l'exemple en main, *VarUse* est la seule procédure racine du programme. Une procédure dans *Gen++* est une série de constructions ou d'opérations sur un noeud courant. Les noeuds sont décorés par des éléments de la hiérarchie de types définies par la spécification *GENII*.

L'outil *Gen++* possède quatre types de construction pour manipuler les graphes syntaxiques : constructions de parcours, constructions impératives, constructions de tests, et des constructions pour les expressions. Dans cette section on va se limiter à décrire les mécanismes de parcours. Les autres constructions sont assez bien documentés dans [DE94].

Les opérateurs de parcours permettent de passer du noeud courant à un nouveau noeud. De plus, ils offrent la possibilité de spécifier une liste d'opérations à exécuter dans le noeud courant. Ces opérateurs sont :

```
<slotname ... list of operation ...>
{typename ... list of operation ...}
[... list of operation ...]
(TRAVERSE typename expression operation)
```

Le premier opérateur permet de passer du noeud courant à une composante de ce noeud. Cette composante peut être soit un noeud terminal (feuille), i.e. qui ne contient pas d'autres noeuds, ou composée. Le premier paramètre de cet opérateur indique le nom du noeud vers lequel on veut aller. L'autre paramètre sert à spécifier la liste des opérations à exécuter dans ce noeud.

Le deuxième opérateur, les accolades, est un itérateur de liste. Il est utilisé lorsque le noeud courant est une liste de noeuds. Le premier paramètre spécifie le type des noeud.

Le deuxième est une liste d'opérations à exécuter dans tous les noeuds de la liste. Tous les noeuds de cette liste doivent être du même type.

Le troisième opérateur, entre crochets, permet de parcourir en profondeur le graphe syntaxique à partir du noeud courant. Le seul paramètre de cet opérateur spécifie la liste des opérations à exécuter dans chaque noeud pendant le parcours.

Le dernier opérateur, *TRAVERSE*, sert à manipuler un noeud qui a été sauvegardé pendant des traitements antécédents. Cet opérateur ne suppose pas qu'on est dans un noeud courant, contrairement aux autres opérateurs. Les paramètres de cet opérateur spécifient le type d'une expression, l'expression elle-même, et une liste d'opérations à exécuter sur les noeuds. Cet opérateur a joué un rôle primordial dans la spécification des détecteurs des patrons de conception.

Dans l'exemple d'illustration, la déclaration *ROOT Cfile* spécifie le type du noeud courant à partir duquel le processus de la traversée commence. Ensuite, nous avons utilisé l'opérateur d'itération sur les listes, pour traiter tous les noeuds de la liste des noeuds de type *Cfile*. L'opérateur de parcours en profondeur est utilisé pour traverser en profondeur le graphe syntaxique à partir de chaque noeud de cette liste. Dans chaque noeud du graphe syntaxique on exécute l'opération suivante :

```
(?IdName
   ( IF (AND ....
   .
   .


)
```

Cette opération interroge chaque noeud pour savoir s'il est du type *IdName*, i.e. s'il représente un identificateur. Ensuite, on teste si le type du parent de ce noeud est *Assignment*, i.e.

une affectation, et si l'identificateur est dans le côté gauche de cette affectation. Finalement on imprime sur l'unité de sortie le numéro de la ligne où l'identificateur a été localisé, son nom, et s'il s'agit d'une affectation ou d'un accès. Par exemple, dans le programme suivant :

```
main()
{
 int x,y,z;
 x=2;
 y++;
 x=z;
}
```

on a deux affectations et un simple accès. Ainsi, notre analyseur va imprimer sur la console :

```
At 2 assignement to x
At 3 assignement to y
At 4 assignement to x
At 4 access of z
```

Les noms des types *Cfile* et *Idname* sont définis dans la spécification *GENII.*

6.2 Implantation des algorithmes

Dans ce travail nous avons implémenté des détecteurs pour les douze patrons de conception suivants : *Abstract Factory, Adapter, Bridge, Chain of Responsability, Composite, Decorator, Factory Method, Mediator, Observer, Proxy, Singleton*, et *Template Method.* Ces détecteurs ont été testés et validés sur quatre systèmes différents. Le chapitre suivant traitera des expérimentations.

Dans cette section, nous allons présenter le détecteur du patron de conception Factory

Method. Cet exemple permet d'illustrer comment le processus d'implémentation des détecteurs a été conduit. Le détecteur est exprimé sous forme de spécification de requêtes écrites dans le langage *Gen++*.

6.2.1 Implémentation du détecteur de *Factory Method*

La spécification du détecteur du patron de conception "Factory Method" est composée de cinq procédures et fonctions. Parmi elles, on retrouve la procédure racine, "FactoryMethod", qui s'exécute automatiquement au début du programme. Chaque procédure est conçue pour effectuer une tâche de détection spécifique. Dans ce qui suit nous allons présenter chacune de ces procédures et montrer comment elles collaborent pour détecter le patron. La procédure racine, dans le jargon de *Gen++*, sert à parcourir toutes les déclarations globales des classes du système. En même temps elle ne considère que les classes qui possèdent au moins un parent. Cette phase de détection est typique de la majorité des détecteurs. Par la suite, on examine les méthodes virtuelles des classes parentes, une par une, pour voir celles qui sont redéfinies par les sous-classes. C'est à ce moment que la procédure racine passe le contrôle à la procédure "ChercherClass" pour vérifier si la sous-classe redéfinit la méthode virtuelle. Il faut noter que cela se fait pour chaque classe "ConcreteClasse", définie dans le système, et pour chaque super-classe "Creator" de la classe "ConcreteCreator", à cause de la possibilité de l'héritage multiple.

L'outil *Gen++* n'offre pas la possibilité d'accéder directement aux sous-classes d'une classe. Dans le graphe syntaxique abstrait, toute classe possède un champ qui indique la liste des super-classes. On est obligé, alors, de faire deux passes : une pour passer d'une classe à ses super-classes et l'autre pour revenir et analyser la classe en question. La spécification complète de la procédure racine est donnée par le programme suivant :

```
ROOTPROC FactoryMethod
PROC FactoryMethod
```

```
ROOT CFile;
{
LOCAL GNODE fonction;
LOCAL GNODE Creator;
LOCAL GNODE I;
LOCAL GNODE ConcreteCreator;
LOCAL GNODE CreatorId;
LOCAL GNODE CreatorWhere;
(ASSIGN nc 0)
<globals
  {Declaration
    [
      (?ClassDef
        (ASSIGN ConcreteCreator $token)
        <bases
          (IF (NOT (NULL $token))
            (THEN
              {BaseSpec (ASSIGN Creator $token)
                <refname <id (ASSIGN CreatorId $token)>
                  <def <where (ASSIGN CreatorWhere $token)>
                    (?ClassDef
                      <members
                        {MemberSpec
                          (?MemberDeclaration
                            <def
                              (?FunctionSpec (ASSIGN fonction $token)
                                <id (ASSIGN I $token)>
                                <attrs
                                  (?FA_Virtual
                                    (CALL chercherClass CreatorId ConcreteCreator
                                                        I CreatorWhere
                                    )
                                  )
                                >
                              )
                            >
                          )
                        }
                      >
                    )
                  >
                >
              }
            )
          )
        >
      )
    ]
  }
>
}
```

La procédure "chercherClass", appelée par la procédure racine, sert à vérifier si une

méthode virtuelle déclarée ou définie dans une super-classe est bien redéfinie dans une sous-classe. Elle prend comme paramètres le nom de la classe "Creator", la définition du noeud qui représente la classe "ConcreteClasse", le nom de la méthode virtuelle sur laquelle porte la recherche et la position dans le système où la classe "Creator" a été repérée. Ces paramètres ne sont pas tous indispensables pour la procédure. Certains d'entre eux sont donnés à la procédure pour qu'elle les communique à d'autres procédures qu'on va voir par la suite. Ces parametres sont *CreatorId, CreatorWhere.* L'implémentation de cette procédure dans le langage *GENOA* est la suivante :

```
PROC chercherClass
ARG GNODE CreatorId;
ARG GNODE ConcreteClass;
ARG GNODE functionid;
ARG GNODE CreatorWhere;
ROOT G_AnyType;
{
  LOCAL GNODE ConcreteCreatorId;
  LOCAL GNODE ConcreteCreatorWhere;
  LOCAL GNODE I;
  LOCAL GNODE fonction;
  (TRAVERSE ClassDef ConcreteClass
    (?ClassDef
      <id (ASSIGN ConcreteCreatorId $token)>
      <where (ASSIGN ConcreteCreatorWhere $token)>
      <members
        {MemberSpec
          (?MemberDeclaration
            <def
              (?FunctionSpec (ASSIGN fonction $token)
                <id (ASSIGN I $token)>
                <attrs
                  (IF (EQUAL functionid I)
                    (THEN
                      (CALL SearchBody fonction ConcreteCreatorId
                                       CreatorId ConcreteCreatorWhere
                                       CreatorWhere
                      )
                    )
                  )
                >
              )
            >
          )
        }
      >
    )
  )
```

```
}
```

Si la méthode analysée est redéfinie dans la sous-classe le contrôle est passé à la procédure "SearchBody". Toutes les informations nécessaires pour localiser le patron dans le système sont sauvegardés et communiqués entre les deux procédures. Les deux procédures "FactoryMethod" et "ChercherClass" représentent la phase de l'analyse structurelle du patron. À partir de cette étape on continue avec d'autres procédures pour analyser la sémantique de la structure ainsi détectée.

La procédure "SearchBody" sert à examiner le corps d'une méthode. Elle commence par isoler le corps de la méthode des autres informations contenues dans le noeud qui la représente. Ensuite, elle passe le contrôle à la procédure "SearchBlockBody". Cet astuce est utilisé, entre autres, pour diminuer la taille de la procédure "SearchBody" et pour implanter la méthode "SearchBlockBody" d'une façon récursive. La récursivité permet une analyse plus exhaustive des blocs imbriqués. L'implantation de la procédure "SearchBody" est donnée par le programme suivant :

```
PROC SearchBody
ARG GNODE TemplateMethod;
ARG  GNODE ConcreteCreatorId;
ARG  GNODE CreatorId;
ARG  GNODE ConcreteCreatorWhere;
ARG GNODE CreatorWhere;
ROOT G_AnyType;
{
  LOCAL GNODE FactoryMethodId;
  (TRAVERSE FunctionSpec TemplateMethod
    <defname
      <def
        <id (ASSIGN FactoryMethodId $token)>
        (?FunctionDef
          <body
            (?Block
              (CALL SearchBlockBody $token ConcreteCreatorId CreatorId
                                    FactoryMethodId ConcreteCreatorWhere
                                    CreatorWhere
              )
            )
          >
        )
```

```
    > >
  )
}
```

La procédure "SearchBlockBody" est centrale pour l'analyse de la sémantique de la structure détectée par les autres procédures, "FactoryMethod" et "ChercherClass". Cette procédure examine les énoncés et les instructions qui composent le corps d'une méthode. Elle cherche essentiellement à localiser l'instruction "return". Une foix cette instruction est trouvée, elle analyse sa valeur de retour. Si la valeur retournée est une création d'une instance d'une classe alors cette classe représente le produit construit. La spécification de cette procédure est donnée par le programme suivant :

```
PROC SearchBlockBody
ARG GNODE Corps;
ARG GNODE ConcreteCreatorId;
ARG GNODE CreatorId;
ARG GNODE FactoryMethodId;
ARG GNODE ConcreteCreatorWhere;
ARG GNODE CreatorWhere;
ROOT G_AnyType;
{
  LOCAL GNODE I;
  LOCAL GNODE ou;
  LOCAL GNODE ConcreteProductWhere;
  (TRAVERSE Block Corps
    <blockbody
      {Statement
        (?Return
          <where (ASSIGN ou $token)>
          <value
            (?New
              <new_type
                (?Typeref
                  <refname
                    (?TypeName
                      <id (ASSIGN I $token)>
                      <def <where (ASSIGN ConcreteProductWhere $token)>>
                      (CALL PrintInfo CreatorId ConcreteCreatorId
                                      I FactoryMethodId ConcreteCreatorWhere
                                      ConcreteProductWhere ou CreatorWhere
                      )
    )
                  >
                )
              >
            )
```

```
        >
      )
      (?If
        <ifTbranch
          (?Block
            (CALL SearchBlockBody $token ConcreteCreatorId
                                  CreatorId FactoryMethodId
                                  ConcreteCreatorWhere CreatorWhere
            )
          )
        >
      ) /* If Statement */
      (?While
        <whilebody
          (?Block
            (CALL SearchBlockBody $token  ConcreteCreatorId
                                  CreatorId  FactoryMethodId
                                  ConcreteCreatorWhere CreatorWhere
            )
          )
        >
      )
      (?Block
        (CALL SearchBlockBody $token ConcreteCreatorId
                              CreatorId FactoryMethodId
                              ConcreteCreatorWhere CreatorWhere
        )
      )
      (?Do
        <dobody
          (?Block
            (CALL SearchBlockBody $token ConcreteCreatorId
                                  CreatorId FactoryMethodId
                                  ConcreteCreatorWhere CreatorWhere
            )
          )
        >
      )
    }
  >
 )
}
```

La procédure "PrintInfo" sert à imprimer les résultats de la détection sur une unité de sortie. Elle affiche des informations qui localisent les composantes du patron de conception dans le système. Entre autres, cette procédure nous informe sur les noms et les positions des classes et les méthodes qui forment le patron en question. Les résultats sont imprimés textuellement pour faciliter l'interface du module de détection avec d'autres systèmes. Par

exemple, dans le cadre du projet *SPOOL*, ces résultats sont ensuite récupérés et analysés pour afficher les instances des patrons sous forme graphique. Cette procédure reflète la structure du patron telle qu'elle est donnée dans le catalogue. Elle est unique pour chaque patron. Sa spécification est donnée par le programme suivant :

```
PROC PrintInfo
ARG  GNODE CreatorId;
ARG  GNODE ConcreteCreatorId;
ARG  GNODE ConcreteProductId;
ARG  GNODE FactoryMethodId;
ARG  GNODE ConcreteCreatorWhere;
ARG  GNODE ConcreteProductWhere;
ARG  GNODE ou;
ARG  GNODE CreatorWhere;
ROOT G_AnyType;
  {
    (ASSIGN nc (+ nc 1))
    (PRINT stdout "%s ) PATTERNMODEL : Factory Method\n" nc)
    (PRINT stdout "CLASS Creator: %s\t%s\n" CreatorId  CreatorWhere)
    (PRINT stdout "CLASS ConcreteCreator: %s\t:%s\n" ConcreteCreatorId
                                                     ConcreteCreatorWhere)
    (PRINT stdout "CLASS ConcreteProduct: %s\t:%s\n" ConcreteProductId
                                                     ConcreteProductWhere)
    (PRINT stdout "Factory Method:%s\t:%s\n" FactoryMethodId ou)
    (PRINT stdout "\n")
  }
```

Les spécifications des autres détecteurs sont semblables à celle du détecteur du patron de conception *Factory Method*. Essentiellement, elles sont composées de procédures qui s'occupent de la détection de la structure du patron, des procédures d'analyse de la sémantique du patron, et une procédure d'affichage des résultats de sorties.

6.2.2 Taille des analyseurs

La table ci-dessous donne la taille (LOC dans le langage *Gen++*) de chaque analyseur ainsi que la taille globale de tous les analyseurs.

Patron	LOC
Abstract Factory	226
Adapter	414
Bridge	754
Chain of Responsability	83
Composite	254
Decorator	122
Factory Method	211
Mediator	199
Observer	227
Proxy	494
Singleton	238
Template Method	300
Total	3522

TAB. 6.1: Taille des analyseurs (LOC dans le langage *Gen++*)

Chapitre 7

Expérimentation et discussion

Ce chapitre présente les expériences faites avec les détecteurs sur des systèmes réels. Cela nous permet de valider notre travail de recherche et de le comparer avec d'autres travaux connus de la littérature. Les mesures de performance calculées en fonction de ces systèmes seront aussi analysées et interprétées. Il sera question de problèmes rencontrés lors de la conduite des expériences. Ce chapitre conclura avec une discussion des travaux futurs pour étendre et améliorer notre travail.

7.1 Systèmes de tests

Dans cette section nous allons introduire les systèmes de test que nous avons utilisés pour valider et comparer nos résultats. Le choix de ces systèmes est basé, en premier lieu, sur leur disponibilité et sur leur richesse en matière de patrons de conception. Par exemple, les systèmes ACE et ET++ sont des cadres d'application documentés par des patrons de conception. Ces deux systèmes sont très différents du point de vue conceptuel. La hiérarchie de classes du système ET++ a la forme d'une arborescence où toutes les classes du système dérivent d'une classe racine "Object". Par contre, le système ACE utilise intensivement les "templates" pour implémenter les structures génériques. Les deux autres systèmes *LEDA*

et *xForms* ont été choisis parce qu'ils étaient conçus et développés sans avoir le concept du patron à l'esprit. Ainsi, nous voulons savoir si nos détecteurs sont capables de détecter des patrons dans ces deux systèmes qui sont relativement conventionnels. De plus, le système "LEDA" a été utilisé pour valider les travaux de Prechelt et al., et il est intéressant pour nous de comparer nos résultats avec les leurs. La figure 7.1 donne la taille de chaque système.

	ET++	ACE	LEDA	xForms
Fichier	276	385	175	133
Macros	622	3665	477	441
Fonction	284	222	302	16
Classes	300	396	222	106
Méthodes	3634	1162	4963	452
LOC	113023	188225	197559	39876

TAB. 7.1: Taille des systèmes de tests

7.1.1 *ET++*

Le système *ET++* est un cadre d'application orienté objet implanté dans le langage C++. Il est composé d'une librairie de classes intégrant l'aspect interface usager, les structures de données de bases et les mécanismes d'entrés/sorties. La hiérarchie de classes du système est une arborescence, semblable à celle de *Smalltalk* où toutes les classes dérivent d'une classe racine commune dénotée par "Object". La classe "Object" offre des services standards que toute classe doit supporter, comme les mécanismes de synchronisation et de clonage. Le choix d'une telle hiérarchie a l'avantage d'éviter l'utilisation des mécanismes des "templates",

souvent difficiles à gérer et à mettre en ouvre.

Le choix du système ET++ se justifie par les raisons suivantes :

- on y trouve l'implémentation de la plupart des patrons de conception du catalogue de Gamma et al. [GHJV94].
- l'étude des patrons qui composent ce système nous a permis de mieux les comprendre, dans un contexte appliqué.
- Gamma, l'un des inventeurs des patrons de conception est parmi les architectes et les concepteurs clefs de ET++.

7.1.2 ACE

ACE (Adaptative Communication Environment) est un cadre d'application orienté objet implémenté dans les langages C++ et Java. Il est conçu pour les applications de télécommunication. Actuellement, il est utilisé pour développer de nombreuses applications de communication en temps réel. Ce cadre d'application est basé sur une couche adaptative qui adapte les mécanismes de communication fournis avec les systèmes d'exploitation *Unix* et *Windows NT*. L'ensemble des classes adaptatives sert de couche d'abstraction sur laquelle est fondé le cadre. Cette couche contient les mécanismes de gestion de mémoire, de gestion des processus, des fils d'exécution (threads), de multiplexage, etc. Les patrons de conception découverts et conçus pour les systèmes sont, à l'état actuel : *ASX* (Adaptative Service eXecutive), *Acceptor, Connector, Corba Handler* et *Service Handler*. La plupart de ces patrons sont de type architectural selon la classification de Buschmann et al. [BMR+96].

Le système ACE a été conçu par patrons de conception. Les patrons qui le composent sont soit spécifiques au domaine soit empruntés des catalogues déjà existants. L'avantage de ACE du point de vue détection est qu'on a une documentation du système par patron qui sert à utiliser et à comprendre facilement le cadre. On sait, donc, a priori l'existence de

certains patrons et on aimerait bien que les détecteurs les reconnaissent. Par exemple, dans la documentation on sait que les patrons de conception *Template Method, Factory Method,* et *Adapter* ont été utilisés pour implémenter le système.

7.1.3 LEDA

Le système *LEDA* est une librairie orientée objet de types de données et d'algorithmes spécifiques au calcul combinatoire. Elle est implantée dans le langage C++. Elle est composée d'une collection de classes intégrant l'aspect interface usager, les structures de données de base et les mécanismes d'entrées/sorties. La hiérarchie de classes du système est aussi une arborescence, semblable à celle de *ET++* où toutes les classes dérivent d'une classe racine commune dénotée par "Object". La classe "Object" offre des services standards que toute classe doit supporter, comme les mécanismes de synchronisation et de clonage. Nous avons choisi ce système pour comparer nos résultats avec ceux de Prechelt et al. qui l'ont utilisé à leur tour pour valider leurs détecteurs.

7.1.4 xForms

Le système *xForms* est un constructeur d'interfaces usager. Il offre la possibilité de spécifier graphiquement une interface usager en incluant les différentes *widgets* dans une forme. À partir d'une spécification graphique le système génère du code C ou C++ qui implémente cette spécification. Nous avons considéré ce système vu sa simplicité et aussi parce qu'il est implémenté sans la notion de patron à l'esprit. Notre but était de savoir si nos détecteurs seront capables de détecter des patrons dans ce genre de système.

7.2 Analyse et interprétation des expériences

Dans cette section, nous allons analyser et interpréter les résultats fournis par les détecteurs des différents patrons de conception.

7.2.1 Mesures de performances des heuristiques

Dans cette section nous allons introduire des mesures de performances des heuristiques de détection. Ce qui nous intéresse le plus ici est de pouvoir mesurer la puissance de détection d'une heuristique ou d'un ensemble d'heuristiques, et non pas son temps d'exécution. Il est clair donc que ces mesures ne s'appliquent pas aux algorithmes de détection. Les mesures que nous proposons sont immédiates et naturelles. Ces mesures ont été déjà utilisées par Prechelt [KP96] dans le but de mesurer la performance de ses détecteurs. Étant donné un patron de conception p nous définissons les paramètres suivants :

- $I(p)$: le nombre exact des implémentations du patron dans le système. Par exemple, I(Template Method) représente le nombre de fois que le patron de conception "Template Method" a été implémenté, consciemment ou inconsciemment, dans le système.
- $J(p)$: le nombre de candidats fourni par le détecteur. Une fois l'analyseur exécuté, J(Template Method) donne le nombre de fois qu'il a détecté le patron de conception "Template Method".
- $K(p)$: le nombre de vraies occurrences parmi $J(p)$.
- $K'(p)$: le nombre de fausses trouvailles. Ce que le détecteur nous fournit comme résultat peut contenir des candidats qui ne sont pas réellement des implémentations du patron "Template Method".

Les paramètres ainsi définis satisfont les contraintes suivantes :

- $0 \leq K(p) \leq J(p) \leq C$, où C est le nombre de classes dans le système
- $K'(p) \leq J(p)$
- $K(p) \leq I(p)$
- $K'(p) = J(p) - K(p)$
- $I(p) - K(p)$ = nombre d'occurrences manquées

De plus, le paramètre $I(p)$ n'est pas connu a priori. Seul $J(p)$ est fourni par le détecteur. Nous définissons la *précision* des détecteurs comme suit :

$$\text{précision} = \frac{\sum_p K(p)}{\sum_p J(p)}.$$

et le *rappel* ("recall") comme suit :

$$\text{recall} = \frac{\sum_p K(p)}{\sum_p I(p)}.$$

Il faut noter que ses mesures sont définies pour l'ensemble des patrons considérés. Elles changent au fur et à mesure qu'on incorpore d'autres patrons.

Les mesures de performance que nous avons donnée sont triviales. Elles nous informent sur la précision globale de tous les détecteurs. Cependant, comme nous allons le constater dans les chapitres suivants, certains patrons de conception, comme "Factory Method" et "Template Method" se détectent mieux que les autres. Néanmoins, ces mesures sont intéressantes pour comparer entre les différentes techniques et approches de détection. L'effort de validation des candidats fournis par l'ensemble des détecteurs est proportionnel à $\sum_p J(p)$. Ce nombre peut être assez élevé pour des systèmes de taille importante. Nous devons réduire ce nombre pour faciliter la validation, et nous devrons le faire sans perte d'information.

Patron	ET++		ACE		LEDA		Xforms	
	J	K	J	K	J	K	J	K
Abstract Factory	25	3	4	2	0	0	0	0
Adapter	122	0	2	2	1	0	20	0
Bridge	5	0	21	0	17	17	0	0
Chain of Responsability	19	0	39	0	48	0	2	0
Composite	8	0	6	0	2	0	1	0
Decorator	8	0	4	0	3	0	1	0
Factory Method	29	29	7	7	0	0	7	7
Mediator	5	0	2	0	2	0	0	0
Observer	8	0	2	0	2	0	0	0
Proxy	0	0	0	0	1	0	3	0
Singleton	0	0	0	0	0	0	0	0
Template Method	1042	1042	0	0	20	20	7	7
Précision	85%		13%		39%		34%	
Temps d'exécution	900		900		3600		600	
Recall	100%		100%		100%		100%	

TAB. 7.2: Résultats des expériences

7.2.2 Résultats des expériences

Dans cette section nous allons donner les résultats des expériences faites avec les quatre systèmes introduits dans la section 7.1 (voir tab. 7.2).

Dans ce tableau nous présentons les paramètres suivants :

- Précision : calculée pour chaque système de test. Elle mesure la performance des douze détecteurs dans leur ensemble. La formule qui calcule cette mesure est donnée dans la

section 7.2.1.

- Temps d'exécution : c'est le temps moyen d'exécution des détecteurs de chaque patron.
- *Recall* : représente le pourcentage des vraies occurrences, parmi toutes les vraies occurrences, que les détecteurs ont détecté. Dans ce travail nous supposons que les détecteurs captent au moins toutes les vraies occurrences des patrons dans le système. En effet, nos détecteurs sont conçues en se basant sur les diagrammes de structure tels qu'ils sont donnés dans le catalogue de Gamma et al. [GHJV93]. Chaque diagramme de structure d'un patron représente la condition nécessaire pour qu'une structure soit considérée comme ce patron de conception. Ces diagrammes de structure sont ceux utilisés dans la plupart des systèmes qui implantent les patrons de conception de ce catalogue, en particulier les systèmes de test que nous avons utilisés. De plus, Kramer et al. [KP96] ont utilisé la même hypothèse pour évaluer les performances de leurs détecteurs. C'est pour cela que *Recall* est toujours de cent pourcent. L formule qui calcule cette mesure est donnée dans la section 7.2.1.

Tout d'abord, les deux patrons de conception "Factory Method" et "Template Method" ont été détectés dans les quatre systèmes à cent pourcent. En effet, ces deux patrons possèdent une sémantique minimale qui les rend facile à détecter, par rapport aux autres patrons. La détection du patron de conception "Factory Method" s'appuie essentiellement sur la détection de l'opérateur "new" dans le corps d'une fonction. Celui du patron "Template Method" se base sur la détection des appels de méthodes virtuelles dans le corps d'une méthode non-virtuelle. La détection de ces constructions dans un programme est très bien supporté par l'outil d'analyse *Gen++*. La détection de ces deux patrons de conception dans un système est d'une grande importance. Elle nous renseigne, entre autres, sur la flexibilité du système considéré. Pour le moment, nous pouvons dire que le système *ET++* est flexible et extensible. Ce système laisse beaucoup de liberté quant aux choix des implémentations des méthodes.

L'intérêt de ces deux patrons pour promouvoir la flexibilité des systèmes a été bien discuté par [Pre96].

La détection du patron de conception "Observer" dépend du type de système analysé. En effet, ce patron établit une architecture et un cadre dans le système. Donc, si on se place dans le contexte d'un cadre nous ne pouvons voir que les classes abstraites et les mécanismes d'interaction entre elles, à moins que le cadre en fournit des classes concrètes par défaut. Ceci a été constaté dans le système *ET++*, par exemple. Le détecteur du patron *Observer* a été conçu en exploitant le cycle fermé (*ConcreteSubject*, *ConcreteObserver*, *Observer*, *Subject*). Or, les classes *ConcreteSubject* et *ConcreteObserver* appartiennent à l'application utilisatrice du cadre et les classes *Observer* et *Subject* appartiennent au cadre lui-même. Normalement, il y a une seule "instance" du patron dans un système et ceci est très bien justifié par la vocation du patron. La détection de deux "instances" de ce patron dans le même système indique qu'on ne réutilise pas assez bien nos ressources.

L'heuristique de détection du patron de conception *Abstract Factory* utilise la même technique que celle utilisée pour la détection du patron *Factory Method*. On cherche à identifier les classes qui possèdent plus qu'une méthode de création, i.e. *Factory Method*. Nous avons pu détecter 3 vraies occurrences sur 25 dans le système *ET++* et 2 vraies occurrences sur 4 dans le système *ACE*. Par contre, aucune vraie occurrence n'a été détectée dans les autres systèmes. La performance du détecteur de ce patron est assez moyenne. Cependant, la détection d'une vraie occurrence de ce patron nous renseigne sur la flexibilité du système.

La détection du patron de conception "Adapter" nous renseigne sur la réutilisation "inter-système". Ceci concorde très bien avec la vocation et la raison d'être du patron. D'après les résultats fournis par les détecteurs de ce patron, nous constatons que le système *ACE* en implémente deux, mais aucun n'est implémenté par *ET++*. En effet, le système *ACE* est conçu par adaptation de systèmes existants.

Le patron de conception "Bridge" possède une sémantique très importante. On trouve plusieurs applications et implantations du patron dans la littérature [GHJV94, Cop92]. Sa détection nous informe, entre autres, comment les abstractions et les implantations sont séparées dans un système. Ce patron a été détecté dans le système "LEDA" à cent pourcent. Nous pouvons dire que dans "LEDA" les abstractions sont séparées de leurs implantations.

La performance des détecteurs des patrons de conception : *Chain of Responsability*, *Composite*, *Decorator Mediator*, *Observer*, et *Proxy* est très médiocre. Cela est dû essentiellement à l'importance de la composante sémantique de ses patrons. Par exemple, l'heuristique qui permet la détection du patron *Composite* se base sur les mécanismes des templates qui sont mals supportées par l'outil *Gen++*.

Dans le cas du patron *Singleton* aucune vraie occurrence n'a été détectée. Toutefois, on peut dire que sa performance est de cent pourcent. En effet, l'implémentation de ce patron est assez standard et l'heuristique qui permet sa détection repose sur ce standard pour son identification.

Nous remarquons que pour certains patrons de conception le nombre de *false positives* est assez grand. Ceci s'explique par le fait que ces patrons sont très riches en sémantique. Ce qui les rend difficile à formaliser et à modéliser et ainsi les implémenter dans *Gen++*. De plus, notre approche basée sur la structure, prend en compte la sémantique seulement partiellement. Donc, notre détection est basée sur des conditions necessaires mais pas suffisantes. Nous remarquons aussi qu'il n y a pas de *false negatives*. En effet, nos détecteurs se basent sur les structures et les styles telles qu'elles sont publiées dans les catalogues. Nos heuristiques ne détectent que les patrons de conception qui ont été implémenté en respectant ces styles.

Kramer et Prechelt ont développé des détecteurs pour les patrons de conception de type structurel, selon la classification de Gamma et al.. Ils ont utilisé le système *LEDA* pour

Patron	SPOOL		PAT	
	J	K	J	K
Adapter	1	0	1	0
Bridge	17	17	59	10
Composite	2	0	6	0
Decorator	3	0	3	0
Proxy	1	0	1	0
Précision	70%		14%	
Temps d'exécution	2		900	
Recall	100%		100%	

TAB. 7.3: Comparaison avec *PAT*

valider et tester leurs détecteurs. Nous avons repris le même système afin de comparer nous résultats avec les leurs. La table 7.3 en donne un résumé. Nous remarquons que dans tous les cas nos détecteurs performent mieux.

7.3 Problèmes rencontrés

Les problèmes rencontrés lors de ce travail de recherche sont de deux types : techniques et conceptuels. Du point de vue conceptuel, il fallait étudier les patrons de conception et extraire leurs sémantique. Pour certains patrons, "Factory Method" et "Template Method" par exemple, la sémantique est facile à comprendre et exploiter. Pour d'autres, il fallait chercher des artifices "maximaux" qui permettent de caractériser le patron d'une façon "unique".

Du point de vue technique, l'outil *Gen++* ne supporte pour le moment que le compilateur de *C++* de ATT. Or, les systèmes étudiés ne supportent pas nécessairement ce compilateur.

Il a fallu modifier ces systèmes sans toucher à leur sémantique pour les compiler. Par exemple, les mécanismes "avancés" des "template" et des "macros" sont mal supportés par le compilateur de ATT (exigé par *Gen++*). La définition des macros paramètrées n'est pas supportée du tout. Cependant, ce style de programmation est très utile et est utilisé dans les systèmes d'envergure. De plus, ce compilateur ne permet pas la déclaration ou la définition d'une classe générique (template) qui dérive récursivement de l'un de ces paramètres. Ce style de programmation a été utilisé dans les systèmes *ACE* et *LEDA*. De plus, l'outil *Gen++* ne reconnaît pas les instantiations des "templates" et ne permet pas d'accéder à leurs paramètres actuels. Ce besoin est vital pour la détection de la plupart des patrons. De plus, le langage *GENOA* est rudimentaire et n'offre qu'un jeu minimal d'instructions pour la manipulation du graphe abstrait. Du point de vue du génie logiciel, il n'offre que les techniques de réutilisation de base, via la définition des procédures et des fonctions. Il n'offre pas de mécanismes de modularité et de la compilation séparée.

Afin de réaliser notre travail dans un temps acceptable, la liste des patrons de conception considérée n'est pas exhaustive. De plus, comme nous l'avons souligné dans les chapitres 6 et 7, certains patrons de conception possèdent une composante sémantique très importante, ce qui rend l'automatisation de leurs détection délicate.

Dans le calcul de la performance des détecteurs le nombre exact des vraies occurrences, $I(P)$, d'un patron dans un système n'est pas connu à priori. Ceci est intrinsèque au problème de la détection puisqu'on ne connaît pas d'avance le système qu'on désire analyser. Ainsi, la mesure *recall* demeure à son tour inconnue. Dans ce travail nous avons supposé que cette mesure est de cent pour cent. Cette hypothèse est justifiable par le faite que nos détecteurs sont basés sur les diagrammes de structures tels qu'ils sont suggères dans le catalogue de Gamma et al. [GHJV93] et que ces structures sont les plus utilisées par les concepteurs et les programmeurs. Cependant, la connaissance du *recall* est très utile pour évaluer les

performances d'un système de détection.

Chapitre 8

Conclusion

Ce travail de recherche apporte une solution à la problématique de la détection des patrons de conception dans des systèmes orientés objet.

Nous avons discuté dans le chapitre deux le paradigme orienté objet, ses principes et ses propriétés. Le chapitre trois a traité de la description des patrons de conception utilisés dans le paradigme orienté objet. Dans le quatrième chapitre nous avons introduit la notion de qualité des logiciels, en particulier dans le cadre des systèmes orientés objet.

Les trois derniers chapitres forment le noyau de notre contribution. Nous avons commencé par la discussion de la problématique de la détection des patrons. Ensuite, nous avons introduit les différentes approches à la détection retrouvées dans la littérature. L'emphase est mise sur la présentation de notre approche ainsi que la conception de nos heuristiques. De plus, nous avons donné des mesures de performance de nos détecteurs. Dans le chapitre suivant, nous avons introduit l'outil *Gen++* que nous avons utilisé pour l'implémentation des heuristiques. Ensuite, nous avons discuté les implémentations de nos détecteurs.

Les contributions majeures de ce travail résident dans les points suivants. Premièrement, nous avons conçue des heuristiques pour détecter les douze patrons de conception : *Abstract Factory, Adapter, Bridge, Chain of Responsability, Composite, Decorator, Factory Method, Mediator, Observer, Proxy, Singleton*, et *Template Method.* Ces patrons de conception sont

parmi les plus utilisés dans divers systèmes orientés objet. Contrairement aux autres approches décrites dans la littérature, notre approche intégre les trois aspects : structurel, sémantique et stylistique. Ce qui explique en grande partie sa robustesse. Deuxièmement, nous avons implémenté ces heuristiques dans le langage *Gen++* pour détecter ces patrons de conception dans le langage *C++*. Finalement, notre approche a été validée et testée sur des systèmes de taille importante (*ACE, ET++, LEDA*, et *xForms*). La comparaison de la performance de nos détecteurs aux autres dans la littérature [KP96] (§7.2) indique clairement que notre approche performe mieux. Entre autres, nous avons pu réduire, d'une façon significative, le nombre de *false positives* et ainsi la précision de la détection se trouve améliorée. Ceci signifie une réduction du temps d'inspection et de verification des candidats.

Cependant, ce travail peut être étendu et amélioré dans le futur. Nous suggérons comme extensions et améliorations :

- Adaptation des détecteurs pour d'autres langages de programmation, comme le langage *Java* et *Smalltalk*. En effet, la conception des heuristiques donnée dans le chapitre quatre est générique, elle ne dépend pas des langages de programmation. Cependant, certains langages comme Java offrent des mécanismes d'introspection qui facilitent la détection de la structure des patrons de conception.
- Raffinement des heuristiques pour réduire d'avantage l'effort de validation.
- Amélioration de l'outil *Gen++* pour supporter d'autres systèmes de compilation et d'autres langages de programmation.
- L'outil *Gen++* ne supporte pas la compilation séparée comme c'est le cas des langages évolués. En effet, la compilation séparé, i.e. la modularité facilite l'entretien, la maintenance, l'évolution et la réutilisation des détecteurs.
- Utilisation de l'analyse dynamique des programmes pour la détection des patrons. En effet, un patron de conception est décrit et défini par un diagramme de classe et un dia-

gramme de comportement. Le diagramme de classe décrit son comportement statique et le diagramme de comportement en définit son comportement dynamique. Il sera, donc, intéressent d'utiliser des analyseurs dynamiques de programmes pour observer le comportement dynamique d'un patron et élaborer des techniques de détection sous-jacentes. L'intégration des deux techniques donnera naissance à des détecteurs plus puissant et plus robuste.

Bibliographie

[AIS+77] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language.* Oxford University Press, New York, NY, 1977.

[Ale79] Christopher Alexander. *The timeless way of building.* Oxford University Press, New York, NY, 1979.

[And94] Bruce Anderson. Patterns : Building blocks for object-oriented architectures. *ACM SIGSOFT Software Engineering Notes*, 19(1) :47–49, January 1994. OOPSLA'93 Workshop Report.

[Bak95] Brenda S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the Second Working Conference on Reverse Engineering*, pages 86–95, Toronto, Canada, July 1995. IEEE.

[BBM96] Victor R. Basili, Lionel C. Briand, and Walcelio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10) :751–761, October 1996.

[BCC+96] Kent Beck, James O. Coplien, Ron Crocker, Lutz Dominick, Gerard Meszaros, Frances Paulisch, and John Vlissides. Industrial experience with design patterns. In *Proceedings of the Eighteenth International Conference on Software Engineering*, pages 103–114, Berlin, Germany, 1996. IEEE.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture - A System of Patterns.* John Wiley and Sons, 1996.

[Boo94] Grady Booch. *Object Oriented Analysis and Design with Applications.* Benjamin/Cummings Publishing Company Inc., Redwood City, CA, 1994. Second edition.

[BRJ96] Grady Booch, James Rumbaugh, and Ivar Jacobson. The unified modelling language for object-oriented development. documentation set version 0.9 addendum, July 1996. Rational Software Corporation, Santa Clara, CA.

[BRJ97] Grady Booch, James Rumbaugh, and Ivar Jacobson. The unified modelling language for object-oriented development. documentation set version 1.0, January 1997. Rational Software Corporation, Santa Clara, CA.

[Bro97] Kyle Brown. *Design Reverse-Engineering And Automated Design Pattern Detection In Smalltalk.* PhD thesis, NC State University, Raleigh, North Carolina, USA, 1997.

[CK94] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6) :476–493, June 1994.

[Coa92] Peter Coad. Object-oriented patterns. *Communications of the ACM*, 35(9) :152–159, September 1992.

[Cop92] James O. Coplien. *Advanced C++ Programming Styles and Idioms.* Addison-Wesley, 1992.

[CS95] James O. Coplien and Douglas C. Schmidt, editors. *Pattern Languages of Program Design.* Addison-Wesley, 1995. (Reviewed Proceedings of the First

International Conference on Pattern Languages of Programming (PLoP'95), Monticello, IL, 1994).

[CSM+79] B. Curtis, S. B. Sheppard, P. Milliman, M. A. Borst, and T. Love. Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. *IEEE Transactions on Software Engineering*, pages 96–104, March 1979.

[DE94] Premkumar T. Devanbu and Laura Eaves. Gen++ - an analyzer generator for c++ programs. Technical report, ATT Labs, 1994.

[Dev92] Premkumar T. Devanbu. GENOA - a customizable, language- and front-end independent code analyzer. In *Proceedings of the Fourteenth International Conference on Software Engineering*, pages 307–317, Melbourne, Australia, 1992.

[EKS95] Johann Eder, Gerti Kappel, and Michael Schrefl. Coupling and cohesion in object-oriented systems. Technical report, Johannes-Kepler University, Linz, Austria, 1995.

[FMvW97] Gert Florijn, Marco Meijers, and Pieter van Winsen. Tool support for object-oriented patterns. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, Jyväskylä, Finland, June 1997. to appear.

[Gam92] Erich Gamma. *Object-oriented Software Development exemplified with ET++ : Design Patterns, Class Library, and Tools.* Springer-Verlag, Berlin, Germany, 1992. Book version of PhD thesis, University of Zürich, Switzerland, 1991 ; in German.

[GHJV93] E. Gamma, R. Helm, R Johnson, and J. Vlissides. Design patterns : Abstraction and reuse of object oriented design. In *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP'93)*, Kaiserlauten, Germany, July 1993.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[Hal77] M.H Halstead. *Elements of Software Science.* Elsevier, Berlin, Germany, 1977.

[HCN97] Rachel Harrison, S. Counsell, and R. Nithi. An overview of object-oriented design metrics. In *Proceedings of the Eighth International Workshop on Software Technology and Engineering Practice (STEP'97)*, pages 230–235, London, England, July 1997.

[HS96] Brian Henderson-Sellers. *Object-Oriented Metrics. Measures of Complexity.* Prentice-Hall, Inc., 1996.

[JCJO92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Oevergaard. *Object-Oriented Software Engineering : A Use Case Driven Approach.* Addison-Wesley, 1992.

[JEJ94] Ivar Jacobson, Maria Ericsson, and Agneta Jacobson. *The Object Advantage. Business Process Reengineering with Object Technology.* Addison-Wesley, 1994.

[JF88] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 4(2) :22–35, June/July 1988.

[KA96] J. Gosling K. Arnold. *The Java Programming Language.* Addison-Wesley, 1996.

[KP88] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *Journal of Object-Oriented Programming*, 1(3) :26–49, August/September 1988.

[KP96] Christian Kramer and Lutz Prechelt. Design recovry by automated search for structural design patterns in object-oriented software. November 1996.

[KTLD97] Kamel Karoui, Nejmeddine Tagoug, Franccois Lustman, and Rachida Dssouli. Design metrics that predict maintainability. In *Eighth Annual Oregon Work-*

shop on Software Metrics, Coeur d'Alene, ID, May 1997. Precision Software Measurement Products.

[Kuh86] Tomas Kuhn. *Structure des révolutions scientifiques.* 1986.

[LH93] Wei Li and Sallie Henry. Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 23 :111–122, 1993.

[McC76] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4) :308–320, December 1976.

[MLM96] Jean Mayrand, Claude Leblanc, and Ettore M. Merlo. Automatic detection of function clones in a software system using metrics. In *Proceedings of the International Conference on Software Maintenance (ICSM'96)*, pages 244–253, Monterey, CA, November 1996. IEEE.

[MS95] Alberto Mendelzon and Johannes Sametinger. Reverse engineering by visualizing and querying. *Software - Concepts and Tools*, 16(4) :170–182, September–December 1995. Springer-Verlag.

[Nav87] J. K. Navlakha. A survey of system complexity metrics. *The Computer Journal*, 30(3) :233–238, June 1987.

[Pre96] Wolfgang Pree. *Framework Patterns.* SIGS Books and Multimedia, 1996.

[Pre97] Roger S. Pressman. *Software Engineering. A Practitioner's Approach.* McGraw-Hill, fourth edition, 1997. ISBN 0-07-052182-4, Math-Info QA 76.6 P74 1997 (RESERVE IFT 2240).

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-oriented Modeling and Design.* Prentice-Hall, Inc., 1991.

[Sch95a] Douglas C. Schmidt. Acceptor and Connector : Design Patterns for Active and Passive Establishment of Network Connections. In *Workshop on Pattern Languages of Object-Oriented Programs at ECOOP '95*, Aarhus, Denmark, August 1995.

[Sch95b] Douglas C. Schmidt. Experience Using Design Patterns to Develop Reuseable Object-Oriented Communication Software. In *Communications of the ACM (Special Issue on Object-Oriented Experiences)*, volume 38, October 1995.

[Sch95c] Douglas C. Schmidt. Reactor : An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design.* Addison-Wesley, Reading, MA, June 1995.

[Sch96] Stephen R. Schach. The cohesion and coupling of objects. *Journal of Object-Oriented Programming*, 8(8) :48–50, January 1996.

[SS95a] Douglas C. Schmidt and Paul Stephenson. Experiences Using Design Patterns to Evolve System Software Across Diverse OS Platforms. In *Proceedings of the* 9^{th} *European Conference on Object-Oriented Programming*, Aarhus, Denmark, August 1995.

[SS95b] Douglas C. Schmidt and Paul Stephenson. Using Design Patterns to Evolve System Software from UNIX to Windows NT. *C++ Report*, 7(3), March/April 1995.

[TK96] Jean Tessier and Rudolf K. Keller. Manager-Agent and Remote Operation : Two key patterns for network management interfaces. In *Collected Papers from the PLoP'96 and EuroPLoP'96 Conferences, Washington University Department of Computer Science, wucs-97-07*, pages 4.8.1–4.8.14, Monticello, IL, September 1996.

[WBVC+90] Allen Wirfs-Brock, John Vlissides, Ward Cunningham, Ralph Johnson, and Lonnie Bollette. Designing reusable designs : Experiences designing object-oriented frameworks. In *Proceedings of the Conference on Object-Oriented Programming : Systems, Languages and Applications, Addendum to the Proceedings*, pages 19–24, Ottawa, Canada, October 1990. Panel Discussion.