Université de Montréal

Support informatique à la compréhension des logiciels orientés objet de taille industrielle

par

Sébastien Robitaille

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
Maître ès sciences (M.Sc)
en informatique

Avril, 2000

Université de Montréal
Faculté des études supérieurs

Ce mémoire intitulé :

Support informatique à la compréhension des logiciels orientés objet de taille industrielle

présenté par :

Sébastien Robitaille

a été évalué par un jury composé des personnes suivantes:

Président-rapporteur :	Jean Vaucher
Directeur de recherche :	Rudolf K. Keller
Membre du jury :	Houari Abdelkri Sahraoui

Mémoire accepté le :.....

Sommaire

L'étude du code source est une des activités les plus fréquentes durant l'entretien et l'évolution des logiciels de grande taille. Le manque de support de la part des outils présentement disponibles rend cette tâche très difficile. Entre autres, la navigation et la recherche à travers le code source sont des activités souvent mal supportées par ces outils : La navigation s'effectue en général en manipulant des éléments de bas niveau, l'exploration est supportée comme une activité à sens unique, et ces outils fournissent très peu d'aide pour récupérer de l'information à un plus haut niveau d'abstraction.

Dans cette recherche, nous avons développé une approche d'aide à la compréhension des logiciels de taille industrielle, basée entre autres sur l'utilisation des concepts orientés objet et sur l'utilisation des patrons de conception. Cette approche vise à permettre la navigation dans différents contextes d'un logiciel et à différents niveaux d'abstraction en s'appuyant sur des outils informatiques appropriés. C'est pourquoi plusieurs outils ont été développés pour valider notre approche.

L'utilisation de notre approche, par l'entremise des outils développés dans le cadre de ce travail, permet d'accélérer significativement le processus de compréhension des logiciels. Les quatre études de cas présentés dans ce mémoire ainsi que l'évaluation qui est faite de l'approche et des outils permettent de montrer comment la compréhension de logiciels peut être accélérée.

Mots clés : génie logiciel, rétro-ingénierie, compréhension des logiciels, outil, système orienté objet, patron de conception, navigation, recherche, visualisation, design.

Table des matières

Liste des tableaux

Liste des figures

Liste des sigles et abréviations

CASCON	Center for Advanced Studies Conference
CSER	Consortium for Software Engineering Research
DnD	Drag & Drop
ICSE	International Conference on Software Engineering
ICSM	International Conference on Software Maintenance
NRC	National Research Council of Canada
NSERC	National Sciences and Research Council of Canada
OSI	Open Systems Interconnection
SPOOL	Spreading Desirable Properties into the Design of Object-Oriented, Large-Scale Software Systems
TCP/IP	Transfer Control Protocol / Internet Protocol
UML	Unified Modeling Language

Je dédie ce mémoire

à

ma mère Micheline

Remerciements

Je tiens à remercier, Monsieur Rudolf K. Keller, professeur à l'Université de Montréal, pour avoir dirigé et supervisé la rédaction de ce mémoire, ainsi que pour m'avoir permis de faire partie de l'équipe du projet SPOOL. Sa patience, son esprit critique ainsi que sa grande attention aux détails m'ont été d'une aide inestimable tout au long de ce travail.

Je me dois de remercier Monsieur Reinhard Schauer pour ses enseignements, ses encouragements et son amitié.

Mes remerciements vont aussi à l'équipe d'analyse de la qualité chez Bell Canada pour leurs encouragements ainsi que leur intérêt marqué envers mon travail.

Le projet SPOOL est organisé par CSER (Consortium for Software Engineering Research), qui est financé par Bell Canada, NSERC (National Sciences and Research Council of Canada) et NRC (National Research Council of Canada). Je me dois aussi de remercier les membres de ces organismes qui ont rendu possible la réalisation de ce travail.

Enfin, je tiens à remercier ma conjointe Jolaine pour son amour, sa patience ainsi que pour les longues soirées et les fins de semaine qu'elle a passées sans moi.

Chapitre 1 : Introduction

La compréhension du code source est une activité qui joue un rôle essentiel et dominant durant la maintenance et l'évolution des systèmes logiciels. Il s'agit d'une activité qui peut prendre beaucoup de temps aux programmeurs, spécialement lorsque ceux-ci sont nouveaux dans un projet ou lorsque le système sur lequel ils travaillent est très grand. Cela peut même avoir des conséquences graves sur l'évolution de certains systèmes. Zawinski, par exemple [46], attribue une des causes principales du ralentissement du projet *Netscape Communicator* au fait qu'il est très difficile pour les nouveaux programmeurs de comprendre ce gros système et que c'est d'autant plus difficile pour eux de rapidement se mettre à contribuer au projet.

1.1 Contexte de ce travail

Le travail présenté dans ce mémoire a été fait dans le cadre du projet SPOOL (*Spreading Desirable Properties into the Design of Object-Oriented Large-scale Software Systems*), un projet CSER (*Consortium for Software Engineering Research*) qui est effectué en collaboration avec Bell Canada. Dans ce projet on s'intéresse principalement à identifier les propriétés désirables dans la conception des systèmes orientés objet à grande échelle, et ce, pour être en mesure d'évaluer leur qualité. C'est pourquoi, dans le cadre de ce projet, un environnement de rétro-ingénierie a été développé pour fournir l'infrastructure informatique nécessaire à l'étude des activités de maintenance, d'évaluation et de compréhension des logiciels orientés objet de grande taille. Ce mémoire s'inscrit donc dans ce contexte en présentant une nouvelle approche de compréhension des logiciels, ainsi que la description de deux outils, le *Design Browser* et le *Context*

Viewer, qui ont été développés spécialement dans le but de supporter cette approche.

1.2 Structure de ce mémoire

Le Chapitre 2 présente différents aspects théoriques reliés à la compréhension de logiciels. Entre autres, il discute de plusieurs modèles cognitifs ainsi que de différents aspects de la compréhension reliés au paradigme orienté objet et à l'utilisation des patrons de conception. Ce chapitre fait aussi un survol des outils d'aide à la compréhension qui sont disponibles et enfin, il présente la problématique qui a donné lieu à ce travail.

Le Chapitre 3 décrit notre approche à la compréhension de logiciels. Cette approche est basée sur la mise en différents contexte de l'information tirée du code source, en permettant la visualisation de cette information à différents niveaux d'abstraction, tels que le niveau structurel ou le niveau de la conception. Ce chapitre fournit aussi une liste de fonctionnalités souhaitables pour des outils supportant notre approche et en décrit ainsi les besoins.

Le Chapitre 4 décrit l'environnement de rétro-ingénierie SPOOL, ainsi que les outils développés dans le but de supporter l'approche décrite au Chapitre 3.

Le Chapitre 5 présente quatre études de cas visant à montrer l'usage qui peut être fait des différents outils de l'environnement SPOOL, et comment ceux-ci peuvent aider à supporter l'approche décrite. Ces quatre exemples d'utilisation se situent à différents niveaux d'abstraction et illustrent donc les aspects principaux de l'approche.

Le Chapitre 6 présente une évaluation de l'approche ainsi que de l'environnement SPOOL et des outils conçus dans le cadre de ce mémoire. Il discute aussi les avantages et limites de l'approche et de l'environnement.

Enfin, le Chapitre 7 résume brièvement ce travail et les contributions majeures, et une discussion des travaux futurs conclut la présentation.

L'Annexe présente les aspects importants de la conception du cadre d'application SPOOL ainsi que l'utilisation qui en a été faite pour développer les outils nécessaires au support de l'approche par mise en contexte.

1.3 Contributions principales

Ce travail apporte deux contributions majeures. Premièrement, l'approche de compréhension présentée dans ce mémoire s'appuie sur des concepts peu exploités à des fins de compréhension jusqu'à présent. La mise en contexte basée sur l'utilisation des concepts du paradigme orienté objet et des patrons de conception est une approche innovatrice qui n'a guère été explorée jusqu'à présent. Notre recherche promet donc d'ouvrir de nouvelles possibilités en termes d'aide à la compréhension à condition que nos concepts soient implémentés dans les environnements de développement disponibles aux programmeurs.

La deuxième contribution majeure de ce travail consiste en la définition et la description des besoins pour le développement d'outils d'aide à la compréhension, en plus de fournir une implémentation modèle répondant à ces besoins. Notons que les deux outils implémentés dans le cadre de ce mémoire et décrits au Chapitre 4 (le *Design Browser* et le *Context Viewer*) ont suscité l'intérêt de l'équipe d'analyse de la qualité de Bell Canada, le partenaire industriel dans le projet SPOOL. Conséquemment, les outils ont été installés chez Bell Canada et seront évalués dans un contexte industriel, constituant ainsi une autre contribution notable de ce mémoire.

Différents aspects du travail présenté dans ce mémoire ont donné lieu à des publications. Entre autres, les contributions majeures de ce mémoire sont résumées dans un article accepté à la conférence internationale ICSM'2000 [27].

ICSM'2000 : *Bridging Program Comprehension Tools by Design Navigation*, mis en nomination pour le prix du meilleur article [27].

CSER : *The SPOOL Design Repository: Architecture, Schema, and Mechanisms* [16].

ICSM'99 : *Hot Spot Recovery in Object-Oriented Software with Inheritance and Composition Template Methods*, Récipiendaire du prix du meilleur article (commandité par *Journal of Software Maintenance*) [28].

ICSE'99 : *Pattern-based Reverse Engineering of Design Components* [19].

Chapitre 2 : La compréhension de logiciels de grande taille

Comprendre un système logiciel est une activité difficile. Elle l'est d'autant plus si le logiciel en question est de taille industrielle, s'il a été écrit par plusieurs personnes et dans des styles variés, s'il fait partie d'un domaine d'application qui ne nous est pas familier, etc. Tous ces critères ont fait que plusieurs théories ont vu le jour. Celles-ci tentent d'expliquer les processus de compréhension utilisés par les humains et tentent d'identifier et de rationaliser leurs étapes pour être en mesure de développer des outils qui supportent mieux les processus en question. Nous verrons donc, dans les sections qui suivent, quelques-uns de ces modèles théoriques, certaines approches de compréhension ainsi que différents outils commerciaux et académiques qui ont été développés dans le passé pour supporter ces approches. Nous discuterons ensuite la problématique liée à l'utilisation de ces outils ainsi qu'au manque de support pour la compréhension des logiciels orientés objet de grande taille.

2.1 Bases théoriques de la compréhension

On peut regarder le problème de la compréhension de logiciels sous plusieurs angles. En effet, jusqu'à présent, plusieurs processus de compréhension ont été décrits dans la littérature et ceux-ci sont en général influencés par beaucoup de facteurs. Les facteurs en question peuvent être de nature cognitive ou être liés à l'approche de compréhension elle-même. Ces facteurs peuvent aussi être liés au paradigme de programmation et au langage utilisés, et peuvent aussi dépendre d'éléments imprévisibles comme le style de programmation utilisé ou la qualité

de la conception d'un système. Dans cette section nous verrons donc certains de ces aspects plus en détail.

2.1.1 Aspects cognitifs

Parmi les modèles de compréhension qui ont été décrits et cités dans la littérature au cours de la dernière décennie, on trouve entre autres des modèles de type *bottom-up,* comme celui décrit par Pennington [25] par exemple. Ces modèles présentent la compréhension comme un processus partant des détails de bas niveau pour finalement remonter à des niveaux plus abstraits. D'autres modèles inverses, de type *top-down,* sont aussi décrits dans la littérature. Soloway et Ehrlich [33], par exemple, ont observé que ce type d'approche est souvent utilisé lorsque le code source est familier. Selon eux, le modèle mental du programmeur se construit de façon *top-down* en créant préalablement une hiérarchie de concepts qu'on retrouve dans le code source. Le modèle de Brooks [4] est un autre modèle de type *top-down* bien connu. Ce dernier présente le processus de compréhension comme une reconstruction du domaine de l'application, suivi d'une association de ce domaine avec les éléments du code source en vérifiant la véracité ou non d'une série d'hypothèses. D'autres modèles, comme celui décrit par Littman [22], sont plus axés sur la distinction entre l'approche du style « au besoin » et l'approche systématique de compréhension.

D'autre auteurs ont combiné différentes approches. C'est entre autres le cas du modèle de Soloway et al. [34], qui fait appel aux concepts d'approche systématique et d'approche au besoin, mais de façon combinée, en utilisant les notions de micro-stratégie et macro-stratégie. Letovsky [20] présente de son coté un modèle combinant les approches *top-down* et *bottom-up* pour décrire le processus d'assimilation du programmeur. Ce composant de son modèle décrit comment le modèle mental du programmeur évolue lorsqu'il utilise sa base de connaissance propre, ainsi que le code source et la documentation du système. Dans le modèle de Letovsky, c'est le programmeur qui décide si son approche de

compréhension est *top-down* ou *bottom-up* en fonction de ses connaissances du système.

Enfin, Von Mayrhauser et Vans [42][43] ont créé un métamodèle de compréhension qui intègre plusieurs aspects des modèles décrits précédemment. Ce métamodèle est constitué en fait de quatre composants. Les trois premiers sont des modèles permettant d'aborder le problème de compréhension à de différents niveaux d'abstraction. Les niveaux sont le domaine, le programme et la situation. Le quatrième composant constitue en fait la base de connaissance nécessaire à la construction des trois modèles précédents. La compréhension se fait donc, selon Von Mayrhauser et Vans, en utilisant chacun de ces modèles au besoin (i.e. au moment jugé opportun par le programmeur).

Les différents types de modèles cognitifs que nous venons de voir ont certaines différences mais ils ont aussi beaucoup de points en commun. Storey [36] a extrait, à partir de ces différents modèles, une liste d'éléments qui influencent les stratégies de compréhension des programmeurs qui font la maintenance de systèmes logiciels. Cette liste se divise en trois types d'éléments : Les caractéristiques du programmeur, les caractéristiques du programme à modifier et les caractéristiques de la tâche à effectuer. À partir de ces différentes caractéristiques, Storey a élaboré et organisé une hiérarchie d'éléments cognitifs, dans le but de guider le développement d'outils d'aide à l'exploration et la compréhension de logiciels. Cette hiérarchie est séparée en deux grandes branches. La première traite de l'amélioration de la compréhension et décrit différents éléments facilitant l'approche *bottom-up*, l'approche *top-down* ainsi que l'intégration de ces deux approches. La deuxième branche traite de la réduction de l'effort cognitif du programmeur en présentant les éléments nécessaires pour faciliter la navigation, pour fournir des indices d'orientation et pour réduire la désorientation. Cet arbre d'éléments cognitifs donne donc une base pouvant servir à la spécification de différents outils d'aide à la compréhension.

Selon Sim et al. [30], le code source d'un système logiciel peut être considéré comme étant un espace d'information complexe. Cette approche permet d'appliquer au domaine des logiciels certains résultats tirés du domaine de la recherche d'information. Par exemple, on remarque qu'il y existe deux styles de navigation: la recherche de quelque chose de précis dans un espace d'information (*searching*) ou bien l'exploration de cet espace sans but précis (*browsing*). Dans le cas des logiciels, il s'agit en fait de deux stratégies de compréhension qui peuvent être utilisées pour supporter certains des modèles mentionnés précédemment, dont l'approche de Von Mayrhauser et Vans qui combine les stratégies *top-down* et *bottom-up*.

2.1.2 Concepts orientés objet

Les langages de programmation ont beaucoup évolués depuis leur invention. Aujourd'hui, on retrouve des langages fournissant un support à différents types de programmation. Entre autres, on trouve des langages supportant la programmation logique comme Prolog, la programmation fonctionnelle comme Lisp ou Scheme, la programmation impérative comme Fortran, Pascal ou C, et finalement, la programmation orientée objet comme C++, Smalltalk ou Java. Ces paradigmes et langages introduisent tous certaines formes de sémantique dans les logiciels par les concepts qui les composent. Par exemple, la programmation logique introduit le concept de prédicat, la programmation orientée objet introduit le concept d'objet, etc. Par conséquent, c'est la nature du paradigme et du langage utilisés pour développer un système qui déterminent quelle information sémantique se retrouve directement dans le code source. Cette information est par conséquent essentielle à la compréhension d'un tel système puisqu'elle est contenue à même le langage de programmation utilisé.

Les notions d'abstraction et d'encapsulation, qui sont directement supportées par le paradigme orienté objet (entre autres par les concepts de classes, d'objets et d'héritage), sont d'une grande aide à la compréhension puisqu'elles sont physiquement représentées dans le code source et peuvent être récupérées

facilement (et même automatiquement). Comprendre un programme écrit dans un langage de bas niveau (comme Assembleur ou C par exemple) peut être une activité très ardue, premièrement parce qu'à la limite il est nécessaire de connaître et de comprendre tous les détails concernant l'architecture de la machine (registres, mémoire, etc.), mais aussi et surtout parce que les concepts importants implantés dans le programme ne sont pas reflétés directement dans le code source. Comprendre de tels programmes nécessite un effort d'abstraction beaucoup plus grand de la part du programmeur que si les différents concepts implémentés pouvaient être extraits du code directement, comme c'est le cas avec les langages orientés objet.

En fait, les concepts d'abstraction et d'encapsulation apportent de l'aide lors de la conception de logiciels, mais ils facilitent aussi la compréhension des logiciels puisqu'ils permettent de découper un système en différents modules qui peuvent être compris les uns indépendamment des autres. Meyer présente la *compréhensibilité modulaire* [23] d'un système comme étant un critère très important, surtout durant les activités de maintenance où les programmeurs doivent travailler avec du code déjà existant. C'est pourquoi il est important que certains des modules d'un système puissent être identifiés directement ainsi que les concepts qu'ils représentent. Les concepts de classes et d'objets présents dans les différents langages orientés objet permettent de récupérer facilement certains de ces modules (les classes, par exemple) et donc facilitent la compréhension de systèmes écrits dans de tels langages.

Comme les concepts d'abstraction et d'encapsulation sont directement supportés par les langages orientés objet, ceci peut faciliter la compréhension des programmes écrits dans ces langages. En fait, les concepts présents dans un langage donnent toujours une certaine sémantique aux éléments d'un programme. Cette sémantique peut être utilisée pour récupérer de l'information importante sur un système, et donc, en faciliter la compréhension. Pour ce qui est des langages orientés objet, on peut par exemple citer l'héritage qui apporte une aide à l'abstraction en permettant la généralisation et la spécialisations de concepts. Le

polymorphisme apporte aussi une certaine sémantique aux programmes puisqu'il permet que la spécification d'une méthode dans le programme ait plusieurs implémentations. La visibilité des attributs et des méthodes d'une classe en C++ ou en Java par exemple, permet de mieux discerner l'encapsulation de certains éléments d'un programme.

Dans la section précédente nous avons vu les aspects cognitifs de la compréhension de logiciels. Plusieurs des modèles présentés font appel à la notion d'abstraction comme partie intégrante du processus de compréhension. Le support direct de cette notion par un langage de programmation, comme c'est le cas pour les langages orientés objet, peut donc être grandement utile à la compréhension de programmes écrits dans ce langage.

2.1.3 Les patrons de conception

Faire la conception d'un système est une chose difficile. C'est d'autant plus difficile si le concepteur a peu d'expérience car il doit trouver de nouvelles solutions à tous les problèmes qu'il rencontre lors de la conception. Même les concepteurs experts sont confrontés à de nouveaux problèmes mais ce qui fait d'eux des experts, c'est justement leur connaissance de certaines solutions aux problèmes courants et leur aptitude à appliquer ces solutions à chaque fois qu'un problème déjà résolu se présente à nouveau.

Un patron de conception peut être vu comme la description d'un de ces problèmes et la description de sa solution. Évidemment, on parle de patron uniquement si le problème en question est récurrent et si la solution proposée est assez générale pour permettre de la réutiliser dans d'autres contextes. En fait, il existe plusieurs types de patrons, à différents niveaux d'abstraction. Par exemple, Buschmann et al. [5] font la distinction entre les patrons architecturaux (*architectural patterns*), les patrons de conception (*design patterns*) et les idiomes de programmation. Les trois paragraphes qui suivent présentent les définitions données par Buschmann et al. à chacun de ces types.

Les patrons architecturaux sont des modèles pour des architectures logicielles concrètes et sont donc des patrons à grande échelle. Ils spécifient les différents sous-systèmes et différents composants d'un système, leurs responsabilités, ainsi que les règles pour organiser les relations entre eux. Le catalogue de Buschmann et al. présente plusieurs patrons architecturaux. *Layer* en est un exemple. Celui-ci est en fait la formalisation du modèle en couches souvent utilisé lors de la définition de protocoles de réseau (TCP/IP ou OSI par exemple). *Reflexion*, *Broker* et *Pipes and Filters* sont d'autres patrons architecturaux relativement bien connus qui sont aussi décrits dans ce catalogue.

Les patrons de conception fournissent un schéma pour raffiner les sous-systèmes ou les composants d'un système logiciel. Tel que défini par Gamma et al.[12], un patron de conception décrit une structure de composants communiquant entre eux et solutionnant un problème de conception récurrent. Ce type de patrons est probablement le plus répandu et il existe des centaines d'exemples dans différents catalogues [5][12] [29], touchant des domaines très variés. Parmi les plus connus, on trouve entre autres *Observer*, *Bridge*, *Iterator*, *State*, *Strategy* et *Visitor* [12].

Les idiomes de programmation sont des patrons de bas niveau, spécifiques au langage de programmation utilisé. Plus précisément, ils décrivent comment implanter les aspects particuliers des composant d'un système, en utilisant les fonctionnalités fournies par le langage. Il existe un exemple bien connu qui est décrit dans le catalogue de Buschmann et al. Il s'agit de l'idiome *Counted Pointer* [5] qui permet de gérer plus facilement la mémoire allouée dynamiquement à des objets partagés en C++. Notons qu'il existe aussi beaucoup d'autres idiomes et pour différents langages, comme par exemple, ceux décrits par Jim Coplien pour le langage C++ [7].

L'utilisation de ces patrons lors de la conception de systèmes logiciels peut être très utile car ceux-ci permettent de trouver rapidement des solutions bien établies aux problèmes rencontrés. Pour ce qui est de la compréhension d'un système, si celui-ci a été développé en utilisant certaines de ces solutions, leur identification

permet de rapidement comprendre les parties du système en question et d'avoir une vue de plus haut niveau de ses parties. L'identification des éléments qui collaborent entre eux pour former cette solution permet aussi de déterminer rapidement qui fait quoi et permet de mieux comprendre leurs interrelations.

2.2 Outils et environnements

Comprendre de grands systèmes logiciels sans utiliser d'outils d'aide à la compréhension est une activité très difficile. Imaginons qu'un programmeur doive s'attaquer à un gros système de plusieurs millions de lignes de code réparties dans plusieurs milliers de fichiers, et qu'il doive n'utiliser qu'un éditeur de texte. La tâche à entreprendre est alors gigantesque, et ceci même si le programmeur connaît déjà certaines parties du système. Évidemment, s'il existe de la documentation sur la partie du système qu'il doit modifier, ceci peut lui faciliter grandement la tâche, mais l'expérience montre que dans les gros systèmes la documentation est rarement à jour, voire souvent inexistante. C'est pourquoi des outils d'aide à la compréhension sont nécessaires et pourquoi beaucoup de personnes travaillent à développer de tels outils.

Outils commerciaux	**Outils académiques**
Discover [9]	*SHriMP* [36][37][38]
Visual Age for Java [41]	*The Portable Bookshelf* [10][14][30]
SNiFF+ [32]	*Rigi* [44]
Source Navigator [35]	*TkSee* [31]

Tableau 1 : Outils commerciaux et académiques

Le Tableau 1 présente certains des outils commerciaux et académiques qui sont représentatifs et dont un des buts principaux est de faciliter la compréhension des systèmes logiciels. Ces outils sont décrits dans les sous-sections qui suivent mais les résultats de leur évaluation ne seront présentés qu'au Chapitre 6 puisque les

critères utilisés pour cette évaluation sont en grande partie basés sur l'approche de compréhension qui sera présentée au Chapitre 3.

2.2.1 Outils commerciaux

Les outils commerciaux développés spécifiquement pour l'aide à la compréhension des systèmes logiciels sont plutôt rares. C'est pourquoi seuls quelques environnements de développement ayant des fonctionnalités pouvant aider à la compréhension seront décrits dans cette section.

Le premier, *Discover* [9], est un environnement d'analyse et de développement qui est commercialisé par la compagnie *Software Emancipation Technology.* C'est en fait, selon la documentation disponible [1][2][39], plus qu'un simple environnement de développement. Il s'agit d'un environnement fournissant une gamme d'outils d'aide aux différentes personnes impliquées dans le développement d'un système logiciel. L'environnement est séparé en plusieurs ensembles d'outils : Par exemple, un de ces ensembles est destiné aux gestionnaires de projets, un autre comprend des outils pour aider l'architecte système, etc. L'ensemble qui nous intéresse plus particulièrement dans le cas de la compréhension de logiciels est l'ensemble qui est destiné aux développeurs. Cet ensemble contient plusieurs outils dont un module d'analyse d'impact de changement et un outil appelé le *Visual Navigator* qui permet de naviguer et faire des recherches dans un système en exécutant des requêtes prédéfinies. Cet outil sert en fait d'interface pour naviguer dans le modèle du code source, qui est stocké dans la base de données de l'environnement.

Le deuxième environnement de développement, *Visual Age for Java* [41], est commercialisé par *IBM Corp.* et intègre plusieurs fonctionnalités comme la gestion automatique de versions par exemple. De plus, le code source des systèmes développés avec cet environnement n'est pas stocké dans des fichiers comme à l'ordinaire. Il est en fait sauvegardé dans la base de données de l'environnement. Du point de vue de la compréhension logicielle, cet environnement fournit certains outils de recherche ainsi que certains diagrammes

permettant de visualiser la hiérarchie de classes ou le graphe d'appels des méthodes par exemple.

L'environnement *SNiFF+* [32] de la compagnie *WindRiver Systems inc.*, ainsi que le *Source Navigator* [35] de *RedHat.com* sont deux environnements de développement fournissant des fonctionnalités qui permettent d'améliorer la compréhension de systèmes logiciels. Entre autres, un diagramme de classes est disponible dans les deux environnements ainsi que de puissants outils pour la recherche d'éléments et pour l'affichage des références croisées.

Cette liste d'environnements n'est évidemment pas exhaustive. Par contre, ces outils reflètent bien l'état de l'art dans le milieu commercial en ce qui a trait au développement et à la compréhension de systèmes logiciels.

2.2.2 Outils académiques

Il y a un grand nombre d'outils académiques qui ont été développés dans le but d'aider et de faciliter la compréhension de logiciels. Ceux-ci abordent plusieurs aspects de la compréhension des systèmes logiciels et ce, pour différents langages et pour différents paradigmes de programmation. Comme il est impossible de présenter tous ces outils, la liste qui suit décrit uniquement quelques-uns d'entre eux qui sont représentatifs de l'état de l'art pour la compréhension des systèmes logiciels.

Le premier outil, *SHriMP* [36][37][38], supporte la navigation d'un système en utilisant des hyper-liens insérés directement dans le code source, un peu à la manière de LXR [21]. De plus, chaque fragment de code source est présenté à l'intérieur des nœuds d'un graphe représentant le système. L'agrandissement de ces nœuds, et donc la visualisation du code source, se fait de façon « emboîtée » par rapport aux autres nœuds du graphe et de façon animée. Cela permet donc de naviguer à travers le système en gardant le contexte visuel de l'endroit où on se trouve.

The Portable Bookshelf [10][14][30] est un environnement académique qui a été conçu dans le but de fournir un format standardisé de documentation aux systèmes logiciels. L'information sur un système est disponible sous la forme d'une page web et peut donc être accédée à partir d'un navigateur comme Netscape. L'environnement permet de naviguer à l'intérieur de diagrammes (appelés *Landscapes*) qui affichent les différents sous-systèmes et fichiers d'un système graphiquement ainsi que certaines des dépendances entre eux, comme les appels de fonctions par exemple. La navigation à l'intérieur de tels diagrammes est assez interactive puisque ceux-ci sont affichés dans le navigateur par un applet Java et parce que des menus contextuels sont disponibles.

Rigi [44] est un outil qui permet la visualisation de diagrammes représentant de façon abstraite les différents aspects d'un logiciel (sous-systèmes, fichiers, etc.). Par contre, contrairement aux outils précédents, celui-ci ne fournit pas de mécanisme de recherche et peut donc être utilisé comme documentation d'un système mais pas comme outil interactif de recherche et navigation.

TkSee [31] est un autre outil pouvant servir à la compréhension de logiciels. Celui-ci fournit des mécanismes de recherche et de navigation intégrés et permet l'affichage du code source.

2.3 Énoncé de la problématique

Dans les sections précédentes, nous avons vu des modèles cognitifs de compréhension logicielle ainsi que certains aspects du paradigme orienté objet et des patrons de conception dans une optique de compréhension. Finalement, nous avons discuté de plusieurs outils développés dans le but de supporter la compréhension logicielle. Malgré le fait que plusieurs outils existent, peu d'entre eux s'attaquent au problème de la compréhension des systèmes logiciels orientés objet. Ceux-ci ne tirent pas avantage non plus de l'information supplémentaire qui est disponible dans le code source de ces systèmes, contrairement à ceux qui sont écrits dans des langages impératifs par exemple. De plus, les outils disponibles

offrent peu de support et peu de flexibilité pour ce qui est de la navigation dans de grands systèmes en plus de ne fournir aucun support de navigation au niveau de la conception du système.

La compagnie de télécommunications Bell Canada dépense plusieurs millions de dollars chaque année pour l'achat et l'entretien de systèmes logiciels écrits en C++, grands de plusieurs millions de lignes de code. Bell Canada veut donc être en mesure de s'assurer que les logiciels en question sont d'une certaine qualité, qu'ils pourront être entretenus et que de nouvelles fonctionnalités pourront être ajoutées dans le futur, vu la croissance rapide du domaine des télécommunications. Il y a donc une équipe d'assurance de la qualité chez Bell Canada qui utilise plusieurs outils et techniques pour évaluer ces systèmes avant leur achat initial, mais aussi pendant leur développement et pendant toutes les phases d'évolution de ces systèmes. Entre autres, les membres de cette équipe utilisent des métriques et des outils graphiques pour faire l'étude des systèmes à un haut niveau d'abstraction. Ils doivent aussi souvent naviguer et rechercher certains éléments dans le code source pour être en mesure de vérifier et de valider certains des résultats obtenus à l'aide de leurs outils. Ils ont donc besoin de techniques pour faire ces vérifications, mais ils ont aussi besoin d'un environnement qui leur permettrait de faire des recherches dans les systèmes et qui rendraient la navigation rapide à travers leurs différents modules, de façon à pouvoir comprendre les détails de la conception originale du système ainsi que certaines des décisions prises par les programmeurs.

L'approche qui sera proposée dans le prochain chapitre vise donc à mieux supporter la compréhension de grands systèmes orientés objet comme on en retrouve en industrie, et de donner des spécifications permettant de développer des outils supportant cette nouvelle approche.

Chapitre 3 : Une approche par mise en contexte

Dans le chapitre précédent, plusieurs modèles cognitifs de compréhension ont été présentés. Les prochaines sections présentent une approche de compréhension par mise en contexte qui est basée sur certains aspects de ces modèles ainsi que sur des concepts tirés du paradigme orienté objet et des patrons de conception. Ce qui est nouveau dans l'approche proposée est que celle-ci est fortement couplée à l'utilisation d'un environnement d'aide à la compréhension. En effet, les différentes démarches proposées dans cette approche nécessitent un support considérable en termes d'outils. De plus, cette approche fait appel aux concepts orientés objet mais fait aussi intervenir la récupération semi-automatique d'éléments liés à la conception des systèmes, ce qui est inexistant dans les environnement de développement qui sont disponibles présentement.

3.1 Démarche et processus

On peut introduire quatre niveaux d'abstraction pour un système logiciel donné. Le code source est le plus bas niveau d'abstraction, c'est-à-dire qu'il représente le niveau le moins abstrait et qu'il est constitué de tous les détails sur le système. À un niveau plus élevé on peut faire abstraction des boucles, des conditions et des détails de bas niveau pour représenter le code source comme un ensemble d'éléments (cf. Figure 1; Éléments du système), comme les classes, structures, fonctions, variables, etc. À un niveau encore plus élevé, on peut voir la structure du système logiciel en terme des différentes relations qui relient les éléments du code source (cf. Figure 1; Structure du système), comme l'héritage, les appels de fonctions, les références, etc. Certaines parties de ces structures font intervenir

plusieurs éléments qui collaborent d'une façon particulière et qui font donc partie de certains concepts liés à la conception du système (cf. Figure 1; Conception du système). Cela représente un niveau d'abstraction encore supérieur. À un niveau encore plus haut, lorsque ces différents concepts sont mis ensemble et que le système au complet peut être vu comme une collaboration de sous-systèmes, on parle du niveau architectural du système. La Figure 1 schématise les trois niveaux qui vont nous intéresser par la suite (i.e. les éléments du code source, la structure logique et physique du système ainsi que sa conception en terme de collaborations d'éléments).

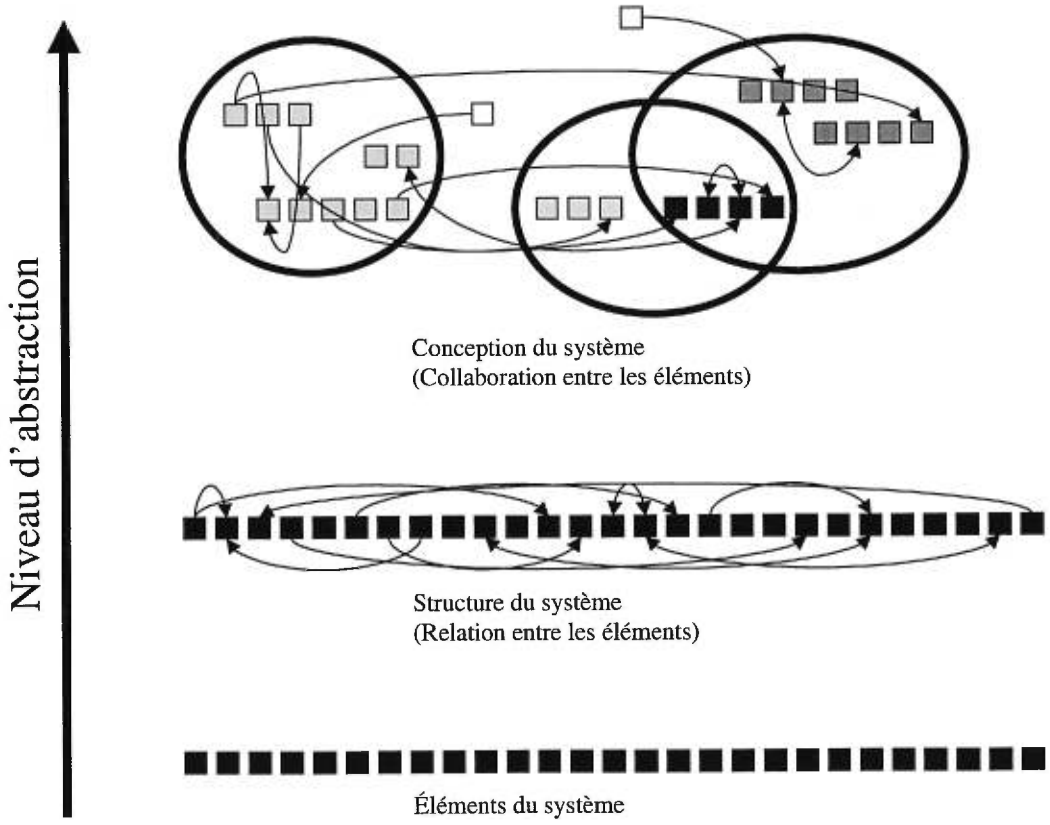

Figure 1 : Les niveaux d'abstraction : Éléments, structure, et conception

Afin de faciliter la compréhension des logiciels, des outils sont nécessaires pour permettre l'extraction de cette information à de plus haut niveaux d'abstraction. La récupération automatique ou semi-automatique de cette information permettrait d'appliquer de façon plus rapide et plus flexible les différentes

approches de compréhension présentés dans le chapitre précédent, que ce soit une approche *top-down*, *bottom-up* ou un mélange des deux. De plus, un passage rapide entre ces niveaux permettrait premièrement de mieux supporter l'approche décrite par Von Mayrhauser et Vans [42][43], mais aussi d'avoir une approche de compréhension plus itérative, c'est-à-dire une approche basée sur la résolution d'hypothèses par mise en contexte aux niveaux d'abstraction supérieurs. Une analogie à cette approche est le traitement de la langue naturelle. En effet, le traitement de la langue naturelle est normalement basée sur une résolution des ambiguïtés du langage par l'analyse à des niveaux supérieurs [23]. Par exemple, si une ambiguïté survient lors de l'analyse morphologique, on cherche à la résoudre au niveau syntaxique. Si l'ambiguïté survient lors de l'analyse syntaxique, la résolution se fait au niveau sémantique. Pour ce qui est de la compréhension des logiciels, les ambiguïtés sont plutôt des incompréhensions ou des manques d'information au niveau du modèle mental du programmeur. Ces manques peuvent souvent être comblés en navigant entre les niveaux d'abstraction du système et en changeant le contexte dans lequel les éléments sont visualisés.

Cette approche nécessite tout de même que des contextes utiles soient prédéfinis. Ces contextes peuvent être de nature structurelle comme les liens d'héritage entre plusieurs classes, ou peuvent être directement liés à la conception du système, comme l'appartenance d'un élément à l'implémentation d'un patron de conception par exemple. Une fois ces contextes définis et connus du programmeur, celui-ci peut les utiliser au besoin pour y visualiser certains éléments et ainsi permettre la mise en contexte de l'information dont il dispose. L'utilisation de contextes personnalisés et créés par le programmeur peut aussi grandement aider à la compréhension. En effet, si le programmeur a la possibilité de mettre plusieurs éléments ensemble dans un panier (un diagramme de dépendances pour visualiser les relations entre ces éléments par exemple), celui-ci pourra se servir de ce nouveau contexte pour de futures explorations du système.

Évidemment, pour être en mesure de manipuler les différents éléments d'un système et pouvoir les placer dans ces différents contextes, un mécanisme doit

permettre de naviguer à travers les différentes relations qui existent entre eux. L'approche de compréhension doit donc aussi être basée sur la possibilité d'exécuter des requêtes simples à partir des éléments d'un système ainsi que d'en faire la recherche.

3.2 Fonctionnalités souhaitables

Les différents outils de compréhension présentés dans le chapitre précédent ne fournissent pas les fonctionnalités nécessaires pour être en mesure de supporter totalement notre approche de compréhension par mise en contexte. En effet, ceux-ci permettent en général de visualiser certaines informations dans un contexte prédéfini (par exemple, dans SNiFF+ on peut visualiser une classe dans sa hiérarchie d'héritage) mais ils sont très restreints en terme d'outils de navigation et ne fournissent pas d'information au niveau de la conception. De plus, il y a une utilisation très faible de l'information particulière qu'on peut tirer du code source de systèmes orientés objet. Ce n'est pas possible non plus de créer des contextes personnalisés à partir de ces outils.

Un environnement intégrant ces fonctionnalités est donc nécessaire. Cet environnement doit aussi fournir un outil flexible de navigation et de recherche qui peut être utilisé pour faire une analyse fine du système, sans avoir à chercher l'information voulue dans le code source directement. Autrement dit, cet outil doit permettre de manipuler les éléments d'un système de façon abstraite en permettant l'exécution de requêtes prédéfinies ou personnalisées sur ces éléments. De plus, pour être complet l'environnement doit aussi faciliter au maximum les aspects cognitifs décrits par Storey [36][37] pour l'exploration de logiciels. Entre autres, l'environnement doit faciliter la navigation, il doit réduire la désorientation, il doit améliorer la compréhension *bottom-up* et *top-down,* il doit permettre d'intégrer plusieurs approches de compréhension et il doit fournir des informations permettant d'orienter la navigation du programmeur.

Donc, en résumé, cette approche de compréhension nécessite un support logiciel pour les fonctionnalités suivantes :

1. Permettre l'abstraction des éléments du système et permettre la recherche et navigation de ces éléments de façon à réduire la désorientation de l'utilisateur.

2. Permettre la restauration automatique ou semi-automatique de différentes vues du système au niveau structurel et au niveau de la conception.

3. Permettre la navigation à travers ces différents niveaux d'abstraction dans les deux sens pour mieux supporter les approches top-down et bottom-up, ainsi que l'intégration des deux approches.

4. Permettre aux éléments du système d'être visualisés dans différents contextes et à différents niveaux d'abstraction.

5. Permettre la création de contextes personnalisés où seuls les éléments voulus sont visualisés.

Lorsqu'un programmeur tente de comprendre un bout de code source, il doit être en mesure d'identifier son rôle dans le système. En utilisant l'approche par mise en contexte, le programmeur peut accélérer de beaucoup sa compréhension en identifiant rapidement la nature des collaborations que ce bout de code entretient avec tout le reste du système.

Chapitre 4 : SPOOL : Une plate-forme supportant l'approche par mise en contexte

Pour être en mesure de supporter l'approche précédemment décrite, plusieurs outils de navigation, de recherche et de visualisation ainsi que plusieurs mécanismes d'interaction entre ces outils sont nécessaires. Ce chapitre décrit l'environnement utilisé, le schéma utilisé pour la base de données de l'environnement, les outils d'analyse qui y sont intégrés, ainsi que les deux outils, soient le *Design Browser* et le *Context Viewer* qui ont été développés dans le but de supporter notre approche d'aide à la compréhension de logiciels orientés objet de taille industrielle.

4.1 L'environnement SPOOL

L'environnement qui a été développé dans le cadre du projet SPOOL est entièrement écrit dans le langage Java et permet de stocker et de visualiser l'information statique extraite du code source de logiciels orientés objet (C++, Java). Il a été conçu principalement pour permettre l'étude de grands systèmes au niveau de la conception, mais il permet aussi d'explorer de tels systèmes à des niveaux moins abstraits, comme au niveau de leur structure ou directement au niveau de leur code source.

L'environnement SPOOL est formé de plusieurs composants tel que montré dans la Figure 2. Le noyau de l'environnement est une base de données orientée objet [26] qui sert de dépôt au reste de l'environnement et qui contient l'information des systèmes à analyser. Le dépôt peut être accédé par une série d'outils de visualisation comme des diagrammes UML, des outils d'analyse de dépendances,

des outils de détection et d'inspection de patrons de conception [19], des outils de recherche et navigation et un outil de calcul de métriques. L'information contenue dans ce dépôt y est introduite en utilisant différentes technologies d'analyse syntaxique. Par exemple, il est possible d'utiliser l'analyseur GEN++ [8] pour extraire l'information du code source puis d'intégrer cette information au dépôt (Le Tableau 2 donne la liste des types d'information extraits et ajoutés au dépôt de SPOOL). D'autres technologies comme l'analyseur Datrix [3] ou l'environnement SNiFF+ [32] permettent aussi d'extraire un certain nombre d'informations qui peuvent être ajoutées au dépôt de l'environnement SPOOL.

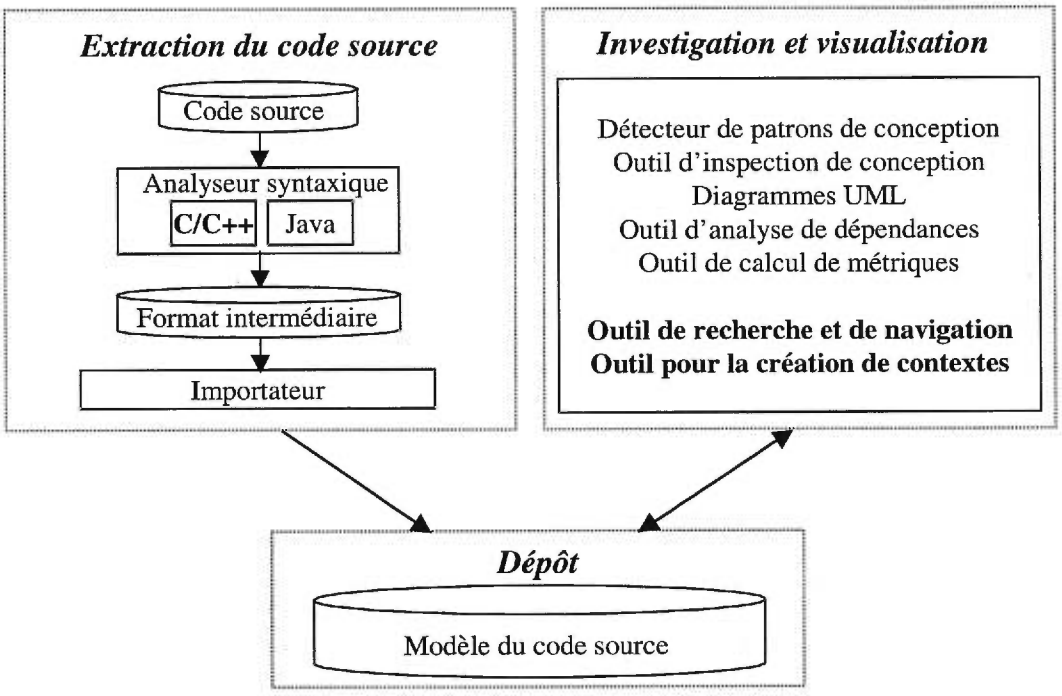

Figure 2 : Architecture de l'environnement SPOOL

En fait, l'information extraite du code source est stockée dans le dépôt sous la forme d'un modèle objet. Ceci signifie que chaque "élément" du code source est représenté dans l'environnement SPOOL comme un objet Java standard, avec son type, ses attributs et ses méthodes. Par exemple, on retrouve les classes, les fichiers, les attributs et les méthodes des systèmes à analyser sous la forme d'objets dans le dépôt, mais aussi les différents types de relations entre ces éléments comme les appels d'opérations, les références, les instanciations, etc.

Comme tous ces objets sont stockés tels quels dans la base de données, différents outils peuvent ainsi les accéder et naviguer à travers eux simplement en suivant leurs interrelations. Cette approche permet d'utiliser l'information contenue dans le dépôt à l'intérieur de n'importe quel programme Java de façon transparente. Les différents outils d'analyse de SPOOL énumérés précédemment utilisent aussi cette approche et permettent de visualiser, de rechercher et de naviguer facilement les différents éléments d'un système.

1.	Fichiers [nom, répertoire]
2.	Classes, structures, unions, types primitifs
3.	Hiérarchie d'héritage
4.	Attributs [nom, type, propriétaire, visibilité]
5.	Opérations et méthodes [nom, type, propriétaire, polymorphisme, sorte]
6.	Paramètres [nom, type]
7.	Types de retour [nom, type]
8.	Appels d'opérations [opération, receveur]
9.	Création d'objets
10.	Utilisation de variables
11.	Relations amies (*friendship* C++)

Tableau 2 : Information contenue dans le dépôt de SPOOL

Deux de ces outils ont été conçus dans le cadre de ce mémoire. Le premier, le *Design Browser*, est un outil permettant la navigation et la recherche, tandis que le second, le *Context Viewer*, est un outil de mise en contexte. Ces deux outils graphiques, qui seront présentés respectivement aux sections 4.4 et 4.5, permettent d'accéder rapidement et de manipuler l'information disponible dans le dépôt de SPOOL à des fins de compréhension.

4.2 Le dépôt de SPOOL

Le schéma utilisé pour le dépôt de SPOOL est en fait une hiérarchie de classes dont une grande partie est empruntée du méta-modèle UML [40]. En étant un méta-modèle pour l'analyse et la conception de logiciels, UML fournit une fondation solide à l'environnement SPOOL comme outil de compréhension. Malgré cela, la rétro-ingénierie effectuée par SPOOL doit débuter à partir du code source, et donc certaines extensions au méta-modèle ont dû être apportées pour être en mesure de supporter certains des aspects spécifiques aux langages de programmation.

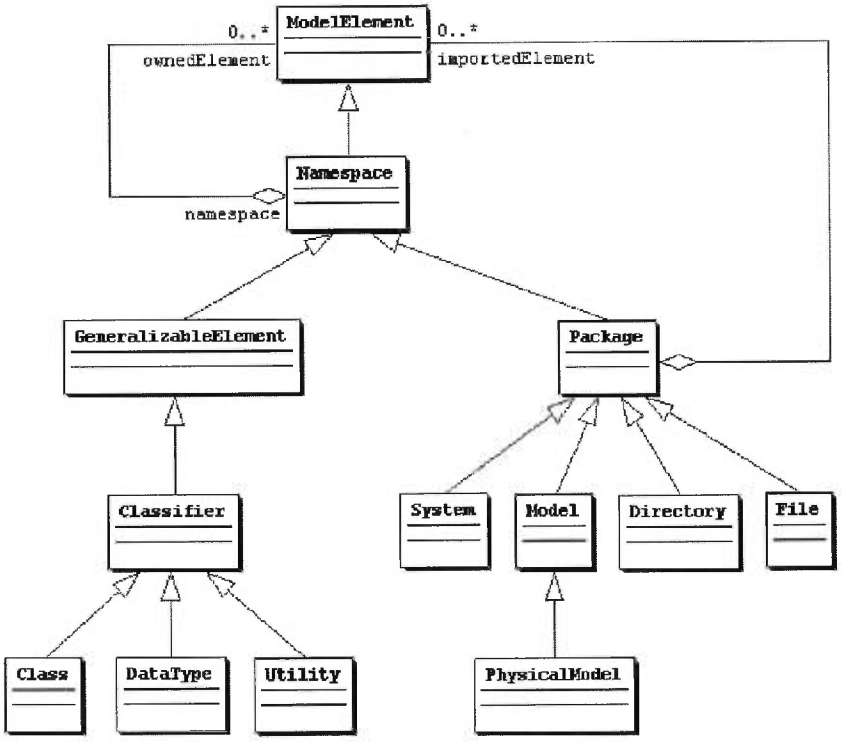

Figure 3 : Schéma du dépôt de SPOOL : Classes de type *Namespace*

Les classes centrales du dépôt de SPOOL définissent la structure de base ainsi que la hiérarchie de contenance des éléments qui sont gérés dans le dépôt (voir Figure 3 et Figure 4). En fait, comme les éléments du système à analyser sont stockés sous le format du méta-modèle UML, on parle plutôt de *ModelElement*, tel que

défini par UML [40]. Au centre de cette hiérarchie, on trouve la classe *Namespace* qui contient une collection de *ModelElements*. Un *GeneralizableElement* définit les nœuds entre les relations de généralisation comme l'héritage par exemple. Un *Classifier* fournit des fonctionnalités (*Features*) qui peuvent être de nature structurelle (*Attributes*) ou de nature comportementale (*Operations* et *Methods*). Un *Package* sert à regrouper des *ModelElements* ensemble.

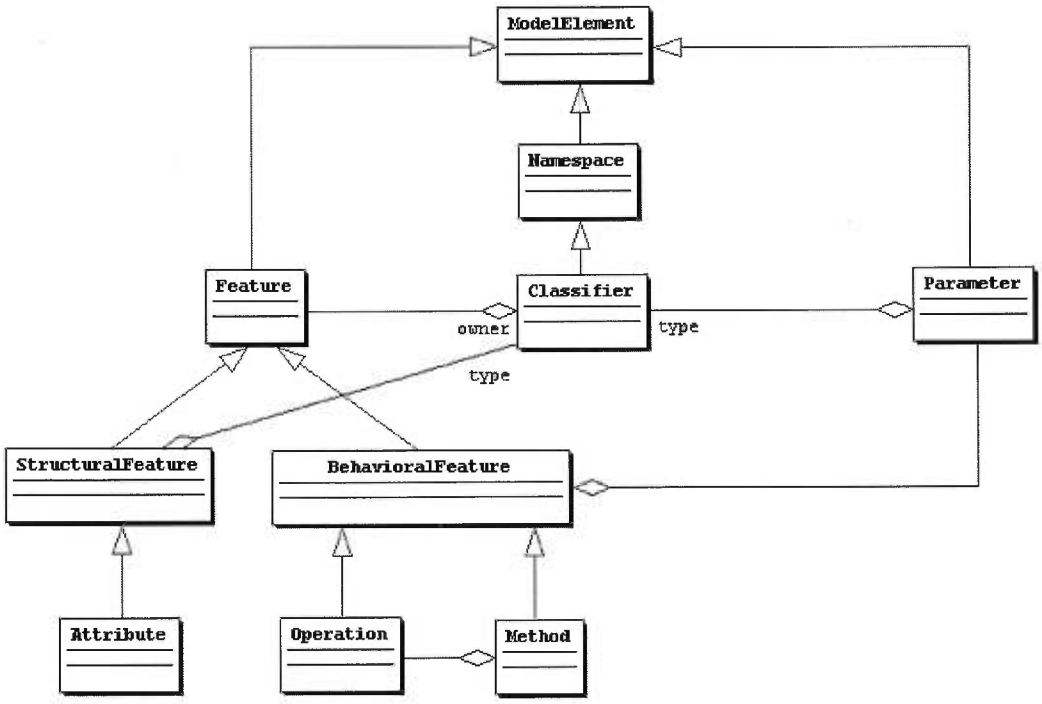

Figure 4 : Schéma du dépôt de SPOOL : Classes de type *Feature*

En fait, seules les classes définies dans le bas de la hiérarchie sont concrètes et donc, seules les classes *Class*, *DataType*, *Utility*, *System*, *PhysicalModel*, *Directory* et *File* de la Figure 3 sont instanciées, ainsi que les classes *Attribute*, *Operation*, *Method* et *Parameter* de la Figure 4. Chacune de ces classes représente un élément particulier d'un système orienté objet :

- *Class* représente une classe, *Directory* un répertoire et *File* un fichier.

- *DataType* représente les types de base du langage comme par exemple *int*, *float* et *char* en C++.

- *Utility* est un concept permettant de classifier les éléments ne faisant pas partie d'une classe comme les variables globales, ou les fonctions libres.

- *System* représente un système logiciel. Le *System* contient normalement un modèle physique et un modèle logique (le modèle logique n'est pas modélisé dans le schéma de SPOOL).

- *PhysicalModel* représente le modèle physique d'un système et contient entre autres les répertoires et les fichiers du système.

- *Attribute* représente les attributs d'une classe (ou d'un *Utility* s'il s'agit de variables libres).

- *Operation* est la signature d'une méthode et contient une liste de paramêtres (*Parameter*). Autrement dit, une méthode a une seule signature mais une signature peut être partagée par plusieurs implémentation et donc dans différentes méthodes.

- *Method* est l'implémentation d'une *Operation* et contient le corps de la méthode. Plusieurs méthodes peuvent implémenter la même signature (*Operation*).

- *Parameter* est un paramètre défini dans la signature d'une méthode (*Operation*).

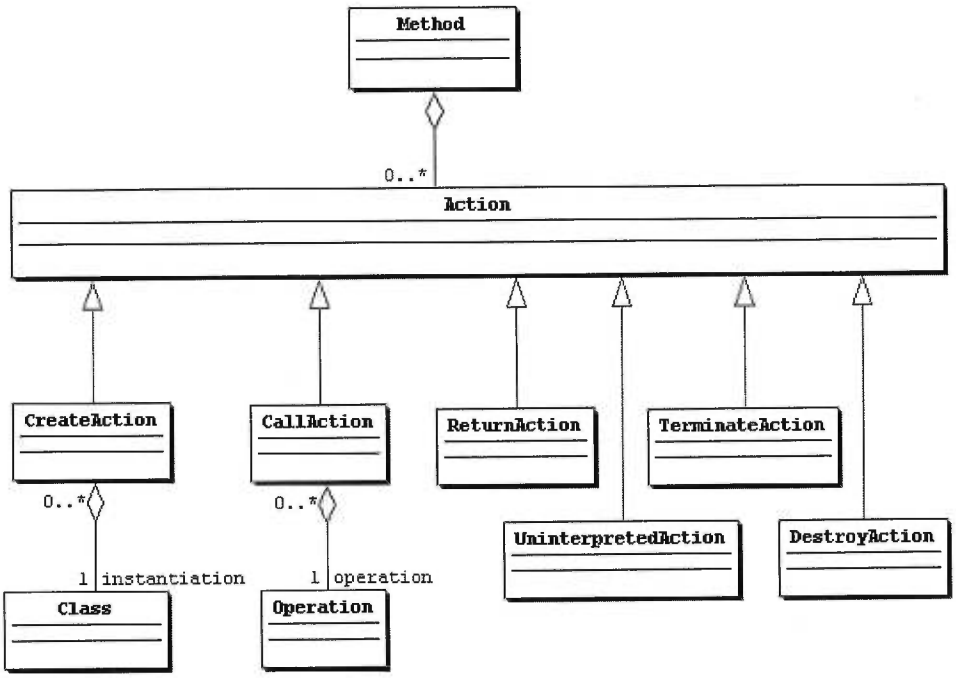

Figure 5 : Schéma du dépôt de SPOOL : Classes de type *Action*

La liste précédente permet de classifier une partie des éléments qu'on retrouve dans les systèmes orientés objet ainsi que de spécifier leurs interrelations. La Figure 5 présente la hiérarchie de classes de SPOOL modélisant les différents types d'actions qui y sont modélisés. Une *Action* est définie par UML comme étant un calcul atomique dont le résultat est de changer l'état du système ou de retourner une valeur [40]. Dans SPOOL, les actions sont utilisées pour décrire ce qui est fait dans le corps des méthodes. Par exemple, l'instanciation d'une classe est modélisée par *CreateAction,* et un appel d'opération est modélisé par *CallAction*. Les autres types d'action (*ReturnAction*, *TerminateAction*, *DestroyAction* et *UninterpretedAction*) ne sont présentement pas utilisés par les outils de SPOOL, et donc cette information n'est pas importée dans le dépôt pour réduire la taille de la base de données et ainsi améliorer la performance des outils d'analyse.

Le schéma décrit dans les paragraphes précédents est basé principalement sur le méta-modèle UML, mais ne le respecte pas à 100%. En effet, comme le méta-modèle à été conçu dans un esprit de conception de logiciels et non de rétro-

ingénierie, certains aspects particuliers des langages de programmation ont dû être ajoutés et certaines modifications ont dû être apportées pour des considérations de performance. Malgré ces quelques différences, la compréhension du schéma de SPOOL est tout de même grandement facilitée grâce à sa ressemblance avec UML.

4.3 Les outils d'analyse de SPOOL

L'environnement SPOOL intègre plusieurs outils, permettant l'étude d'un système à plusieurs niveaux d'abstraction (code source, structure et conception). Les sous-sections qui suivent présentent brièvement certains des outils qui ont été développés ou simplement intégrés dans le cadre du projet SPOOL.

4.3.1 Analyse au niveau du code source

Le niveau d'abstraction le plus bas qu'on peut explorer d'un système à des fins de compréhension est le code source lui-même. L'environnement SPOOL ne permet pas directement l'exploration du code source puisque aucun éditeur n'est présentement implémenté. Par contre, un mécanisme permettant d'interagir avec l'environnement de développement SNiFF+ [32] est disponible. Ce mécanisme permet de visualiser les éléments du code source via les différents outils de SNiFF+ (éditeur de code source, hiérarchie de classes, etc.). Ce pont entre les outils de SPOOL et ceux de SNiFF+, en plus de permettre l'accès au code source, ajoute à l'environnement SPOOL toute la puissance de SNiFF+ pour l'inspection et la navigation à travers des systèmes logiciels.

4.3.2 Analyse au niveau de la structure

Lorsqu'on visualise un logiciel, on distingue normalement deux types de structures. La première est la structure physique du logiciel, c'est-à-dire l'emplacement des fichiers sources dans une hiérarchie de répertoires, l'emplacement des différentes structures et classes dans le système, etc. La deuxième structure est celle qui a trait à l'emplacement logique des éléments entre

eux. Par exemple, l'emplacement des différentes classes dans l'arbre d'héritage ou simplement l'emplacement d'une méthode dans un graphe d'appels.

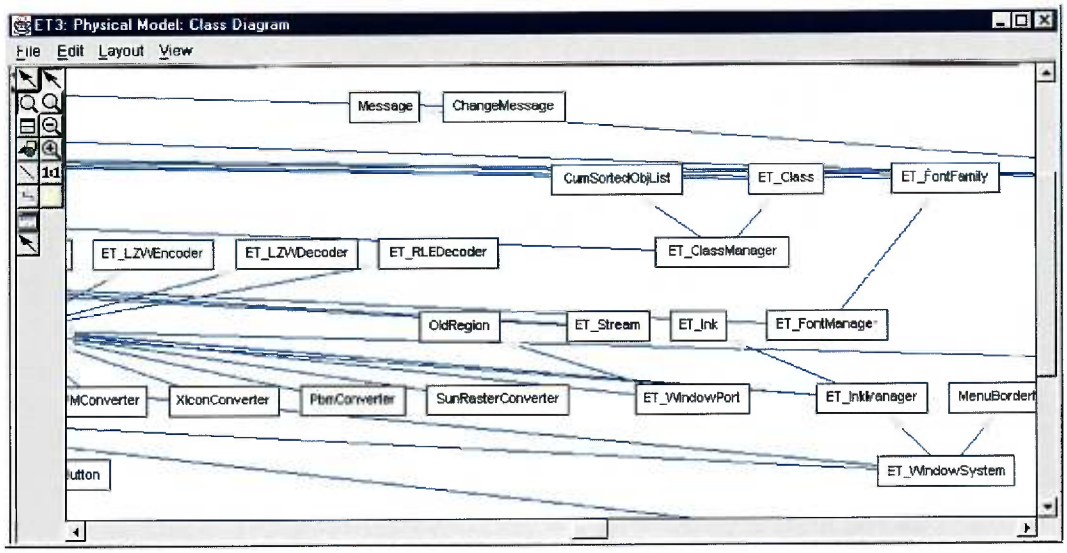

Figure 6 : Diagramme de classes avec le lien de création d'objets

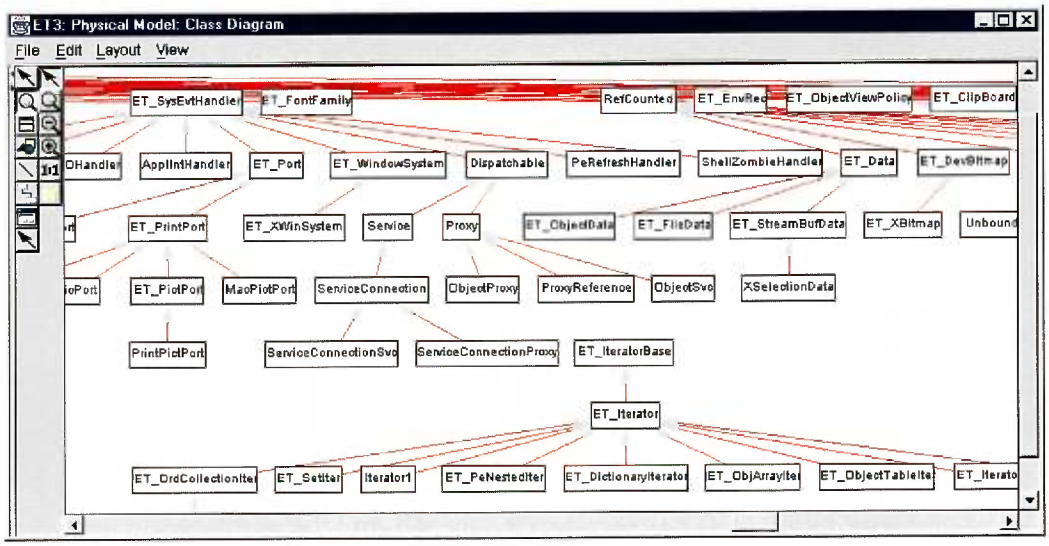

Figure 7 : Diagramme de classes avec le lien d'héritage

L'environnement SPOOL permet de visualiser certains des aspects structurels des logiciels et aussi de combiner l'aspect physique et l'aspect logique. Par exemple, l'environnement permet de visualiser l'arbre de répertoires et l'emplacement des différents fichiers, classes et structures du système dans cet arbre. Il permet aussi

de visualiser ces différents éléments en faisant apparaître leurs interrelations (héritage, appel d'opérations, création d'objets, références, etc.) sous forme d'un graphe. La Figure 6 montre une partie du diagramme de classes du système ET++ en affichant seulement la création d'objets, tandis que la Figure 7 affiche le même diagramme mais avec les liens d'héritage.

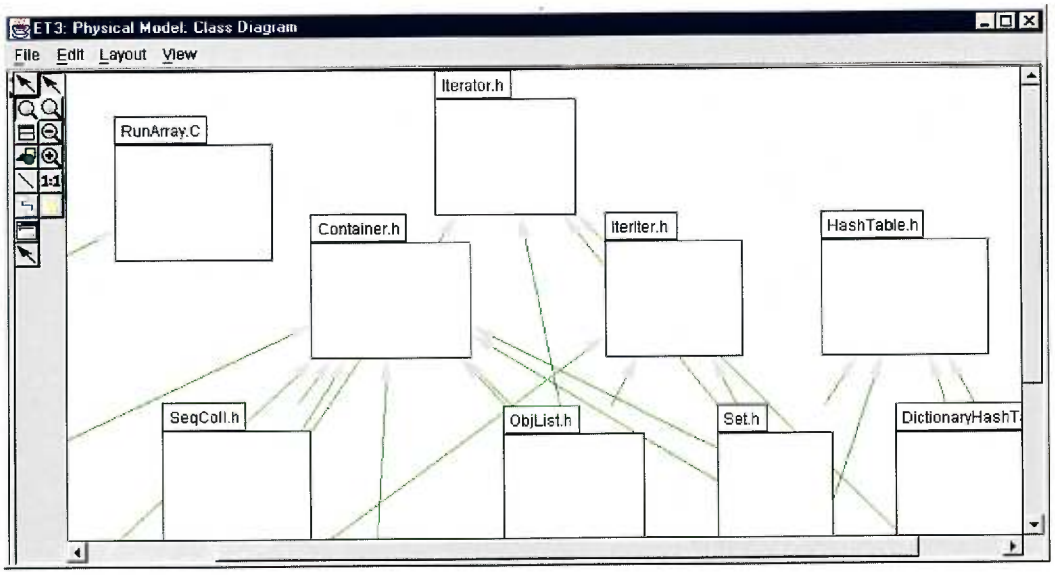

Figure 8 : Diagramme de dépendances

L'environnement fournit aussi un mécanisme d'agrégation permettant de visualiser les dépendances à un plus haut niveau. Par exemple, les appels d'opération peuvent être accumulés au niveau des classes, des fichiers ou même des répertoires comme montré dans la Figure 8. Cette approche permet entre autres d'extraire certaines informations liées à la conception du système, ou même de déceler certains problèmes, comme par exemple, un couplage imprévu entre deux répertoires ou deux fichiers.

4.3.3 Analyse au niveau de la conception

Pour récupérer et analyser les détails de la conception d'un système, une simple visualisation de sa structure ou de son code source n'est pas suffisante. Certaines informations supplémentaires sont nécessaires afin d'extraire ou de retrouver les différents composants et collaborants à la conception originelle du système.

L'approche suivie dans le projet SPOOL consiste à détecter la structure de composants de conception et à permettre leur visualisation et leur inspection à l'aide d'outils [19]. Il est évidemment très difficile de récupérer automatiquement des composants de conception si ceux-ci sont de très haut niveau (comme les patrons architecturaux [5] par exemple). C'est pourquoi SPOOL supporte la récupération de certains patrons de conception (entre autres certains patrons de Gamma et al. [12]) qui ont une structure simple, qui sont d'un niveau assez bas et qui sont détectables par une analyse statique (i.e. qui ne sont pas hautement dynamique comme le patron *Chain of Responsability*, par exemple, qui est très difficile à détecter si on ne possède pas d'informations sur le comportement dynamique du système). La récupération d'éléments de conception s'effectue en exécutant des requêtes sur le dépôt de SPOOL et donc en détectant certaines structures correspondant à l'implémentation généralement utilisée pour les patrons choisis. Évidemment, un certain calibrage des requêtes doit être mis en place pour éviter que celles-ci ne soient trop générales ou trop spécifiques. En effet, un mauvais calibrage des requêtes d'un coté ou de l'autre entraîne soit beaucoup de bruit, soit un danger de ne pas récupérer certaines instances importantes du patron.

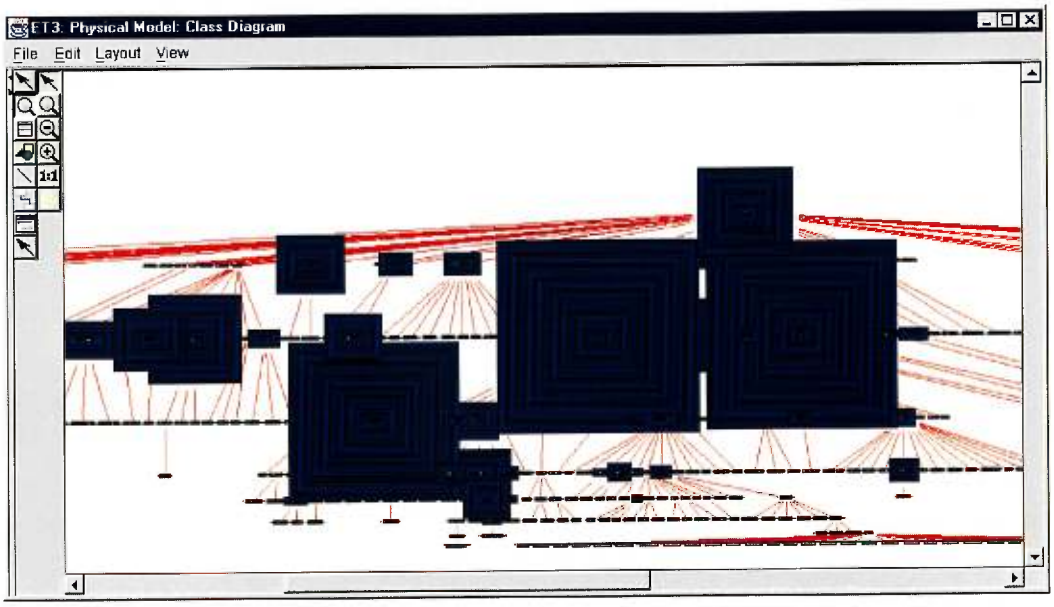

Figure 9 : Détection de patrons de conception

Une fois que l'exécution d'une requête de détection est terminée, les résultats doivent être visualisés pour permettre une inspection plus en profondeur par l'usager. Dans SPOOL, la visualisation des résultats d'une requête se fait directement dans le diagramme de classes du système. En effet, dans la définition de la requête, on détermine lequel des éléments de la structure recherchée sera le représentant du patron (autrement dit, le représentant est l'élément auquel on associe l'instance du patron détectée). SPOOL peut donc, à partir de cette information, repérer l'élément en question dans le diagramme de classes, et dessiner une boîte autour de celui-ci pour chaque instance trouvée. Autrement dit, plus une classe participe à d'instances d'un patron, plus la boîte qui l'entoure est grande. Par exemple, la Figure 9 montre les *Template Methods* [12] trouvés dans le système ET++. Notons toutefois que la façon dont les boîtes sont dessinées autour des éléments dans le diagramme est flexible et peut être modifiée selon la requête afin de tenir compte de la nature particulière d'un patron de conception. Par exemple, si une classe est identifiée comme participante à un patron, on peut vouloir dessiner une boîte autour d'elle pour chacune de ses méthodes, faisant ainsi ressortir seulement les plus grandes classes dans le diagramme. Cette façon de visualiser l'information dans le contexte du diagramme de classes est très importante car elle permet d'avoir une première impression sur l'endroit où se trouvent les instances du patron de conception et, dans beaucoup de cas, d'identifier où se trouvent les classes importantes dans le système.

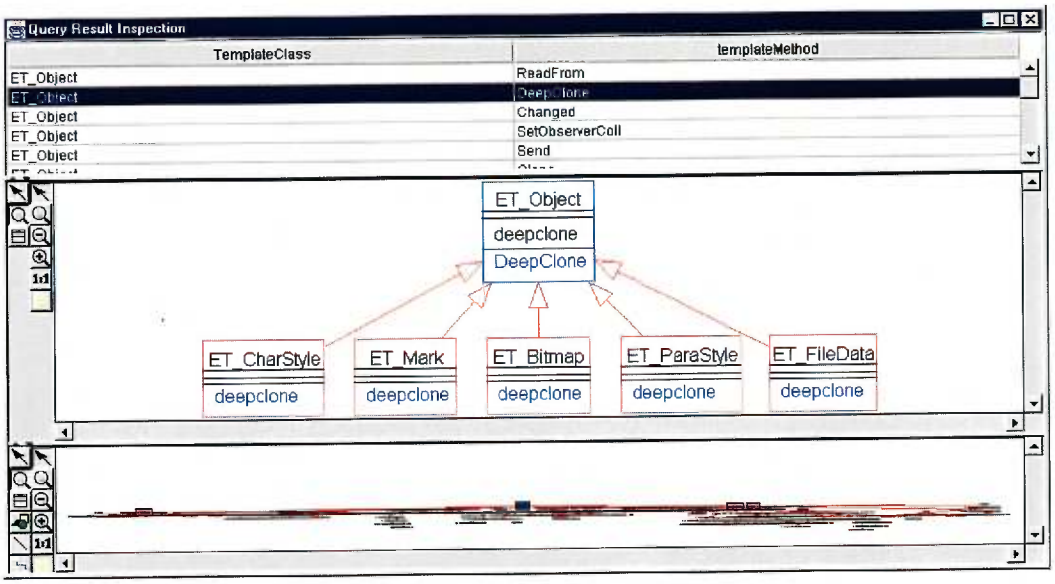

Figure 10 : Inspection de patrons détectés

Après cette étape, directement dans le diagramme de classes, il est possible de sélectionner un des éléments qui est entouré de boîtes, et d'inspecter, à l'aide d'un outil d'inspection, les différentes instances du patron détecté qui lui sont associées. Cet outil, montré dans la Figure 10, affiche les informations en trois sections. La section du haut affiche sous forme d'une liste toutes les instances du patron parmi lesquelles l'élément choisi est le représentant. La section du centre affiche, sous la forme d'un diagramme de collaboration, tous les éléments participant à l'instance sélectionnée dans la première section. Finalement, la section du bas présente chacun des éléments de la deuxième section dans le contexte de leur hiérarchie de classes, ce qui permet de bien identifier les composants ainsi que leur emplacement.

En se servant des différents outils de l'environnement SPOOL, un utilisateur peut détecter certaines structures particulières dans le système et les inspecter pour déterminer si ces structures font réellement partie d'un élément de conception ou non. En fait, même si une instance détectée à l'aide de l'environnement SPOOL s'avère ne pas être réellement le patron de conception recherché (i.e. l'intention n'est pas celle du patron), on peut souvent en tirer de l'information très

importante tout de même, surtout si la structure est relativement complexe comme celle de *Bridge* ou de *Observer* par exemple [12].

4.4 La navigation et la recherche dans SPOOL

Le *Design Browser* est un outil intégré à l'environnement SPOOL qui a été développé dans deux buts principaux. Le premier but était de rendre la navigation entre les éléments et les relations d'un système plus flexible en créant une abstraction de ces éléments à un plus haut niveau que le code source, en fournissant une liste de requêtes prédéfinies, en fournissant des mécanismes de composition des requêtes et un mécanisme d'ajout pour de nouvelles requêtes. Le deuxième but était de pouvoir utiliser le *Design Browser* comme outil d'exploration à lui seul mais aussi de pouvoir l'utiliser conjointement avec le reste des outils d'analyse de SPOOL. Cette approche d'intégration était pour permettre, de façon simple et transparente, la visualisation des éléments du *Design Browser* dans différents outils et donc dans différents contextes.

4.4.1 Description globale

L'interface usager du *Design Browser*[1] est séparée en trois sections: *Starting Point*, *Queries* et *Results* (voir Figure 11). La première et la dernière section peuvent contenir des listes d'éléments qui sont contenus dans un système. Par exemple, on peut y trouver des classes, des méthodes, des attributs, des paramètres, des fichiers, des répertoires, etc. Ces *ModelElements* sont tous représentés de façon abstraite dans le *Design Browser* par leur nom et par une petite icône de couleur représentant leur type, ce qui permet de les identifier visuellement et rapidement. Le Tableau 3 donne un aperçu des icônes et des types d'éléments qui sont disponibles dans le *Design Browser*.

[1] Une description détaillée des différentes fonctionnalités du *Design Browser* est aussi disponible dans le manuel d'utilisation [10].

A	*Attribut de classe*
C	*Classe*
E	*Énumération (Spécifique à C/C++)*
F	*Fichier*
I_A	*Attribut Local à une méthode*
M	*Méthode (UML: implémentation d'une opération)*
O	*Opération (Est composé principalement d'une signature et peut être implémentée par plusieurs méthodes «polymorphisme»)*
P	*Paramètre d'une opération*
P_M	*Model Physique (UML: Un modèle de système est composé d'un modèle physique et d'un modèle logique)*
P_R	*Type Primitif (int, float, char, etc.)*
S	*Système logiciel*
U	*Union (Spécifique à C/C++)*
U_t	*Utilité (UML: Regroupement de ce qui n'est pas orienté objet - variables globales, fonctions libres, etc.)*
	*Répertoire*

Tableau 3 : Les icônes du *Design Browser*

La liste au centre du *Design Browser* contient la liste des requêtes qui est suggérée par l'outil selon les *ModelElements* sélectionnés. Par exemple, si l'élément sélectionné dans le *Starting Point* est une classe, alors il est possible d'exécuter différentes requêtes pour trouver ses méthodes, ses attributs, ses sous-classes et super-classes, les classes qui y sont instanciées, les opérations qui y sont invoquées, etc. La liste de requêtes est créée dynamiquement ce qui permet de voir uniquement les requêtes qui sont pertinentes aux types des éléments sélectionnés. Une fois l'exécution d'une requête terminée, les résultats sont affichés dans la section *Results* sous la forme d'une nouvelle liste de *ModelElements*. Dans l'exemple de la Figure 11, la classe *ET_Backgrounditem* a été sélectionnée dans la liste de gauche, faisant ainsi apparaître la liste de requêtes au centre. L'exécution de la requête *Operation called* a ensuite permis de récupérer les quatre opérations appelées dans cette classe et de les afficher dans la section de droite. Pour des raisons de flexibilité, l'outil permet le déplacement des éléments entre les listes par « drag & drop » (DnD) ce qui rend possible la

composition de requêtes. Par exemple, on peut exécuter une requête, déplacer certains éléments du résultat dans la première section et exécuter une nouvelle requête à partir de ces éléments. Si l'usager le préfère, il peut aussi démarrer un second *Design Browser* et copier seulement (par DnD) les éléments voulus dans cette nouvelle instance de l'outil.

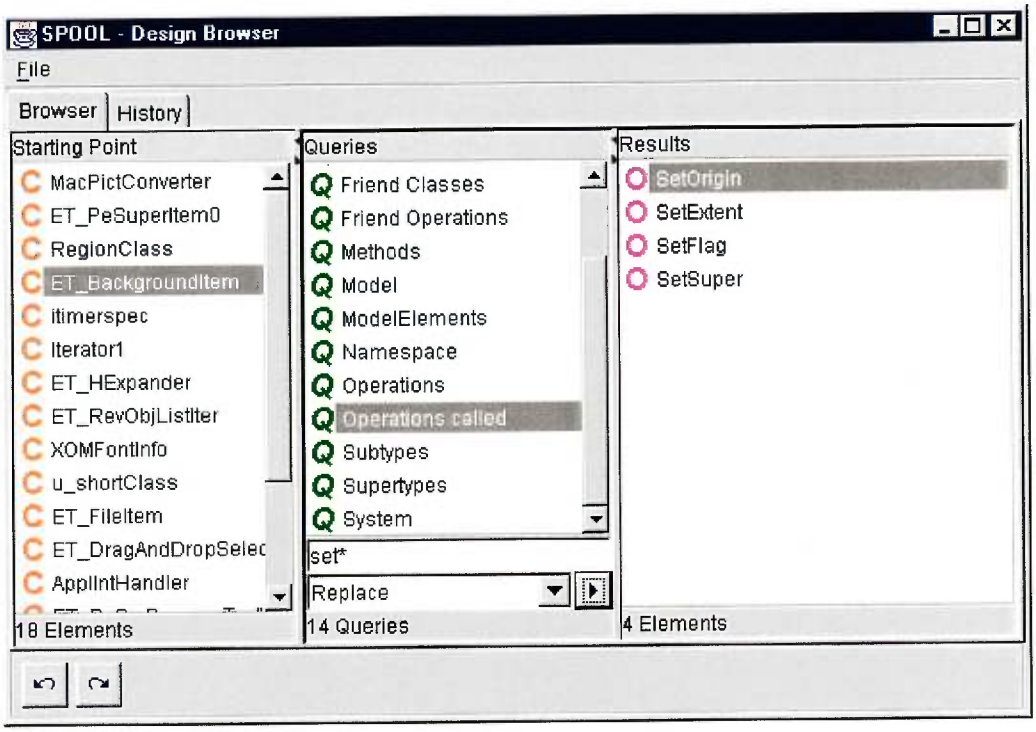

Figure 11 : Le *Design Browser* de SPOOL

Il est aussi possible d'effectuer certaines actions complémentaires à l'exécution d'une requête. Par exemple, on peut appliquer un filtre sur le résultat en tapant la chaîne de texte recherchée dans le champ situé sous la liste de requêtes. Ceci a pour effet de supprimer du résultat les *ModelElements* dont le nom ne correspond pas à la chaîne entrée comme filtre. On peut aussi définir le type d'opération qui sera effectuée sur les éléments de la section *Results* si celle-ci contient toujours des éléments d'une requête précédente. Les opérations supportées sont celles possibles sur les ensembles, c'est-à-dire remplacement, union, intersection et différence. De plus, ces opérations sont aussi disponibles lors du déplacement des

éléments par DnD grâce à un menu contextuel qui apparaît lorsque les éléments sont relâchés.

4.4.2 Mécanisme d'historique

Comme le *Design Browser* permet de déplacer les éléments d'une liste à l'autre ou d'un *Design Browser* à l'autre, il est essentiel d'avoir un mécanisme d'historique permettant de revenir sur ses pas et de récupérer des résultats précédents. Contrairement aux navigateurs web traditionnels qui ne conservent en mémoire qu'un historique linéaire du chemin suivi, le *Design Browser* conserve tout l'arbre parcouru. En effet, lorsqu'on fait un retour en arrière dans les navigateurs traditionnels et qu'on suit un nouveau chemin, la branche originale de l'historique est perdue, ce qui oblige l'utilisateur à retrouver lui-même le chemin parcouru la première fois. Le *Design Browser* conserve donc tout l'arbre parcouru en mémoire, et permet à l'usager de choisir parmi les différents chemins possibles. Des boutons *Back* et *Forward* (voir Figure 12; icônes du bas) permettent donc de se déplacer dans l'historique mais, à la différence des navigateurs standards qui ne gardent que le dernier endroit visité, le *Design Browser* permet de choisir parmi plusieurs chemins possibles. Une liste des chemins ordonnée de façon chronologique, apparaît sous la forme d'un menu qui permet à l'usager de sélectionner l'état voulu.

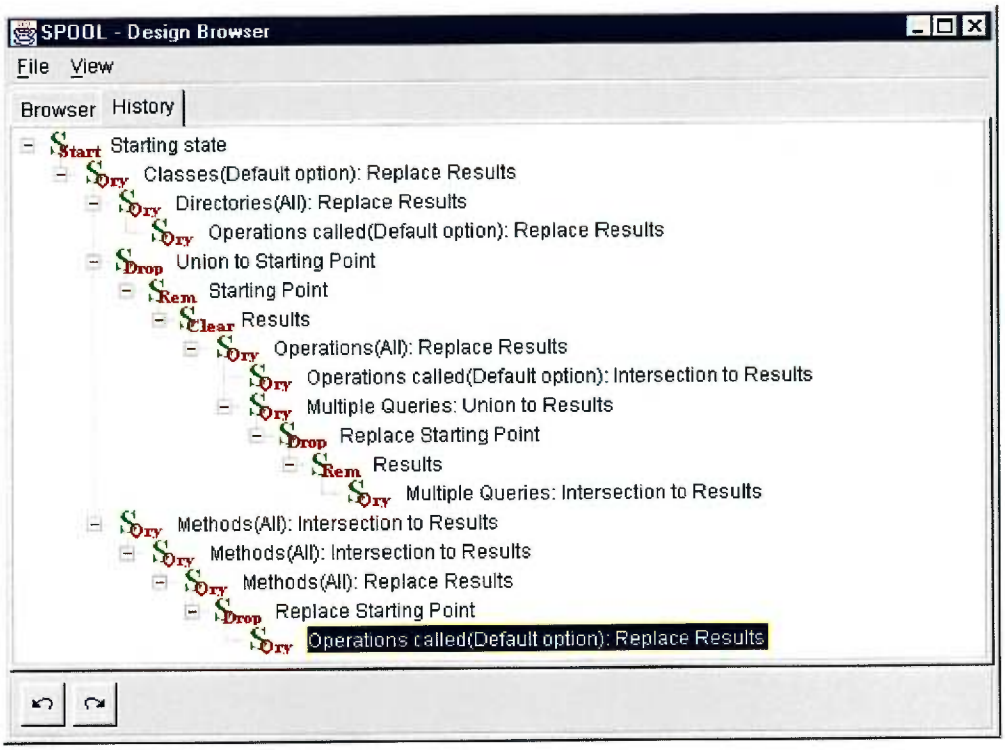

Figure 12 : Diagramme d'historique du *Design Browser*

L'approche « boutons » n'est pas très pratique lorsque l'arbre devient complexe ou si beaucoup de choix sont possibles. De plus, il doit y avoir un étiquetage efficace des états qui apparaissent dans l'historique, puisque c'est le seul moyen dont dispose l'usager qui lui permet d'identifier le chemin à suivre. Pour résoudre ces problèmes, le *Design Browser* permet la visualisation de l'historique sous forme d'un arbre avec, pour chaque nœud, une icône décrivant le type d'action effectuée et une étiquette décrivant textuellement le *quoi*, le *où* et le *comment* de l'action (voir Figure 12). Les types d'actions qui sont possibles dans le *Design Browser* sont l'exécution d'une requête (S_{Qry}), l'ajout d'éléments dans une des listes par DnD (S_{Drop}) ou bien la suppression d'éléments d'une liste par l'usager (S_{Clear} et S_{Rem}). Lorsqu'une de ces actions est exécutée, un nouvel élément est ajouté à l'historique et est représenté dans l'arbre par une de ces icônes. L'icône S_{Start} représente l'état de départ du *Design Browser* et elle apparaît lorsque celui-ci est démarré à partir d'un des différents endroits de l'environnement et à partir d'un ou plusieurs éléments (voir Section 4.4.4). L'étiquette qui apparaît avec

chaque état est elle aussi définie en fonction du type d'action. Par exemple, si l'action est une requête, c'est le nom de celle-ci, l'option de la requête qui a été choisie, ainsi que le type d'opération (union, intersection, etc.) qui sont affichés (Notons que si plusieurs requêtes sont exécutées en même temps, un seul état est créé et le texte « *Multiple queries* » est affiché comme nom). S'il s'agit d'un DnD, c'est le type d'opération qui est affiché ainsi que le nom de la section du *Design Browser* qui est affectée par le DnD. Enfin, s'il s'agit d'une suppression d'éléments, uniquement le nom de la section du *Design Browser* impliquée est affichée (i.e. *Starting Point* ou *Results*). À partir du diagramme affichant l'historique du *Design Browser*, l'usager a aussi la possibilité de restaurer un ancien état directement. Pour récupérer cet état particulier, il suffit de le sélectionner dans le diagramme présenté à la Figure 12, et de faire apparaître un menu contextuel avec la souris qui permet de remettre le *Design Browser* dans l'état voulu.

L'usager peut au besoin cloner l'historique en cours pour l'utiliser dans un autre *Design Browser*. Ce mécanisme permet d'utiliser le même historique pour plusieurs instances du *Design Browser* et donc d'avoir accès à tous les états à partir de tous les outils.

4.4.3 Recherche et navigation par requêtes

Le *Design Browser* se veut un outil flexible de recherche et de navigation. À la base, celui-ci permet d'exécuter des requêtes simples et prédéfinies pour la navigation, mais il permet aussi d'en ajouter de nouvelles plus complexes et personnalisées. Le mécanisme qui est utilisé pour les requêtes repose entièrement sur la représentation interne du dépôt de SPOOL et par conséquent, sur l'implémentation du schéma du dépôt qui est basé sur le méta-modèle UML. De plus, comme le code source des systèmes à analyser est représenté de cette façon et que les éléments sont modélisés par des objets, plusieurs requêtes sont déjà disponibles sous forme de méthodes au sein même de ces objets. Par exemple, considérons *Class*, *Attribute* et *Method*, les classes Java du schéma qui

représentent les éléments UML correspondants. Les instances de ces classes représentent donc les éléments réels d'un système. Supposons que `aClass` est une instance particulière de *Class* et que `aMethod` est une instance de *Method*, alors :

- `aClass.getFeatures()` retourne l'ensemble de toutes les instances de *Operation*, *Method* et *Attribute* contenues dans cette classe.

- `aMethod.getCalledOperations()` retourne la liste des instances de *Operation* qui représentent les opérations appelées par cette méthode.

Les requêtes prédéfinies dans le *Design Browser* sont donc bâties en utilisant les nombreuses méthodes disponibles pour chaque *ModelElement* du schéma. Toutefois, la navigation à travers les éléments du code source en utilisant uniquement des requêtes prédéfinies atteint rapidement ses limites. C'est pourquoi le *Design Browser* permet d'écrire et d'ajouter de nouvelles requêtes à la liste. Évidemment, l'utilisateur qui veut ajouter une requête doit connaître le langage Java pour pouvoir l'écrire ainsi que le schéma du dépôt de SPOOL. Mais puisque le schéma suit de très près le méta-modèle UML, cette deuxième exigence n'est pas un obstacle majeur. Le code source Java qui suit est un exemple typique de l'implémentation d'une requête du *Design Browser*. Il s'agit d'une requête qui, étant donné un certain élément sélectionné dans le *Design Browser* (une classe par exemple), retourne les super-types de cet élément (les super-classes par exemple):

```
package spool.tools.designBrowser.baseQueries;

import spool.tools.designBrowser.*;
import spool.repository.model.ifc.*;
import COM.POET.odmg.*;
import COM.POET.odmg.collection.*;

/**
 * Cette requête retourne les super-types
 * du Classifier selectionné
 * @version     1.0 / 26 Mai 1999
 * @author      Sébastien Robitaille
 */

public class GetSupertypes extends DefaultBrowserQuery
{
    /**
     * Constructeur de la requête. Ne fait qu'initialiser le nom.
     */
    public GetSupertypes() {
        setName("Supertypes");
    }

    /**
     * Méthode facultative utilisée par le Design Browser
     * pour déterminer les options disponibles dans cette requête
     * Dans cet exemple, il y a deux options: Direct et All
     */
    public String[] getOptions() {
        String[] options = { "Direct", "All" };
        return options;
    }

    /**
     * Méthode appelée par le Design Browser pour exécuter la
     * requête. Le paramêtre "option" qui est passé à la méthode
     * est déterminé par le Design Browser en fonction de l'option
     * qui a été sélectionnée par l'usager
     */
    public void execute(String option) {

        // Test si l'élément sélectionné dans le Design Browser
        // existe et qu'il est d'un type généralisable (voir UML [40])
        // Si c'est le cas, on appelle simplement la bonne méthode
        // sur l'objet en fonction de l'option, i.e., tous les
        // super-types ou seulement les super-types directs.
        if ((getElement() != null) &&
             (getElement() instanceof UmlGeneralizableElement)) {

            if (option.equals("All")) {
                setResults(((UmlGeneralizableElement)getElement())
                    .allSupertypes());
            }
            else if (option.equals("Direct")) {
                setResults(((UmlGeneralizableElement)getElement())
                    .getSupertypes());
            }
        }
    }
}
```

Figure 13 : Exemple d'une requête du *Design Browser*

Les étapes pour ajouter une nouvelle requête sont simples[2]. Premièrement, l'utilisateur doit écrire une sous-classe de la classe *Query* qui est prédéfinie dans SPOOL, et implémenter la méthode abstraite *execute* qui y est définie. La seconde étape consiste à ajouter la requête au *Design Browser*. Pour ce faire, l'utilisateur doit fournir, dans un fichier de configuration, le nom de la classe contenant la requête (avec le nom de son package Java s'il y a lieu) ainsi que les sortes d'éléments sur lesquels la requête peut être exécutée. Une fois ces deux étapes terminées, le *Design Browser* utilise les mécanismes de réflexion fournis par Java [15] pour instancier la nouvelle requête et permettre sa visualisation et son exécution.

Parmi les requêtes déjà fournies par le *Design Browser*, on retrouve celles qui permettent de naviguer à travers la hiérarchie physique de contenance des éléments. Par exemple, une méthode se trouve dans une classe, la classe est dans un fichier, le fichier est dans un répertoire, etc. On peut donc, à partir d'un répertoire, exécuter une requête qui trouve toutes les classes contenues dans celui-ci. On peut aussi, par exemple, exécuter une requête trouvant tous les attributs contenus dans une classe, un fichier, un répertoire et même dans tout le système. D'un autre coté, il est aussi possible de remonter à travers cette hiérarchie en exécutant une requête qui trouve le contenant direct d'un élément (par exemple, trouver la classe contenant une méthode particulière). Cette série de requêtes combinée au mécanisme de filtrage décrit précédemment, permet d'effectuer des recherches puissantes et flexibles dans un système. En effet, les requêtes permettent de trouver le *quoi* (quels éléments), le filtre permet de préciser la recherche (quels noms) et l'élément de départ de la requête spécifie le *où* (dans quel espace), ce qui ajoute un aspect structurel important au type de recherche qui peut être fait à l'aide de l'outil. De plus, comme le filtre qui permet de spécifier le nom peut aussi contenir des caractères spéciaux comme * et ? représentant des caractères quelconques, ceci permet une plus grande flexibilité. Par exemple, la Figure 14 montre comment les méthodes d'accès aux attributs (*get/set*) peuvent

[2] Une description détaillée des étapes à suivre pour l'ajout de nouvelles requêtes dans le *Design Browser* est disponible dans [45].

être récupérées dans un certain contexte (un fichier dans l'exemple). Notons que l'implémentation actuelle du mécanisme de filtrage du *Design Browser* est insensible à la différence entre les minuscules et les majuscules.

Cette combinaison de recherche textuelle et structurelle est une fonctionnalité typiquement absente dans les outils de recherche qui sont entièrement basés sur une recherche textuelle, comme les outils de la famille *grep* sur Unix par exemple. Cette capacité de recherche structurelle est principalement due à l'expressivité du schéma qui est utilisé dans SPOOL, et donc le travail qui doit être fait au niveau des requêtes du *Design Browser* est grandement simplifié.

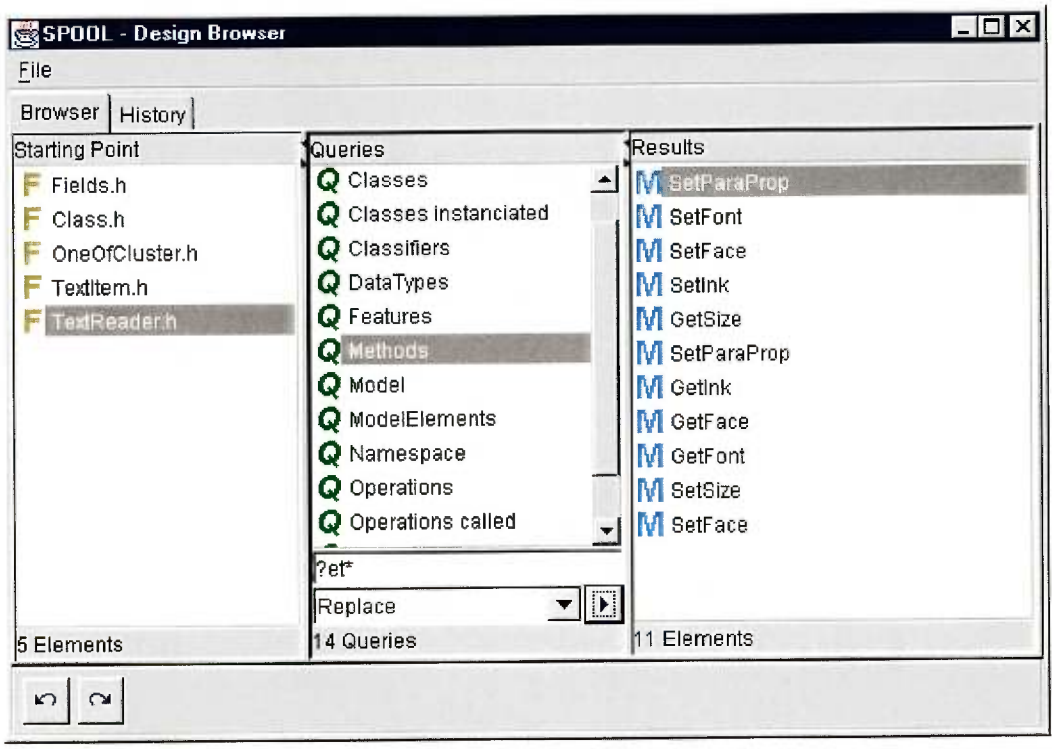

Figure 14 : Recherche textuelle avec le *Design Browser*

Le *Design Browser* peut aussi être utilisé pour retrouver certains éléments propres à la conception du système comme certains patrons de conception par exemple. En effet, nous avons trouvé plusieurs instances du patron *Iterator* [12] dans le système ET++ [13] tout simplement en cherchant l'expression *iter** dans le nom des méthodes. Pour donner un autre exemple, le patron *Observer* peut souvent

être identifié en cherchant pour des noms de méthodes contenant **observ**, **upda** ou **notif** dans un système. Notons tout de même que pour détecter des patrons de conception en général, une méthode systématique et contrôlée par l'humain est nécessaire [19]. Néanmoins, cette méthode simple peut permettre de trouver rapidement quelques instances de patrons de conception et peut fournir un bon point de départ pour de plus amples études. En plus d'aider dans la compréhension de la conception d'un système, le *Design Browser* est aussi utilisé dans le cadre de la récupération de *Hot Spots* dans des cadres d'application [27], dans le cadre d'analyses de l'impact de changement [6] ainsi que dans le cadre de l'évaluation des effets du remplacement de méthodes [16] dans des systèmes orientés objet.

4.4.4 Mécanisme d'interaction entre les outils

Le *Design Browser*, tel que décrit dans les sous-sections précédentes, est un outil qui permet de naviguer et de faire des recherches dans des systèmes. Celui-ci a été développé comme un outil complet en soi, c'est-à-dire qu'il peut être utilisé pour inspecter des systèmes sans avoir recours aux autres outils et mécanismes de l'environnement SPOOL. Par contre, pour être en mesure de supporter l'approche de compréhension telle que décrite dans le chapitre 3, le *Design Browser* peut aussi être utilisé conjointement aux outils d'inspection de l'environnement SPOOL et ainsi permettre la navigation entre différents niveaux d'abstraction. Pour permettre la collaboration entre les outils, plusieurs mécanismes ont été implémentés. L'interaction se fait majoritairement par l'entremise du mécanisme Drag & Drop (DnD) mais aussi par des menus qui ont été ajoutés à l'environnement.

La première sorte d'interaction présente est celle qui permet de démarrer le *Design Browser* à partir de n'importe quel élément dans n'importe quel outil de SPOOL. Il suffit en effet de cliquer sur un élément pour voir apparaître un menu contextuel permettant le démarrage du *Design Browser* avec l'élément en question dans la section *Starting point*. Comme l'environnement SPOOL permet

l'inspection des systèmes à plusieurs niveaux d'abstraction, la possibilité de démarrer le *Design Browser* à partir de n'importe quel niveau apporte un support direct à l'approche *top-down* de compréhension. Par exemple, on peut sélectionner une des classes du diagramme de la Figure 6 en cliquant avec le bouton droit de la souris, puis sélectionner *Browser* dans le menu contextuel qui apparaît. Cela a pour effet de démarrer une instance du *Design Browser* contenant uniquement la classe en question dans sa section *Starting Point* (voir Figure 11), permettant ainsi d'effectuer des requêtes à partir de cette classe.

Le deuxième type d'interaction est celui qui permet de déplacer les éléments entre les outils par simple DnD. Par exemple, on peut prendre un élément du *Design Browser* et le déplacer dans un diagramme de SPOOL ou bien faire cette opération à l'inverse. En fait, tous les outils de SPOOL permettent de démarrer un DnD, mais c'est l'outil receveur qui décide ce qui sera fait des éléments transmis. Si un élément est déplacé vers un diagramme, le diagramme peut le refuser automatiquement (si on tente d'ajouter un fichier dans un diagramme de classes par exemple), il peut le copier (ajouter l'élément dans le diagramme) ou simplement permettre de le localiser s'il existe déjà (en ajoutant une boîte de couleur autour de l'élément dans le diagramme). Si le *Design Browser* est l'outil receveur d'un DnD, alors un menu permettant différentes opérations apparaît (remplacement, union, intersection, etc.). Par exemple, on peut utiliser ce mécanisme pour déplacer des éléments d'une liste à l'autre du *Design Browser*, ou encore entre plusieurs instances du *Design Browser* (La Figure 16 de la Section 5.1 illustre le cas ou la classe *ET_Object* est déplacée par DnD de la section *Results* d'une instance du *Design Browser* (fenêtre 1) vers la section *Starting Point* d'une autre instance de l'outil (fenêtre 2)).

Ce deuxième mécanisme d'interaction rend possible les deux aspects de l'approche de compréhension décrits à la Section 3.1, c'est-à-dire la navigation avec mise en contexte et la composition de contextes personnalisés. La mise en contexte se fait lorsqu'on visualise un élément dans un contexte existant ou dans un contexte prédéfini mais créé sur demande. Par exemple, on peut prendre une

classe qui à été trouvée à l'aide du *Design Browser*, et la localiser dans le diagramme de classes complet ou partiel du système par simple DnD (voir Section 5.1). La composition de contextes permet de mettre ensemble un certain nombre d'éléments d'un système pour visualiser leurs interrelations. Ce deuxième aspect permet de rapidement visualiser les dépendances entre des sous-systèmes et même entre différents éléments d'un même sous-système. Par exemple, on peut récupérer certains fichiers d'un système à l'aide du *Design Browser* et les déposer, par DnD, dans un diagramme d'analyse de dépendances initialement vide, et ainsi visualiser de façon interactive les dépendances entre ces fichiers (voir Section 5.3).

C'est donc ce deuxième mécanisme d'interaction qui apporte un support à l'approche *bottom-up* de compréhension car il permet de remonter certains éléments de bas niveau du système à des niveaux d'abstraction plus élevés. Et donc, la combinaison des deux mécanismes permet de supporter à la fois une approche *top-down* et une approche de compréhension avec mise en contexte de type *bottom-up*.

4.5 La navigation par mise en contextes dans SPOOL

Dans les sections précédentes, nous avons vu comment le *Design Browser* peut être utilisé conjointement au reste de l'environnement SPOOL et comment, grâce à cette collaboration, il est possible de naviguer tout en restant dans un certain contexte. Présentement, le nombre de différents contextes disponibles dans l'environnement SPOOL est restreint. C'est pourquoi le *Context Viewer*, un nouvel outil intégré à l'environnement, a été développé. Cet outil à pour but de permettre l'affichage d'une série de contextes prédéfinis (possiblement synchronisés entre eux) et disponibles aux différents niveaux d'abstraction supportés par l'environnement SPOOL. La Figure 15 montre comment le *Context Viewer* permet l'affichage d'un contexte particulier (ici, plusieurs éléments ont été déposés par DnD dans l'outil, permettant ainsi de voir la hiérarchie de contenance des éléments en question, les uns par rapport aux autres).

4.5.1 Contexte au niveau du code source

Le premier contexte nécessaire est évidemment le code source lui-même. Présentement, le seul moyen d'accéder au code source d'un système à partir de SPOOL est via le mécanisme d'accès à l'environnement SNiFF+. Le *Context Viewer* ne permet pas encore de visualiser le code directement. Dans l'avenir, on voudrait par exemple pouvoir simplement déplacer un élément par DnD dans le *Context Viewer* pour voir immédiatement apparaître le code source à l'endroit où l'élément est défini. Même si cette fonctionnalité n'est pas encore disponible, le passage vers SNiFF+ permet tout de même de visualiser les éléments voulus dans le code source, et donc d'utiliser ce contexte particulier.

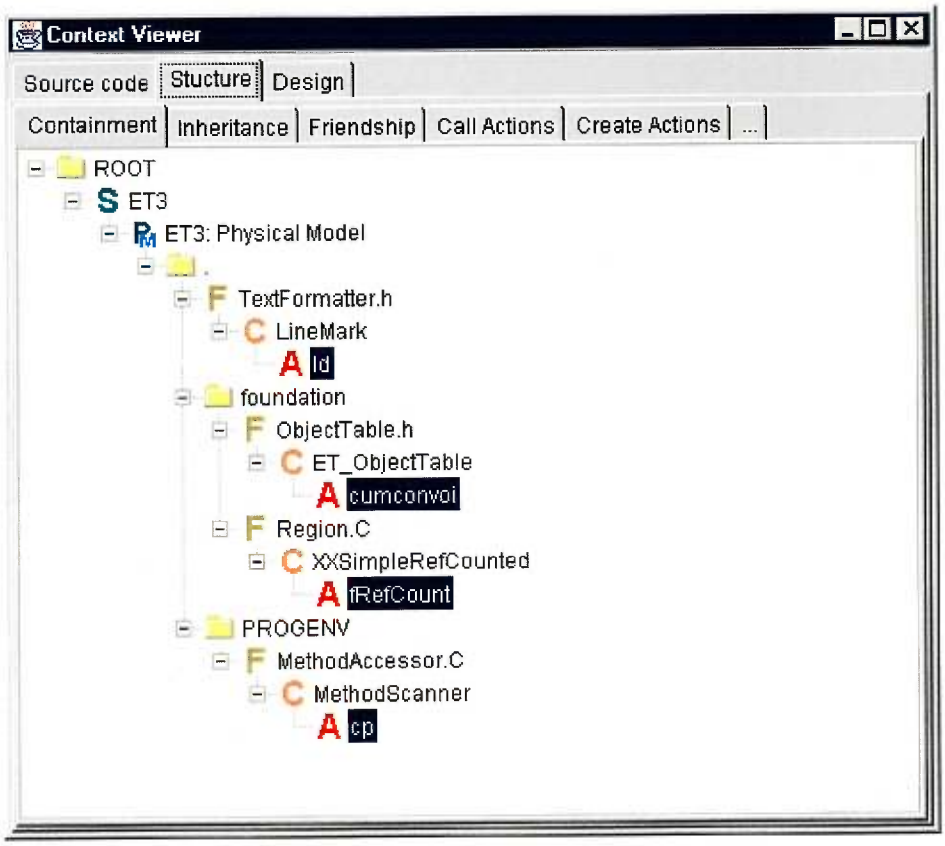

Figure 15 : Le *Context Viewer* de SPOOL

4.5.2 Contexte au niveau de la structure

Pour ce contexte, on distingue une fois de plus les deux types, c'est-à-dire la structure logique et la structure physique des systèmes. Évidemment, on peut aussi utiliser des contextes faisant appel à un mélange des deux types. Par exemple, la hiérarchie de contenance est un type de contexte purement physique. En effet, celui-ci permet de visualiser où se trouve physiquement la définition d'un élément. L'exemple présenté précédemment à la Figure 15 montre, sous forme d'arbre, où se trouvent certains attributs dans un système (dans quelle classe, dans quel fichier, dans quel répertoire, etc.). Autrement dit, la hiérarchie de contenance nous permet de visualiser un élément par rapport à l'élément qui le contient, de façon récursive. Pour les systèmes orientés objet, d'autres types de contextes sont possibles, dont l'arbre d'héritage, le graphe d'appel d'opérations et le graphe de création d'objets. On peut aussi avoir des contextes plus complexes comme l'affichage des implémentations d'une opération (les méthodes) ainsi que les usages faits d'un attribut d'une classe. Certains langages particuliers offrent aussi d'autres possibilités de contextes. Par exemple, la notion d'amis (*friendship*) dans le langage C++ suggère la création d'un contexte permettant de visualiser l'accessibilité de certains éléments par les autres. On peut donc imaginer une multitude de contextes possibles qui pourraient aider à la compréhension. Le genre de contextes qu'on peut visualiser est généralement intimement lié au paradigme de programmation (orienté objet dans notre cas) et au langage utilisés. De plus, on peut envisager la visualisation d'éléments dans différents contextes ayant trait à des aspects dynamiques du système. Par contre, ceci n'est pas disponible présentement dans l'environnement puisque le dépôt de SPOOL ne contient que des informations extraites de façon statique du code source.

4.5.3 Contexte au niveau de la conception

L'environnement SPOOL permet de faire la détection de différents éléments liés à la conception (certains patrons de conception par exemple). On peut donc exécuter certaines requêtes sur le système, permettant ainsi de trouver des

structures particulières qui peuvent ou non collaborer à des concepts liés à la conception du système. Comme le résultat de ces requêtes peut aussi être stocké dans le dépôt de SPOOL, il est possible de vérifier si un élément en particulier fait partie d'une ou plusieurs de ces structures. Le *Context Viewer* permet donc de visualiser des éléments en termes de composants liés à la conception. Par exemple, si un élément fait partie du patron de conception Observer, alors un simple DnD de cet élément dans le *Context Viewer* nous permettra de le visualiser dans le contexte des différents éléments collaborant à ce patron. De la même façon, on peut synchroniser les différentes vues du *Context Viewer* pour permettre de visualiser, pour un ou plusieurs éléments en particulier, quels sont les patrons ou structures auxquels ils collaborent, permettant ainsi de savoir comment ils participent à la conception du système. Cette nouvelle approche devrait grandement accélérer la compréhension, surtout lorsqu'on cherche à déterminer et comprendre le rôle que joue certains éléments en particulier dans un système.

Les différents contextes décrits dans les sections précédentes ont tous été conçus mais seuls quelques contextes majeurs ont été implementés dans SPOOL jusqu'ici. Parmi les contextes qui restent à implémenter, on trouve ceux traitant de la structure d'un système, comme l'affichage des classes par dépendances. Par exemple, le contexte affichant toutes les super-classes et toutes les sous-classes d'une classe à déjà été implémenté dans SPOOL, mais le même type de contexte devra être implémenté pour toutes les formes de dépendance (appels d'opérations, créations d'objets, références, *friendship*, etc.). De plus, des contextes au niveau des méthodes et des opérations d'un système (comme les graphes d'appels, ou les contextes affichant le polymorphisme par exemple) devront aussi être implémentés, en plus des contextes liés à la conception décrits précédemment.

Chapitre 5 : Études de cas

Le chapitre précédent décrivait comment l'environnement SPOOL supporte la navigation et la recherche à travers les éléments et les relations d'un système, par l'entremise du *Design Browser*. Il décrivait aussi comment les différents mécanismes de l'environnement, combinés aux fonctionnalités du *Context Viewer*, peuvent servir à l'étude de systèmes logiciels en supportant l'approche de compréhension par mise en contexte. Ce chapitre présente, en quatre sections, quatre études de cas qui montrent comment le *Design Browser* et le *Context Viewer* peuvent être utilisés dans l'environnement SPOOL pour inspecter des systèmes logiciels, et comment ceux-ci peuvent aider à comprendre des aspects importants de ces systèmes. Le système qui a été utilisé pour ces études de cas est le cadre d'application ET++ [13], un système qui comporte quelques 720 classes réparties dans plus de 480 fichiers. Les deux premières études de cas traitent de la navigation par mise en contexte à l'aide de la visualisation dans SPOOL. Le troisième traite de la création de contextes dans le cadre d'une analyse de dépendances. Finalement, le quatrième traite de l'analyse d'un système au niveau de la conception par la récupération de patrons de conception et de leur analyse.

5.1 Navigation par mise en contexte simple

L'environnement SPOOL peut être utilisé de façon flexible pour faire l'étude d'un système. Dans cette section, les outils de SPOOL sont utilisés afin d'étudier certaines particularités liées au clonage des objets, tel qu'implémenté dans le système ET++. Comme point de départ de cette étude, prenons la classe *ET_Object* de ET++ qui est la super-classe racine de la majorité des autres. En ayant une racine unique, les différentes classes de ET++ peuvent utiliser des

mécanismes communs. Par exemple, la classe *ET_Object* fournit les mécanismes nécessaires pour permettre aux objets de s'observer les uns les autres en suivant le patron de conception *Observer* [12], elle fournit des mécanismes de comparaison entre les instances, elle permet la conservation de méta-information sur le système, etc. Elle fournit aussi un mécanisme de clonage, permettant de dupliquer les objets, par l'entremise d'une méthode *Clone*. On pourrait donc être intéressé à voir s'il y a des classes qui redéfinissent cette méthode et, s'il en existe, de savoir pourquoi elles le font.

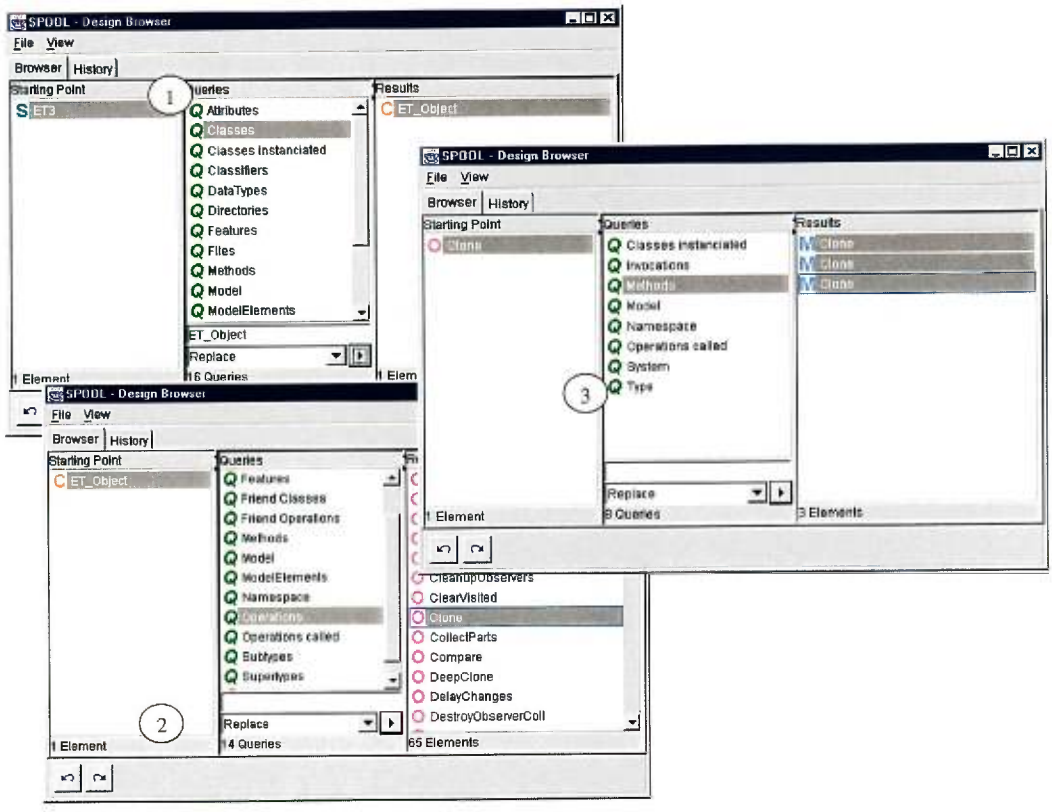

Figure 16 : Exploration à l'aide du *Design Browser*

Les différentes fenêtres de la Figure 16 permettent de voir les premières étapes nécessaires à l'exploration de ET++ dans ce but. La fenêtre 1 montre comment, en partant du système, on peut exécuter une requête qui récupère toutes les classes et comment à l'aide du mécanisme de filtrage du *Design Browser* on peut trouver directement la classe recherchée. La fenêtre 2 montre comment récupérer toutes les opérations de la classe en question, et finalement la fenêtre 3 illustre comment

récupérer les différentes implémentations de l'opération *Clone*. Notons que ces trois étapes sont reproduites dans des fenêtres différentes du *Design Browser* pour les besoins de la présentation, mais il est possible d'utiliser uniquement une instance de l'outil en déplaçant les éléments voulus de la liste de droite vers celle de gauche.

On peut maintenant utiliser le résultat obtenu dans la dernière fenêtre pour continuer l'exploration. Puisqu'on possède les trois méthodes associées à l'opération *Clone*, on peut utiliser le mécanisme d'interaction disponible dans SPOOL pour les visualiser dans le diagramme de classes du système, comme montré dans fenêtre 1 de la Figure 17.

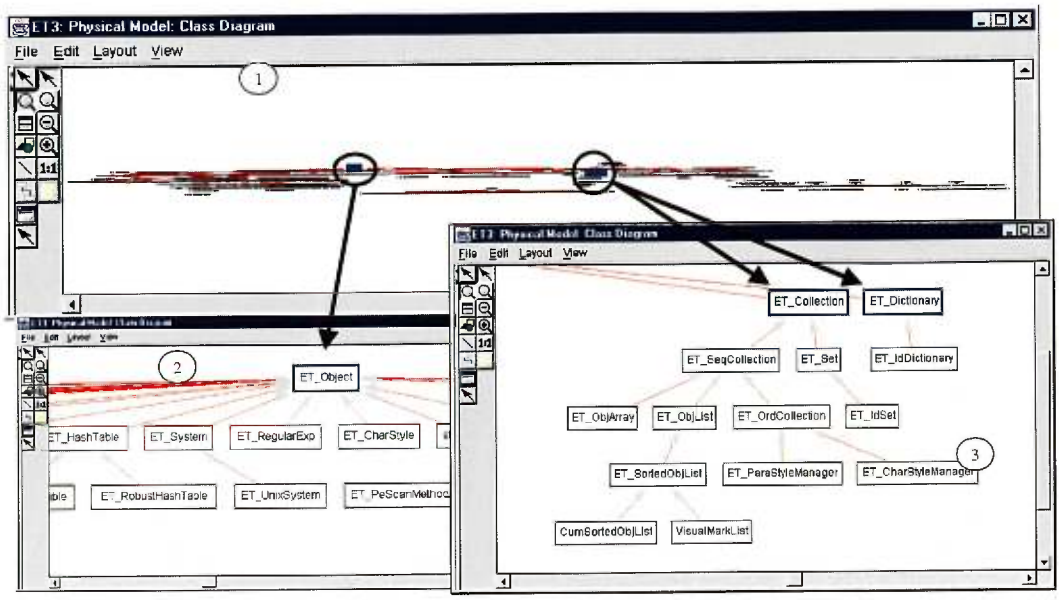

Figure 17 : Mise en contexte dans un diagramme de classes

En faisant un DnD des trois méthodes dans le diagramme, les classes qui les contiennent sont mises en évidence par une boîte de couleur permettant de les repérer facilement (fenêtre 1). Ensuite, on peut agrandir au besoin les parties du diagramme qui nous intéressent à l'aide du mécanisme d'agrandissement et de rétrécissement qui est disponible dans tous les diagrammes de SPOOL. La fenêtre 2 nous montre un agrandissement de la première boîte qui est en fait la classe *ET_Object*. Ce résultat était attendu puisque cette classe fournit le mécanisme de

clonage par défaut à toutes les sous-classes. Ce qui est plus intéressant, ce sont les classes visualisés dans la fenêtre 3. En effet, les classes *ET_Collection* et *ET_Dictionary* qui sont visualisés redéfinissent l'opération *Clone*, ce qui signifie que toutes leurs sous-classes auront la nouvelle version de cette fonctionnalité plutôt que celle qui est définie par défaut dans la classe *ET_Object*.

Une fois qu'on a récupéré cette information, on peut se demander pourquoi les deux classes visualisées introduisent un comportement différent. L'explication vient du fait que les classes *ET_Collection* et *ET_Dictionary* représentent toutes les deux le concept de contenant pour d'autres objets. Il est donc important de pouvoir non seulement cloner le contenant mais aussi de pouvoir ajouter au clone tous les éléments qui sont à l'intérieur de l'original. De plus, les concepts de collection et de dictionnaire étant assez différents (dans un dictionnaire, on associe une clé à une valeur, contrairement à une collection où les objets ne sont pas liés entre eux), leurs mécanismes de clonage respectifs sont différents. Le bout de code en C++ qui suit est tiré de ET++ et illustre le mécanisme de clonage tel que défini dans la classe *ET_Dictionary*[3] :

```
1.  Object *Dictionary::Clone()
2.  {
3.          Dictionary *clone;
4.          Object *key, *value;
5.          clone= (Dictionary *) New();
6.          DictionaryIterator next(this);
7.          while (key= next()) {
8.                  value= next.GetValue();
9.                  clone->PutAtKeyIfAbsent(value, key);
10.         }
11.         return clone;
12. }
```

Dans ET++, le clonage d'un dictionnaire se fait donc en créant un nouvel objet de ce type (ligne 5) puis, en faisant une itération sur les clés du dictionnaire original (lignes 6 à 10), on ajoute les valeurs une à une au nouveau dictionnaire (ligne 9).

[3] Certaines classes du système ET++ sont nommées avec le préfixe « ET_ » dans l'environnement SPOOL alors que dans le code source on les retrouve sans ce préfixe. La raison de cette différence

L'environnement SPOOL permet donc de mettre l'information extraite d'un système dans différents contextes, ce qui permet d'avoir différentes vues de celui-ci, facilitant ainsi la compréhension. L'exemple présenté précédemment montre que l'environnement SPOOL permet de trouver et de visualiser rapidement ce qu'on cherche, mais qu'il faut aussi une certaine analyse de la part de l'utilisateur pour faire une bonne interprétation des résultats.

5.2 Navigation par mise en contexte avancée

Dans la section précédente, nous avons vu comment la visualisation d'éléments obtenus à l'aide du *Design Browser* pouvait permettre de comprendre rapidement certains aspects d'un système. Leur visualisation dans un contexte plus grand (le diagramme de classes du système par exemple) permet de mieux les situer, surtout si le système est très grand. Par contre, on peut vouloir limiter le surplus d'information qui est affiché et ne garder que l'essentiel. Le *Context Viewer* a été conçu dans cette optique, c'est-à-dire réduire le bruit lors de la visualisation des éléments d'un système. L'exemple qui suit illustre l'utilisation du *Context Viewer* à des fins d'étude du système ET+ par la visualisation de certaines classes dans deux contextes différents. Ceci permet ainsi d'identifier le rôle de certaines de ces classes de façon directe.

En général dans un système orienté objet de grande taille, la structure physique ne correspond pas à la structure logique. Autrement dit, plusieurs classes peuvent être dans le même fichier, et des classes se trouvant physiquement dans un répertoire peuvent avoir des liens d'héritage avec des classes d'un autre répertoire. Cette différence entre structure physique et logique est donc tout à fait normale et attendue. Par contre, certaines de ces « différences » peuvent être très utiles pour aider à comprendre un système si celles-ci sont visualisées de façon efficace. Prenons l'exemple des deux classes *ET_Collection* et *ET_Dictionary* que nous avions dans l'exemple précédent. Nous avons vu que ces deux classes

---

vient du fait que l'information disponible dans SPOOL provient du code pré-compilé de ET++ et que certaines macros ajoutent ce préfixe aux noms des classes.

redéfinissent le comportement de clonage tel que défini dans leur super-classe *ET_Object*. Par le diagramme de classes montré à la Figure 17, on peut rapidement voir quelles sont les sous-classes qui héritent de ce nouveau comportement. Par contre, on ne voit pas où ces classes sont physiquement. Évidemment, on peut toujours utiliser le *Design Browser* pour les localiser dans le système, mais l'utilisation du *Context Viewer* permet de récupérer et de visualiser cette information plus efficacement.

La Figure 18 nous montre comment récupérer toutes les super-classes et toutes les sous-classes de *ET_Collection* et *ET_Dictionary* en exécutant deux requêtes de façon simultané, ce qui permettra ensuite de les visualiser dans le *Context Viewer*.

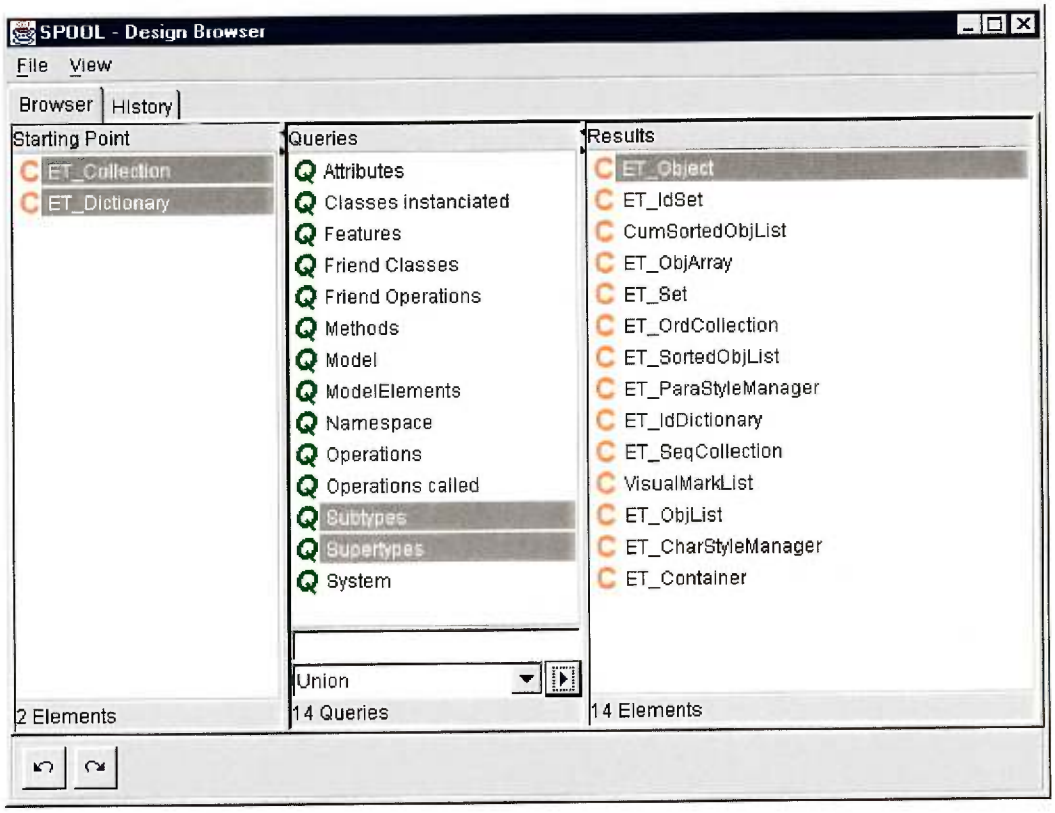

Figure 18 : Récupération de l'héritage à l'aide du *Design Browser*

Comme le montre la Figure 19, les classes récupérées à l'aide du *Design Browser* peuvent être déplacées dans le *Context Viewer* dans les sections *Containment* et *Inheritance* pour continuer l'étude. De ces deux diagrammes on peut tirer un

nombre important d'informations utiles à la compréhension. Par exemple, la fenêtre 2 nous montre seulement les sous-classes et les super-classes de *ET_Collection* et *ET_Dictionary*, ce qui permet de voir tout de suite leur profondeur dans la hiérarchie. On voit aussi tout de suite que ces deux classes ont une seule super-classe (l'héritage multiple étant permis en C++) et qu'elle est commune aux deux. Évidemment, ces informations étaient aussi disponibles dans le diagramme de classe complet du système, mais elles étaient beaucoup plus diffuses.

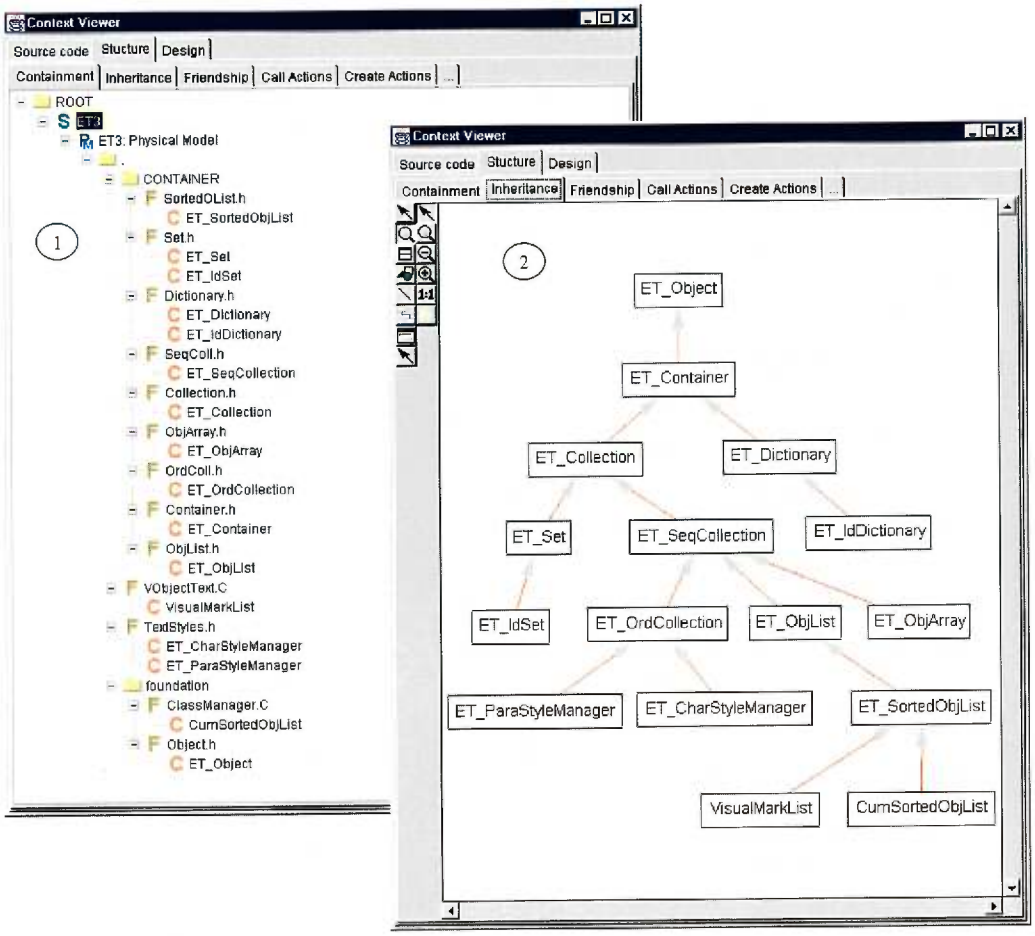

Figure 19 : Localisation des éléments avec le *Context Viewer*

Le diagramme montré à la fenêtre 1 permet de visualiser les différentes classes dans leur hiérarchie de contenance. On peut donc voir tout de suite quelles sont les classes qui sont dans le même répertoire ou dans le même fichier. De plus, en

comparant les diagrammes de la fenêtre 1 et de la fenêtre 2, on peut extraire d'autres informations utiles. Par exemple, on voit que les classes qui ne sont pas des feuilles de l'arbre d'héritage dans la fenêtre 2 sont toutes dans le répertoire *CONTAINER* à la fenêtre 1 (à l'exception de *ET_Object*). Cette observation s'explique du fait que toutes ces classes représentent des structures de données et que les classes au bas de la hiérarchie sont en fait des spécialisations de ces structures à des fins concrètes. Par exemple, on voit que *ET_CharStyleManager* et *ET_ParaStyleManager* semblent être spécialisées pour gérer des collections ordonnées de styles de texte. En fait, cette déduction vient seulement du fait que ces deux classes héritent de *ET_OrdCollection* qui représente une collection ordonnée et que le fichier où elles sont définies s'appelle *TextStyles.h.* Évidemment, une inspection plus approfondie du code source (en utilisant le mécanisme d'accès à SNiFF+ par exemple) permet de confirmer cette hypothèse.

On voit donc par cet exemple que l'utilisation des mécanismes de mise en contexte lors d'une étude peut être d'une aide considérable à la compréhension. Cependant, le *Context Viewer* de SPOOL ne fournit présentement que quelques contextes, ce qui restreint l'utilisation qu'on peut en faire. Par contre, on peut imaginer que si l'outil nous permettait de visualiser plusieurs autres contextes, un utilisateur expérimenté pourrait augmenter considérablement sa vitesse de compréhension. Par exemple, si on pouvait visualiser le polymorphisme en montrant seulement les nouvelles méthodes ou les méthodes réécrites dans des sous-classes, on pourrait tout de suite voir dans notre exemple quelles sont les spécialisations qui sont apportées à *ET_OrdCollection* dans les sous-classes *ET_CharStyleManager* et *ET_ParaStyleManager*. Cela permettrait entre autres de rapidement visualiser la nature des spécialisations dans une hiérarchie de classes, et de détecter éventuellement certaines défectuosités de la conception d'un système, en plus d'aider à sa compréhension.

5.3 Analyse de dépendances par création de contextes

Nous avons vu précédemment comment la visualisation par contextes prédéfinis peut être utile à la compréhension. Maintenant, nous allons voir un exemple pour lequel le contexte qui sera utilisé n'est pas prédéfini. Autrement dit, le contexte sera construit de façon semi-automatique par l'usager, en utilisant le *Design Browser* comme fournisseur d'information à différents diagrammes. De cette façon nous serons en mesure d'identifier ce qui semble être une anomalie dans la conception de ET++ par une simple analyse de dépendances. Puis, nous pourrons ensuite comprendre les raisons qui ont poussé les développeurs à agir de la sorte.

Le point de départ de cette étude est le diagramme montré dans la fenêtre 1 de la Figure 20. Celui-ci montre les liens d'héritage entre les classes, mais accumulés au niveau des répertoires contenants les fichiers et le code source du système. Autrement dit, si deux répertoires dans le diagramme sont reliés par un lien d'héritage, cela signifie qu'au moins une classe contenue dans un de ces répertoires hérite d'une classe contenue dans l'autre répertoire. De cette façon, on peut rapidement visualiser différents types de dépendances entre les répertoires et entre les fichiers. De plus, à l'aide de la boîte de dialogue montrée à la fenêtre 2, on peut même choisir le type de lien à visualiser. Dans l'exemple de la Figure 20, seules les généralisations sont affichées, c'est-à-dire uniquement les liens d'héritage.

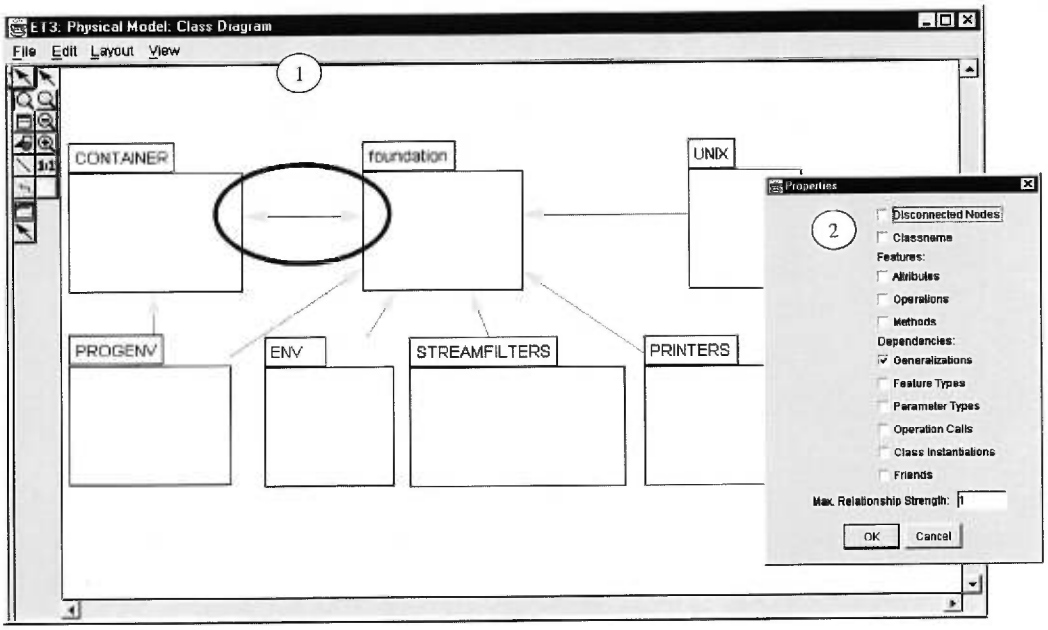

Figure 20 : Accumulation des dépendances au niveau des répertoires

L'information que nous avons recueillie dans les exemples des deux sections précédentes nous permet de comprendre pourquoi la majorité des liens d'héritage du diagramme sont unidirectionnels. En effet, comme la classe *ET_Object* est la classe racine des autres et que celle-ci se trouve dans le répertoire *foundation*, on s'attend à ce que plusieurs répertoires aient un lien d'héritage dans cette direction. De plus, le répertoire *foundation* comme son nom l'indique, contient les éléments qui servent de fondation au reste du système, ce qui explique pourquoi les liens d'héritage sont unidirectionnels. Malgré cela, il y a tout de même un des liens du diagramme qui ne l'est pas. On est donc porté à se demander pourquoi certaines classes du répertoire *foundation* héritent des classes contenues dans le répertoire *CONTAINER*.

Encore une fois, pour être en mesure d'aller chercher l'information dont on a besoin, on peut utiliser la mise en contexte. Cette fois-ci par contre, l'approche consiste à créer le contexte plutôt que d'en utiliser un qui est prédéfini. Comme on cherche à inspecter les classes de *foundation* qui héritent de *CONTAINER*, on veut mettre ces différents éléments dans un même contexte. Ceci peut être obtenu de différentes façons. L'une d'elles consiste à utiliser le *Design Browser* pour

récupérer toutes les classes contenues dans le répertoire *foundation*. Une fois ces classes récupérées, on peut tout simplement les déplacer dans un diagramme de classes initialement vide, comme montré dans la fenêtre 1 de la Figure 21. Ensuite, on peut déplacer le répertoire *CONTAINER* par DnD de la même façon, pour que celui-ci se retrouve dans le même diagramme (fenêtre 2). Enfin, la fenêtre 3 montre un agrandissement de la partie du diagramme qui nous permet maintenant de voir les classes qui héritent de *CONTAINER*.

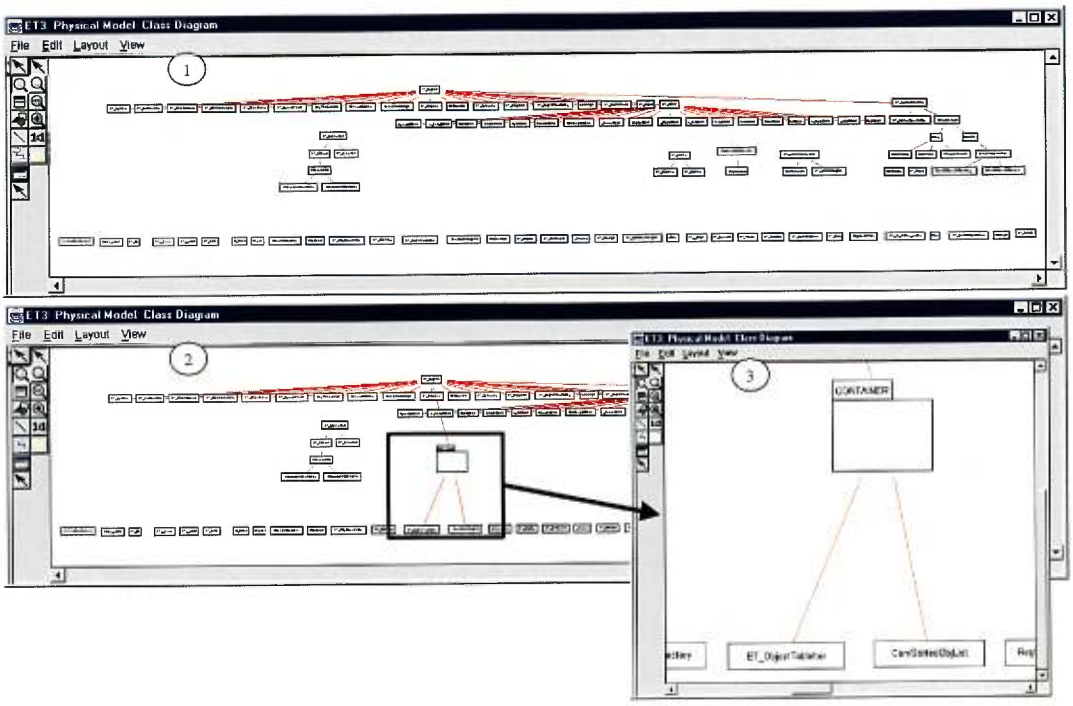

Figure 21 : Mise en contexte de différents éléments

Comme on le voit dans le diagramme, il y a uniquement deux classes du répertoire *foundation* qui héritent de *CONTAINER*. De plus, une de ces classes nous est déjà familière puisque nous l'avons rencontrée dans les deux exemples précédents. Comme le laissent suggérer les diagrammes de la Figure 19, la classe *CumSortedObjList* semble être utilisée comme structure de données à des fins de gestion de méta-information. En effet, dans le système ET++, de la méta-information sur le système est disponible lors de l'exécution, d'une façon semblable à ce que permet le mécanisme de réflexion disponible en Java [15]. C'est probablement pourquoi les concepteurs de ce système ont préféré introduire

un couplage du répertoire *foundation* vers le répertoire *CONTAINER* en réutilisant une structure de données déjà existante plutôt que de la réécrire.

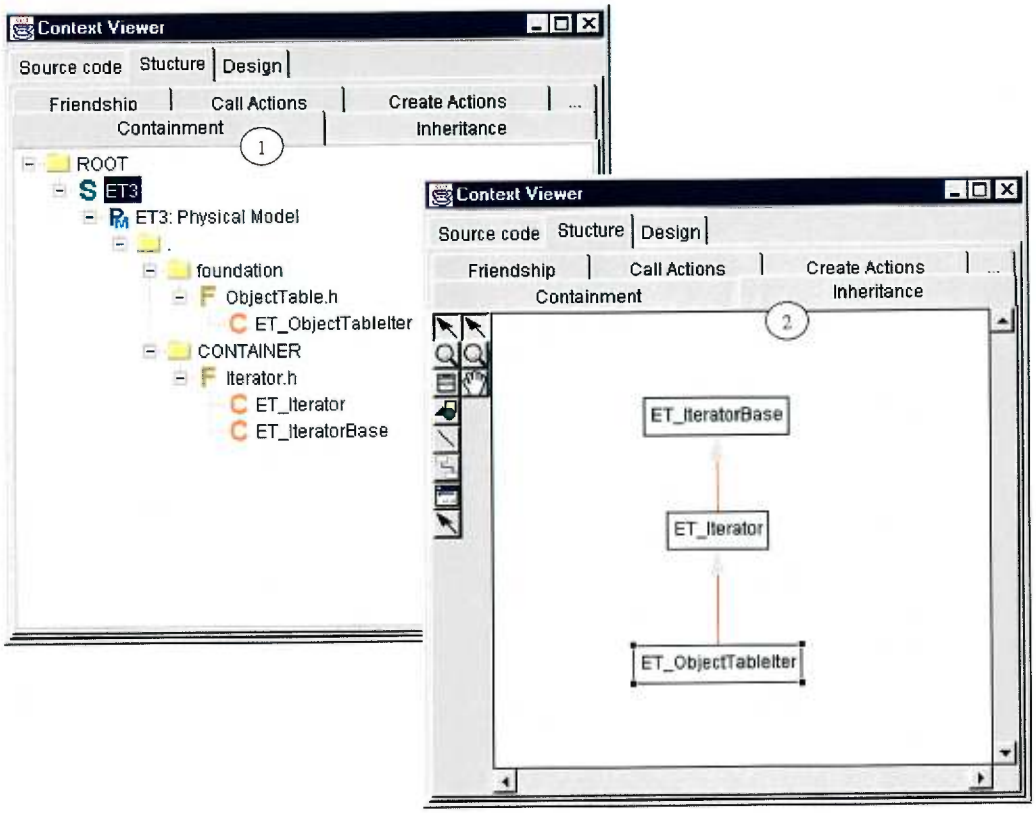

Figure 22 : Nouvelle utilisation du *Context Viewer*

Pour ce qui est de la classe *ET_ObjectTableIter*, on doit récupérer plus d'informations pour être en mesure d'identifier la raison de sa présence dans le répertoire *foundation*. La Figure 22 nous montre à nouveau le *Context Viewer* mais cette fois-ci, ce sont les sous-classes et les super-classes de *ET_ObjectTableIter* qui y sont mises en contexte. On voit tout de suite que sa super-classe immédiate est la classe *ET_Iterator* qui est en fait une implémentation de base pour l'utilisation du patron de conception *Iterator* [12]. Ce patron est normalement utilisé lorsqu'on veut donner accès aux objets contenus dans une collection, mais sans donner accès à la collection elle-même. On peut donc déduire encore une fois que les concepteurs de ET++ ont eu besoin d'une fonctionnalité qui était déjà implémentée dans le répertoire *foundation*. Ils

l'ont simplement réutilisée même si cela a entraîné un couplage supplémentaire de *foundation* vers *CONTAINER*.

5.4 Exploration de la conception

Dans beaucoup de systèmes orientés objet, les décisions clés liées à la conception sont basées sur des implémentations de patrons de conception bien connus, comme ceux de Gamma et al. [12], de Buschmann et al. [5] et de Schmidt [29]. La détection de certains de ces patrons dans un système comme ET++ permet de comprendre rapidement les concepts qui y ont été implémentés. La version actuelle de SPOOL permet de chercher plusieurs des patrons de Gamma et al. dans un système (voir Tableau 4), tout simplement en considérant leur structure. Malheureusement, la détection de la structure d'un patron n'assure pas qu'il s'agisse en effet du patron qu'on tente de détecter. Pour que ce soit le cas, il faut vérifier que l'intention derrière la structure en question est bel et bien celle du patron. L'exemple qui suit illustre cet aspect de la récupération d'éléments liés à la conception d'un système. Entre autres, nous verrons dans le cas du patron de conception *Bridge*, comment il est possible d'utiliser la mise en contexte dans SPOOL pour vérifier si une structure détectée représente réellement un *Bridge* dans le système.

Template Method
Factory Method
Strategy / State
Bridge
Observer
Proxy
Adapter

Tableau 4 : Quelques patrons de conception détectés par SPOOL

Gamma et al. décrivent l'intention du patron de conception *Bridge* comme étant de découpler une abstraction de son implémentation pour que chacune d'elles puissent varier indépendamment. Le *Bridge* est une technique normalement utilisée pour éviter la réplication de hiérarchies et l'explosion du nombre de classes qui en résulte, lorsque différentes variations d'un même concept peuvent être implémentées de façon différentes. La Figure 23 présente la structure des classes qui collaborent dans la structure de ce patron de conception. Normalement, les différentes opérations définies dans l'abstraction délèguent leurs fonctionnalités à des opérations définies dans l'implémentation. Dans la Figure 23, on voit que la méthode *Operation* définie dans la classe *Abstraction* fait un appel à *OperationImp* de la classe *Implementor* via le lien d'agrégation *imp*, et on voit que cette opération est redéfinie dans les sous-classes *ConcreteImplementor1* et *ConcreteImplementor2*.

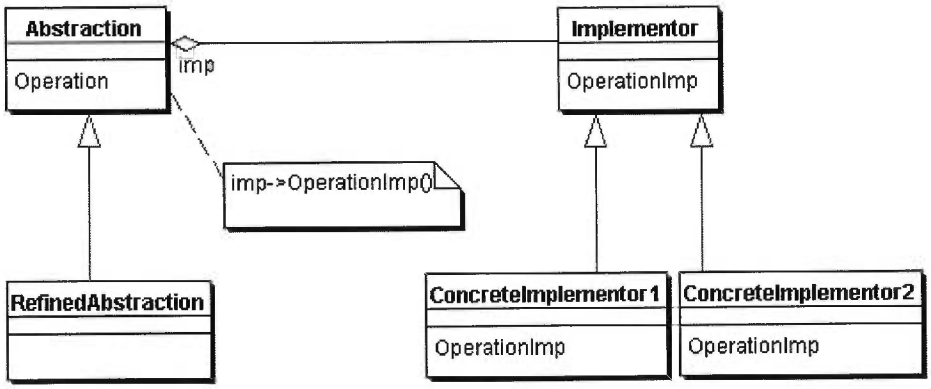

Figure 23 : Structure du patron de conception *Bridge*

SPOOL permet la détection de structures complexes dans des systèmes orientés objet [19]. On peut donc inspecter le système ET++ à l'aide d'une requête détectant la structure décrite à la Figure 23. Cependant, la détection de cette structure dans un système n'assure pas qu'il s'agisse effectivement du patron de conception. En effet, pour que la structure soit effectivement un Bridge, il faut que l'intention derrière sa conception soit de découpler l'abstraction de l'implémentation. Par exemple, SPOOL permet de détecter 46 instances de la

structure du *Bridge* dans ET++ [19], mais il est évident que seules quelques unes de ces structures ont réellement l'intention du *Bridge*. La fenêtre 1 de la Figure 24 montre le diagramme de classes du système ET++ dans lequel des boîtes ont été dessinées autour des classes détectées comme représentant la classe *Abstraction* de la structure d'un *Bridge*. La grosseur des boîtes est calculée en fonction du nombre de méthodes de la classe *Abstraction* qui délèguent leur fonctionnalité à celles de la classe *Implementor*. Ceci permet de visualiser les classes déléguant beaucoup de leurs fonctionnalités, et donc de rapidement identifier les *Bridges* potentiels. En effet, si seulement quelques méthodes délèguent leur fonctionnalité, la probabilité qu'il s'agisse d'un Bridge est moins grande, surtout si la classe contient beaucoup d'autres méthodes qui sont appelées à partir d'autres endroits dans le système.

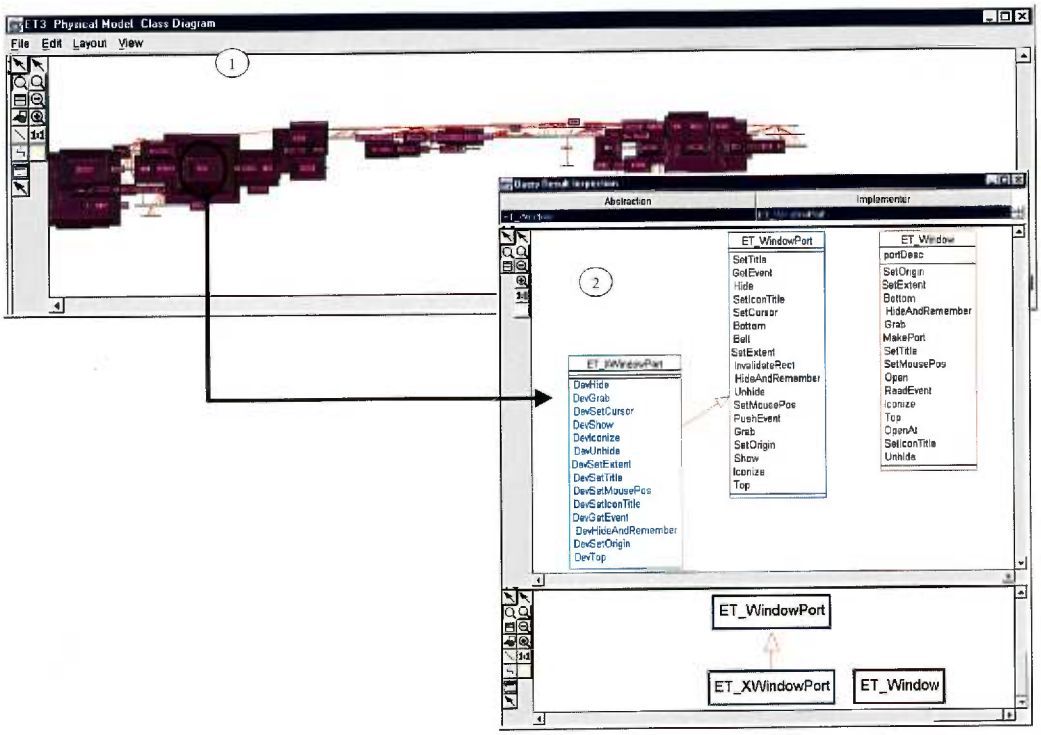

Figure 24 : Détection et inspection du patron de conception *Bridge*

La fenêtre 2 montre l'outil d'inspection de SPOOL démarré à partir d'une des boîtes du diagramme de la fenêtre 1. Cet outil nous montre, avec différentes couleurs prédéfinies, les différents éléments qui participent à la structure détectée.

Par exemple, la classe représentant l'abstraction est entourée en rouge (*ET_Window*), tandis que celle représentant l'implémentation est en bleu (*ET_WindowPort*), et sa spécialisation est en vert (*ET_XWindowPort*). L'inspection de la structure à l'aide de cet outil permet d'aider à identifier s'il s'agit effectivement d'un *Bridge*. La première étape consiste à simplement regarder les conventions de noms utilisées pour tenter d'avoir une idée de ce que sont les fonctionnalités reliées à ces classes. Évidemment, dans beaucoup de cas on ne peut pas se fier uniquement aux noms de méthodes et de classes pour confirmer ou infirmer la présence d'un patron de conception. Par contre, l'utilisation d'un outil comme le *Design Browser* peut aider dans cette tâche en donnant la flexibilité permettant d'inspecter d'autres aspects de la structure détectée. Dans le cas du *Bridge*, on peut par exemple utiliser l'outil pour vérifier certaines propriétés. La fenêtre 1 de la Figure 25 montre comment, à l'aide du *Design Browser*, on peut récupérer les opérations définies dans la classe *ET_WindowPort* qui participent à la structure détectée (L'icône dénotée *CR* signifie que l'objet est un *ClassifierRole* et que, dans ce cas particulier, celui-ci sert à représenter la classe *ET_WindowPort* mais avec seulement les opérations participant au patron détecté).

Ensuite, la fenêtre 2 nous montre comment récupérer les différentes méthodes du système qui invoquent les opérations trouvées précédemment. Comme la classe représentant l'*Implementor* du *Bridge* est normalement accédée via la classe représentant l'*Abstraction*, on peut calculer de cette façon la proportion des appels de ces opérations qui le sont réellement via l'abstraction. En visualisant les différentes méthodes dans le diagramme de classes du système (fenêtre 3), on peut enfin voir si les appels sont fait majoritairement à partir de la classe *Abstraction* ou non.

À partir d'un diagramme de classe (comme celui de la Figure 25), on peut agrandir celui-ci pour identifier les classes entourées par des boîtes et vérifier si leur interaction avec la structure détectée fait en sorte que l'intention n'est pas celle du patron de conception. On peut aussi utiliser le *Context Viewer* pour

vérifier ces classes plus en détail. Par exemple, la Figure 26 montre d'une part les différentes méthodes trouvées avec le *Design Browser* dans leur hiérarchie de contenance (fenêtre 1) et d'autre part le diagramme de classes contenant comme contexte uniquement les classes impliquées et leurs super-classes.

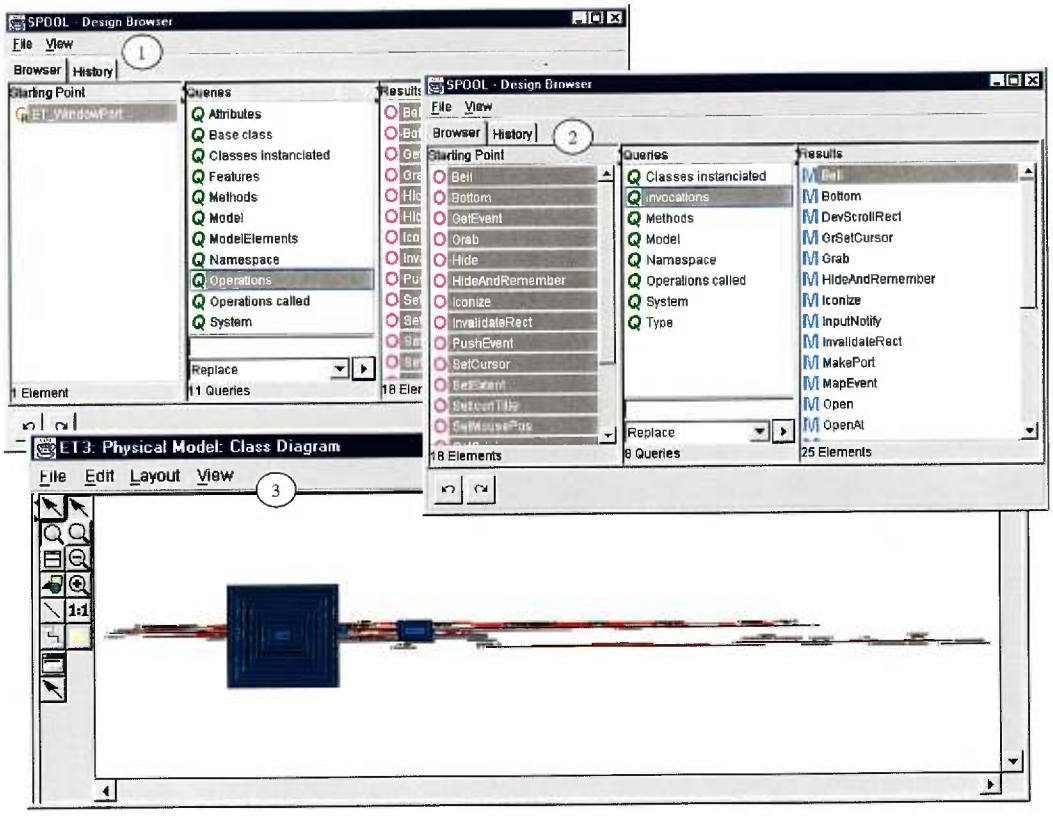

Figure 25 : Inspection du *Bridge* à l'aide du *Design Browser*

À partir de cette dernière figure, on peut voir que la majorité des méthodes se trouvent dans la classe *ET_Window*, ce qui est bon signe puisque c'est la classe qui représente l'abstraction du *Bridge*. Trois autres méthodes sont dans la classe *ET_WindowPort*, ce qui n'est pas un problème puisque cette classe représente la classe *Implementor* du patron de conception et parce que ses méthodes peuvent évidemment en appeler d'autres à l'intérieur de la même classe sans briser l'intention du *Bridge*. Les méthodes contenues dans les classes *ET_Port* et *ET_XWindowPort* ne brisent pas non plus l'intention puisque, comme on le voit dans la fenêtre 2 de la Figure 26, ces deux classes sont en fait la super-classe et la sous-classe de *ET_WindowPort* respectivement. La seule méthode qui reste est

définie dans un *Utility*, ce qui veut dire que c'est une fonction ne faisant partie d'aucune classe. Par contre, on voit que cette méthode est définie dans le fichier *Port.h*, le même fichier qui contient la classe *ET_Port*.

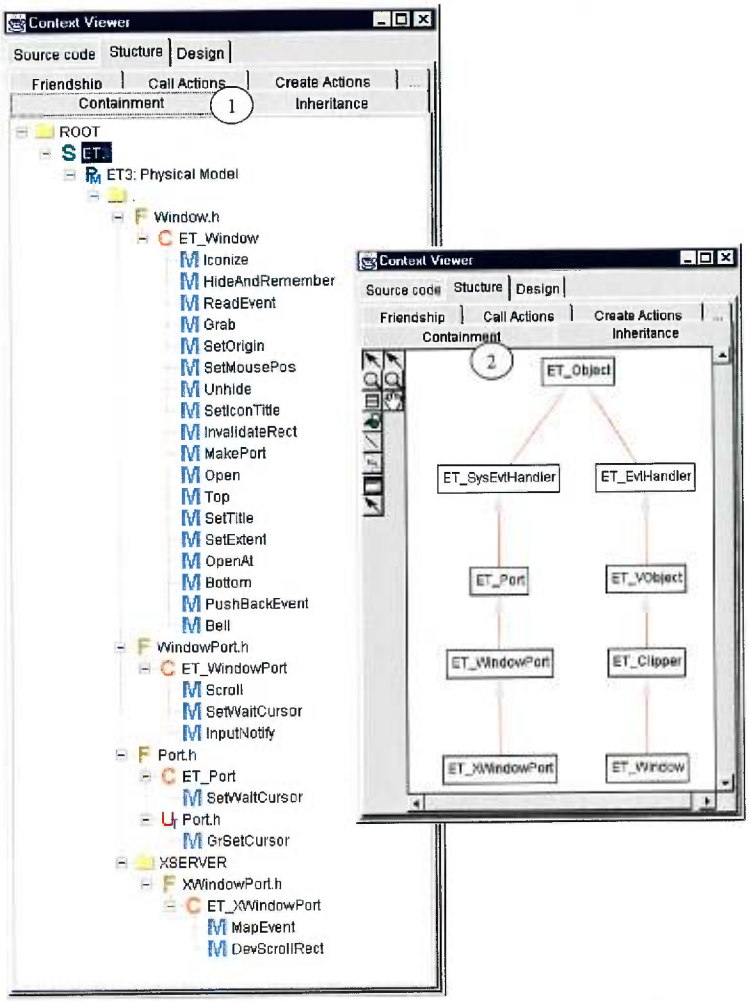

Figure 26 : Inspection du *Bridge* à l'aide du *Context Viewer*

Après ces différentes explorations, on peut donc affirmer avec une bonne probabilité que les classes *ET_Window*, *ET_WindowPort* et *ET_XWindowPort* trouvées lors de la détection de la structure de la Figure 23, implémentent bel et bien ce patron de conception. Évidemment, seule une inspection détaillée du code source permettrait de l'affirmer à 100% si aucune autre source de documentation n'est fournie. Heureusement, pour ce qui est du système ET++ de la documentation existe, qui confirme l'existence de ce *Bridge*. De plus, Gamma et

al. [12] présentent cette instance du système ET++ dans la section « known uses » de leur description du patron *Bridge*, ce qui est une autre confirmation. Dans ET++, la classe *ET_Window* représente l'abstraction d'une fenêtre dans une application graphique, tandis que la classe *ET_XWindowPort* est un « port » de cette fenêtre pour le système de fenêtrage *XWindow*. La classe *ET_WindowPort* sert de super-classe à tous les ports possibles d'une fenêtre. Par exemple, on pourrait avoir une autre sous-classe de *ET_WindowPort* qui s'appellerait *ET_MotifWindowPort* et qui fournirait l'implémentation d'une fenêtre pour le système Motif.

Chapitre 6 : Évaluation

Les deux contributions majeures de ce mémoire sont l'approche de compréhension par mise en contexte et le développement des outils qui ont été intégrés à l'environnement SPOOL comme support pour la compréhension des logiciels. Ce chapitre discute d'abord notre approche par rapport aux modèles de compréhension présentement acceptés par la communauté scientifique (cf. Section 2.1), puis présente une comparaison des outils développés dans SPOOL avec les environnements les plus représentatifs de l'état de l'art dans le domaine (cf. Section 2.2). Finalement, les avantages et limites de l'approche et des outils développés sont présentés.

6.1 Approche de compréhension

L'approche de compréhension présentée dans ce mémoire est principalement basée sur le fait qu'on peut visualiser des éléments de bas niveau dans des contextes de plus haut niveau comme la structure physique d'un système ou comme sa conception. On peut donc voir cette approche comme un complément aux différents modèles de compréhension présentés dans le Chapitre 2 qui sont des processus *bottom-up*. En effet, comme l'approche prévoit la possibilité de visualiser des éléments à différents niveaux d'abstraction, cela signifie qu'on peut prendre des éléments « bien connus » du système et les mettre dans des contextes plus larges afin de permettre une compréhension plus globale. De plus, si des contextes sont prédéfinis et connus du programmeur (comme l'affichage automatique des super-classes et des sous-classes d'une classe par exemple, ou bien l'affichage des patrons de conceptions auxquels participe un élément),

l'utilisation de ces contextes apporte aussi un support considérable à la compréhension *bottom-up*.

Sous une autre perspective, la visualisation des éléments du code source qui ne sont « pas bien connus » du programmeur dans des contextes qui lui sont connus, est en fait une façon de supporter une approche *top-down* de compréhension. De plus, comme notre approche définit les plus hauts niveaux d'abstraction en termes de collaborations ou de relations entre des éléments du code source, c'est aussi possible d'utiliser une approche *top-down* en faisant directement l'étude de ces éléments de bas niveau.

La mise en contexte peut être utilisée aussi bien lors d'une approche systématique que lors d'une approche « au besoin ». En effet, notre approche peut être utilisée lorsqu'un programmeur tente de comprendre systématiquement tous les éléments et relations d'un module ou d'un sous-système, ou lorsque celui-ci tente de comprendre seulement certaines parties, de façon moins structurée. En fait, notre approche de compréhension peut être intégrée, à différents degrés, dans tous les modèles présentés au Chapitre 2 puisque ceux-ci font intervenir des aspects liés à la compréhension qui sont de plus haut niveau et plus généraux.

On peut donc voir notre approche comme une technique d'étude des grands systèmes plutôt que comme un modèle de compréhension. C'est ce qui rend cette approche utilisable dans différents contextes de compréhension, peu importe le type d'influence qui peut affecter la stratégie de compréhension adoptée. Entre autres, les caractéristiques décrites par Storey et discutées dans le Chapitre 2 sont de nature très générale, ce qui permet à notre approche d'être utilisée pour réduire leurs effets négatifs sur la compréhension.

6.2 Environnement SPOOL

Les éléments cognitifs décrits par Storey [36] et discutés dans le Chapitre 2 ont pour but de supporter la création d'un modèle mental par le programmeur et de faciliter la compréhension de logiciels. Certains de ces éléments permettent

d'améliorer la compréhension *bottom-up* et *top-down*. Selon Storey, les outils doivent, à cette fin, fournir des mécanismes d'abstraction, indiquer les relations syntaxiques et sémantiques entre les éléments d'un système, supporter la compréhension basée sur les hypothèses ainsi que fournir des aperçus du système à différents niveaux d'abstraction. Il faut aussi qu'ils aident dans la construction de plusieurs modèles mentaux par le programmeur (tel que décrit dans le modèle de Von Mayrhauser et Vans [42][43]).

Ces différents éléments cognitifs sont tous supportés à différentes échelles par notre approche et par l'environnement SPOOL. En effet, comme décrit précédemment, notre approche supporte les stratégies *top-down* et *bottom-up* tandis que l'environnement, par l'entremise des différents outils, permet la création de plusieurs abstractions d'un système, permet la visualisation des relations entre les éléments, etc.

Les autres éléments cognitifs présentés par Storey sont des éléments ayant pour but de réduire le travail cognitif que doit faire un programmeur dans son processus de compréhension. Par exemple, un outil de compréhension doit faciliter la navigation en fournissant des mécanismes d'exploration dirigés ou arbitraires, ce qui est bien supporté par le *Design Browser*. Selon Storey, un outil doit aussi réduire la désorientation et donner des indices au programmeur pour que celui-ci s'oriente dans un système. Il faut donc que l'outil donne de l'information sur le lieu de la navigation ainsi que le chemin qui y a conduit. Ceci est supporté dans SPOOL principalement par le *Context Viewer* qui donne beaucoup d'information permettant de s'orienter, ainsi que par le *Design Browser* avec son mécanisme d'historique, qui permet de garder le chemin parcouru et de récupérer les états précédents. Les derniers éléments cognitifs présentés par Storey sont la réduction de l'effort du programmeur à l'ajustement de l'interface usager ainsi que la présentation des différentes informations visualisées dans un style efficace. Les outils développés dans SPOOL présentent les éléments de façon claire et concise dans les différents diagrammes et outils, entre autres grâce à l'abstraction qui est faite des éléments du code source. Par contre, comme dans

toute interface usager, il y a certains aspects qui pourraient être améliorés. En particulier, la multiplication des fenêtres ralentit souvent l'étude d'un système puisqu'il faut souvent chercher parmi plusieurs d'entre elles avant de retrouver la bonne (ce qui est aussi le cas lors de l'utilisation d'outils commerciaux comme SNiFF+ par exemple). L'environnement SPOOL fournit un mécanisme de fenêtres internes qui règle partiellement le problème mais qui pourrait encore être amélioré.

L'environnement SPOOL supporte donc plutôt bien les différents aspects cognitifs reliés à la compréhension des logiciels. Pour être en mesure de mieux évaluer notre environnement, une étude des outils commerciaux et académiques décrits au Chapitre 2 à été effectuée, soit à partir de copies d'évaluation, soit à partir de la documentation disponible pour chacun d'eux. Dans ce qui suit, les résultats de cette enquête sont présentés.

La nature de l'évaluation des outils qui a été faite est basée sur trois aspects principaux. Le premier porte sur la possibilité de naviguer dans un système (particularité nécessaire selon Sim et al. pour la compréhension d'un espace d'information [30]) à l'aide de requêtes variées. Le deuxième aspect porte sur les mécanismes de recherche, c'est-à-dire la possibilité de rechercher des éléments du code source dans un système, mais aussi la possibilité de faire ces recherches de façon structurelle (par exemple chercher un attribut à un endroit particulier comme dans un fichier par exemple). Le dernier aspect concerne plus particulièrement notre approche et porte sur la possibilité de faire un pont entre différentes abstractions d'un système [27]. Autrement dit, ce dernier aspect concerne la création d'abstractions par l'outil et la navigation entre eux. Les Tableau 5 et Tableau 6 présentent les résultats de cette évaluation par rapport à des outils commerciaux et académiques respectivement, en donnant une pondération (forte (+), médium (~) et faible (-)) à chaque critère et pour chaque outil.

		Discover	Visual Age for Java	SNiFF+	Source-Navigator
Navigation (browsing)	Requêtes prédéfinies	+	~	~	~
	Requêtes personnalisées	-	-	-	-
Recherche (searching)	Recherche textuelle	+	+	+	+
	Recherche structurelle	+	~	~	~
Pont (bridging)	Création d'abstractions	~	~	~	~
	Navigation en contexte	-	-	-	-

Tableau 5 : Évaluation d'outils commerciaux

En résumé, des requêtes prédéfinies sont disponibles plus ou moins dans tous les outils commerciaux évalués (souvent accessibles via des menus) mais sont en général peu nombreuses. Par contre, aucun de ces outils ne permet la création de nouvelles requêtes personnalisées, ce qui limite le genre d'exploration qui peut être fait à l'aide de ces outils. Les aspects liés à la recherche d'éléments dans un système sont en général bien supportés par les outils commerciaux, mais l'aspect structurel pourrait être amélioré pour certains d'entre eux. De plus, tous les outils permettent de visualiser de l'information à de plus haut niveaux d'abstraction que le code source (par exemple, tous ces outils permettent au moins l'affichage d'un diagramme de classes), mais aucun ne fournit vraiment d'information au niveau de la conception comme le fait SPOOL. Finalement, aucun outil ne fournit vraiment de mécanisme pouvant permettre la navigation entre les niveaux ni aucun mécanisme de créations de contextes, ce qui fait qu'aucun de ces outils ne supporte directement notre approche de compréhension par mise en contexte.

		SHriMP	The Portable Bookshelf	Rigi	TkSee	SPOOL
Navigation (browsing)	Requêtes prédéfinies	~	~	~	+	+
	Requêtes personnalisées	-	+	-	-	+
Recherche (searching)	Recherche textuelle	-	+	-	+	+
	Recherche structurelle	-	~	-	-	+
Pont (bridging)	Création d'abstractions	~	~	~	-	+
	Navigation en contexte	-	-	-	-	+

Tableau 6 : Évaluation d'outils académiques

Du coté des outils académiques, on voit que l'aspect exploration par requêtes est plutôt bien supporté mais qu'encore une fois les requêtes sont peu nombreuses et qu'il n'est pas possible de créer des requêtes personnalisées dans ces outils. En fait, l'outil *The Portable Bookshelf* est une exception car il fournit un langage permettant de créer et de personnaliser ses propres requêtes [30]. Par contre, le résultat de ces requêtes est purement textuel, ce qui réduit la flexibilité de l'outil pour la recherche et la navigation car il est difficile d'exécuter de nouvelles requêtes à partir des informations trouvées. La recherche textuelle est aussi en général moins bien supportés dans ces outils que dans les outils commerciaux, puisque deux des outils sur quatre n'ont aucun mécanisme intégré le permettant. De plus, aucun d'entre eux ne fournit de mécanisme de recherche structurelle (à l'exception de *The Portable Bookshelf* par l'entremise de son langage de requête). Finalement, à l'exception de TkSee, tous les outils académiques supportent aussi la création de certaines d'abstractions, mais encore une fois, aucun d'entre eux ne fournit d'abstractions au niveau de la conception, ni ne permet de naviguer à travers ces abstractions ou de naviguer par mise en contexte.

Les résultats de cette évaluation démontrent donc que les différents aspects de la compréhension des logiciels qui sont supportés par l'environnement SPOOL sont en général mal supportés ou absents des outils commerciaux et académiques présentement disponibles. Évidemment, ces différents outils ont chacun leurs

points forts pour lesquels ils ont été développés, et ce sont ces points forts qu'il faudrait identifier dans une recherche future, de façon à en introduire les points essentiels dans l'environnement SPOOL.

La performance des outils d'aide à la compréhension est aussi un point très important. En effet, comme le processus d'exploration du code source des logiciels se fait en plusieurs petites étapes, il est important que celles-ci soient assez rapides pour éviter la confusion et la désorientation de l'usager. Lorsque les systèmes à analyser sont très grands, la performance en devient d'autant plus importante. C'est pourquoi nous avons conduit une expérience visant à évaluer le temps d'attente de l'usager entre chaque petite étape, autrement dit, le temps nécessaire pour exécuter une requête dans SPOOL. Les résultats de cette expérimentation (décrite dans [27]) sont que le temps d'exécution pour une requête de base dans le *Design Browser* sur des systèmes relativement grands (plusieurs centaines de milliers de lignes de code) est en moyenne dans les 5 secondes, ce qui est plutôt bien. Des requêtes plus complexes, comme la détection de patrons de conception, peuvent par contre prendre plusieurs minutes à exécuter, tout dépendant de la grosseur du système et de sa complexité (plus il y a de dépendances, d'appels d'opérations, de références et de pointeurs, plus les chemins à parcourir sont nombreux et plus la requête peut prendre du temps). Par exemple, la détection du patron de conception *Template Method* a pris jusqu'à 6 minutes sur le Système A[4], un système qui nous est fourni par Bell Canada et qui est d'une taille d'environ un demi-million de lignes de code. Ce temps d'exécution est tout de même raisonnable si on pense que pour trouver les instances de ce patron, la requête doit récupérer toutes les classes du système ainsi que toutes les méthodes de chacune de ces classes. Elle doit ensuite récupérer les appels qui sont faits dans chaque méthode et trouver les opérations qui reçoivent l'appel. Pour chacune de ces opérations, la requête doit vérifier si celle-ci est définie dans la même classe et si elle est implémentée dans au moins une sous-classe.

[4] Le vrai nom du Système A ne peut être dévoilé pour des raisons de confidentialité.

6.3 Avantages et limites

Les avantages de l'approche ainsi que de l'environnement SPOOL ont été illustrés par les quatre études de cas du Chapitre 5. En résumé, on peut dire que l'utilisation d'un environnement où les différents outils de navigation et de recherche sont intégrés et peuvent communiquer entre eux permet d'attaquer le problème de la compréhension logicielle de façon plus efficace. L'utilisation d'un outil flexible et générique (le *Design Browser*) pour naviguer et faire des recherches dans un système rend la tâche de compréhension d'un programmeur plus simple et permet d'accélérer le processus. La mise en contexte (via le *Context Viewer*) permet de rapidement situer les éléments d'un système les uns par rapport aux autres et permet aussi de rapidement comprendre les comportements associés à ces éléments ainsi que leurs collaborations.

Ces outils ont été présentés lors d'une démonstration à la conférence CASCON 1999, organisée par IBM, et ont été grandement appréciés par les visiteurs et en particulier par la communauté CSER. Bruno Laguë, Directeur du groupe d'assurance de la qualité chez Bell Canada et intervenant dans le projet SPOOL, a même déclaré que ces outils feraient l'envie des fournisseurs de *Discover*, environnement dont les licences sont vendues à plus de 160 000$ US au printemps 2000 (rappelons ci-contre, que Discover comprend tout un environnement qui va beaucoup au-delà de la navigation et la recherche).

Malgré son succès, il y a plusieurs limites reliées à l'utilisation de l'environnement SPOOL. Une de ces limites est liée au fait que l'environnement SPOOL est un environnement d'investigation et d'évaluation plutôt qu'un environnement de développement. En effet, l'utilisation de SPOOL sur un système comme ET++ nécessite plusieurs étapes de préparation avant de pouvoir l'utiliser comme utilisateur. Entre autres, le code source doit être analysé puis l'information extraite doit être importée dans le dépôt de SPOOL, ce qui peut prendre plusieurs heures, dépendant de la grosseur du système à analyser. À ce jour, l'environnement SPOOL ne peut donc pas être utilisé comme environnement de développement, ce

qui signifie qu'un programmeur voulant se servir de SPOOL devrait l'utiliser conjointement à un autre environnement (comme SNiFF+ par exemple).

Une autre limite reliée à l'utilisation de l'environnement vient du fait que celui-ci ne fournit actuellement aucun mécanisme qui permettrait de conserver l'information et les connaissances acquises dans le dépôt. En effet, la sauvegarde des différents contextes visualisés pourrait servir à la fois de rappel pour le programmeur qui les a créés et d'une forme de documentation pour les nouveaux arrivants.

Malgré ces limites, les résultats de l'évaluation précédente montrent que l'environnement SPOOL, utilisé à des fins de navigation, de recherche et de mise en contexte, peut faciliter de façon significative le processus de compréhension des logiciels.

Chapitre 7 : Conclusion

La compréhension du code source de grands systèmes logiciels s'effectue normalement en plusieurs étapes. Le programmeur qui est confronté à un système dont il ne connaît pas tous les détails, doit s'appuyer sur l'aide des programmeurs experts qui l'entourent et sur la documentation disponible, en plus de faire ses propres études dans le code source du système. Malheureusement, l'expérience montre que ces deux sources d'information ne sont pas toujours disponibles et que souvent la documentation n'est pas à jour. Les programmeurs ont donc besoin de techniques et d'outils leurs permettant d'aller chercher le plus d'information possible à partir de la seule source d'information fiable qui est disponible, c'est-à-dire le code source. C'est donc dans cette optique qu'a été effectué le travail qui est décrit dans ce mémoire, en présentant un support informatique à la compréhension des logiciels orientés objet de taille industielle.

7.1 Synthèse

L'approche de compréhension par mise en contexte présentée dans ce mémoire vise à minimiser les efforts qui doivent être faits par un programmeur afin de comprendre un système logiciel. La navigation entre les différentes abstractions qui peuvent être extraites du code source d'un système est essentielle si on veut maximiser la compréhension du système en un temps raisonnable. L'approche présentée dans ce mémoire fait appel principalement à trois niveaux d'abstraction qu'on retrouve dans un système logiciel et qui peuvent être extraits de façon semi-automatique: les éléments du code source, les structures physique et logique ainsi que les éléments liés à des aspects de la conception.

L'environnement SPOOL a été développé dans le but de permettre des recherches en termes de maintenance, d'évaluation et de compréhension de logiciels. Les deux outils, le *Design Browser* et le *Context Viewer*, ont été développés dans le cadre de ce mémoire comme partie intégrante de cet environnement, permettant ainsi de supporter l'approche de compréhension par mise en contexte. Ces deux outils contribuent à eux seuls à solutionner beaucoup des problèmes reliés à la navigation et à la recherche dans les systèmes logiciels. Entre autres, comme le *Design Browser* permet de faire de la recherche structurelle et qu'il permet de combiner ses capacités de recherche à une navigation arbitraire dans un système, le programmeur est supporté dans beaucoup de ses activités de compréhension. Le *Context Viewer*, de son coté, aborde le problème de l'excès d'information qui est normalement extrait des grands systèmes. Grâce à ses multiples contextes, cet outil permet de filtrer l'information et de visualiser uniquement ce qui est nécessaire au programmeur. Finalement, l'intégration de ces deux outils au reste de l'environnement ainsi que le mécanisme d'interaction qui a été implémenté pour qu'ils communiquent entre eux, rend possible la navigation entre les différentes abstractions d'un système et permet de faire le pont entre les niveaux de compréhension du programmeur.

Les quatre études de cas présentés au Chapitre 5 ainsi que l'évaluation de l'approche et de l'environnement décrite au Chapitre 6 ont montré comment notre approche et l'utilisation de l'environnement SPOOL pouvaient faciliter la compréhension d'un système logiciel. De plus, l'intérêt démontré par l'équipe d'analyse de la qualité de Bell Canada envers les outils développés dans le cadre de ce projet sont une confirmation supplémentaire de l'utilité de ces outils dans un milieu industriel de développement et d'analyse.

7.2 Travaux futurs

Pour corriger les aspects liés aux limites de l'approche et de l'environnement décrites à la Section 6.3, plusieurs travaux futurs peuvent être entrepris. En effet, un chemin à explorer serait celui de faire de l'environnement SPOOL un

environnement de développement. Ceci signifie que l'environnement devra fournir un éditeur de code source et que les changements dans le code devront être reflétés dans le dépôt de SPOOL pour que l'information qui s'y trouve soit mise à jour. Pour ce faire, il faudra que les fichiers source soient analysés à chaque enregistrement, et donc il faudra qu'un analyseur syntaxique soit intégré à SPOOL.

D'un autre point de vue, certains mécanismes ont déjà été intégrés dans l'environnement SPOOL pour permettre le stockage d'informations permettant l'analyse des versions d'un logiciel. De cette façon, les outils développés dans le cadre de ce mémoire pourront être réutilisés à des fins de navigation et recherche à travers l'histoire des logiciels.

Pour permettre un meilleur support à notre approche de navigation en contexte, nous prévoyons aussi ajouter un mécanisme d'historique ainsi qu'un mécanisme permettant de sauvegarder les contextes pour des fins de documentation et de réutilisation. De plus, plusieurs contextes supplémentaires pourront être ajoutés au *Context Viewer* dans le futur. Par exemple, des contextes extraits à partir d'information dynamique d'un système ou encore des contextes reliés à des « tranches » de code source (*slicing*). On peut même envisager la création « manuelle » de contextes à différents niveaux comme le niveau architectural par exemple.

De plus, tous les outils développés et faisant partie de l'environnement SPOOL pourraient aussi profiter de l'ajout de certains « raccourcis » clavier ou certains mécanismes comme « copier-coller » qui sont normalement utilisés dans les applications avec interface graphique. Ceci permettrait de réduire encore plus l'effort cognitif qui doit être fait par les utilisateurs lors d'une étude.

L'ajout de nouvelles requêtes dans le *Design Browser* est probablement l'aspect le moins évident pour les utilisateurs éventuels. Pour résoudre ce problème, deux directions sont possibles. La première consiste à définir un langage de requêtes qui pourrait permettre aux utilisateurs ne connaissant ni UML ni Java de tout de

même écrire des requêtes personnalisées. La deuxième direction consisterait à fournir une interface où l'utilisateur pourrait composer graphiquement ses requêtes en faisant du DnD d'éléments dans un diagramme, et en ajoutant des relations et des contraintes à ces éléments. Dans un cas ou dans l'autre, l'utilisation et la personnalisation de l'environnement SPOOL en serait grandement facilitées.

Dans un cadre plus général, il y a aussi deux axes d'amélioration de l'environnement SPOOL qui peuvent être envisagés. Le premier est lié au fait que présentement, la quantité et le type d'informations sur les systèmes à analyser qui peuvent être stockés dans le dépôt de SPOOL sont relativement réduits. On pourrait probablement ouvrir de nouvelles possibilités de recherche en ajoutant plus d'information dans le dépôt et ainsi permettre la création de beaucoup d'autres contextes qui pourraient être utilisés à des fins de compréhension et d'analyse. De cette façon, on pourrait même atteindre les limites du faisable en termes de quantité de données pouvant être supportée aujourd'hui par les technologies de gestion de bases de données. L'autre axe d'amélioration consisterait à continuer la tentative d'abstraction qui a été amorcée par l'utilisation du *Design Browser* et la détection de patrons de conception. En permettant de naviguer à travers des concepts de plus en plus haut niveau, on pourrait même envisager la possibilité de construire un outil spécialisé pour naviguer à travers ces concepts et donc créer une sorte de *Architecture Browser* qui serait la suite logique du *Design Browser* en termes d'outil d'abstraction.

Finalement, comme application concrète de ce qui a été accompli dans ce mémoire, le *Design Browser* ainsi que le *Context Viewer* ont déjà commencé à servir pour d'autres axes de recherche. Par exemple, des travaux dans le projet SPOOL visant à calculer et visualiser l'impact de changements qui sont apportés à un système utilisent déjà le *Design Browser* comme outil d'inspection. D'autres champs de recherche, comme la détection et la visualisation de chaînes de récursion ont dernièrement débutés et font aussi usage de cet outil.

Bibliographie

[1] Barr, J., Product Review "Reengineer/SET", Software Development Magazine, August 1996. On-line at <http://www.sdmagazine.com/breakrm/products/reviews/s968r2.shtml>.

[2] Barr, J., Product Review "DISCOVER 3.0", Software Development Magazine, March 1996. On-line at *<http://www.sdmagazine.com/breakrm/products/reviews/s963r2.shtml>*

[3] Bell Canada. DATRIX abstract semantic graph - reference manual. Montreal, Quebec, Canada. January 1999. Available on request from *<datrix@qc.bell.ca>*.

[4] Brooks, R., Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies 18*, pages 543-554. 1987.

[5] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M., Pattern-Oriented Software Architecture – A System of Patterns. *John Wiley and Sons*. 1996.

[6] Chaumun, A., Kabaili, H., Keller, R. K., and Lustman, F., A Change Impact Model for Changeability Assessment in Object-Oriented Systems. In *Proceedings of the Third Euromicro Working Conference on Software Maintenance and Reengineering*, pages 130-138, Amsterdam, The Netherlands. March 1999.

[7] Coplien, J. O., C++ Idioms Patterns, In *Pattern Languages of Program Design 4*, pages 167-197, Addison Wesley, Reading, MA. 2000. On-line at <http://www.bell-labs.com/~cope/Patterns/C++Idioms/EuroPLoP98.html>.

[8] Devanbu, P. T. GENOA – a customizable, language and front-end independent code analyzer. In *Proceedings of the 14th International Conference on Software Engineering (ICSE'92)*, pages 307-317. Melbourne, Australia. 1992.

[9] Discover online documentation, Software Emancipation Technology. On-line at *<http://www.setech.com/>*.

[10] Dufresne, P., Robitaille, S., and Keller, R. K. SPOOL Design Browser documentation. *Technical Report GELO-127*, Université de Montréal, April 2000.

[11] Finnigan, P., Holt, R., Kalas, I., Kerr, S., Kontogiannis K., Müller, H., Mylopoulos, S., Perelgut, S., Stanley, M., and Wong, K., The Software Bookshelf. *IBM Systems Journal*, Vol. 36, No. 4, pages 564-593. November 1997. On-line at *<http://www-turing.cs.toronto.edu/pbs/papers/bsbuild.html>*.

[12] Gamma, E., Helm, R., Johnson, R. and Vlissides, J., Design Patterns: Elements of Reusable Object-Oriented Software. *Addison-Wesley*. Menlo Park, CA. 1995.

[13] Gamma, E. and Weinand, A., ET++: A Portable C++ Class Library for a UNIX Environment. Tutorial notes. *OOPSLA '90*. Ottawa, ON, Canada. October 1990.

[14] Holt, R., Software Bookshelf: Overview and construction. March 1997. On-line at *<http://www-turing.cs.toronto.edu/pbs/papers/bsbuild.html>*.

[15] Java documentation, Sun Microsystems Inc. On-line at *<http://java.sun.com/>*.

[16] jKit/GO. Online documentation. Instantiations, Tualatin, OR. On-line at *<http://www.instantiations.com/>*.

[17] Keller, R. K., Knapen, G., Laguë, B., Robitaille, S., Saint-Denis, G., and Schauer, R., The SPOOL design repository: Architecture, schema, and mechanisms. In *Hakan Erdogmus and Oryal Tanir,* editors, *Advances in Software Engineering. Topics in Evolution, Comprehension, and Evaluation.* Springer-Verlag, 2000. 28 pages. To appear.

[18] Keller, R. K., and Schauer, R., Towards a Quantitative Assessment of Method Replacement. In *Proceedings of the Fourth Euromicro Working Conference on Software Maintenance and Reengineering*, pages 141-150, Zurich, Switzerland. February 2000.

[19] Keller, R. K., Schauer, R., Robitaille, S., and Pagé, P., Pattern-Based Reverse Engineering of Design Components. In *Proceedings of the 21th International Conference on Software Engineering (ICSE'99)*, pages 226-335, Los-Angeles, CA, USA. May 1999.

[20] Letovsky, S., Cognitive processes in program comprehension. *Empirical Studies of Programmers*, pages 58-79, Ablex, Norwood, NJ. 1986.

[21] Linux Cross-Reference, documentation set. On-line at *<http://lxr.linux.no/>*.

[22] Littman, D. C., Pinto, J., Letovsky, S. and Soloway, E., Mental models and software maintenance. *Empirical Studies of Programmers*, pages 80-98, Ablex, Norwood, NJ. 1986.

[23] Luger, G. F. and Stubblefield, W. A., Artificial Intelligence, Structures and Strategies for Complex Problem Solving. *Addison-Wesley Longman Inc.* Reading, MA. 1998.

[24] Meyer, B., Object-Oriented Software Construction. *Prentice-Hall.* 1988.

[25] Pennington, N. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology 19*, pages 295-341. 1987.

[26] POET Java ODMG Binding documentation. Poet Software Corporation. San Mateo, CA, USA. On-line at *<http://www.poet.com>*.

[27] Robitaille, S., Schauer, R., and Keller, R. K., Bridging Program Comprehension Tools by Design Navigation. In *Proceedings of the International Conference on Software Maintenance (ICSM'00)*, San Jose, CA, October 2000. IEEE. To appear.

[28] Schauer, R., Robitaille, S., Keller, R. K. and Martel, F., Hot Spot Recovery in Object-Oriented Software with Inheritance and Composition Template Methods. In *Proceedings of the International Conference on Software Maintenance (ICSM'99)*, pages 220-229. Oxford, England. August 1999.

[29] Schmidt, D., Design patterns for concurrent, parallel, and distributed systems. On-line at *<http://siesta.cs.wustl.edu/~schmidt/patterns-ace.html>*.

[30] Sim, S. E., Clarke, C. L. A., Holt, R. C. and Cox, A. M., Browsing and Searching Software Architectures. In *Proceedings of the International Conference on Software Maintenance (ICSM'99)*, pages 381-390. Oxford, England. August 1999.

[31] Singer, J., Lethbridge, T., Vinson, N. and Anquetil N., An Examination of Software Engineering Work Practices. In *Proceedings of CASCON'97*, pages 209-223. Toronto, ON, Canada. 1997.

[32] SNiFF+ documentation set. On-line at *<http://www.windriver.com>*.

[33] Soloway, E. and Ehrlich, K., Empirical studies of programming knowledge, *Transactions on Software Engineering*, Volume 10, Number 5, pages 595-609. 1984.

[34] Soloway, E., Pinto, J., Letovsky, S., Littman, D. and Lampert, R. Designing documentation to compensate for delocalized plans. Communications of the ACM, Volume 31 , Issue 11, pages 1259-1267. 1988.

[35] Source-Navigator documentation set. On-line at *<http://www.cygnus.com/sn/>*.

[36] Storey, M.-A. D., A Cognitive Framework For Describing and Evaluating Software Exploration Tools. PhD Thesis, Technical Report, School of Computing Science, Simon Fraser University. December 1998.

[37] Storey, M.-A. D., Fracchia, F. D. and Müller, H. A., Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Systems and Software*, 44(3):171-185. January 1999.

[38] Storey, M.-A. D. and Müller, H. A., Manipulating and documenting software structures using SHriMP views. In *Proceedings of the International Conference on Software Maintenance (ICSM'95)*, pages 275-284. Opio (Nice), France. October 1995.

[39] Tilley, S. R., Discovering DISCOVER. Technical report CMU/SEI-97-TR-012.
Pittsburgh, PA. October 1997. On-line at *<http://www.sei.cmu.edu/publications/documents/97.reports/97tr012/97tr012title.htm>*.

[40] UML, Documentation set version 1.1. September 1997. On-line at *<http://www.rational.com>*.

[41] Visual Age for Java online documentation, IBM Corporation. On-line at *<http://www.ibm.com>*.

[42] Von Mayrhauser, A. and Vans, A. M., Program comprehension during software maintenance and evolution, IEEE Computer, pages 44-55. 1995.

[43] Von Mayrhauser, A. and Vans, A. M., Comprehension processes during large scale maintenance. In *Proceedings of the International Conference on Software Engineering (ICSE'94)*, pages 39-48, Sorrento, Italy. May 1994.

[44] Wong, K. and Müller, H., Rigi User's Manual—Version 5.4.4, University of Victoria, Victoria, Canada, June 1998. On-line at *<ftp://ftp.rigi.csc.uvic.ca /pub/rigi/doc/rigi-5.4.4-manual.pdf>*.

[45] Yin, R. and Keller, R. K. The SPOOL query cookbook: How to write and add queries to the SPOOL environment. *Technical Report GELO-128*, Université de Montréal, April 2000.

[46] Zawinsky, J., resignation and postmortem. 1999. On-line at *<http://www.jwz.org/gruntle/nomo.html>*.

Annexe : La conception des outils de SPOOL

Cette annexe décrit les concepts importants derrière la conception du *Context Viewer* et du *Design Browser*. Comme ces deux outils ont été conçus à l'aide du cadre d'application de SPOOL, c'est celui-ci qui est présenté en grande partie dans cette annexe.

L'environnement SPOOL a été développé en utilisant le patron architectural MVC [5], c'est-à-dire que les fonctionnalités reliées à l'affichage sont découplées des données à afficher. La Figure 27 montre, sous la forme d'un diagramme UML, les classes importantes de la conception sous-jacente. La classe *Element* est la classe racine du cadre d'application, un peu comme *Object* est la classe racine dans ET++. Cette classe fournit les méthodes nécessaires aux éléments pour s'observer entre-eux, en fournissant une implémentation de base au patron *Observer* [12]. Les classes *ViewElement* et *ModelElement* représentent respectivement l'aspect *View* et l'aspect *Model* du patron MVC. De plus, le concept de *ViewElement* peut être implémenté à priori par n'importe quel composant graphique puisque la classe *ViewElement* est liée par l'agrégation *viewImp* à une classe de type *Object* (*Object* est la super-classe de toutes les classes en Java [15]). Par exemple, certains ViewElements de SPOOL sont implémentés par des composants de *Swing* qui sont disponibles dans les librairies standards de Java [15] (i.e. le concept de *ViewPanel* peut être implémenté par un *JPanel* de Java).

Dans l'implémentation du cadre d'application SPOOL, un *ModelElement* peut être associé à plusieurs *ViewElements* mais pas l'inverse. Conceptuellement, un *View* est la représentation visuelle d'un *Model,* ce qui fait que plusieurs

représentations peuvent exister pour les même données, mais plusieurs *Models* ne peuvent pas être représentés par un seul et unique *View*. L'aspect *Controller* du patron MVC est aussi implémenté dans SPOOL et est représenté par la classe ViewController, qui est liée par agrégation à la classe *ViewElement*. Notons que la classe *ModelElement* de la Figure 27 est en fait la même classe que celle qui est décrite à la Section 4.2. Cette classe représente deux concepts : le *Model* du patron MVC et le ModelElement tel que défini par UML[5].

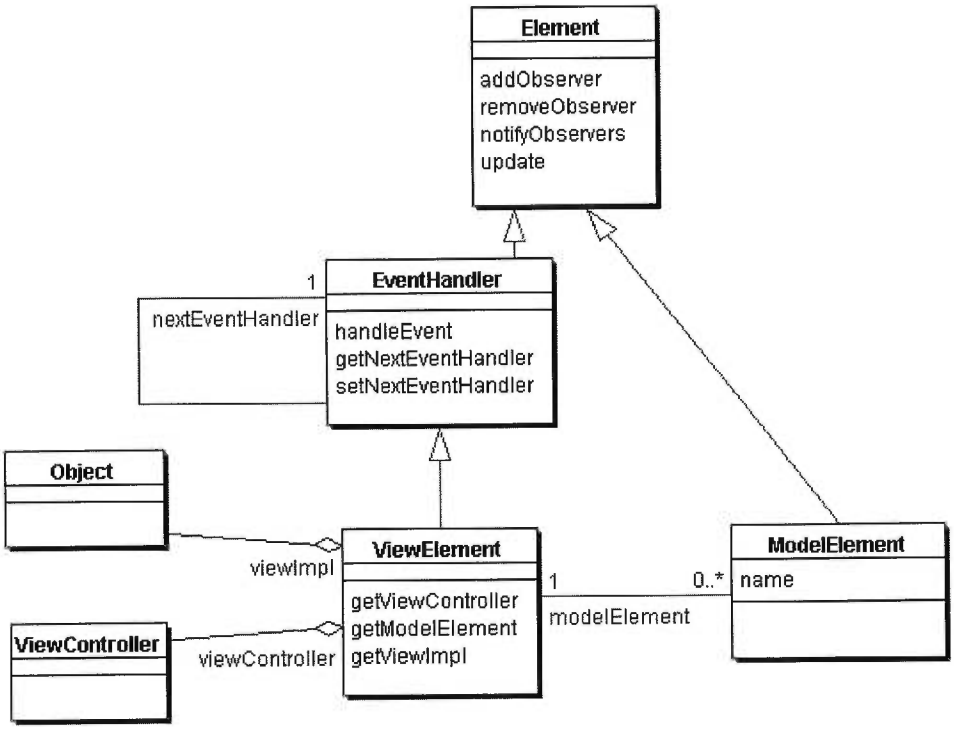

Figure 27 : *Model* vs *View* dans SPOOL

La classe *EventHandler* de la Figure 27 fournit les mécanismes nécessaires à la création de différents canaux de communication dans l'application, suivant le patron de conception *Chain of Responsability* décrit par Gamma et al. [12]. Ces mécanismes permettent entre autres aux *ViewElements* de SPOOL de pouvoir

[5] Ces deux concepts devront être séparés dans une version ultérieure de SPOOL pour permettre la distinction entre les éléments d'un système à analyser (autrement dit les éléments faisant partie du méta-modèle UML) et les autre types d'éléments qui peuvent être visualisés dans SPOOL.

transmettre les événements qu'ils ne savent pas comment traiter, au prochain *EventHandler* dans la chaîne, si celui-ci ne sait pas comment les traiter.

Il y a plusieurs spécialisations possibles du concept représenté par la classe *ViewElement*. Entre autres, dans SPOOL, il existe les concepts de *ToolBar*, de *MenuBar*, de *Glyph* et de *ViewPanel,* comme le montre la Figure 28. *ToolBar* et *MenuBar* représentent respectivement la barre de menu et la barre d'outils (généralement des boutons) qu'on retrouve dans les applications graphiques. Par exemple, les boutons *Back* et *Forward* du *Design Browser* décrits à la Section 4.4.2, sont définis dans l'implémentation d'un *ToolBar*, tandis que les menus de cet outil sont définis dans l'implémentation d'un *MenuBar*.

Un *ViewPanel* est un *ViewElement* qui peut en contenir d'autres (il s'agit en fait de l'implémentation du patron de conception *Composite* [12]). Par exemple, un *ViewPanel* peut contenir d'autres *ViewPanels* ou n'importe quelle autre spécialisation de la classe *ViewElement*. Un *Glyph* représente un objet graphique qui ne peut pas en contenir d'autres, contrairement à *ViewPanel*. La Figure 29 et la Figure 30 montrent les spécialisations de ces deux concepts.

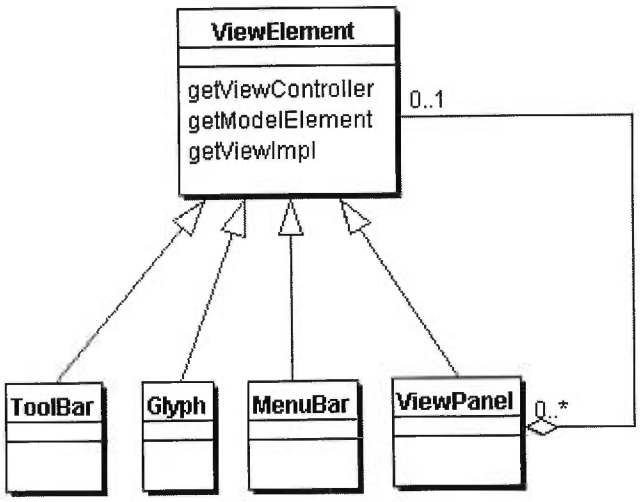

Figure 28 : La classe *ViewElement* de SPOOL

Il existe plusieurs sortes de *Glyphs* dans SPOOL. Un *GListNode* représente un nœud dans une liste d'éléments tandis qu'un *GTreeNode* représente un nœud dans un arbre d'éléments. Par exemple, les éléments et les requêtes apparaissant dans les listes du *Design Browser* sont des *GListNodes* tandis que les états apparaissant dans l'historique de cet outil sont des *GTreeNodes*. *GGO* est une classe abstraite représentant les *Glyphs* implémentés à l'aide du cadre d'application jKitGO [16], qui est utilisé dans SPOOL pour dessiner les éléments des diagrammes de classes, des diagrammes de collaborations, etc. Cette classe a trois spécialisations principales, GGOConnection, GGONode et GBoundingBox qui sont respectivement la représentation d'une connexion, d'un nœud et d'une boîte entourante dans un diagramme dessiné à l'aide de JKitGo. Par exemple, les différents éléments graphiques du diagramme de classes du *Context Viewer* sont des implémentations de ces concepts (les classes sont des *GGONodes* et les liens d'héritage sont des *GGOConnections*).

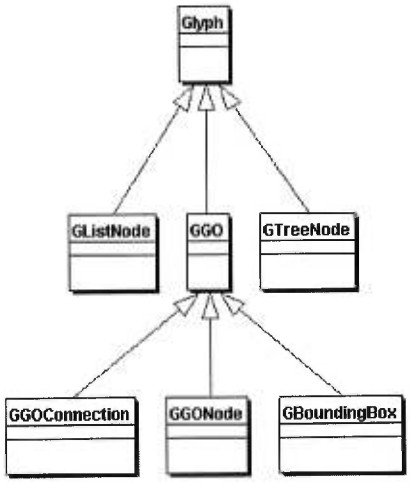

Figure 29 : La classe *Glyph* de SPOOL

ViewPanel est aussi un concept qui a plusieurs spécialisations, comme le montre la Figure 30. Un *ViewFrame* représente un fenêtre graphique et un *Dialog* représente une boîte de dialogue. La classe *BasicViewPanel* fournit certaines fonctionnalités de base à d'autres spécialisations du concept *ViewPanel*. Celles-ci sont montrées dans la Figure 31.

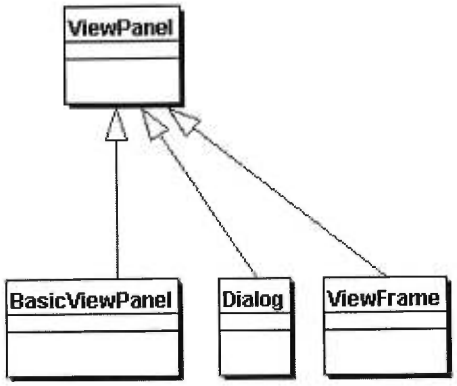

Figure 30 : La classe *ViewPanel* de SPOOL

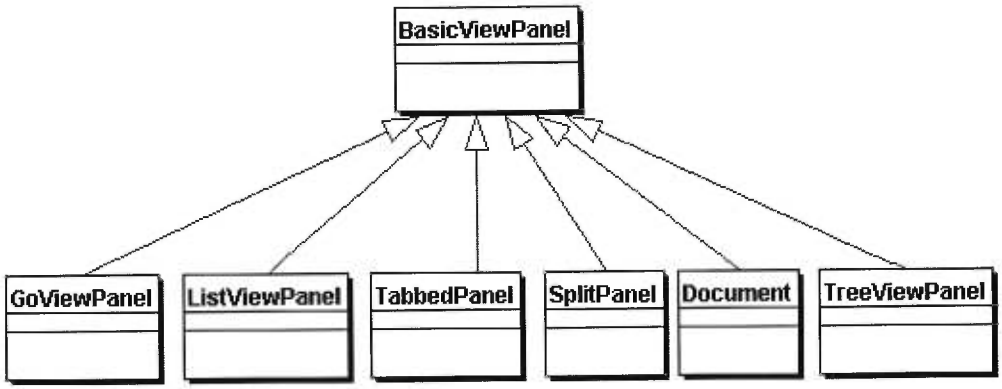

Figure 31 : La classe *BasicViewPanel* de SPOOL

La classe *GoViewPanel* représente un panneau qui peut contenir des *Glyphs* de type *GGO* tandis que les classes *ListViewPanel* et *TreeViewPanel* représentent des panneaux qui peuvent contenir respectivement des *Glyphs* de type *GListNode* et *GTreeNode* (voir Figure 29). *TabbedPanel* et *SplitPanel* représentent des panneaux contenant d'autres panneaux (des éléments de type *ViewPanel*). *SplitPanel* est utilisé entre autres dans le *Design Browser* pour séparer les listes d'éléments, tandis que *TabbedPanel* y est utilisé pour permettre l'affichage de l'historique (voir Figure 12). La classe *Document* représente un panneau qui est normalement contenu dans un *ViewFrame* et donc, qui peut contenir un *ToolBar*,

un *MenuBar*, ainsi que plusieurs autres *ViewPanels*, dépendant de la spécialisation du *Document*. Les spécialisations du concept *Document* sont aussi des panneaux et contiennent normalement plusieurs autres sortes de *ViewPanels*. Quelques-unes des spécialisations présentes dans SPOOL sont montrées dans la Figure 32.

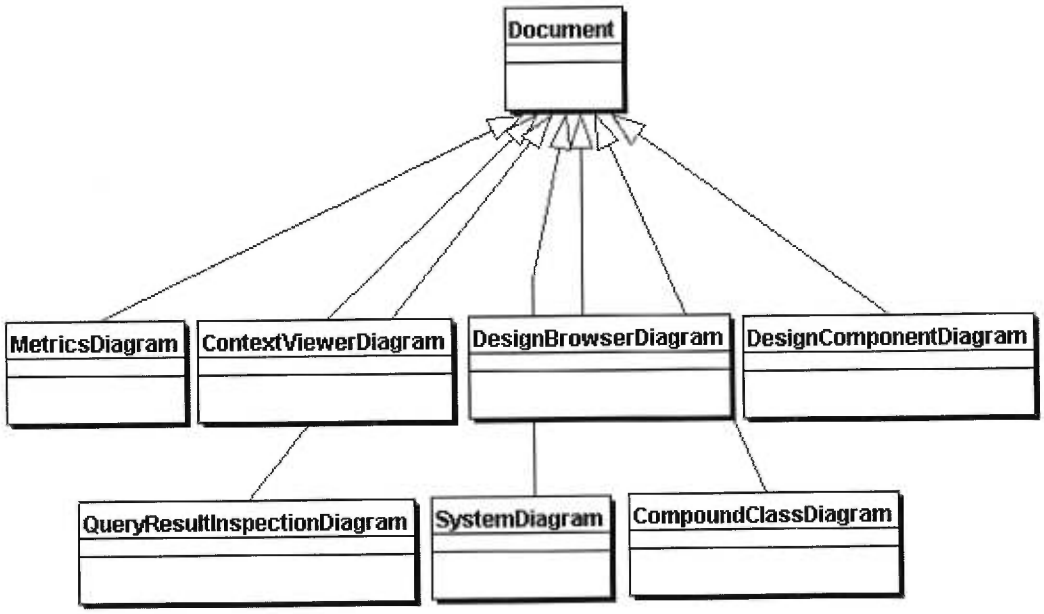

Figure 32 : La classe *Document* dans SPOOL

Chacune des classes montrées dans la Figure 32 représente un panneau contenu dans la fenêtre principale d'un outil de SPOOL. Parmi celles-ci, on trouve les classe *DesignBrowserDiagram* et *ContextViewerDiagram* qui sont respectivement les panneaux principaux du *Design Browser* et du *Context Viewer*. Ce sont ces spécialisations de la classe *Document* qui contiennent les différents *ViewPanels* qu'on peut identifier dans les fenêtres des outils de SPOOL. Pour ce qui est du *Design Browser* et du *Context Viewer*, il y a plusieurs spécialisations différentes du concept de *ViewPanel* qui ont été *implémentées* dans SPOOL. Celles-ci sont montrés dans la Figure 33.

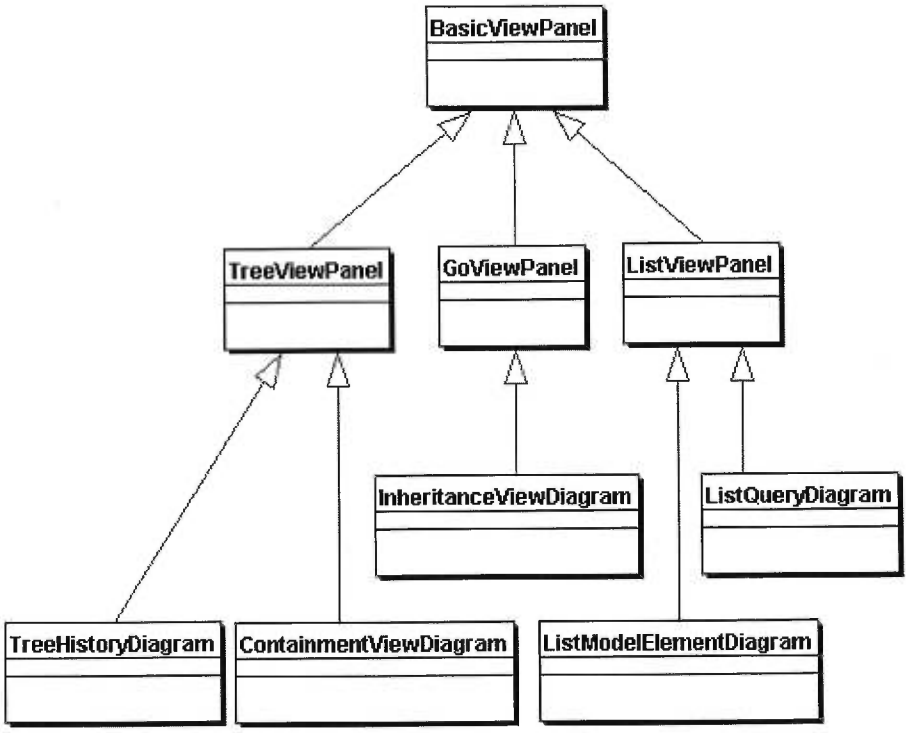

Figure 33 : Les *ViewPanels* du *Design Browser* et du *Context Viewer*

- *TreeHistoryDiagram* représente le panneau de l'historique du *Design Browser*
- *ListQueryDiagram* et *ListModelElementDiagram* représentent les deux types de listes du *Design Browser*, respectivement la liste de requêtes et les deux listes affichant des éléments
- *InheritanceViewDiagram* et *ContainmentViewDiagram* représentent les deux types de contextes présentement disponibles dans le *Context Viewer.*

Pour illustrer les collaborations entre ces classes dans les outils de SPOOL, la Figure 34 montre les objets impliqués lors de l'exécution d'une instance du *Design Browser.*

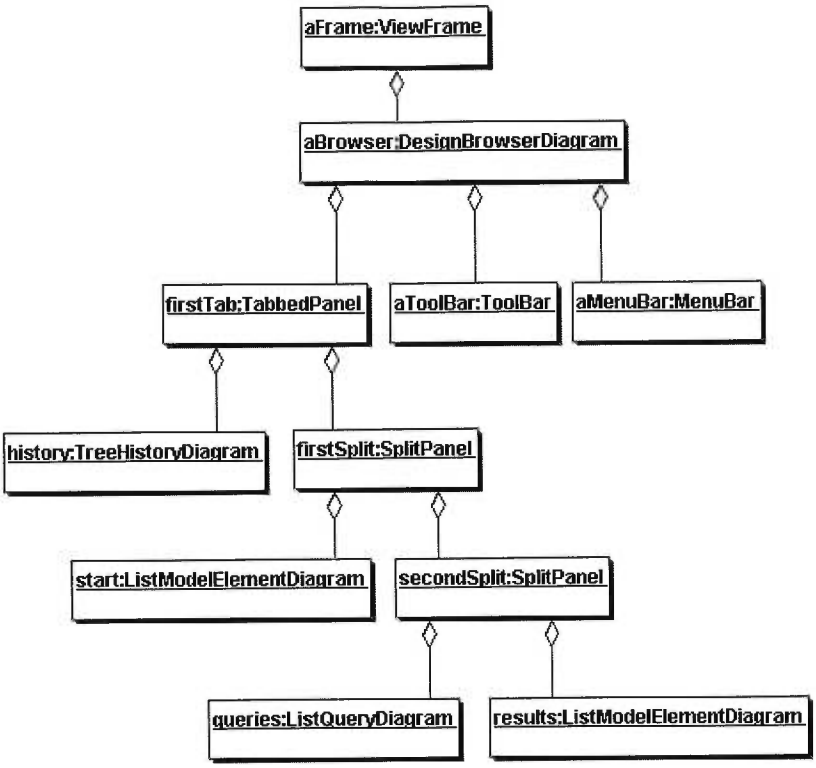

Figure 34 : Les objets d'une instance du *Design Browser*

Lorsqu'un *Design Browser* est lancé, les objets montrés à la Figure 34 sont créés ainsi que les liens d'agrégation qui les relient. À partir de cette figure, on voit qu'il y a un objet de type *ViewFrame* (la fenêtre de l'outil) qui contient un objet de type *DesignBrowserDiagram* (une spécialisation de la classe *Document*). Ce *Document* contient à son tour trois objets : un objet de type *ToolBar* (la barre d'outils), un objet de type *MenuBar* (la barre de menus) ainsi qu'un objet de type *TabbedPanel*. Le *TabbedPanel* contient l'historique (TreeHistoryDiagram) ainsi qu'un objet de type *SplitPanel*. Ce *SplitPanel* contient un objet de type *ListModelElementDiagram* (la liste de départ du *Design Browser*) ainsi qu'un second *SplitPanel* qui contient à son tour deux objets : un objet de type *ListQueryDiagram* (la liste de requêtes) ainsi qu'un objet de type *ListModelElementDiagram* (la liste de résultats).

Les objets qui sont dans le bas de l'arbre de composition de la Figure 34 contiennent à leur tour des éléments de type *Glyph* (comme décrit

précédemment). Ces objets ne sont pas montrés dans la figure puisque ceux-ci peuvent être très nombreux (tous les éléments d'un système par exemple) et qu'ils peuvent être ajoutés et enlevés dynamiquement de l'outil lors de l'exécution des outils.

Cette dernière figure illustre l'utilisation qui est faite des différentes classes du *Design Browser* décrites précédemment. Comme toutes ces classes sont des spécialisations de *ViewElement* et donc de *EventHandler*, celles-ci communiquent entre elles par des canaux d'événements prédéfinis. Pour simplifier, l'objet de type *DesignBrowserDiagram* est le « *nextEventHandler* » de tous les autres objets composant l'outil. La même stratégie est aussi appliquée au *Context Viewer*, ce qui permet de traiter les événements des outils de façon localisée. Le mécanisme d'observation de la classe racine *Element* peut aussi être utilisé pour avertir les différents *ViewElements* lorsque leur *ModelElement* associé est modifié. Finalement, le mécanisme de composition des panneaux permet de facilement modifier le contenu d'un outil. Par exemple, de nouveaux contextes peuvent être ajoutés de façon flexible au *Context Viewe*r.