UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

RECONCEPTION DE SYSTÈMES ORIENTÉS-OBJET BASÉE SUR L'ANALYSE DES CLONES

présenté par: BALAZINSKA Magdalena

en vue de l'obtention du diplôme de: Maître ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de:

M. GRANGER Louis, M.Sc., président
M. MERLO Ettore, Ph.D., membre et directeur de recherche
M. BOUDREAULT Yves, M.Sc.A., membre

UNIVERSITÉ DE MONTRÉAL

RECONCEPTION DE SYSTÈMES ORIENTÉS-OBJET BASÉE SUR L'ANALYSE DES CLONES

MAGDALENA BALAZINSKA
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE ET DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)
NOVEMBRE 1999

à Michel

REMERCIEMENTS

J'aimerais avant tout remercier le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG) ainsi que Bell Canada pour le financement du projet. J'aimerais aussi remercier mon directeur de recherche, Ettore Merlo, pour m'avoir appuyée et conseillée pendant ces deux années de maîtrise et sans qui le projet n'aurait pas été possible.

J'aimerais également remercier Michel Dagenais et Kostas Kontogiannis pour leur collaboration dans les articles publiés.

J'aimerais finalement remercier Janice Singer, Susan Sim, mes parents et mon fiancé pour m'avoir aidée à persévérer, même pendant les phases les plus difficiles du projet.

RÉSUMÉ

Dans les systèmes orientés-objet, la réutilisation d'une fonctionnalité n'est pas toujours effectuée par héritage ou par appel d'une composante configurable. Il arrive que les programmeurs copient directement le code source et le modifient pour satisfaire leur nouveau besoin. Une telle activité de réutilisation implicite produit des fragments de code très similaires, appelés des clones.

La présence de clones dans un système n'est pas nécessairement désirable puisque la synchronisation de leur maintenance peut être difficile. Les clones sont aussi des liens implicites entre des classes qui partagent une fonctionnalité.

Ce mémoire présente les résultats d'un projet de reconception de systèmes orientés-objet basée sur l'analyse des clones. Le principal objectif de la recherche était d'étudier la possibilité d'utiliser les informations de clonage pour des actions de réingénierie pouvant aider la maintenance du logiciel.

Les principales contributions de la recherche sont le développement d'une nouvelle taxinomie des clones servant à la mesure des opportunités de réingénierie, le développement d'un nouvel algorithme de comparaison de fragments de code permettant l'extraction de différences sémantiques ainsi que le développement et l'étude de plusieurs méthodes de reconception basée sur les clones.

ABSTRACT

In object-oriented systems, re-use isn't always performed through inheritance or use of configurable components. Programmers often proceed with a manual copy-paste-modify to satisfy their need for re-use. Such an activity produces very similar code fragments called clones.

The presence of clones in a system isn't necessarily desirable as the synchronization of their maintenance is difficult. Clones also represent implicit links between classes sharing a functionality.

This thesis presents the results of a project on object-oriented system redesign based on clone analysis. The main goal of the research was to study the use of clones for reengineering actions that could improve system maintainability.

The main contributions of this research are the development of a new clone classification scheme useful to assess reengineering opportunities, the development of a new matching algorithm that allows to extract semantic differences between clones and the development and study of several redesign approaches based on clones.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LISTE DES ANNEXES

INTRODUCTION

Dans les systèmes orientés-objet, la réutilisation du code source est possible grâce à différents mécanismes tels l'héritage, les librairies et la composition d'objets. Certaines approches, comme l'utilisation de patrons de conception [21], facilitent plus particulièrement la réutilisation.

Cependant, les programmeurs ont parfois besoin de se servir de composantes qui n'ont pas été conçues pour être réutilisées. Cela se produit principalement lorsque les systèmes sont dans leur phase d'expansion et que de nouvelles spécifications doivent être satisfaites périodiquement [21].

Lorsqu'une telle situation survient, le module et la composante visés devraient être restructurés pour que la composante puisse être correctement réutilisée. Encore mieux, le système complet pourrait être réorganisé à la lumière de l'ensemble des nouvelles spécifications qu'il satisfait ou qu'il doit satisfaire. Les classes pourraient être restructurées en des composantes générales et leurs interfaces rationalisées. Un tel processus de consolidation [21] permettrait au système de devenir plus flexible, plus facile à étendre en termes de fonctionnalités et surtout plus facile à maintenir.

Malheureusement, il arrive souvent que la technique de réutilisation employée soit plutôt un copier-coller du code source de la composante désirée. Cette approche produit des fragments de code très similaires que l'on appelle des clones, et qui vont subir des activités de maintenance et de modification indépendantes [30].

0.1 Existence de clones dans les systèmes informatiques

Les recherches antérieures [4,4,30] ont montré qu'environ 5 à 15% du code source des systèmes de grande taille est formé par des clones.

La présence d'une quantité si élevée de code dupliqué n'est pas désirable. Cela augmente inutilement la taille des systèmes, mais surtout cela rend difficile la synchronisation des actions de maintenance effectuées sur les différentes copies. Il est aussi vrai que le clonage rend implicite les liens entre des éléments qui partagent une même fonctionnalité.

0.2 Recherches antérieures sur les clones

Plusieurs chercheurs se sont déjà intéressés à la détection des clones et à leur utilisation dans différents contextes, dont la compréhension de programmes, la documentation, l'évaluation de la qualité de systèmes ainsi que la restructuration des systèmes et des processus.

Plusieurs techniques de détection de clones ont été étudiées. Certaines sont basées sur le texte formant les fichiers sources des logiciels. Johnson [26] a développé une méthode d'identification des copies exactes de fragments de chaînes de caractères en utilisant des empreintes (*fingerprints*). L'outil "Dup" [4] de Baker rapporte autant les copies exactes de sections de code que celles qui diffèrent dans la substitution systématique d'un ensemble de noms de variables et de constantes par un autre. Ducasse et al. [17] font appel à la fois à une correspondance entre chaînes de caractères et à la visualisation pour l'identification de copies exactes de codes sources.

D'autres approches, comme celles de Mayrand et al. [33] et de Kontogiannis et al. [28], permettent la détection de blocs d'instructions similaires en utilisant des métriques liées à la disposition des instructions, aux expressions formant les instructions, au flux de contrôle, aux variables utilisées, aux variables définies, etc.

Dans [28], Kontogiannis et al. font aussi la détection de clones en utilisant deux méthodes de mise en correspondance de patrons (*pattern matching*): le *Dynamic Programming Matching*, qui calcule l'alignement optimal entre deux fragments de code et un mécanisme de concordance statistique entre les patrons d'une description abstraite du code et le code source lui-même.

Une autre technique de détection de clones est celle de Baxter et al [5]. Elle est basée sur la comparaison des sous-arbres de l'AST (Arbre de syntaxe abstraite) du système. Une fonction de *hashing* est tout d'abord utilisée pour diviser les sous-arbres en groupes de sous-arbres similaires. Les sous-arbres de chaque groupe sont ensuite exhaustivement comparés un à un.

Plusieurs applications de la détection des clones ont été étudiées. Johnson [26] visualise les fragments de chaînes de caractères redondants pour faciliter la tâche de compréhension des systèmes de grande taille. Mayrand et al. [33], ainsi que Lagüe et al. [30], documentent le phénomène de clonage dans le cadre de l'évaluation de la qualité de systèmes. Lagüe et al. [30] ont aussi évalué les bénéfices de l'incorporation de la détection des clones aux processus de maintenance de logiciel.

0.3 Motivation de la recherche

La littérature montre donc que les recherches antérieures se sont penchées beaucoup plus sur la détection des clones dans les systèmes informatiques que sur l'utilisation subséquente de ces informations. Il y a eu avant tout des efforts de visualisation et de documentation. Cependant, de telles applications restent au niveau de la constatation du phénomène de clonage dans les systèmes. Lagüe et al. [30] ont amené les applications vers la gestion du phénomène. Ils contrôlent donc le clonage, mais toujours sans y remédier activement.

L'objectif de la recherche présentée dans ce mémoire est d'étudier l'utilisation des clones comme base d'actions de réingénierie utiles à la maintenance des systèmes. Nous voulons faciliter l'entretien des logiciels en éliminant les duplications de code source et en permettant une réutilisation plus appropriée des fonctionnalités clonées que la copie manuelle de leurs codes sources.

Le chapitre 1 de ce mémoire décrit plus en détails le problème abordé et les solutions étudiées. Le chapitre 2 présente une nouvelle taxinomie des clones servant de base à l'analyse des possibilités de reconception. Dans le chapitre 3, les détails d'un algorithme de comparaison de fonctions clones sont présentés. Cet algorithme permet de déterminer les différences sémantiques entre les clones. Le chapitre 4 décrit la reconception en soit tandis que le chapitre 5 présente les expériences de reconception conduites; le chapitre 6 discute des résultats obtenus. Finalement, le chapitre 7 compare les travaux effectués dans le cadre de ce projet, avec les recherches existant dans les domaines connexes.

CHAPITRE 1

CONTRIBUTION DE LA RECHERCHE

1.1 Définition du problème

Le problème abordé dans cette recherche est la reconception de systèmes orientés-objet basée sur l'analyse des clones. Nous étudions l'utilisation d'information sur la présence de clones dans un système pour reconcevoir ce dernier et le rendre ainsi plus facile à maintenir. Un système plus facile à maintenir est un système pour lequel l'introduction de nouvelles fonctionnalités représente un effort et un degré de difficulté faibles. C'est aussi un système facile à corriger sans risque d'effets secondaires inopinés.

Les objectifs précis du projet étaient les suivants:

- Rendre explicites les réutilisations implicites de codes sources par clonage.
- Éliminer les duplications par l'union des parties communes des clones.
- Conserver les caractéristiques uniques de chaque variante d'une méthode clonée.
- Faciliter la réutilisation et les modifications futures des fonctionnalités clonées.
- Ne pas modifier les interfaces des parties du système transformées.

1.2 Contributions originales de la recherche

Ce mémoire présente donc l'étude des reconceptions automatiques des systèmes qui unissent les parties communes des clones et paramétrisent leurs différences. Une partie des solutions analysées est basée sur la modification de la hiérarchie des classes. Les éléments communs des clones sont remontés vers la racine de la hiérarchie, tandis que les différences sont descendues vers les feuilles. L'autre partie des techniques étudiées transforme la réutilisation implicite par duplication de code source en une réutilisation explicite de composantes générales, configurables, réutilisables et indépendantes des clones originaux.

L'union des parties communes des clones a déjà été étudiée par Baxter et al. dans [5]. Leur approche, basée sur les macros, permet l'élimination de toutes les duplications et par le fait même la réduction de la quantité de codes sources dans le système. Néanmoins, l'utilisation de macros présente plusieurs limitations. Elle n'est applicable qu'à des langages qui supportent les macros; mais, plus important encore, chaque modification lexicale d'une macro nécessite une vérification manuelle afin d'assurer que la modification sémantique désirée est correctement répendue à tous les endroits où la macro est utilisée.

L'approche que nous proposons est basée sur les différences sémantiques entre les clones. Elle permet donc à la reconception de garder la signification initiale du code lors des transformations. Les modifications subséquentes n'ont donc pas besoin d'être vérifiées manuellement. Cette nouvelle approche est la principale contribution de la recherche.

Dans le cadre de la reconception proposée, plusieurs autres contributions originales ont été effectuées:

- Un nouvel algorithme de comparaison de fragments de code source à été développé. Cet algorithme est simple, présente une complexité relativement faible de $\theta(n^2)$ (qui peut même être réduite, comme il le sera discuté plus loin), permet une fine granularité de comparaison et est applicable à n'importe quel langage procédural ou orienté-objet.

- Une approche originale d'extraction de différences sémantiques entre fragments de code, basée sur l'algorithme mentionné ci-dessus, a aussi été développée. Cette approche permet de comparer des fragments de codes et de déterminer leurs différences en termes d'éléments du langage de programmation, tels les expressions, les instructions, les déclarations, etc.

- Une nouvelle taxinomie des clones a été élaborée. Cette taxinomie permet de mesurer les opportunités de réingénierie dans un système.

- Plusieurs types de reconception basée sur les clones ont été proposés. Certains font usage de la hiérarchie des classes tandis que d'autres transforment les clones en composantes réutilisables.

- Un processus de reconception automatique a également été développé.

1.3 Définitions

Les résultats de l'étude présentés dans les chapitres suivants reposent sur plusieurs termes et concepts-clés que nous définissons dans cette section.

1.3.1 Clone

Il existe plusieurs définitions d'un clone. Dans ce projet, nous définissons un clone comme étant un segment de code source identique ou similaire à un autre.

Le degré de similitude nécessaire pour que deux morceaux de code soient considérés des clones est arbitraire et dépend de l'application.

Bien que la ressemblance puisse être le fruit du hasard, surtout lorsque les clones sont de petite taille, elle est souvent due à une activité consciente de clonage c'est-à-dire de copie, puis de modification d'un morceau de code.

1.3.2 Différence sémantique entre clones

Nous considérons comme différence sémantique entre clones toute différence syntaxique exprimée en termes de la composante du langage de programmation à laquelle elle correspond.

Soit les deux méthodes suivantes:

```
public void myMethod(int a, int b, int c) {
    ...
    doSomething(a + b);
```

```
    ...
}
et
public void myMethod(int a, int b, int c) {
    ...
    doSomething(a - c);
    ...
}
```

La différence $\{< +b >, < -c >\}$ sera considérée comme une différence dans l'argument d'une méthode.

1.4 Cadre technique du projet

L'étude complète a été réalisée sur des systèmes Java. Les outils développés ont été implantés avec JDK 1.1.7. Le parseur utilisé a été généré à l'aide de Javacc version 0.8.

CHAPITRE 2

TAXINOMIE DES CLONES

Avant d'étudier les possibilités de réingénierie des systèmes basées sur des informations de clonage, nous avons examiné et classé une quantité importante de clones détectés dans plusieurs systèmes réels. Une telle classification nous a permis de mesurer les opportunités de réingénierie présentes dans les systèmes et de déterminer les types de clones dont la transformation serait profitable.

2.1 Taxinomies existantes

Mayrand et al. ont présenté une taxinomie des clones dans [33]. Ils ont défini une échelle ordinale de huit niveaux de clonage correspondant au degré de similitude entre des fonctions clones. Ce degré est déterminé par les différences des fonctions clones dans leurs noms, leurs dispositions physiques, les expressions qui les forment et leurs flux de contrôle. Cette taxinomie était développée pour des objectifs d'évaluation de la qualité des systèmes logiciels.

Pour des activités de réingénierie, d'autres informations sur les clones doivent être prises en considération. En effet, pour transformer les clones, une connaissance détaillée de leurs caractéristiques, notamment de leurs différences sémantiques, est nécessaire.

2.2 Taxinomie proposée

2.2.1 Données d'observation

La taxinomie présentée ici a été développée après un examen manuel de quelques 800 clones, extraits de six systèmes dont le code source est disponible gratuitement:

- JDK [42], un kit de développement de Sun Microsystems comptant quelques 145 000 lignes de code.
- SableCC [18], un générateur de parseurs de l'université McGill de 32 000 lignes de code.
- ANTLR [32], un générateur de parseurs développé par l'Institut MageLang et comptant 25 000 lignes de code.
- SWING [43], une boîte à outils pour le développement d'interfaces utilisateurs produite par Sun Microsystems et totalisant 215 000 lignes de code.
- KFC [48], une boîte à outils pour le développement d'interfaces utilisateurs d'une longueur de 57 000 lignes de code, produite par K. Yasumatsu.
- HTTPCLIENT [45] un fureteur de R. Tshalaer dont le code source est d'une longueur de 21 000 lignes de code.

Les clones utilisés pour l'élaboration de la taxinomie ont été extraits des systèmes par l'approche de Patenaude et al. [39]. Cette approche extrait et groupe des méthodes similaires en utilisant des métriques pour leur comparaison.

2.2.2 Catégories développées

Durant la phase d'observation, les différences existant entre les clones ont été listées. Il a été noté qu'une grande partie des clones étaient fortement semblables ou même identiques. Les deux catégories suivantes ont donc été définies:

- Identiques: des méthodes clones sont dites identiques lorsqu'il n'existe aucune différence entre elles, même pas dans leurs noms.

- Différences superficielles: les méthodes se distinguent seulement par des différences qui n'affectent ni leurs valeurs de sortie ni leurs comportements. Il s'agit de différences dans les noms des méthodes, dans les noms de leurs paramètres et dans ceux de leurs variables locales.

Pour les autres clones, trois catégories de différences sont clairement ressorties: les différences n'affectant qu'un jeton lexical à la fois, les différences affectant des séquences de jetons et les différences affectant les attributs des méthodes, comme leurs modificateurs (static, public, etc.) ou leurs listes d'exceptions. Le premier groupe de différences a été subdivisé en sept catégories, selon la signification des jetons appartenant aux différences:

- Appels de méthode: les différences entre les clones correspondent à des appels de méthodes. Autrement dit, lorsque deux clones diffèrent uniquement dans les appels qu'ils font à des méthodes, ils appartiennent à cette catégorie.

- Variables globales: les différences correspondent à des utilisations de variables non-locales ou de constantes définies à l'extérieur des méthodes.

- Type de la valeur de retour: la seule différence entre les méthodes clonées est le type de la valeur retournée.

- Types de paramètres: les différences affectent le type d'un ou de plusieurs paramètres des méthodes.

- Variables locales: les types des variables locales déclarées varient entre les clones.

- Constantes: les différences affectent les constantes numériques, booléennes ou autres directement encodées dans les méthodes.

- Utilisation de types: les différences correspondent à des types manipulés de manière explicite dans des expressions telles que *instanceof*, ou dans des changements de type (*typecast*).

Certaines méthodes clones contiennent plusieurs des types de différences couverts par les catégories précédentes. Pour ces clones, les catégories 10 à 12 ont été définies:

- Changements d'interface: les différences correspondent à des appels de méthodes, des variables globales, des types de paramètres et/ou le type de la valeur de retour.

- Changements d'implantation: les différences se situent dans les types de variables locales, les constantes utilisées et/ou les types explicitement manipulés.

- Changements d'interface et d'implantation: les différences affectant un jeton à la fois correspondent à n'importe lesquelles des différences décrites dans les catégories précédentes.

Aucun patron de correspondance entre les différences et les entités du langage de programmation similaire n'a été identifié pour les différences couvrant plus d'un jeton. Il a cependant été noté que la plupart des clones contenaient seulement une ou deux différences de ce type. Les catégories 13, 14 et 15 ont donc été définies comme:

- Une longue différence: une seule entité (une expression, une instruction ou autre) diffère d'un clone à l'autre.

- Deux longues différences: exactement deux entités sont affectées par les différences.

- Plusieurs longues différences: trois entités ou plus sont affectées par les différences.

Certaines entités étant plus importantes en taille que d'autres, un seuil a été fixé quant aux pourcentages des méthodes qui peuvent être affectés par des différences pour que les méthodes puissent encore être considérées comme des clones. En

Tableau 2.1: Taxinomie des méthodes clones

Numéro	Nom de la catégorie
1	Identiques
2	Différences superficielles
3	Appels de méthodes
4	Variables globales
5	Type de la valeur de retour
6	Types de paramètres
7	Variables locales
8	Constantes
9	Utilisation de types
10	Changements d'interfaces
11	Changements d'implantation
12	Changements d'interface et d'implantation
13	Une longue différence
14	Deux longues différences
15	Plusieurs longues différences
16	Une longue différence, interface et implantation
17	Deux longues différences, interface et implantation
18	Plusieurs longues différences, interface et implantation

effet, pour deux méthodes complètement différentes, nous pourrions dire qu'elles diffèrent d'une seule entité, soit le corps des méthodes, ce qui n'aurait pas beaucoup de signification pour la réingénierie. Le seuil à été arbitrairement fixé à 30%, mais il pourrait être raffiné par l'utilisateur.

Il a aussi été remarqué que certaines méthodes contenaient à la fois les deux types de différences, celles affectant un jeton à la fois et celles couvrant des séquences de jetons. Alors, les catégories 16 à 18 ont été définies comme les correspondants des catégories 13 à 15, mais incluant en plus les différences définies dans la catégorie 12 ("Changements d'interface et d'implantation").

Finalement, les différences affectant la liste des exceptions et les modificateurs des méthodes n'ont pas été utilisées dans la taxinomie parce qu'elles n'affectent pas directement les choix des actions de réingénierie. Elles ont été laissées comme paramètres pour les phases de réingénierie.

Le tableau 2.1 résume les catégories définies. Toutes les méthodes similaires qui ne sont pas couvertes pas la taxinomie ne sont présentement pas catégorisées et sont gardées pour des recherches futures.

Nous avons appliqué cette taxinomie à plusieurs systèmes de grande taille, ce qui nous a permis d'analyser les phénomènes de clonage du point de vue de la réingénierie. Les résultats de cette expérience sont présentés dans le chapitre 5.

CHAPITRE 3

COMPARAISON DE CLONES

Le fondement du processus de réingénierie basée sur les clones est la connaissance des différences sémantiques entre ces derniers. En effet, une telle connaissance permet d'isoler les différences, de les manipuler et de les déplacer, et de transformer ainsi le système en le gardant cohérent et compréhensible du point de vue d'un programmeur.

Ce chapitre présente un nouveau processus et un nouvel algorithme qui permettent l'extraction des différences sémantiques entre les clones.

3.1 Processus d'extraction des différences

Le processus d'extraction des différences sémantiques est présenté à la figure 3.1. Il consiste en quatre étapes principales:

- L'analyse lexicale. Cette étape permet d'obtenir les vecteurs de jetons formant les deux fragments de code à comparer.

- La mise en correspondance. Cette étape détermine quels jetons correspondent dans les deux fragments de code et lesquels doivent être enlevés, ajoutés ou remplacés pour transformer l'un des fragments de code en l'autre.

- Analyse syntaxique. Cette étape permet d'obtenir les arbres syntaxiques abstraits (AST) des deux fragments de code.

- La projection des différences sur les AST. Finalement, les différences obtenues sous forme de séquences de jetons sont projetées sur les AST afin de les faire correspondre à des entités significatives du langage de programmation.

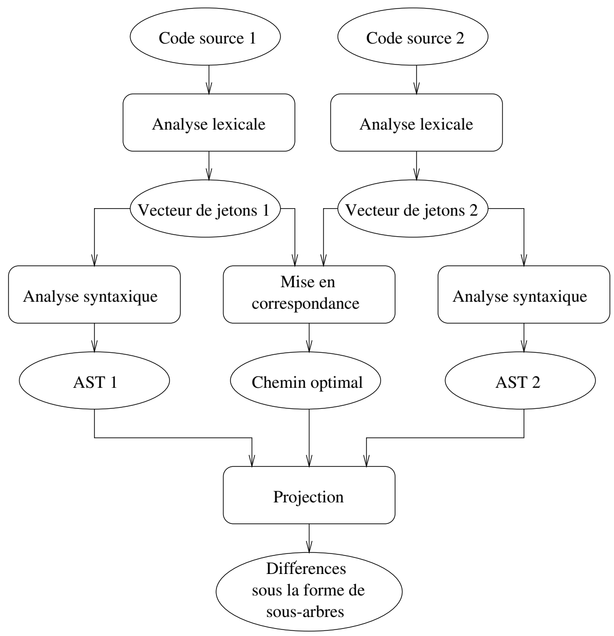


Figure 3.1: Processus d'extraction des différences sémantiques.

Ce processus est adéquat pour la comparaison de deux fragments de code. Il arrive souvent qu'un groupe de clones soit constitué de plus de deux fontions. Dans ce cas, pour comparer simultanément toutes les fonctions, une d'entre-elles est choisie comme référence, et toutes les autres sont comparées à elle. Une étape d'union des ensembles de différences est ensuite ajoutée.

Les sections suivantes expliquent les différentes étapes du processus, les justifient et exposent les algorithmes sur lesquels elles sont basées.

3.2 Abstraction du code source

Pour pouvoir analyser la signification des énoncés formant un programme, il est nécessaire de représenter ce dernier à un plus haut niveau d'abstraction que la suite de caractères qui forment les fichiers sources.

Plusieurs schémas de représentation ont été proposés dans la littérature. Parmi ceux-ci se retrouvent les cadres (*frames*) [36], les graphes annotés de flux de contrôle et de flux de données [46], les arbres syntaxiques abstraits [35], les formules logiques des dépendances de programmes [10] et les tuplets relationnels basés sur un modèle du domaine du langage [34].

Nous avons choisi d'utiliser les arbres syntaxiques abstraits pour les raisons suivantes:

- Ils sont le produit direct du processus d'analyse syntaxique et ne requièrent donc pas de calculs ni de manipulations additionnels pour être obtenus.
- Il est facile d'en extraire les entités du langage correspondant aux différences entre les clones.
- Ils sont faciles à manipuler lors d'activités de reconception et de réingénierie.
- Ils représentent un format de données manipulables par la machine.

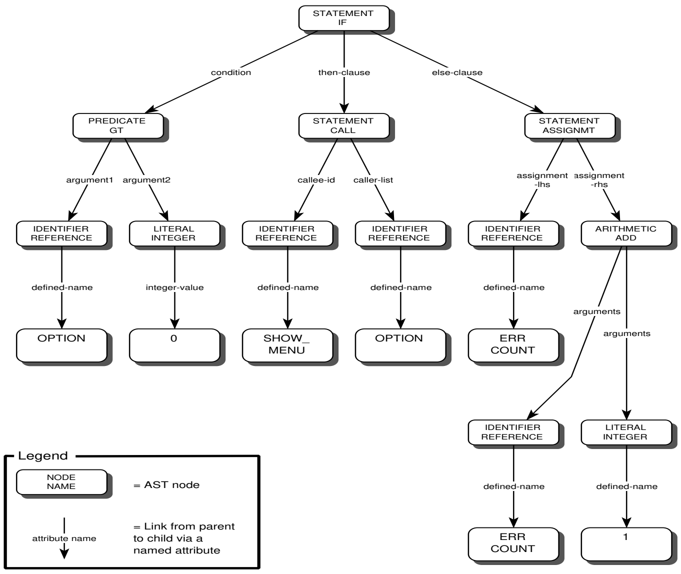


Figure 3.2: Un exemple de AST.

Un exemple d'AST est présenté à la figure 3.2.

3.3 Mise en correspondance

L'algorithme de comparaison utilisé est basé sur l'algorithme de mise en correspondance dynamique (*Dynamic Pattern Matching*) de Kontogiannis et al. [28]. Une modification fondamentale y a cependant été apportée. Au lieu d'aligner des entités syntaxiquement structurées comme des énoncés, l'algorithme aligne des entités syntaxiquement non-structurées, les jetons. C'est seulement par la suite que l'alignement obtenu est projeté sur les AST des fragments de code et que la correspondance optimale structurée est obtenue.

Kontogiannis et al. utilisent en effet les instructions comme éléments de comparaison. Celles-ci sont d'abord abstraites en des ensembles de caractéristiques. La correspondance est alors réalisée sur les vecteurs de caractéristiques correspondant aux blocs d'instructions. Les caractéristiques utilisées incluent des valeurs de métriques ainsi que des propriétés spécifiques de flux de données et de contrôle. L'approche que nous avons adoptée utilise les jetons du langage de programmation comme unités de comparaison. Cela permet aussi une granularité d'analyse affinée.

3.3.1 Données en entrée

La correspondance dynamique est donc effectuée sur deux vecteurs de jetons formant les fragments de code. Soit $t1_i$ le $i^{ème}$ jeton du premier fragment de code et soit $t2_j$ le $j^{ème}$ jeton du second fragment. Nécessairement i et j doivent être plus petits que les tailles respectives des deux vecteurs. Alors, ces vecteurs peuvent être définis comme: $v1 =< t1_1, t1_2, ..., t1_n >$ et $v2 =< t2_1, t2_2, ..., t2_m >$ où n et m sont les longueurs des fragments de code.

À partir des deux vecteurs, une grille est construite. Cette grille sera utilisée pour les résultats partiels de la mise en correspondance dynamique. La grille est constituée par des éléments du type:

type: ElementGrille

cout: Entier

precedent: ElementGrille

end

où:

- **cout** est le coût de la correspondance optimale entre les sous-séquences <v1[1],...,v1[i]> et <v2[1],...,v2[j]>

- **precedent** est une référence utilisée pour la mémorisation du chemin dans la grille formant la correspondance optimale.

Les fonctions *cout(grille[i][j])* et *precedent(grille[i][j])* associent à chaque élément de la grille les valeurs des champs *cout* et *precedent* correspondants.

Avant le début de la mise en correspondance, la première ligne et la première colonne de la grille sont initialisées aux coûts correspondant au pire cas, c'est-à-dire au cas où tous les jetons seraient différents. La correspondance optimale sera déterminée à partir de ce pire cas.

3.3.2 Coeur de l'algorithme

L'algorithme prend donc en entrée la grille et les deux vecteurs de jetons $v1$ et $v2$. Il retourne la correspondance optimale entre les vecteurs $v1$ et $v2$ sous la forme d'un chemin dans la grille. À ce chemin correspond un coût que nous appelons aussi *distance* et qui est défini comme la quantité minimale de jetons qui doivent être insérés, enlevés ou remplacés pour transformer un des vecteurs en l'autre.

L'algorithme est complètement défini avec les pré- et post- conditions suivantes:

Pré-conditon P:

$$cout(grille[i][0]) = i \ \forall i \in [0, n] \wedge$$

$$cout(grille[0][j]) = j \ \forall j \in [0, m] \wedge$$

$$cout(grille[i][j]) = 0 \ sinon \wedge$$

$$precedent(grille[i][j]) = nil \ \forall i \in [0, n] \ et \ j \in [0, m] \wedge$$

$$v1[i] = t1_i \ \forall i \in [1, n] \wedge$$

$$v2[j] = t2_j \ \forall j \in [1, m].$$

La post-condition suivante doit être vraie après exécution de la mise en correspondance:

$$cout(grille[0][0]) = 0 \wedge$$

$$cout(grille[i][j]) = d(< v1[1], ..., v1[i] >, < v2[1], ..., v2[j] >)$$

$$\forall i \in [1, n] \wedge \forall j \in [1, n]$$

où d() est la fonction de la distance

entre deux vecteurs de jetons.

Une fois la correspondance optimale trouvée, la grille contient les distances entre toutes les paires de sous-séquences correspondant aux i et j premiers jetons des vecteurs, pour toutes les valeurs de i et j plus petites que la taille des vecteurs. La distance pour les fragments de code complets est donnée par:

$$cout(grille[taille(v1)][taille(v2)].$$

Le coeur de l'algorithme qui permet de satisfaire $P \longrightarrow Q$ est défini par la fonction *match()*. L'algorithme de cette fonction est présenté à la figure 3.3.

fonction match(c: Grille; v1,v2: Sequence) => (coût: Entier)
  pour (i $\leftarrow$ 1 *à* taille(v1))
    pour (j $\leftarrow$ 1 *à* taille(v2))
      coutTemp $\leftarrow$ calculCout(v1[i],v2[j])

$$\text{c[i][j].cout} \leftarrow \min \begin{cases} c[i-1][j].cout + 1, \\ c[i][j-1].cout + 1, \\ c[i-1][j-1].cout + coutTemp \end{cases}$$

$$\text{c[i,j].precedent} \leftarrow \begin{cases} c[i-1][j], \\ c[i][j-1], \\ c[i-1][j-1] \quad \text{selon le cout minimal} \end{cases}$$

  retourne c[taille(v1)][taille(v2)]

Figure 3.3: Méthode principale de l'algorithme de mise en correspondance.

La fonction *match()* effectue une itération sur tous les éléments de la grille et calcule les distances entre toutes les sous-séquences $s1$ et $s2$ de $v1$ et $v2$. Pour chaque calcul, la fonction utilise les valeurs des distances optimales calculées pour les sous-séquences plus courtes. En effet, la distance entre $s1$ et $s2$ est le minimum parmi:

- la distance entre $front(s1)$ (tous les éléments de $s1$ sauf le dernier) et $s2$ augmentée du coût de l'effacement du dernier élément de $s1$ (coût de une unité).

- la distance entre $s1$ et $front(s2)$ augmentée du coût de l'insertion du dernier jeton de $s2$ (coût de une unité).

- la distance entre $front(s1)$ et $front(s2)$ augmentée du coût de la mise en correspondance des derniers éléments de $s1$ et $s2$, donnés par $< last(s1) >$ et $< last(s2) >$. Ce coût est calculé par la fonction *calculCoût()*.

La fonction *calculCoût()* compare deux jetons $k1$ et $k2$ en évaluant l'expression suivante:

$$
\begin{aligned}
(((Type(k1) = Type(k2)) \wedge (Type(k1) \neq Identificateur) \\
\wedge (Type(k1) \neq Constante)) \\
\vee((Type(k1) = Type(k2)) \wedge (Valeur(k1) = Valeur(k2)) \\
\wedge((Type(k1) = Identificateur) \vee (Type(k1) = Constante))))
\end{aligned}
$$

Ainsi, durant leur comparaison, les égalités des types et des valeurs des jetons sont testées. Deux jetons correspondent s'ils appartiennent au même type, sauf s'ils sont des identificateurs ou des constantes. Dans ces deux derniers cas, ils doivent aussi avoir la même valeur. Par exemple, le jeton “if”, peut seulement correspondre à un autre jeton “if” tandis qu'un identificateur peut seulement correspondre à un autre identificateur dont la valeur représente la même chaîne de caractères.

L'algorithme de comparaison tel que présenté a une complexité de $\theta(n * m)$, où n et m sont les tailles respectives, en quantité de jetons, des deux fragments de code. Pour des fragments similaires (des clones), $n \approx m$ et la complexité est approximativement $\theta(n^2)$. Cette complexité peut être réduite par l'utilisation de ce qui s'appelle un *beam search*. Il s'agit d'éliminer les possibilités de correspondance qui se situent trop loin de la diagonale de la grille, et qui demandent donc l'insertion ou l'effacement d'une quantité de jetons plus grande qu'un certain seuil. Cette optimisation est possible dans le cas de la comparaison de clones, puisque la solution se situe nécessairement près de la diagonale, les fragments comparés étant similaires.

3.3.3 Exemple

```
———— Copy 1 ——————
public int getSoLinger() throws SocketException {
  Object o = impl.getOption( SocketOptions.SO_LINGER);
  if( o instanceof Integer) {
    return(( Integer) o). intValue();
  }
  else {
    return - 1;
  }
}
———— Copy 2 ——————
public synchronized int getSoTimeout()
                              throws SocketException {
  Object o = impl.getOption( SocketOptions.SO_TIMEOUT);
  if( o instanceof Integer) {
    return(( Integer) o). intValue();
  }
  else {
    return 0;
  }
}
```

Figure 3.4: Exemple de méthodes clones en Java. Cet exemple est extrait de JDK 1.1.5, du fichier Socket.java et de la classe Socket.

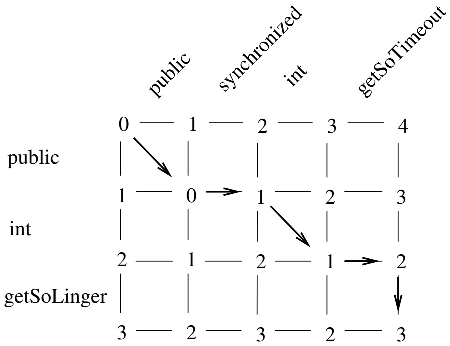


Figure 3.5: Grille après l'alignement des fragments de code v1 = <public, int, getSoLinger> et v2 = <public, synchronized, int, getSoTimeout> des méthodes présentées à la figure 3.4. Les flèches indiquent la correspondance optimale entre les fragments de code.

Soient les deux fragments de code de la figure 3.4; la figure 3.5 présente une partie de la grille après leur mise en correspondance. Les numéros apparaissant à chaque $grille[i][j]$ sont les valeurs des champs *ElementGrille.cout* des éléments correspondants de la grille. La correspondance optimale peut être représentée comme un chemin dans la grille: p = <<0,0>, <1,1>, <1,2>, <2,3>, <2,4>, <3,4>>. L'algorithme retourne donc la correspondance optimale sous la forme de chemin dans la grille où chaque jeton du chemin est associé à une action permettant la transformation d'un des vecteurs de jetons en l'autre. Ces actions sont l'insertion(flèche horizontale), l'effacement (flèche verticale) ou la substitution (flèche diagonale).

3.3.4 Groupement des jetons en séquences

La sortie de l'algorithme présenté plus haut n'est pas le résultat final de la comparaison des fragments de code. En effet, pour pouvoir ensuite les projeter sur l'AST, il faut regrouper en séquences les jetons consécutifs auxquels correspond une même action.

Il est facile d'extraire ces séquences de la grille. Il suffit de suivre le chemin optimal à partir du dernier élément ($grille[taille(v1)][taille(v2)]$) jusqu'au premier élément ($grille[0][0]$) en utilisant les champs *precedent* pour se déplacer. Les jetons consécutifs auxquels correspond une même action (insertion, effacement) ou qui correspondent sont groupés en séquences. Par la suite, les actions d'insertion et d'effacement consécutives sont remplacées par des actions de substitutions. Pour l'exemple de la figure 3.5, une telle transformation retourne le chemin sous la forme suivante: < {<public>,<public>, correspondance}, {ϵ,<synchronized>, insertion}, {<int>, <int>, correspondance}, {<getSoLinger>, <getSoTimeout>, substitution}>, où ϵ correspond à une séquence vide.

La figure 3.6 donne la correspondance optimale, en termes de séquences de jetons pour les fragments de code complets.

3.4 Union des ensembles de différences

L'algorithme présenté précédemment permet de comparer deux fragments de code. Les groupements de clones contiennent souvent plus de deux clones. Il faut

<public> dans la méthode originale et <public> dans le clone	Correspondance(pas d'action)
ϵ et <synchronized>	Insertion
<int> et <int>	Correspondance
<getSoLinger> et <getSoTimeout>	Substitution
< ( ) throws ... SocketOptions .> et < ( ) throws ... SocketOptions .>	Correspondance
<SO_LINGER> et <SO_TIMEOUT>	Substitution
<) ; ... else { return> et <) ; ... else { return>	Correspondance
<- 1> et <0>	Substitution
<; } }> et <; } }>	Correspondance

Figure 3.6: Correspondance optimale entre les deux fragments de code de la figure 3.4.

donc trouver un moyen de regrouper les différences et les correspondances, afin d'identifier le code commun à tous les clones ainsi que les particularités de chacun.

Pour effectuer cette opération, nous procédons en deux étapes. D'abord, tous les fragments de code sont comparés à un même fragment, choisi au hasard dans le groupement de clones. Ensuite, ce fragment est utilisé comme référence pour déterminer ce à quoi correspondent toutes les différences identifiées. Autrement dit, chaque différence est liée à son correspondant dans la méthode de référence. Ce correspondant est ensuite utilisé pour propager la différence aux autres méthodes.

3.5 Projection des différences sur l'AST

La dernière étape de l'extraction des différences sémantiques est la projection des différences sous forme de séquences de jetons sur les AST des méthodes comparées. Cela permet de déterminer les entités du langage de programmation qui correspondent aux différences.

Les étapes suivantes sont utilisées pour la projection:

- Identification des noeuds des AST auxquels appartiennent les jetons formant les différences.
- Pour les séquences, identification du premier ancêtre commun des noeuds formant la différence. L'ancêtre identifié doit être significatif, c'est-à-dire qu'il doit représenter une entité concrète du langage de programmation.
- Fusions des différences imbriquées.

L'algorithme et le processus de mise en correspondance présentés dans cette section permettent de comparer deux fragments et d'en extraire les différences sémantiques. Celles-ci prennent la forme de sous-arbres des AST des fragments de code correspondant à des entités significatives du langage de programmation.

CHAPITRE 4

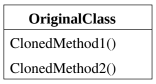
RECONCEPTION

L'objectif de la reconception basée sur les clones est de transformer ces derniers en des structures plus appropriées. Il s'agit en fait de factoriser les parties communes du code source tout en gardant le comportement unique de chacun des clones.

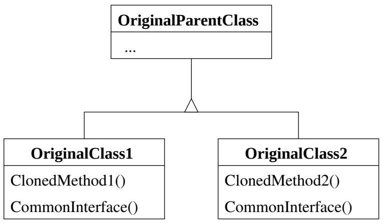
Deux axes de reconception sont étudiés ici. Le premier implique la transformation de la hiérarchie des classes pour grouper les parties communes des clones dans les classes ancêtres ou bien séparer les différences dans des classes enfants. Le second axe explore la transformation des clones en de nouvelles entités configurables, réutilisables, mais surtout indépendantes de la hiérarchie des classes existantes.

Ce chapitre commence par explorer les organisations des classes contenant les clones qui sont différenciées dans le choix du type de reconception. Sont ensuite discutées plusieurs possibilités de reconception permettant la factorisation des parties communes des clones, ainsi que la paramétrisation de leurs différences selon les deux axes de reconception mentionnés. Certaines solutions nécessitent le découplage de la méthode générale de son contexte et la paramétrisation de ce dernier. Ce problème est discuté et plusieurs possibilités de découplage sont présentées.

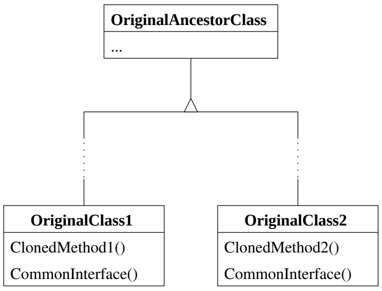
Finalement, les relations et les contraintes entre les organisations des classes



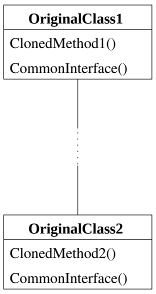
(a) Une seule classe.



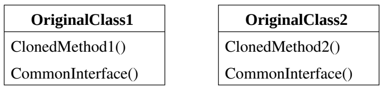
(b) Même parent.



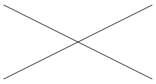
(c) Ancêtre commun.



(d) Une même branche.



(e) Aucune classe n'a de parent.



(f) Toute autre organisation.

Figure 4.1: Organisations des classes différenciées lors de la reconception.

contenant les clones, les choix de reconception, ainsi que la nécessité et les possibilités de découplage, sont examinés. Des solutions de reconception complètes sont aussi présentées.

4.1 Cas différenciés

Les méthodes clones appartiennent à des classes qui possèdent différentes relations de parenté. Ces liens de parenté, ou positions relatives dans la hiérarchie des classes du système, déterminent ce que nous appelons les organisations des classes contenant les clones.

Parmi toutes les organisations de classes possibles, nous avons identifié celles qui présentent un intérêt du point de vue de la réingénierie. Ce sont seulement ces organisations que nous différencions. La figure 4.1 présente ces différentes organisations. Des groupements de deux clones sont présentés afin d'alléger les figures. Les organisations présentées peuvent facilement être étendues à plus de deux clones.

Ainsi, tous les clones peuvent appartenir à une même classe 4.1(a). Il sera alors possible de faire des transformations locales à la classe ou encore de lui créer des descendants. Les clones peuvent aussi appartenir à des classes soeurs 4.1(b) ou de manière plus générale, à des classes ayant un ancêtre commun 4.1(c). Dans ce cas, l'ancêtre sera une cible de choix pour les éléments communs des clones. L'organisation où aucune classe n'a de parent 4.1(e) se rapproche du cas précédent puisque la création d'un parent sera possible.

Une situation différente se présente lorsque les classes contenant les clones appartiennent toutes à une même branche de la hiérarchie des classes 4.1(d). Cette situation peut néanmoins être la base d'une redistribution des éléments de données à différents niveaux de la hiérarchie.

Les classes peuvent aussi être liées par d'autres relations plus complexes, ou présenter un réseau impliquant simultanément plusieurs des relations décrites ci-dessus. Dans ces cas, l'utilisation des particularités des relations pour la réingénierie sera plus difficile, ce qui nous a amené à les regrouper sous un cas général (présenté en 4.1(f)).

En plus des clones, il se peut que les classes partagent d'autres attributs ou méthodes. Ces derniers forment alors une interface commune aux classes que nous appelons *CommonInterface*.

Finalement, il est important de noter que dans les organisations 4.1(b) à 4.1(e), il faut que chaque classe contienne exactement un clone pour que les solutions décrites ci-dessous s'appliquent directement. Sinon, nous tombons dans le cas général. Bien entendu, ce cas général peut être modifié pour tenir compte de l'organisation particulière des clones, mais il s'agira alors d'une solution hybride.

4.2 Factorisation des parties communes et paramétrisation des différences

Pour réussir la factorisation des parties communes des clones tout en préservant le comportement unique de chacun d'eux, les différences qui existent entre les clones doivent être encapsulées dans des méthodes et utilisées comme paramètres du code général commun. L'ensemble de ces méthodes qui encapsulent les différences définit une interface appelée *DiffInterface*.

Une fois les différences encapsulées, les parties communes des clones peuvent être unies de l'une des trois façons suivantes:

- Si tous les clones appartiennent à une même classe, chaque fragment de code commun peut être encapsulé dans une méthode et remplacé par un appel à celle-ci. Les méthodes communes peuvent alors toutes être gardées dans la classe. Les méthodes encapsulant les différences peuvent être gardées dans la classe ou déplacées vers de nouvelles classes enfants.

- Si un ancêtre commun existe ou peut être créé, le code commun paramétrisable peut être placé dans cet ancêtre. Les méthodes encapsulant les différences peuvent alors être gardées dans les classes originales.

- Finalement, le code commun peut aussi être placé dans une nouvelle classe, indépendante de celles contenant les clones. Les différences peuvent alors être gardées dans les classes originales (ce qui est possible lorsqu'un seul clone se trouve dans chaque classe), ou bien elles peuvent être placées dans de nouvelles classes agrégées par les anciennes.

Les sous-sections suivantes décrivent chacune de ces approches en détails.

4.2.1 Factorisation et paramétrisation au sein d'une seule classe

Lorsque toutes les méthodes clones appartiennent à une même classe, la duplication du code peut être vue comme une simple distribution inadéquate des responsabilités entre les méthodes de la classe. La reconception a donc pour objectif de redistribuer ces responsabilités au sein de cette classe. Il s'agit donc de garder l'interface commune des clones, leurs parties communes, ainsi que leurs différences, dans la classe.

Une des possibilités de redistribution consiste à diviser les méthodes clones en parties communes et en différences, puis à encapsuler à la fois les différences et les parties communes dans de nouvelles méthodes. Les corps des méthodes clones deviennent alors:

```
clonedMethodi() {
    partieCommune1();
    diff1versionA();
    partieCommune2();
    diff2versionA();
    partieCommune3();
    ...
}
```

Les méthodes de type *partieCommunei* forment une interface que nous appelons *CloneCommonInterface*, tandis que les méthodes qui encapsulent les différences forment l'interface *DiffInterface*. La figure 4.2 présente une classe contenant deux méthodes clones après application d'une telle réingénierie. Toutes les méthodes clones appellent alors les mêmes méthodes de la *CloneCommonInterface* et les méthodes correspondantes de leur version de la *DiffInterface*.

Cette solution permet une redistribution des responsabilités dans de petites méthodes simples, qu'il devient facile d'agencer de multiples façons pour obtenir

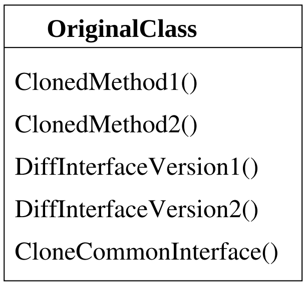


Figure 4.2: Factorisation et paramétrisation au sein d'une même classe.

les différents comportements souhaités. Cette solution est encore plus flexible que la création d'une méthode générale et paramétrisable.

Les nouvelles petites méthodes gardent le même contexte que les méthodes originales. Le problème de découplage mentionné en introduction ne se pose donc pas. Par contre, la difficulté de la séparation des méthodes clones en parties communes et en parties différentes dépend grandement des types des différences.

4.2.2 Paramétrisation de différences par création de classes enfants

Les clones d'une même classe peuvent aussi être éliminés par la création, dans la classe, d'une méthode générale et configurable. La difficulté réside alors dans l'efficacité de la paramétrisation des différences.

Certains langages de programmation, comme le C/C++, permettent l'utilisation de pointeurs à fonctions, et par ce fait leur passage comme paramètres. Dans d'autres langages, cela est impossible. Il devient alors nécessaire de grouper les méthodes formant les différences dans des structures ou des classes. Une référence à une instance d'une de ces classes est alors passée en paramètre à la méthode générale.

Ces nouvelles classes peuvent être agrégées par la classe originale, ce qui correspond au cas de reconception général (4.5) avec la différence que la classe *Clone-Handler* est la classe originale.

Une autre solution consiste à déclarer les classes contenant les différences comme enfants de la classe originale.

Dans cette solution, les corps des méthodes clones seront remplacés par:

```
clonedMethodi() {
    SousClassi objet = new SousClassi(this);
    objet.clonedMethod();
}
```

Chaque méthode clone doit instancier la sous-classe qui lui correspond, c'est-à-dire celle qui redéfinit les méthodes de manière à refléter les différences de ce clone. La sous-classe est ensuite utilisée pour répondre à la requête reçue. L'instanciation doit avoir lieu dans les méthodes afin d'éviter une boucle infinie d'appels au constructeur de la classe d'origine.

La figure 4.3 présente une classe et ses enfants après une telle transformation.

Le principal inconvénient de cette solution est la nécessité de découpler les méthodes encapsulant les différences de leur contexte. En effet, chaque différence doit manipuler les attributs, non pas de la classe qui la contient, mais plutôt de la classe mère, comme c'était le cas initialement. Les classes enfants doivent donc agréger la

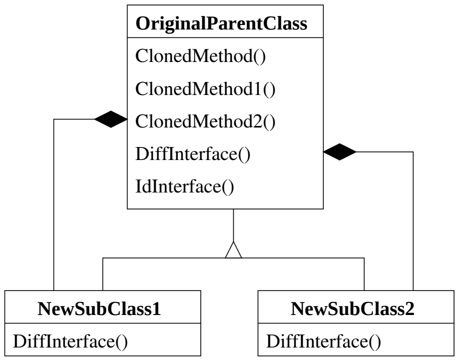


Figure 4.3: Paramétrisation des différences par l'utilisation de classes enfants.

classe parent et lui rediriger les opérations de manipulation de données non-locales aux méthodes.

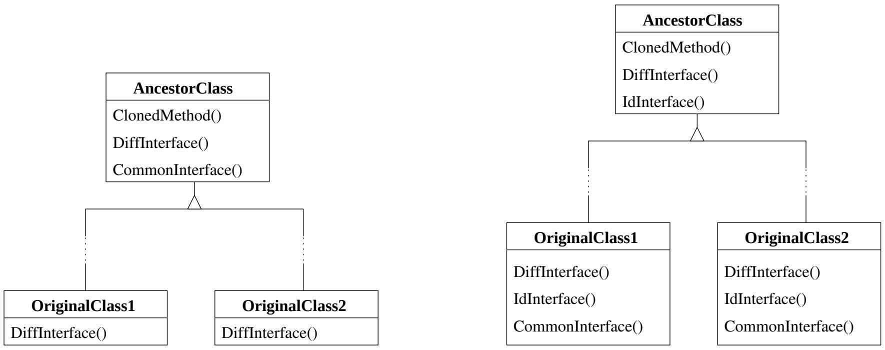
Un tel découplage est discuté en détails plus loin (c.f. section 4.3). Il consiste en l'encapsulation des manipulations de données dans des méthodes qui forment ce que nous appelons la *IdInterface*.

4.2.3 Factorisation à l'aide d'un ancêtre

Lorsque les classes qui contiennent les clones ont le même parent, un même ancêtre ou encore n'ont aucun parent, et qu'un parent commun peut leur être ajouté, la solution décrite dans cette section peut être appliquée lors de la reconception.

La solution consiste à déplacer les parties communes des clones vers l'ancêtre, et à laisser dans les classes d'origine les méthodes de la *DiffInterface* seulement. La factorisation se fait donc par l'utilisation d'un ancêtre, tandis que la paramétrisation tient en la redéfinition des méthodes de la *DiffInterface* dans les classes originales.

Lorsque les fragments communs des clones sont ainsi remontés vers un ancêtre, les attributs manipulés par ces fragments font partie de ce que nous avons appelé au début du chapitre l'interface *CommonInterface* des clones. Ils devraient donc aussi être remontés. Cependant, la difficulté d'une telle opération peut être évitée par le découplage de la méthode commune de son contexte. Ce découplage se manifestera par l'utilisation d'une *IdInterface* pour la manipulation des données, ainsi que par la redéfinition, dans chaque classe originale, des méthodes de cette interface. La figure 4.4 présente la solution avec et sans le découplage.



(a) Sans découplage. (b) Avec découplage.

Figure 4.4: Factorisation des fragments communs des clones dans un ancêtre commun.

4.2.4 Utilisation d'une classe externe

Lorsque l'organisation des classes originales ne permet pas l'utilisation de la hiérarchie des classes pour la factorisation des fragments communs dans un ancêtre, il faut avoir recours à une classe extérieure. Cette classe contiendra la méthode générale qui sera paramétrisable avec les différences et le contexte des clones.

Plusieurs organisations sont possibles pour la relation entre les classes d'origine et la nouvelle classe. Nous avons choisi d'explorer les patrons de conception pour former cette relation. L'utilisation de patrons de conception connus permet de créer un code facile à comprendre.

Nous proposons et décrivons trois solutions. Dans toutes ces solutions, les corps des méthodes clones seront remplacés par:

```
clonedMethodi(listeParametres) {
    cloneHandler.clonedMethod(listeParametres, diffContexteClone)
}
```

Le *cloneHandler* est un nouvel attribut de toutes les classes contenant des clones. Il est configuré avec un contexte qui est en fait l'instance de la classe originale à laquelle il appartient (c.f. section 4.3). La requête partagée lui est déléguée avec les paramètres qu'elle requiert, ainsi que l'information quant aux différences à utiliser.

4.2.4.1 Patron de conception *Strategy*

Dans cette solution, la paramétrisation des différences est effectuée par le patron de conception *Strategy*.

La figure 4.5 présente la structure des classes après reconception. Les méthodes clones délèguent les requêtes à une méthode générale de la nouvelle classe *Clone-Handler*. Cette classe est configurée avec une stratégie de type *DiffStrategy*, qui s'occupe de parties de la méthode générale correspondant aux différences entre les clones. Les classes qui représentent les stratégies redéfinissent donc les méthodes de l'interface *DiffInterface*. Elles sont agrégées par les classes d'origine afin de connaître leur contexte.

Une optimisation triviale permettant la réduction de la quantité d'instances de la classe *CloneHandler* consiste à passer la stratégie à chaque appel à la méthode générale, plutôt que de configurer chaque instance de la classe *CloneHandler* avec une stratégie différente.

Il est important de remarquer que la méthode de la nouvelle classe *CloneHandler* devra également recevoir son **contexte** en paramètre afin de faire correspondre, à chaque appel, les identificateurs aux structures de données appropriées. Cette remarque est également vraie pour les deux autres solutions présentées dans les sections suivantes. Les possibilités de découplage sont discutées à la section 4.3.

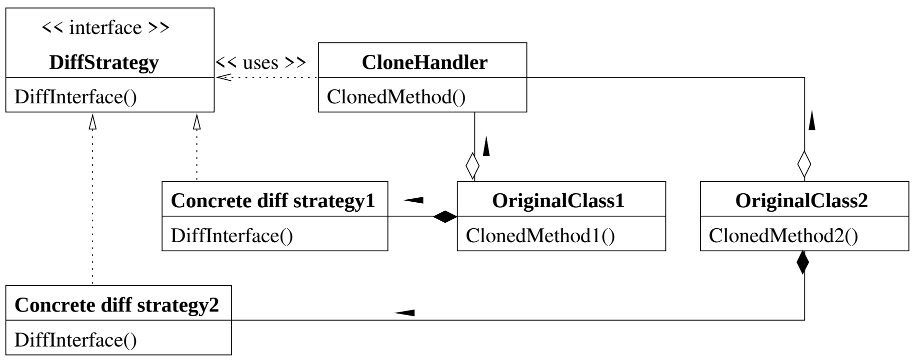


Figure 4.5: Patron de conception *Strategy* utilisé pour la paramétrisation des différences.

4.2.4.2 Patron de conception *Flyweight*

Cette solution est quasi-identique à la précédente, sauf qu'elle profite de la possibilité de transformer la composante configurable en *Flyweight*, réduisant ainsi la quantité d'instances de cette classe à une seule. La figure 4.6 présente cette solution.

Il est important de remarquer qu'il y a un coût à cette optimisation résultant de la nécessité de créer une classe *FlyweightFactory* qui gère les requêtes d'instanciation du *flyweight*.

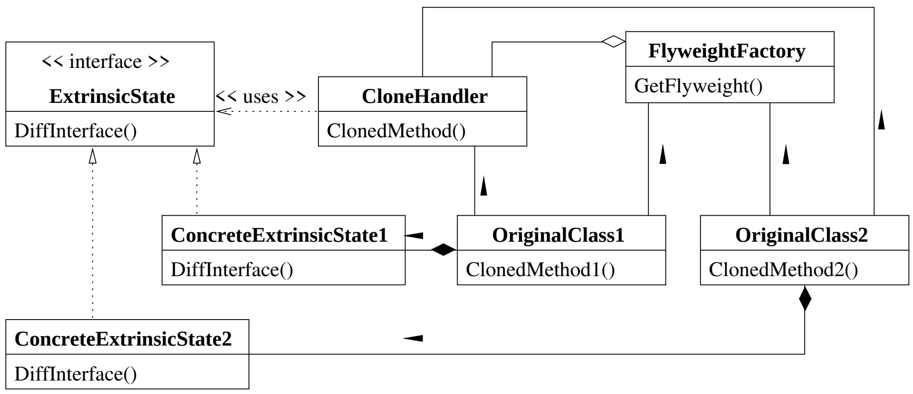


Figure 4.6: Patron de conception *Flyweight* utilisé pour la paramétrisation des différences.

4.2.4.3 Patron de conception *Template method*

Finalement, même lorsqu'une classe nouvelle et indépendante contient la méthode générale, des classes enfants peuvent être utilisées pour la paramétrisation des différences. En effet, ces classes peuvent redéfinir les méthodes de la *DiffInterface*. La figure 4.7 présente cette solution.

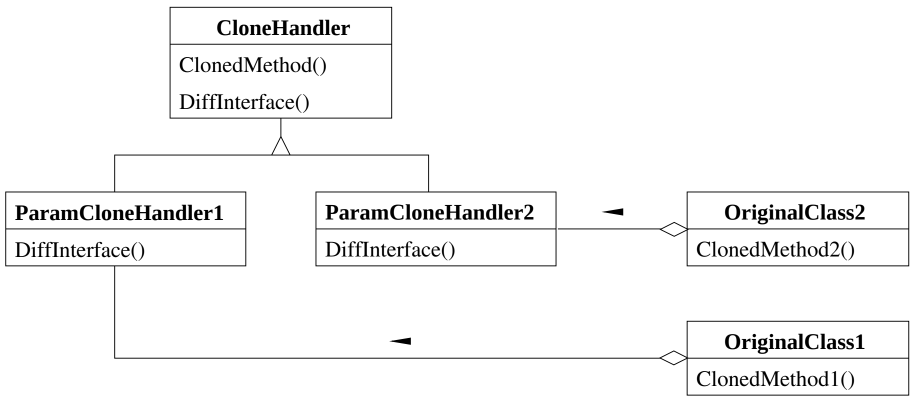


Figure 4.7: Patron de conception *Template method* utilisé pour la paramétrisation des différences.

Chaque clone original va rediriger ses requêtes vers la méthode générale de la sous-classe, qui redéfinit les différences de manière à refléter les comportements de ce clone.

4.3 Découplage du contexte

Dans quelques-unes des solutions de reconception décrites plus haut, il est nécessaire de découpler la nouvelle méthode générale de son contexte. En effet, la méthode doit changer la signification des identificateurs pour qu'ils correspondent à ceux d'une des classes originales, selon la provenance de l'appel qui lui est fait. Un tel découplage peut être effectué en encapsulant les opérations de manipulation de données extérieures dans des méthodes. L'ensemble de ces méthodes forme alors l'interface *IdInterface*.

Une fois l'interface *IdInterface* obtenue, la reconception peut être terminée par:

- L'utilisation de la hiérarchie des classes. Chaque sous-classe redéfinit les méthodes de l'interface tel que présenté à la figure 4.4(b).

- Le patron de conception *Strategy* tel que présenté à la figure 4.8.

- Le patron de conception *Flyweight* qui est, encore une fois, une variante de la solution précédente (figure 4.9).

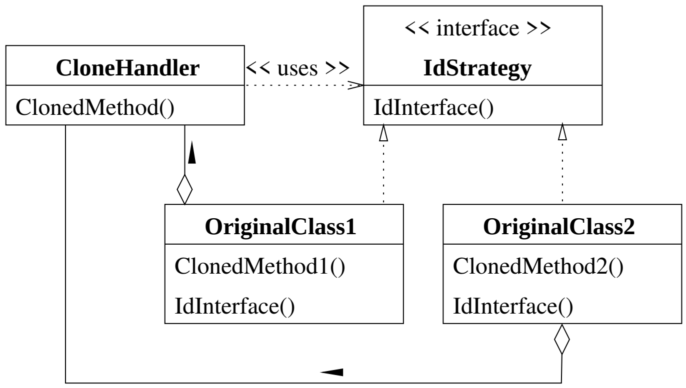


Figure 4.8: Patron de conception *Strategy* utilisé pour le découplage de la méthode générale de son contexte.

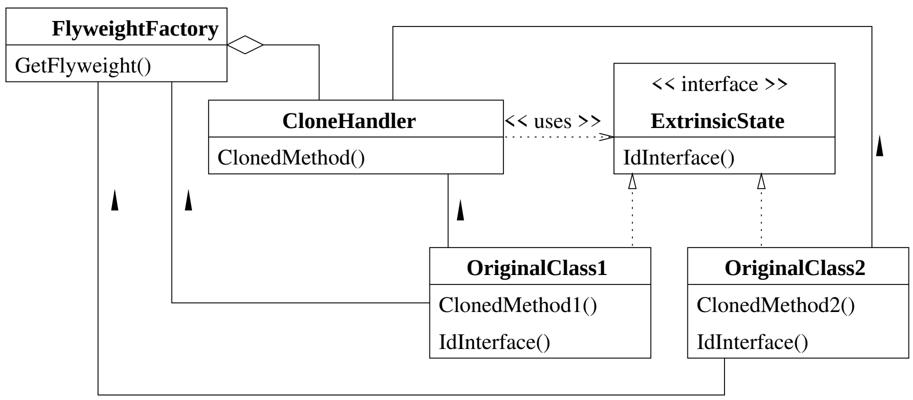


Figure 4.9: Patron de conception *Flyweight* utilisé pour le découplage de la méthode générale de son contexte.

Le choix de l'une ou l'autre des méthodes de découplage sera déterminé par le choix de la reconception principale factorisant les parties communes des clones.

Les relations entre les choix de reconception et de découplage sont discutés plus loin.

Tableau 4.1: Tableau de références pour les différentes organisations de classes

Nom	Numéro de figure
Organisations initiales	
Une seule classe	4.1(a)
Même parent	4.1(b)
Ancêtre commun	4.1(c)
Une même branche	4.1(d)
Aucun parent	4.1(e)
Autre	4.1(f)
Factorisations/Paramétrisations	
Dans une seule classe	4.2
Dans des classes enfants	4.3
Dans un ancêtre commun	4.4
Strategy	4.5
Flyweight	4.6
Template method	4.7
Découplage du contexte	
Dans un ancêtre	4.4(b)
Strategy	4.8
Flyweight	4.9

Le tableau 4.1 liste l'ensemble des organisations différenciées, des solutions de factorisation et des solutions de découplage du contexte avec les numéros des figures qui leur correspondent. Dans les sections suivantes, ces numéros seront utilisés pour les identifier.

Tableau 4.2: Reconceptions possibles en fonction des organisations initiales

Organisations initiales	Possibilités de reconception					
	4.2	4.3	4.4	4.5	4.6	4.7
4.1(a)	P2	P3	I1	P1	P4	P1
4.1(b)	I2	I2	P6	P1	P5	P1
4.1(c)	I2	I2	P6	P1	P5	P1
4.1(d)	I2	I2	P6	P1	P5	P1
4.1(e)	I2	I2	P6	P1	P5	P1
4.1(f)	I2	I2	I3	P1	P5	P1

4.4 Choix de solutions complètes

Cette section présente les relations entre les organisations de classes différenciées, les choix de reconception pour la factorisation des fragments communs, ainsi que les choix de reconception quant au découplage du contexte.

4.4.1 Choix de la reconception pour la factorisation

Les organisations initiales ont un grand impact sur les choix de reconception permettant la factorisation des parties communes des clones et la paramétrisation de leurs différences. Le tableau 4.2 montre les combinaisons des organisations initiales et des choix de reconception.

La légende pour les tableaux 4.2 et 4.3 est la suivante:

- P1: Solution possible et adéquate.
- P2: Peut être une très bonne solution selon les types de différences.
- P3: Possible, mais de nouveaux objets sont temporairement créés à chaque

appel. Ceci n'est pas vraiment un problème en Java grâce au ramasse-miettes.

- P4: Possible mais inutile, puisqu'une seule instance est nécessaire de toute façon.
- P5: Possible, mais valant la peine seulement lorsque beaucoup de clones sont unis.
- P6: Solution possible et adéquate. Moins d'analyse est nécessaire lorsque le découplage est utilisé.
- P7: Solution possible, mais qui sacrifie la performance. Le découplage n'est pas vraiment nécessaire dans ce cas.
- I1: Impossible. Les parties communes des clones sont déjà dans une même classe.
- I2: Impossible. Les clones n'appartiennent pas à la même classe.
- I3: Aucun ancêtre commun n'existe ni ne peut être créé.
- I4: Découplage inutile.
- I5: Impossible. Les mauvaises données seraient manipulées.
- I6: Combinaison impossible.
- M: Choix obligatoire.

Tableau 4.3: Possibilités de découplage selon les choix de reconception

Possibilités de reconception	Découplage obligatoire	Possibilités de découplage		
		4.4(b)	4.8	4.9
4.2	Non	I4	I4	I4
4.3	Oui	I5	M	I6
4.4	Non	P7	I6	I6
4.5	Oui	I6	M	I6
4.6	Oui	I6	I6	M
4.7	Oui	I6	M	I6

4.4.2 Choix du type de découplage

Une fois que la méthode de factorisation des parties communes et de paramétrisation des différences a été choisie, la nécessité et la méthode de découplage du contexte sont presque complètement définies. Le tableau 4.3 résume ces contraintes.

4.5 Solutions complètes avec patrons de conception

D'après le tableau 4.3, pour les solutions utilisant les patrons de conception, le découplage du contexte est obligatoire et les solutions sont uniques. Les figures 4.10 à 4.12 montrent les solutions complètes pour ces reconceptions.

4.6 Formalisation des reconceptions

Dans cette section, nous adoptons une approche plus formelle, et nous définissons précisément les organisations et transformations décrites dans les sections précédentes.

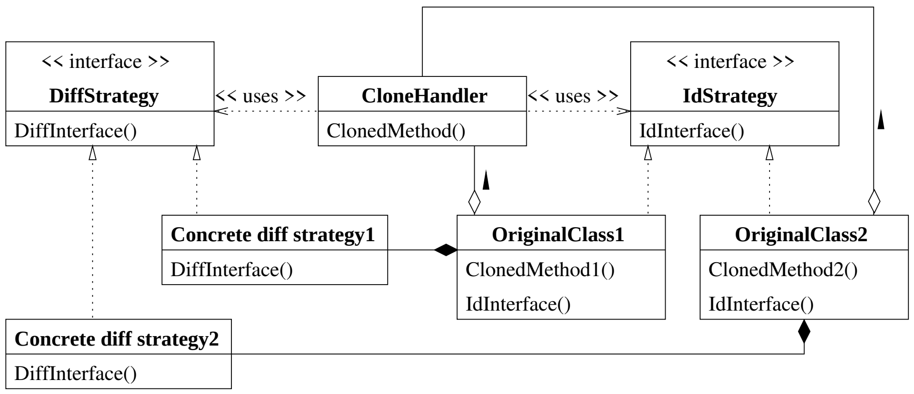


Figure 4.10: Solution complète basée sur *Strategy*.

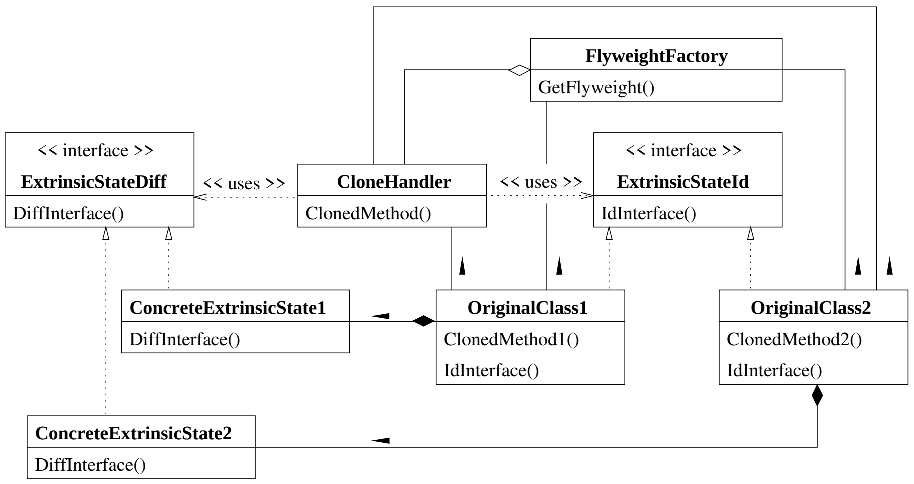


Figure 4.11: Solution complète basée sur *Flyweight*.

4.6.1 Définitions

Dans la notation utilisée plus loin, P(s) dénote l'ensemble puissance de s, c'est-à-dire l'ensemble de tous les sous-ensembles de s. $x[c_1, ..., c_n]$ dénote la projection de x sur ses composantes c_1 à c_n. Les éléments entre $< ... >$ représentent des tuplets, et entre $\{...\}$ des ensembles. Certains ensembles sont définis à l'aide de la notation suivante: $ensemble = \{D \mid P \bullet E\}$ où D est une déclaration, P est un prédicat restreignant les valeurs possibles des éléments de l'ensemble et E est

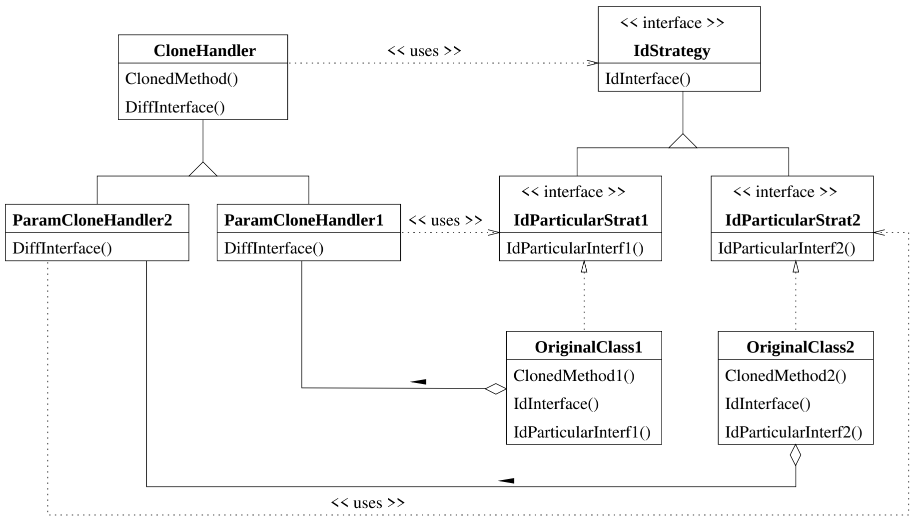


Figure 4.12: Solution complète basée sur *Template method*.

l'expression dénotant un élément. Lorsque l'expression E correspond directement à la déclaration, elle est omise.

- Soit $Classes = \{c_i\}$ l'ensemble des classes du système.
- Soit $Interfaces = \{t_i\}$ l'ensemble des interfaces du système.
- Soit $Methods = \{n_i\}$ l'ensemble des méthodes définies dans le système.
- Soit $Clones = \{m_i\}$ l'ensemble des méthodes dans le groupement de clones.
- Soit $Variables = \{v_i\}$ l'ensemble des variables définies dans le système.

Nous définissons:

- $Class : Methods \rightarrow Classes$ comme la classe à laquelle appartient une méthode.

- $Parent : Classes \rightarrow Classes$ comme le parent d'une classe.

- $Ancestors : Classes \rightarrow P(Classes)$ comme l'ensemble des ancêtres d'une classe.

- $Implementation : Classes \rightarrow P(Interfaces)$ comme l'ensemble des interfaces supportées par (implantées dans) une classe.

- $DefinedMethods : Classes \rightarrow P(Methods)$ comme l'ensemble des méthodes définies dans une classe.

- $Defined : Classes \rightarrow P(Variables)$ comme l'ensemble des attributs d'une classe.

- $Associations : Classes \rightarrow P(Classes)$ comme l'ensemble des classes auxquelles une classe est liée par des associations.

- $Aggregations : Classes \rightarrow P(Classes)$ comme l'ensemble des classes agrégées par une classe.

- $Uses : Classes \rightarrow P(Interfaces)$ comme l'ensemble des interfaces utilisées par une classe.

- $VarUsed : Methods \rightarrow P(Variables)$ comme l'ensemble des variables utilisées dans le corps d'une méthode.

- $Calls : Methods \rightarrow P(Methods)$ comme l'ensemble des méthodes appelées à partir du corps d'une méthode.

- $IdMethods : Clones \rightarrow P(Methods)$ comme l'ensemble des méthodes introduites pour un clone afin d'encapsuler ses accès à des données extérieures.

- $DiffMethods : Clones \rightarrow P(Methods)$ comme l'ensemble des méthodes introduites pour un clone afin d'encapsuler ses différences.

- $DiffClass = \{m : Clones, c : Classes \mid DiffMethods(m) =$
  $= DefinedMethods(c) \bullet m \rightarrow c\}$. Donc, DiffClass associe, à chaque méthode clone, une nouvelle classe qui va contenir toutes les méthodes de l'ensemble DiffMethods(m). Pour certaines solutions, DiffClass n'existe pas.

Finalement,

- $ClonedMethod = merge(Clones)$ est définie comme la méthode générale résultant de l'unification des corps des clones.

- $newMethod()$ est définie comme la transformation des corps des méthodes clones en de nouveaux corps tels que définis dans les sections précédentes.

- $getAncestor()$ est définie comme la méthode qui trouve l'ancêtre commun des classes contenant les méthodes clones, ou qui crée un nouvel ancêtre si aucune des classes n'a de parent.

- $CommonInterface$ est définie comme l'ensemble des attributs et méthodes qui ne sont pas directement des clones, mais dont les définitions sont dupliquées dans les classes contenant les clones. Elle est donc l'union de deux ensembles ensembles, et se définit comme:

$$
\begin{aligned}
CommonInterface = \{n \in Methods \mid \\
& \forall m \in Clones, n \in Calls(m) \\
& \wedge \; n \in DefinedMethods(Class(m))\} \\
& \cup \{v \in Variables \mid \forall m \in Clones, v \in VarUsed(m) \\
& \wedge \; v \in Defined(Class(m))\}
\end{aligned}
\tag{4.1}
$$

4.6.2 Cas différenciés

Les organisations initiales présentées à la figure 4.1 peuvent être définies par les pré-conditions suivantes sur le système:

- Organisation 4.1(a)

$$\forall m, n \in Clones, \quad Class(m) = Class(n) \tag{4.2}$$

- Organisation 4.1(b)

$$\forall m, n \in Clones, \quad Parent(Class(m)) = Parent(Class(n)) \tag{4.3}$$

- Organisation 4.1(c)

$$\begin{aligned}
&\forall m, n \in Clones, \quad \exists x, y \in Classes \\
&\mid x \in Ancestors(Class(m)) \\
&\wedge \; y \in Ancestors(Class(n)) \\
&\wedge \; x = y
\end{aligned} \tag{4.4}$$

- Organisation 4.1(d)

$$\begin{aligned}
&\forall m, n \in Clones, \quad Class(n) \in Ancestors(Class(m)) \\
&\vee \; Class(m) \in Ancestors(Class(n)) \\
&\vee \; Class(m) = Class(n)
\end{aligned} \tag{4.5}$$

- Organisation 4.1(e)

$$\forall m, n \in Clones, \quad Ancestors(Class(m)) = \varnothing \wedge Ancestors(Class(n)) = \varnothing \tag{4.6}$$

- Organisation 4.1(f)

$$true \tag{4.7}$$

4.6.3 Sélection du type de reconception

Les choix de reconception présentés aux figures 4.2 à 4.7 peuvent être définis avec les post-conditions sur le système ci-dessous. Les noms de composantes (classe, interface ou autre) suivis d'un apostrophe représentent la composante après transformation. Cette notation est empruntée de [25]

- Choix de reconception 4.2

$$\begin{aligned}
&\forall m \in Clones, c = Class(m) \\
&\wedge\ DiffInterfaceVersion_i = DiffMethods(m) \\
&\wedge\ DiffInterfaceVersion_i \subset DefinedMethods(c') \\
&\wedge\ ClonedMethod \in DefinedMethods(c') \\
&\wedge\ m' = newMethod()
\end{aligned} \tag{4.8}$$

- Choix de reconception 4.3

$$\begin{aligned}
&\forall m \in Clones, c = Class(m) \\
&\wedge\ ClonedMethod \in DefinedMethods(c') \\
&\wedge\ DiffMethods(m)[Signature] * \{\} \subset DefinedMethods(c') \\
&\wedge\ \ < c', DiffClass(m) > \ \in Aggregations \\
&\wedge\ \ < DiffClass(m), c' > \ \in Parents \\
&\wedge\ m' = newMethod()
\end{aligned} \tag{4.9}$$

- Choix de reconception 4.4(a) (L'aspect découplage de cette solution sera présenté plus loin).

$$
\begin{aligned}
&\forall m \in Clones, c = Class(m)\\
&\wedge\ AncestorClass = getAncestor()\\
&\wedge\ ClonedMethod \in Methods(AncestorClass')\\
&\wedge\ DiffMethods(m)[Signature] * \{\} \subset DefinedMethods(AncestorClass')\\
&\wedge\ DiffMethods(m) \subseteq DefinedMethods(c')\\
&\wedge\ < c', AncestorClass' > \ \in Ancestors\\
&\wedge\ m' \notin DefinedMethods(c')
\end{aligned}
\tag{4.10}
$$

lorsque le découplage n'est pas utilisé, la post-condition suivante doit aussi être vérifiée:

$$
\begin{aligned}
&\forall m \in Clones, c = Class(m)\\
&\wedge\ CommonInterface \in AncestorClass'\\
&\wedge\ CommonInterface \notin c'
\end{aligned}
\tag{4.11}
$$

- Choix de reconception 4.5

$$
\begin{aligned}
&\forall m \in Clones, c = Class(m)\\
&\wedge\ DiffStrategy = DiffMethods(m)[Signature]\\
&\wedge\ DiffStrategy \in Interfaces\\
&\wedge\ \exists CloneHandler : Classes \mid \{ClonedMethod\} =\\
&= DefinedMethods(CloneHandler)\\
&\wedge\ \ < CloneHandler, DiffStrategy > \ \ \in Uses\\
&\wedge\ \ < c', DiffClass(m) > \ \ \in Aggregations\\
&\wedge\ \ < DiffClass(m), DiffStrategy > \ \ \in Implementations\\
&\wedge\ \ < c', CloneHandler > \ \ \in Aggregations\\
&\wedge\ m' = newMethod()
\end{aligned}
\tag{4.12}
$$

- Choix de reconception 4.6

$$
\begin{aligned}
&\forall m \in Clones, c = Class(m) \\
&\wedge\ \exists FlyweightFactory : Classes \mid DefinedMethods(FlyweightFactory) = \\
&= \{GetFlyweight()\} \\
&\wedge\ ExtrinsicDiffState = DiffMethods(m)[Signature] \\
&\wedge\ \{ExtrinsicDiffState\} \subseteq Interfaces \\
&\wedge\ \exists CloneHandler : Classes \mid DefinedMethods(CloneHandler) = \{ClonedMethod\} \\
&\wedge\ < CloneHandler, ExtrinsicDiffState > \ \in Uses \\
&\wedge\ < FlyweightFactory, CloneHandler > \ \in Aggregations \\
&\wedge\ < c', DiffClass(m) > \ \in Aggregations \\
&\wedge\ < c', CloneHandler > \ \in Associations \\
&\wedge\ < c', FlyweightFactory > \ \in Associations \\
&\wedge\ < DiffClass(m), ExtrinsicDiffState > \ \in Implementation \\
&\wedge\ m' = newMethod()
\end{aligned}
\tag{4.13}
$$

- Choix de reconception 4.7

$$\begin{aligned}
&\forall m \in Clones, c = Class(m)\\
&\wedge\ \ Empty = DiffMethods(m)[Signature] * \{\}\\
&\wedge\ \ \exists CloneHandler : Classes \mid DefinedMethods(cloneHandler) =\\
&= \{ClonedMethod, Empty\}\\
&\wedge\ \ < c', DiffClass(m) > \ \ \in Aggregations\\
&\wedge\ \ < DiffClass(m), CloneHandler > \ \ \in Parent\\
&\wedge\ \ m' = newMethod()
\end{aligned} \tag{4.14}$$

4.6.4 Choix du découplage du contexte

Les solutions de découplage présentées aux figures 4.4(b) à 4.6 peuvent être définis avec les post-conditions sur le système suivantes:

- Solution présentée à la figure 4.4(b)

$$
\begin{aligned}
&\forall m \in Clones, c = Class(m)\\
&\wedge\ AncestorClass = getAncestor()\\
&\wedge\ ClonedMethod \in DefinedMethods(AncestorClass')\\
&\wedge\ IdMethods(m)[Signature] * \{\} \subset DefinedMethods(AncestorClass')\\
&\wedge\ < c', AncestorClass' > \ \in Ancestors\\
&\wedge\ IdMethods(m) \subseteq DefinedMethods(c')\\
&\wedge\ m' = newMethod()
\end{aligned}
\tag{4.15}
$$

- Solution présentée à la figure 4.8

$$
\begin{aligned}
&\forall m \in Clones, c = Class(m)\\
&\wedge\ IdStrategy = IdMethods(m)[Signature]\\
&\wedge\ \{IdStrategy\} \subseteq Interfaces\\
&\wedge\ \exists CloneHandler : Classes \mid DefinedMethods(CloneHandler) =\\
&= \{ClonedMethod\}\\
&\wedge\ \ < CloneHandler, IdStrategy > \ \ \in Uses\\
&\wedge\ IdMethods(m) \subseteq DefinedMethods(c')\\
&\wedge\ \ < c', IdStrategy > \ \ \in Implementations\\
&\wedge\ \ < c', CloneHandler > \ \ \in Aggregations\\
&\wedge\ m' = newMethod()
\end{aligned}
\tag{4.16}
$$

- Solution présentée à la figure 4.9

$$
\begin{aligned}
&\forall m \in Clones, c = Class(m)\\
&\wedge\ \exists FlyweightFactory : Classes \mid DefinedMethods(FlyweightFactory) =\\
&= \{GetFlyweight()\}\\
&\wedge\ ExtrinsicIdState = IdMethods(m)[Signature]\\
&\wedge\ \{ExtrinsicIdState\} \subseteq Interfaces\\
&\wedge\ \exists CloneHandler : Classes \mid DefinedMethods(CloneHandler) = \{ClonedMethod\}\\
&\wedge\ < CloneHandler, ExtrinsicIdState > \ \in Uses\\
&\wedge\ < FlyweightFactory, CloneHandler > \ \in Aggregations\\
&\wedge\ IdMethods(m) \subseteq DefinedMethods(c')\\
&\wedge\ < c', ExtrinsicIdState > \ \in Implementation\\
&\wedge\ < c', CloneHandler > \ \in Associations\\
&\wedge\ < c', FlyweightFactory > \ \in Associations\\
&\wedge\ m' = newMethod()
\end{aligned}
\tag{4.17}
$$

Dans ce chapitre, les possibilités de reconception, ainsi que leurs applicabilités et leurs contraintes, ont été discutées. L'applicabilité des solutions dépend des relations entre les classes contenant les clones, que nous avons appelées organisations initiales. Les contraintes sont principalement liées à la nécessité de découpler les méthodes générales de leur contexte. Les organisations initiales, ainsi que les différentes transformations, ont aussi été formellement décrites afin d'éviter toute ambiguïté.

CHAPITRE 5

EXPÉRIENCES

5.1 Classification des clones de plusieurs systèmes réels

La taxinomie des clones décrite dans le chapitre 2 a été implantée dans l'outil SMC (Similar Methods Classifier). L'outil a été développé en Java avec JDK 1.1.7. Nous avons appliqué SMC aux six systèmes utilisés pour le développement de la classification.

La figure 5.1 présente le processus de l'expérience. Pour chaque système, l'approche de Patenaude et al. [39] a été appliquée pour l'identification, à l'aide de métriques, des regroupements de méthodes similaires. Ces regroupements forment les données d'entrée de SMC. Pour chaque regroupement, SMC détermine précisément les types des relations de clonage, et classe les clones dans la catégorie qui correspond au type déterminé.

L'objectif de l'expérience de classification des clones dans les systèmes réels était de valider la taxinomie développée, ainsi que de mesurer les opportunités de réingénierie basée sur les clones qui existent dans les systèmes. Nous avons défini une opportunité de réingénierie comme étant un groupe de méthodes clones auxquelles une action de réingénierie concrète peut être appliquée. Plusieurs mesures ont été prises pendant l'expérience: la quantité des groupes de méthodes qui

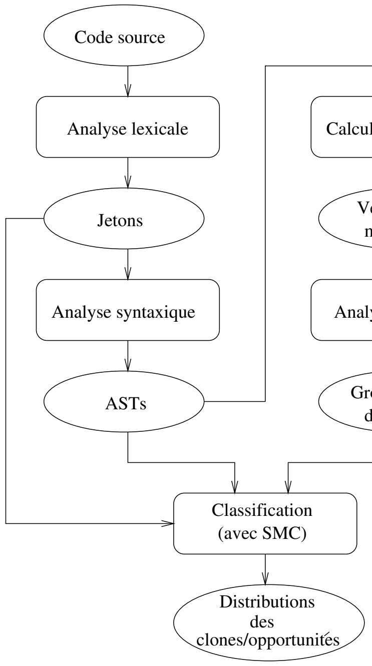


Figure 5.1: Processus de classification.

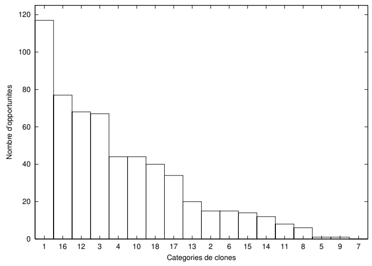
appartiennent à chaque catégorie de clone, le nombre de méthodes formant chaque groupe, ainsi que les tailles en nombre de lignes de code de ces méthodes.

Certaines méthodes étaient associées à plusieurs catégories de relation de clonage; nous avons calculé les différentes valeurs pour chaque catégorie séparément.

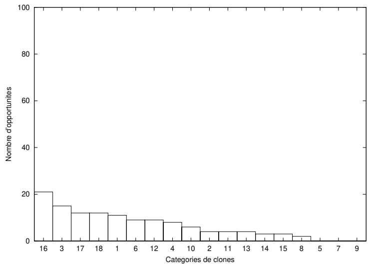
5.1.1 Résultats généraux

Les résultats détaillés de l'expérience sont présentés dans le tableau 5.1. Les figures 5.2(b) à 5.2(g) montrent les distributions des opportunités de réingénierie dans les différents systèmes. La figure 5.2(a) montre les résultats pour l'ensemble des systèmes.

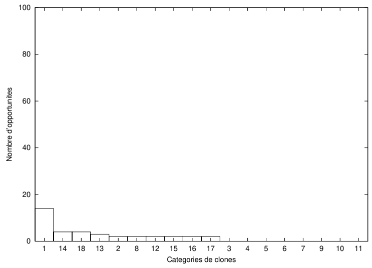
D'après la figure 5.2(a), la catégorie 1 ("Identiques") contient beaucoup plus



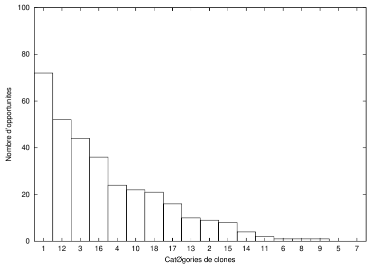
(a) Tous les systèmes.



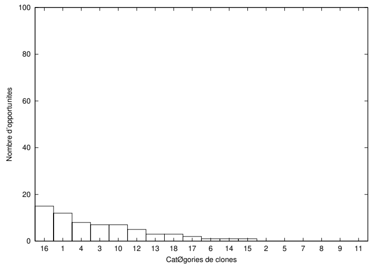
(b) JDK.



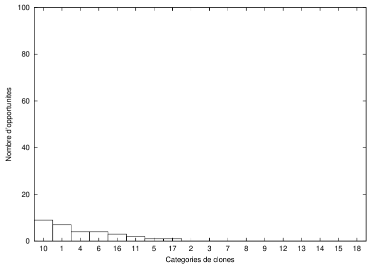
(c) ANTLR.



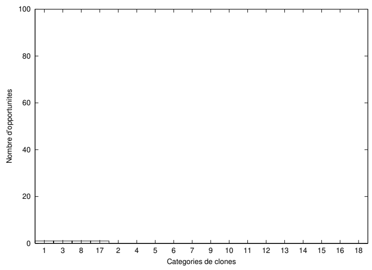
(d) SWING.



(e) KFC.



(f) SABLECC.



(g) HTTPCLIENT.

Figure 5.2: Distribution des opportunités dans des systèmes réels.

Tableau 5.1: Résultat de la classification des méthodes clones dans six systèmes. La première colonne indique le numéro des catégories. La seconde colonne indique la quantité de code source couverte par les clones de la catégorie, tandis que la troisième colonne traduit cette quantité en pourcentage de la proportion du système couverte par les clones. Les colonnes quatre et cinq représentent respectivement la quantité de méthodes et de groupes de méthodes (opportunités) de chaque catégorie.

	JDK—				SABLECC				ANTLR			
Catégorie	LOC	Pourc. clones	Méthodes	Groupes	LOC	Pourc. clones	Méthodes	Groupes	LOC	Pourc. clones	Méthodes	Groupes
1	432	5.9	42	11	954	35.5	94	7	593	24.2	51	14
2	42	0.6	8	4	0	0.0	0	0	40	1.6	4	2
3	464	6.4	36	15	0	0.0	0	0	0	0.0	0	0
4	198	2.7	26	8	476	17.7	49	4	0	0.0	0	0
5	0	0.0	0	0	12	0.4	2	1	0	0.0	0	0
6	229	3.2	24	9	1640	61.0	164	4	0	0.0	0	0
7	0	0.0	0	0	0	0.0	0	0	0	0.0	0	0
8	54	0.7	4	2	0	0.0	0	0	44	1.8	4	2
9	0	0.0	0	0	0	0.0	0	0	0	0.0	0	0
10	162	2.2	21	6	2016	75.0	215	9	0	0.0	0	0
11	117	1.6	13	4	70	2.6	4	2	0	0.0	0	0
12	354	4.9	35	9	0	0.0	0	0	375	15.3	29	2
13	134	1.8	10	4	0	0.0	0	0	69	2.8	11	3
14	56	0.8	6	3	0	0.0	0	0	198	8.1	8	4
15	135	1.9	9	3	0	0.0	0	0	157	6.4	4	2
16	742	10.2	58	21	52	1.9	7	3	363	14.8	29	2
17	233	3.2	28	12	14	0.0	2	1	173	7.1	7	2
18	334	4.6	30	12	0	0.0	0	0	201	8.2	9	4
Total	N/A	N/A	N/A	120	N/A	N/A	N/A	31	N/A	N/A	N/A	37

	SWING				KFC				HTTP			
Catégorie	LOC	Pourc. clones	Méthodes	Groupes	LOC	Pourc. clones	Méthodes	Groupes	LOC	Pourc. clones	Méthodes	Groupes
1	3332	14.5	245	72	294	10.7	29	12	42	17.0	2	1
2	274	1.2	24	9	0	0.0	0	0	0	0.0	0	0
3	1793	7.8	217	44	228	8.3	16	7	63	25.5	3	1
4	1420	6.2	192	24	135	4.9	20	8	0	0.0	0	0
5	0	0.0	0	0	0	0.0	0	0	0	0.0	0	0
6	12	0.05	2	1	117	4.3	9	1	0	0.0	0	0
7	0	0.0	0	0	0	0.0	0	0	0	0.0	0	0
8	28	0.1	2	1	0	0.0	0	0	21	8.5	3	1
9	28	0.1	2	1	0	0.0	0	0	0	0.0	0	0
10	1115	4.8	133	22	140	5.1	16	7	0	0.0	0	0
11	88	0.4	4	2	0	0.0	0	0	0	0.0	0	0
12	2729	11.9	269	52	111	4.0	13	5	0	0.0	0	0
13	314	1.4	21	10	40	1.5	6	3	0	0.0	0	0
14	68	0.3	8	4	16	0.6	2	1	0	0.0	0	0
15	278	1.2	22	8	144	5.2	16	1	0	0.0	0	0
16	2684	11.7	363	36	450	16.4	51	15	0	0.0	0	0
17	669	2.9	75	16	40	1.5	4	2	50	20.2	2	1
18	712	3.1	53	21	201	7.3	21	3	0	0.0	0	0
Total	N/A	N/A	N/A	323	N/A	N/A	N/A	65	N/A	N/A	N/A	4

d'opportunités que n'importe laquelle des autres catégories. Ce premier résultat est aussi vrai pour deux des systèmes pris séparément, ANTLR et SWING. Dans les autres systèmes, sauf JDK, la catégorie 1 n'est pas la première, mais ressort néanmoins comme l'une des plus importantes. Ce résultat est particulièrement intéressant, puisque les clones identiques seront évidemment les plus faciles à manipuler pendant la réingénierie.

Les trois catégories suivantes en terme de quantité d'opportunités sont les catégories 16 ("Une longue différence, interface et implantation"), 12 ("Changements d'interface et d'implantation") et 3 ("Appels de méthodes"). Néanmoins, ce résultat est fortement influencé par SWING dont la taille est beaucoup plus importante (200 000 lignes de code) que celle des autres systèmes (300 000 lignes en tout). Si nous considérons donc plutôt les cinq premières catégories de chaque système et que nous extrayons ensuite les trois catégories qui reviennent le plus souvent, nous obtenons toujours les catégories 16 et 3, mais au lieu de la catégorie 12, la catégorie 4 ("Variables globales") semble jouer un rôle important. Cela nous amène à tirer deux conclusions:

- Les résultats généraux ne reflètent pas les distributions des opportunités dans les systèmes pris séparément.

- Néanmoins, en plus de la catégorie 1, les catégories 16, 4 et 3 tendent à contenir la majorité d'opportunités. Donc, quel que soit le système, la réingénierie de ces catégories devrait avoir un impact significatif.

Par ailleurs, la catégorie 7 ("Variables locales") est vide pour tous les systèmes étudiés. Il semble donc que les variables locales ne soient jamais modifiées sans autre impact sur les méthodes. Cette catégorie pourrait donc être supprimée.

Finalement, les opportunités de réingénierie, soient les groupes de méthodes clones, ne sont pas de valeurs équivalentes. Certaines couvrent une plus grande partie du système en terme de quantité de lignes de code. D'autres contiennent plus de méthodes ou des méthodes plus longues. Par exemple, d'après les résultats présentés au tableau 2.1, la catégorie 8 contient deux opportunités impliquant 54 lignes de code, tandis que la catégorie 2 en contient 4 mais qui englobent seulement 42 lignes de code. La catégorie 8 de ANTLR contient 2 opportunités correspondant à 4 méthodes tandis que la catégorie 12 contient 29 méthodes pour le même nombre d'opportunités.

Ce résultat implique que l'analyse des distributions doit être faite sous différentes perspectives, et, selon l'impact recherché, différentes catégories de clones vont ressortir comme cibles de réingénierie recommandées.

5.1.2 Résultats particuliers aux systèmes

Les résultats présentés dans la section précédente sont vrais quel que soit le système étudié. Cependant, les histogrammes des figures 5.2(b) à 5.2(g) montrent que les distributions des opportunités varient d'un système à l'autre indiquant ainsi que plusieurs des caractéristiques des mesures dépendent des systèmes. Certains systèmes, comme SABLECC, ont tous leurs clones dans quelques catégories tandis que d'autres, dont JDK, ont des distributions beaucoup plus équilibrées. Ce résultat est extrêmement important, puisqu'il montre que les décisions de réingénierie nécessitent l'étude de la distribution des clones.

5.2 Reconception automatique

Pour montrer la faisabilité des reconceptions proposées dans le chapitre 4, nous avons implanté les types de reconception basés sur les patrons de conception *Strategy* et *Template method* dans l'outil CloRT (*Clone Reengineering Tool*).

L'outil CloRT a été développé en Java avec JDK 1.1.7. Pour les AST, l'outil utilisait un parseur Java généré avec Javacc version 0.8.

Nous avons appliqué CloRT à 26 des 120 opportunités de réingénierie identifiées dans JDK 1.1.5. La limite quant aux cas traités est dûe aux analyses nécessaires à l'encapsulation des différents types de différences. Pour pouvoir traiter tous les cas, il faudrait pousser encore plus la capacité d'analyse de CloRT, ou bien le greffer sur un compilateur commercial. Cependant, les clones utilisés suffisent à montrer qu'un processus automatique de reconception est possible, et à mesurer l'impact de la reconception sur les systèmes.

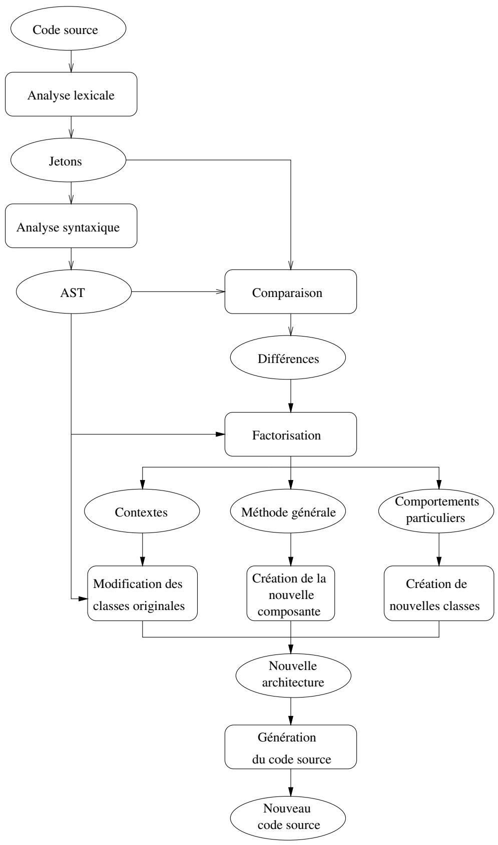


Figure 5.3: Processus de reconception.

5.2.1 Processus de reconception

Le processus de reconception utilisé est présenté à la figure 5.3. Les premières étapes du processus sont celles utilisées lors de la comparaison des clones et de l'extraction de leurs différences sémantiques. Une fois ces différences obtenues, les codes sources des fragments sont unis en une composante générale. Les différences sont paramétrisées et forment les *comportements particuliers*, tandis que les opérations dépendantes du contexte sont découplées. Par la suite, les classes originales sont modifiées pour comprendre les opérations de contexte, les nouvelles classes con-

Tableau 5.2: Distribution des clones formant la base de la reconception

Catégorie	Quantité de groupes
1	9
3	7
4	7
10	2
13	1
Total	26

tenant les différences sont créées, tandis que la méthode générale est incluse dans une nouvelle classe. La nouvelle architecture du système est ainsi créée. À partir de cette nouvelle conception, il est facile de regénérer le code source, ce qui est fait.

Ce processus a été appliqué aux 26 groupes de clones. Des exemples de code source de JDK avant et après reconception sont présentés aux annexes I à III. Pour chacune des reconceptions, le code source généré a été compilé avec succès pour former une nouvelle version de JDK. Des programmes simples qui utilisaient certaines des méthodes modifiées ont été exécutés avec succès dans les environnements reconçus.

5.2.2 Distribution des clones reconçus

Le tableau 5.2 présente le détail des catégories et quantités des clones transformés. Les 26 groupes de clones étaient formés de 56 méthodes appartenant à 37 classes.

5.2.3 Interaction avec l'utilisateur

Bien que le processus de reconception se veuille automatique, nous avons fait appel à l'utilisateur dans les situations suivantes:

- Lorsque plusieurs méthodes d'une même classe portent le même nom et ont exactement le même nombre de paramètres, choisir la bonne méthode, dans la table des symboles lors de la reconception, nécessite l'évaluation des types des expressions passées comme paramètres. Afin d'éviter cette analyse complexe et rarement nécessaire, nous demandons à l'utilisateur de choisir la bonne méthode parmi les possibilités identifiées par l'outil, lorsque la situation survient. Sur 26 groupes de clones, une interaction de ce type était nécessaire. Cette interaction pourrait être enlevée si l'analyse des types d'expressions était ajoutée à l'outil.

- Lorsque des objets de types différents sont accédés à partir des méthodes clones et que ces accès doivent être encapsulés, l'utilisateur doit choisir entre l'utilisation d'un ancêtre commun ou d'une interface commune dans la composante générale, ou bien l'encapsulation d'une plus grande partie de code couvrant la distinction de type. Cette interaction pourrait facilement être éliminée par l'imposition du second choix. Quatre de ces interactions ont été nécessaires lors de la reconception.

- Lorsque la première solution est choisie dans la situation précédemment

Tableau 5.3: Impact de la reconception basée sur *Strategy*

variation de la quantité de codes sources (lignes de code)	+2123
classes créées	80
interfaces créées	48
méthodes créées	171

décrite, l'outil identifie l'ensemble des classes ancêtres et des interfaces communes aux types des objets considérés. Si plus d'un choix est possible, l'utilisateur doit sélectionner le type approprié parmi les types possibles. Avec un peu plus d'analyses sur les utilisations des objets, le type approprié pourrait être sélectionné automatiquement. Deux interactions de ce type ont eu lieu durant chaque reconception.

- Finalement, la reconception nécessite l'ajustement de la visibilité de certaines méthodes et attributs. 14 modifications devaient être effectuées dans le cas de la reconception avec *Template method* et 11 avec *Strategy*.

Le processus de reconception expérimentale nécessitait donc quelques interactions avec l'utilisateur. Il serait néanmoins possible d'éliminer ces interactions en augmentant la quantité d'analyses effectuées par l'outil. Ce serait cependant coûteux en temps et en complexité.

5.2.4 Reconception avec le patron *Strategy*

Le tableau 5.3 présente les résultats de la reconception en utilisant le patron de conception *Strategy*. Nous pouvons voir que la reconception augmente finalement la quantité de code source dans le système, et ce de manière significative. Une telle augmentation se produit souvent dans des systèmes reconçus. En effet, ce résultat est en accord avec les pratiques de conception actuelles, qui tendent à créer des systèmes contenant plus de petites classes et de petites méthodes que des systèmes à plus fort couplage.

L'augmentation de la quantité de codes sources provient principalement de deux aspects de la reconception. Tout d'abord, une grande quantité de méthodes sont ajoutées pour l'encapsulation des différences et du contexte. Aussi, les interfaces et classes sont un coût constant pour chaque groupe de clones. Ce coût devient proportionnellement plus faible avec l'augmentation de la quantité de clones dans un groupe, et de la longueur des clones par rapport aux quantités de différences et d'opérations dépendantes du contexte.

5.2.5 Reconception avec le patron *Template method*

Le grand avantage de la reconception basée sur le patron de conception *Template method* est que les classes contenant les différences sont indépendantes des classes originales. Elles peuvent donc être partagées et réutilisées. Le tableau 5.4 présente les résultats de cette reconception.

Tableau 5.4: Impact de la reconception basée sur *Template method*

variation de la quantité de codes sources (lignes de code)	+3223
classes créées	80
interfaces créées	80
méthodes créées	204

Comme le montre les résultats, la reconception ajoute encore plus de codes sources. Cela est dû à la nécessité de découpler le contexte dans les différences aussi bien que dans le code commun.

D'après les résultats, nous pouvons voir qu'une grande partie du nouveau code source est liée aux interfaces et aux classes créées. Les autres reconceptions que nous avons proposées dans le chapitre 4 seraient donc probablement plus avantageuses, parce qu'elles ne nécessitent pas la création de nouvelles classes, mais se basent plutôt sur le déplacement des attributs, des méthodes et des parties communes des clones entre les différentes classes déjà existantes. Cependant, les reconceptions proposées ici sont applicables à tous les cas de clonage.

Nous pouvons aussi conclure que le choix de la méthode de reconception des clones a un impact important sur le système reconçu.

5.2.6 Compréhension du système reconçu

Le nouveau code du système reconçu (c.f. annexes II et III) est basé sur des patrons de conception. Les classes créées comportent uniquement des méthodes courtes, effectuant chacune une seule tâche précise. Ce code devrait donc être facile à comprendre pour un programmeur. L'aspect le plus difficile à saisir réside dans la stratégie d'attribution des noms aux classes et aux méthodes. Ces noms sont présentement générés par la machine, par concaténation des noms des clones et de leurs classes. Une telle stratégie génère malheureusement des noms complexes qui alourdissent le code. Cependant, cette stratégie peut facilement être remplacée.

Finalement, la reconception appliquée à JDK 1.1.5 montre qu'il est possible de reconcevoir automatiquement des systèmes en se basant sur l'information de clonage. Il est possible de regrouper le code partagé par les clones tout en préservant les particularités de chacun. Le nouveau code source généré est facile à comprendre, car il repose sur des patrons de conception connus. Les approches expérimentées augmentent néanmoins la quantité de codes sources dans le système.

CHAPITRE 6

DISCUSSION

Nos expériences montrent donc qu'il est possible de transformer de manière automatique des clones en des composantes réutilisables et paramétrisables. Ce résultat est tout à fait nouveau, puisque les autres études sur les clones ne font que les visualiser, les gérer ou, au mieux, les remplacer par des macros.

La difficulté de reconception des clones dépend des différences qui les séparent. La classification que nous avons développée permet donc d'étudier le phénomène de clonage dans un système avec une perspective de reconception.

6.1 Avantages de la reconception

6.1.1 Maintenance

La reconception basée sur les clones permet de transformer l'architecture d'un système de manière à éliminer les duplications, à rendre explicites les liens entre les classes qui partagent la fonctionnalité clonée et à faciliter la réutilisation des nouvelles composantes paramétrisables.

Nous avons proposé plusieurs types de reconception qui se distinguent par leur applicabilité et la facilité de réutilisation qu'ils permettent. En effet, toutes les solutions ne sont pas toujours applicables. Certaines solutions nécessitent l'existence d'un ancêtre commun ou la possibilité d'en créer un. Créer des composantes paramétrisables favorise plus la réutilisation générale que de transformer la hiérarchie des classes, puisqu'une nouvelle composante accessible par tous est générée. Cependant, la reconception de la hiérarchie des classes a un impact moindre en terme de variation de la taille des systèmes.

La reconception est en accord avec le processus de *"Problem mining"* de Lagüe et al. [30], qui consiste à gérer la présence des clones dans un système en gardant les liens entre ces derniers et en affichant toutes les méthodes clones à chaque fois que l'une d'elles est modifiée. Cela permet au programmeur de déterminer si la modification doit être propagée aux autres copies de la méthode modifiée ou non. Notre méthode de reconception permet de propager les modifications effectuées aux parties communes des clones de manière automatique, puisque ces parties sont unies en une composante unique. Les modifications des aspects particuliers des clones restent locales. Le programmeur n'a donc pas besoin de se soucier des impacts de ces modifications sur les autres variantes de la fonctionnalité.

De plus, comme la nouvelle structure sépare clairement le code commun des différences, il est facile de déterminer comment incorporer des modifications pour qu'elles n'affectent qu'une seule méthode, ou bien plusieurs méthodes à la fois. Si des nouvelles différences sont apportées, toute la structure nécessaire à les accueillir est déjà présente. Le programmeur doit simplement s'assurer d'insérer les méthodes dans les bonnes classes et de placer l'appel à ces méthodes au bon endroit dans la composante commune. Ainsi, la modification et l'expansion des fonctionnalités clonées sont simples après la reconception.

Finalement, un autre avantage important de notre approche est la comparaison des clones qui est à sa base. Cette comparaison permet de déterminer les différences sémantiques entre deux fragments de code, soit les différences interprétées en termes des composantes du langage de programmation auxquelles elles correspondent. Une telle interprétation permet donc de poursuivre des reconceptions qui gardent la bonne signification du code source et qui produisent un nouveau système facilement compréhensible et modifiable par un programmeur.

Ces caractéristiques du processus de reconception proposé devraient donc jouer dans l'amélioration de la maintenabilité des systèmes, c'est-à-dire la facilité d'y ajouter de nouvelles fonctionnalités ou d'y corriger des erreurs sans risque d'effets secondaires inattendus.

6.1.2 Applicabilité

L'approche que nous proposons peut être appliquée indépendamment des techniques de détection des clones. En effet, la comparaison des clones peut être effectuée sur n'importe quels fragments de code, peu importe comment ils ont été obtenus. Cela est important, puisqu'il existe plusieurs techniques de détection de clones. Les groupes désirant essayer notre approche de reconception peuvent donc le faire sans devoir changer de méthode de détection.

6.1.3 Gestion du risque

Finalement, l'une des plus importantes caractéristiques de la reconception basée sur l'analyse des clones est qu'elle est partielle. Elle n'affecte donc jamais le système dans son ensemble. Cette caractéristique est importante pour la gestion du risque lié à toute reconception. La reconception est, en effet, ciblée et contrôlée. Seuls certains fichiers très précis sont modifiés, et ces changements n'ont pas de répercussions sur le reste du système, puisque les interfaces des classes restent identiques.

6.2 Limites d'une approche automatique

6.2.1 Manque de flexibilité

La principale limite de l'approche proposée est, comme l'ont montré les expériences, son impact sur la taille du système. Cet impact est particulièrement important pour les types de reconception que nous avons implantés. En effet, l'utilisation de la hiérarchie des classes pour les reconceptions aurait un impact moindre puisqu'aucune classe ni interface n'aurait besoin d'être créée. Seules les classes existantes seraient utilisées. Cependant, les reconceptions basées sur les patrons de conception peuvent être préférables dans certaines situations, comme lorsque l'on désire unir des clones sans créer de liens étroits entre les classes auxquelles ils appartiennent. Cela montre une limite d'une méthode de reconception complètement automatique qui ne permettrait pas de choisir le type des reconceptions. Une telle approche ne profiterait pas de la capacité d'un programmeur à juger de la meilleure solution pour chaque cas de clonage. Notre approche, au contraire permet de sélectionner la reconception la plus appropriée pour chaque groupe de clones. Pour le moment, le choix se limite à deux reconception mais il pourrait être étendu sans que l'approche ne doive être modifiée.

6.2.2 Performance

Les redirections ajoutées par les reconceptions basées sur les patrons de conception ont un impact sur la performance des méthodes transformées. Des modules critiques ne devraient donc pas être transformés. Encore une fois, l'utilisateur devrait avoir le choix de reconcevoir certains clones ou non.

6.2.3 Complexité des outils d'analyse

L'encapsulation des différences et des opérations qui dépendent du contexte nécessite une analyse détaillée des fragments de code à encapsuler. Il faut déterminer les paramètres à passer, et surtout leurs types. Ces derniers doivent être les mêmes dans toutes les variantes des méthodes pour que les signatures soient les mêmes. Cela demande parfois de recourir à des ancêtres communs ou de changer la portée de l'encapsulation pour faire disparaître les divergences de type. Bref, ces opérations demandent une analyse détaillée et coûteuse à implanter.

Par contre, ces opérations sont simples du point de vue d'un programmeur qui peut tenir compte facilement des différentes contraintes, et peut prendre les meilleures décisions pour chaque cas de figure.

6.3 Approche interactive

À la lumière des expériences effectuées, nous pouvons conclure qu'une approche interactive unissant les forces d'un outil automatique et d'un humain permettrait d'obtenir les meilleurs résultats pour la reconception de systèmes. L'outil pourrait en effet présenter toutes les informations nécessaires à la prise de décision, qui, elle, pourrait être effectuée par l'humain.

6.3.1 Sélection des candidats à la reconception

En premier lieu, l'approche de reconception que nous avons élaborée permet de classer les clones en catégories reflétant leurs différences et leurs difficultés de reconception. Une telle classification permet donc à un humain de juger des meilleurs candidats pour la reconception en terme d'effort et de gain.

Par la suite, l'outil pourrait aussi fournir des informations détaillées quant à la longueur des clones, aux quantités de différences, aux types des différences, etc. permettant encore mieux de comparer les candidats à la reconception.

Un outil pourrait donc faciliter le choix des groupes de clones servant de base à la reconception, et même l'ordonnancement de ces derniers par niveau de priorité.

6.3.2 Proposition des reconceptions applicables

Une fois des groupes de clones sélectionnés pour la reconception, l'outil pourrait proposer différentes reconceptions applicables à chaque groupe. En effet, l'applicabilité des solutions est déterminée par des pré-conditions précises sur les clones candidats.

Le programmeur pourrait alors juger de la situation et déterminer la meilleure solution. Une telle situation permettrait d'éviter les cas où des clones simples sont remplacés par des structures complexes, comme cela peut se produire dans une approche automatique.

L'ajout de nouvelles reconceptions possibles nécessiterait alors la définition des pré-conditions régissant leur applicabilité.

6.3.3 Liste des actions à effectuer

Une fois une solution de reconception choisie, l'outil pourrait en préciser les actions à effectuer en déterminant les opérations relatives aux contextes ainsi que les différences précises entre les clones.

L'utilisateur pourrait alors effectuer ces transformations en sachant que la liste des actions est exhaustive et qu'il peut être sûr de ne rien oublier. Il aurait aussi toute la flexibilité nécessaire pour déterminer les meilleures méthodes pour encapsuler les différences et les opérations de contexte. Cela résulterait probablement en des solutions plus efficaces que ce qui peut être produit de manière automatique.

6.3.4 Automatisation de certaines actions

Finalement, l'outil pourrait néanmoins effectuer certaines tâches de manière automatique. Les encapsulations implantées actuellement peuvent être des solutions par défaut, que l'utilisateur pourrait choisir pour certaines transformations.

Finalement, la reconception des clones peut aider à améliorer la maintenabilité d'un système. Une solution automatique est possible, mais reste coûteuse en analyse et peu flexible. Une approche semi-automatique serait probablement plus avantageuse, et produirait des résultats plus intéressants.

CHAPITRE 7

TRAVAUX RELIÉS

Notre étude touche plusieurs aspects de la réingénierie et du génie logiciel qui ont déjà été abordés par d'autres chercheurs dans des contextes différents. Dans ce chapitre, nous comparons nos recherches à celles existant déjà dans ces domaines connexes.

Nous abordons, en premier lieu, les recherches portant sur l'extraction de la structure des systèmes et sur leurs reconceptions, puisque c'est le coeur de notre projet. Par la suite, nous examinons les autres travaux sur l'extraction de différences sémantiques entre fragments de code. Nous discutons aussi de quelques recherches portant sur la fusion de codes sources et les comparons à notre factorisation des éléments communs des clones. Nous comparons finalement les travaux existant sur les patrons de conception à notre utilisation de ces derniers.

7.1 Transformation de l'architecture des systèmes

L'approche de reconception basée sur les clones que nous proposons extrait partiellement la structure des systèmes. Elle extrait la hiérarchie des classes et des interfaces ainsi que les relations de clonage. Elle utilise ensuite la structure extraite pour reconcevoir partiellement le système.

Dans cette section, nous comparons cette facette de notre étude à d'autres études sur la récupération de la structure des systèmes et sa reconception. Ces activités sont devenues relativement importantes dans le cycle de vie des logiciels [13], et sont donc les sujets de nombreuses recherches, dont nous mentionnons quelques-unes des plus importantes.

7.1.1 Extraction de la structure à partir du code source

L'extraction de la structure des systèmes ou de leurs spécifications à partir du code source est appelée rétro-ingénierie. La principale différence entre la reconception basée sur les clones que nous proposons et les autres recherches existant dans le domaine de la rétro-ingénierie se situe au niveau des informations utilisées pour rebâtir la structure des systèmes.

La plupart des autres recherches [8, 14, 22] se concentrent sur l'identification d'entités (modules, fonctions, classes, etc.) et de leurs relations. Cependant, les relations extraites sont toujours explicites comme des échanges de données ou des invocations de méthodes.

L'extraction de la structure utilisée dans la reconception basée sur les clones diffère des autres études en ce qu'elle extrait des liens implicites entre les modules et les fonctions. Ces liens sont les relations de clonage.

Cependant, certaines études, comme celles de Harris et al. [22] ou celles de Choi et Scacchi [14], pourraient extraire de telles informations, puisqu'elles étudient le fonctionnement intérieur des modules, et notamment leurs descriptions fonctionnelles. Elles pourraient faire correspondre ces descriptions et ainsi créer des liens entre différentes composantes. Kontogiannis et al. ont utilisé, dans [28], la concordance statistique entre les patrons d'une description abstraite du code et le code source lui-même pour l'identification de relation de clonage. Contrairement à Kontogiannis et al., nous faisons des correspondances directes de code source à code source pour établir les liens de clonage, sans passer par une description intermédiaire.

Les études dont l'objectif est la récupération de la structure des systèmes extraient ou sont capables d'extraire, à la demande de l'utilisateur, beaucoup plus d'informations que la reconception basée sur les clones. Nous extrayons uniquement les relations de clonage ainsi que la hiérarchie des classes et interfaces du système. Les autres études extraient les relations liées aux appels de méthodes, aux passages de paramètres, aux communications inter-processus et bien d'autres.

7.1.2 Reconception

Plusieurs études dans le domaine de la reconception des logiciels s'apparentent à notre étude. Nous comparons ici notre approche de reconception à celles qui existent dans les domaines de la transformation des systèmes procéduraux en systèmes orientés-objet, et de la transformation des structures de classes de systèmes déjà orientés-objet.

7.1.2.1 Du procédural vers l'orienté-objet

Les études qui transforment les systèmes procéduraux en systèmes orientés-objet [15,40] cherchent avant tout à regrouper les fonctions et les variables existant déjà dans les systèmes en des entités significatives qui vont former la hiérarchie des classes, des modules et des sous-systèmes.

Les fonctions et variables sont regroupées sous la base de leurs relations d'appels de fonctions et d'accès aux variables.

Ces approches ne transforment donc pas la structure du système, mais y ajoutent plutôt des niveaux d'abstraction. Au contraire, nos restructurations ne modifient pas les niveaux d'abstraction. Elles manipulent ceux existant déjà. Par contre, notre approche change l'implantation des composantes, c'est-à-dire leur code source, ce qui n'est pas le cas des autres études.

7.1.2.2 Réorganisation des classes

La réorganisation des classes est une activité très importante lors de la maintenance, et surtout de la réingénierie. Elle permet la consolidation de la structure des systèmes, pour que ces derniers reflètent mieux l'ensemble de leurs spécifications et soient plus faciles à modifier et entretenir. La restructuration des classes fait donc l'objet de nombreuses études [6,7].

Contrairement à la migration du code procédural en OO, les recherches portant sur la réorganisation de la hiérarchie des classes d'un système fusionnent certaines méthodes et variables. Elles font donc plus qu'ajouter un niveau d'abstraction à la structure du système, elles la modifient réellement.

Dans [11], Casais propose la réorganisation de la hiérarchie des classes basée sur l'analyse des redéfinitions de méthodes. La réorganisation consiste en des décompositions d'étapes d'abstraction en plusieurs sous-étapes afin de diminuer la valeur sémantique de chaque opération. Elle consiste aussi en des restructurations qui extraient les propriétés partagées par plusieurs classes et les fusionnent dans un nouvel ancêtre commun. Cette dernière activité est similaire à notre restructuration de la hiérarchie des classes lors de la fusion des parties communes des clones. La principale différence réside dans la granularité des composantes communes traitées. Casais déplace des variables et des méthodes complètes. Notre approche permet de fusionner des méthodes clones similaires sans nécessairement être identiques. Nous extrayons les éléments communs des méthodes clones et permettons leur fusions Autrement dit, nous restructurons les méthodes clones en des méthodes générales et paramétrisables, et nous inférons de nouvelles méthodes encapsulant les différences et le contexte d'exécution.

Chae et Kwon [12] étudient la cohésion des classes par l'analyse des relations, des définitions et des utilisations des variables d'une classe par les méthodes de la classe. Ils transforment la structure des classes en changeant les regroupements des méthodes et variables. La cohésion est utilisée comme critère de déplacement. Chae et Kwon, contrairement à Casais, transforment aussi l'intérieur de certaines méthodes. Ils remplacent en fait des séquences de manipulations d'attributs d'une classe par un appel à une méthode de cette même classe, qui sera en charge de l'ensemble des manipulations. Contrairement à notre approche, cette transformation n'a pas pour objectif la fusion de code commun, mais simplement le déplacement de certaines opérations d'une classe vers une autre plus appropriée, sans aucune autre modification.

Opdyke et Johnson [38] proposent l'ajout de classes abstraites dans les systèmes. Ces dernières regroupent les méthodes et variables communes à plusieurs classes existantes. Encore une fois, la différence principale avec nos recherches est que les méthodes communes remontées vers l'ancêtre doivent être identiques, tandis que nous fusionnons des méthodes comportant des différences. De plus, Opdyke et Jonhson créent de nouvelles classes abstraites, alors que nous réutilisons, si possible la hiérarchie des classes existantes, et créons de nouveaux ancêtres seulement lorsque nécessaire.

Johnson et Opdyke explorent aussi, dans [27], la restructuration de classes de la forme agrégation/composante. Autrement dit, ils étudient les déplacements de variables et de méthodes entre une classe et une de ses agrégées. Cela recoupe notre traitement du cas général de clonage, où nous créons une nouvelle composante, vers laquelle nous déplaçons la méthode générale obtenue de la fusion des clones. Deux principales différences distinguent nos travaux de ceux de Johnson et Opdyke. Tout d'abord, nous ne modifions nullement l'interface des composantes originales, et n'avons donc pas de réajustement à faire ailleurs que dans les méthodes directement transformées. Ensuite, nous utilisons la composante pour y fusionner plusieurs méthodes clones, et non pour simplement y déplacer intégralement des méthodes existant déjà.

7.1.2.3 Reconception incrémentielle

Finalement, l'une des différences majeures entre nos travaux et ceux précédemment mentionnés est que notre réingénierie est partielle. Elle n'affecte qu'une partie d'un système à la fois. Elle peut donc s'inscrire dans un processus de reconception incrémentielle.

D'après Olsem [37], la réingénierie incrémentielle a comme avantages:

- Des résultats plus immédiats et concrets;
- moins de risques et un meilleur recouvrement en cas d'erreur; et
- une meilleure flexibilité.

7.2 Identification de différences sémantiques

Lorsque nous comparons des méthodes clones, nous identifions les séquences de terminaux qui doivent être ajoutées, enlevées ou remplacées pour transformer l'un des fragments de code en l'autre. Par la suite, nous déterminons la signification de chaque différence en les projetant sur l'AST du programme. Nous distinguons alors entre les différences superficielles, celles qui n'affectent ni le comportement du fragment de code ni les valeurs qu'il produit (différences dans les noms de variables locales et des paramètres) et les autres.

Horwitz [23] prend une approche différente pour déterminer les différences sémantiques entre deux fragments de code, et surtout pour les distinguer des différences textuelles. L'approche est basée sur les graphes de représentation de programmes et sur un algorithme de partition. Cet algorithme groupe les composantes dont le comportement est identique dans les deux fragments de code (c'est-à-dire une même séquence de valeurs est soit produite, soit assignée). L'identification des différences est alors simple lorsque les correspondances des énoncés sont fournies par l'éditeur, mais possible même lorsqu'elles ne le sont pas.

La principale différence avec notre approche est l'identification de ce qui constitue une différence. Dans notre cas, nous partons des différences textuelles dont nous éliminons les superficielles. Dans l'approche de Horwitz, le comportement de chaque élément comparé est considéré. Autrement dit, si une modification engendre une séquence différente de valeurs vrai/faux au niveau d'une expression, cette expression sera considérée comme une différence même si elle correspond à une expression textuellement identique dans l'autre fragment. Cette divergence dans les approches est dûe à la manière dont la fusion des fragments de code est ensuite validée et effectuée. Comme nous encapsulons les différences, nous n'avons pas besoin de nous soucier de leurs répercussions sur le reste du programme. Horwitz et al. [24] effectuent une fusion directe sans modification du code source. Ils doivent donc considérer de telles différences.

Par contre, notre approche détermine comme différence même de simples modifications qui n'ont aucune répercussion sur le comportement d'un fragment de code. Par exemple, si la manière dont un calcul est effectué est changée (ex: $a = 3 + b$ devient $a = b + 3$), nous allons considérer le calcul comme une différence, ce qui n'est pas le cas pour l'approche de Horwitz et al.

7.3 Fusion de codes sources

Les recherches portant sur la fusion de fragments de code source s'intéressent surtout à la maintenance des programmes. Gallagher et Lyle [19,20] décomposent un programme en plusieurs parties dont ils contraignent les modifications afin que celles-ci n'affectent qu'une partie du système à la fois. Ils refusionnent le tout après modifications. Yang et al. [47] et Horwitz et al. [24] permettent au contraire des modifications indépendantes et arbitraires sur un même fragment de code, puis fusionnent ces modifications lorsque possible.

Plus précisément, Gallagher et Lyle [19,20] facilitent la maintenance en permettant la décomposition d'un programme en deux sous-programmes, dont le premier remplit une partie des spécifications et le second, l'autre partie. La décomposition est basée sur la technique de *slicing* et génère deux programmes partiellement indépendants. Il existe alors certains types de modification qui permettent de changer le premier sous-programme sans affecter l'autre. Le programmeur peut effectuer seulement ces types de modifications. Une fois les modifications effectuées, les deux sous-programmes sont fusionnés. Les contraintes sur les modifications sont telles que cette fusion est un simple entrelacement d'énoncés.

Cette technique unit donc deux parties d'un même programme pour constituer une composante avec un comportement unique. Notre fusion des parties communes des clones fusionne, au contraire, des éléments communs de composantes indépendantes, tout en gardant le comportement particulier de chacune.

Par ailleurs, les contraintes sur les modifications assurent une fusion directe. Dans notre cas, il n'y a aucun contrôle au moment de la création des fragments de code. Toute l'analyse est donc faite pendant la fusion.

Horwitz et al. [24] déterminent s'il est possible de créer une version du code source d'une composante accommodant le comportement d'une version de base de la composante, et de deux variantes résultant d'activités de modifications indépendantes de la version de base. Lorsque cela est possible, ils créent le nouveau code source unique. Le code source final est le résultat de la fusion de trois *slices* (tranche) soit de la *slice* (tranche) qui reflète le comportement de la version de base préservé dans les deux variantes, ainsi que le comportement unique de chaque variante par rapport à la version de base.

Yang et al. [47] ont amélioré l'approche de Horwitz et al. quant à la définition d'une modification du code source. Ils distinguent entre les modifications qui préservent la sémantique et celles qui ne la préservent pas, permettant ainsi la fusion dans un plus grand nombre de cas.

Encore une fois, ces approches partent de plusieurs versions d'une même composante, tandis que nous partons de composantes distinctes qui partagent une partie de leur code source. Les approches décrites plus haut produisent donc une composante à comportement unique, tandis que nous produisons une composante dont le comportement est paramétrisable pour refléter les comportements des fragments de code originaux.

Par ailleurs, nous pouvons fusionner les parties communes de n'importe quels fragments de code, tandis que les autres approches ne fusionnent que des fragments qui ne présentent pas de conflits quant à leur comportement.

D'un autre côté, nous paramétrisons des différences même lorsque ces dernières ne présentent pas de conflit et pourraient être directement intégrées l'une dans l'autre.

7.4 Patrons de conception

Une partie de notre étude s'intéresse à la transformation des méthodes clones en des composantes indépendantes, paramétrisables et réutilisables. Pour ce faire, nous nous basons sur les patrons de conception *Strategy, Flyweight et Template method*. Dans cette section, nous comparons cette facette de notre projet aux autres recherches portant sur les patrons de conception.

La plupart des recherches sur les patrons de conception [2,3,9,29,41,44] s'intéressent à l'identification de ces derniers dans des systèmes orientés-objet. Pour ce faire, les approches extraient certaines structures explicitement présentes dans le code source comme des relations d'héritage ou d'agrégation, des passages d'objets en paramètres, et d'autres. L'objectif est de détecter les patrons de conception implantés dans le système par les concepteurs et les programmeurs. Notre approche, au contraire, infère des patrons de conception à partir de relations implicites, les relations de clonage. Il s'agit non seulement de patrons de conception qui n'existent pas encore dans le système, mais de patrons auxquels les concepteurs n'ont peut-être même pas songé. De plus, notre approche n'est pas passive et ne fait pas seulement déterminer la position des patrons dans le système; elle transforme ce dernier pour faire apparaître les patrons inférés.

Dans une autre ligne de recherche, Lano et Malik [31]utilisent les patrons de conception pour transformer du code procédural en OO. Ils identifient les patrons de conception procéduraux directement dans le code source, puis ils les font correspondre à des patrons de conception OO. Par exemple, une séquence de SI-SINON permettant le choix entre plusieurs algorithmes pourra être remplacée par une structure de type *Strategy*. Cela ressemble à notre approche d'inférence de patrons de conception dans un système, mais avec plusieurs différences majeures. En premier lieu, l'approche de Lano et Malik est, pour le moment, seulement théorique. De plus, notre approche infère des patrons à partir de l'information de clonage, tandis que la leur se base sur la présence de patrons procéduraux.

Finalement, Cinnéide et al. [16] introduisent des patrons de conception dans des systèmes qui n'en contiennent pas. Cette approche est complémentaire à la nôtre. Au lieu de conseiller où introduire des patrons de conception et lesquels introduire, les auteurs automatisent les transformations en laissant le programmeur déterminer quel patron doit être inséré et où il doit l'être. De plus, nous insérons des patrons pour remplacer des structures que l'ont peut considérer comme inappropriées, tandis que Cinnéide et al. remplacent des structures équivalentes à des patrons par les patrons eux-mêmes.

CONCLUSION

Dans ce mémoire, nous avons présenté les résultats d'une recherche portant sur l'utilisation des informations de clonage pour la reconception de systèmes orientés-objet. Nous avons présenté une nouvelle taxinomie des clones permettant l'étude du phénomène de clonage dans un système, du point de vue de la reconception. Nous avons aussi présenté un nouvel algorithme, développé pour permettre d'extraire les différences entre les méthodes clones tout en leur donnant la signification des composantes du langage de programmation auxquelles elles correspondent. Nous avons aussi présenté plusieurs possibilités de reconception que nous avons appliquées à JDK, un système réel de grande taille. Ces reconceptions permettent d'éliminer les duplications en factorisant les parties communes des clones. Elles permettent aussi de rendre le partage des fonctionnalités clonées explicites.

La reconception automatique basée sur l'analyse des clones est néanmoins limitée par la complexité d'analyse nécessaire à la couverture de tous les cas de clonage. Elle n'est pas non plus flexible, en ce qu'elle ne permet pas à l'utilisateur de modifier la reconception selon les cas particuliers qui peuvent se présenter. Cela est un problème, puisque les approches basées sur les patrons de conception, bien qu'applicables dans tous les cas, accroissent la quantité de codes sources et ne devraient donc pas nécessairement être toujours utilisées.

Dans les travaux futurs, il serait intéressant d'étudier la reconception interactive où un logiciel pourrait calculer et déterminer toutes les informations nécessaires à la reconception. Il pourrait aussi déterminer la liste des actions de reconception, mais il laisserait l'utilisateur choisir les types de reconceptions et effectuer les actions avec des outils spécialisés.

BIBLIOGRAPHIE

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley, 1988.

[2] G. Antoniol, R.Fiutem, and L.Cristoforetti. Design pattern recovery in object-oriented software. In *Proceedings of the 6th International Workshop on Program Comprehension. IWPC'98.* IEEE Computer Society Press, 1998.

[3] G. Antoniol, R.Fiutem, and L.Cristoforetti. Using metrics to identify design patterns in object-oriented software. In *Fifth International Software Metrics Symposium.* IEEE Computer Society Press, Nov. 1998.

[4] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the 2nd Working Conference on Reverse Engineering.* IEEE Computer Society Press, July 1995.

[5] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance 1999*, pages 368–377. IEEE Computer Society Press, 1998.

[6] P. L. Bergstein. Object-preserving class transformations. *SIGPLAN Notices*, 26(11):299–313, 1991.

[7] P. L. Bergstein. Maintenance of object-oriented systems during structural evolution. *Theory and Practice of Object-Oriented Systems*, 3(3):1–28, 1997.

[8] T. J. Biggerstaff. Design recovery for maintenance and reuse. *IEEE Computer*, 22(7):36–49, July 1989.

[9] K. Brown. Design reverse-engineering and automated design pattern detection in Smalltalk.

. http://www2.ncsu.edu/eos/info/tasug/kbrown/thesis2.htm.

[10] G. Canfora, A. Cimitile, and A. DeLucca. Software salvaging based on conditions. In *Proceedings of the International Conference on Software Maintenance 1994*, pages 424–433. IEEE Computer Society Press, 1994.

[11] E. Casais. An incremental class reorganization approach. In *Proceedings of ECOOP'92, European Conference on Object-Oriented Programming*, pages 114–132, 1992.

[12] H. S. Chae and Y. R. Kwon. Assessing and restructuring of classes based on cohesion. In *Proceedings of the Asian-Pacific Conference on Software Engineering*, pages 76–82, 1996.

[13] E. J. Chikofsky and J. H. Cross II. Reverse engineering and design recovery. *IEEE Software*, 7(1):13–17, Jan. 1990.

[14] S. C. Choi and W. Scacchi. Extracting and restructuring the design of large systems. *IEEE Software*, 7(1):66–71, Jan. 1990.

[15] W. Chu and S. Patel. Software restructuring by enforcing localization and information hiding. In *Proceedings of the International Conference on Software Maintenance 1992*, pages 165–172, 1992.

[16] M. Ó. Cinnéide and P. Nixon. A methodology for the automated introduction of design patterns. pages 463–472, 1999.

[17] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. pages 109–118, 1999.

[18] E. Gagnon, Sable Research Group, School of Computer Science, McGill University. Sablecc 2.5: Object-oriented compiler framework. http://www.sable.mcgill.ca/sablecc/.

[19] K. Gallagher. Conditions to assure semantically consistent merges in linear time. 1991.

[20] K. Gallagher. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, 1991.

[21] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley, 1997.

[22] D. R. Harris, H. B. Reubenstein, and A. S. Yeh. Reverse engineering to the architectural level. In *Proceedings of the 17th International Conference on Software Engineering*. IEEE Computer Society Press, Apr. 1995.

[23] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proceeding of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, 1990.

[24] S. Horwitz, J. Prins, and T. Reps. Integrating non-interfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, 1989.

[25] J.B.Wordsworth. *Software development with Z*. Addison-Wesley, 1992.

[26] J. H. Johnson. Identifying redundancy in source code using fingerprints. *CASCON'93*, pages 171–183, October 1993.

[27] R. E. Johnson and W. F. Opdyke. Refactoring and aggregation. ftp://st.cs.uiuc.edu/pub/papers/refactoring/refactor-aggregation.ps.

[28] K. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Journal of Automated Software Engineering*, 3:77–108, March 1996.

[29] C. Krämer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Proceedings of the 3rd Working Conference on Reverse Engineering*. IEEE Computer Society Press, Nov. 1996.

[30] B. Lagüe, D. Proulx, E. Merlo, J. Mayrand, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Proceedings of the International Conference on Software Maintenance 1997*, pages 314–321. IEEE Computer Society Press, 1997.

[31] K. Lano and N. Malik. Reengineering legacy applications using design patterns. In *Proceeding of the 8th IEEE International Workshop on Software Technology and Engineering Practice*, pages 326–338, 1997.

[32] MageLang Institute. Antlr 2.2.3.: Predicated-ll(k) parser generator. http://www.antlr.org.

[33] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the International Conference on Software Maintenance 1996*, pages 244–253. IEEE Computer Society Press, 1996.

[34] H. Müller. Understanding software systems using reverse engineering technology perspectives from the Rigi Project. *CASCON'93*, pages 217–226, October 1993.

[35] P. Newcomb and P. Scott. Requirements for advanced year 2000 maintenance tools. *IEEE Computer*, pages 52–57, March 1997.

[36] J. Ning, A. Engberts, and W. Kozaczynski. Automated support for legacy code understanding. *Communications of the ACM*, 37(5):50–57, 1994.

[37] M. Olsem. An incremental approach to software systems. *Journal of Software Maintenance: Research and Practice*, 10(3):181–202, 1998.

[38] W. F. Opdyke and R. E. Johnson. Creating abstract superclasses by refactoring. In *Proceedings of CSC'93: The ACM 1993 Computer Science Conference*, pages 66–73, 1993.

[39] J.-F. Patenaude, E. Merlo, M. Dagenais, and B. Lague. Extending software quality assessment techniques to Java systems. In *Proceedings of the 7th International Workshop on Program Comprehension. IWPC'99*. IEEE Computer Society Press, 1999.

[40] S. Pidaparthi and G. Cysewski. Case study in migration to object-oriented system structure using design transformation methods. In *1st Euromicro Working Conference on Software Maintenance and Reengineering CSMR 97*. IEEE Computer Society Press, Mar. 1997.

[41] J. Seemann and J. Wolff. Pattern-based design recovery of Java software. *Software engineering notes*, 23, 1998.

[42] Sun Microsystems Inc. Jdk 1.1.5.: Java development kit.

[43] Sun Microsystems Inc. Swing component set 1.0.2. http://www.javasoft.com/products/jfc/tsc/.

[44] P. Tonella and G. Antoniol. Object-oriented design pattern inference. pages 230–238, 1999.

[45] R. Tschalaer. Httpclient 0.3.:http client library. http://www.innovation.ch/java/HTTPClient.

[46] L. Wills. Automated program recognition by graph parsing. MIT Technical Report 1358, MIT, AI Laboratory, 1993.

[47] W. Yang, S. Horwitz, and T. Reps. A program integration algorithm that accomodates semantics preserving transformations. *ACM Transactions on Software Engineering and Methodology*, 1(3):310–354, 1992.

[48] K. Yasumatsu. Kfc 1.0 beta: Kazuki yasumatsu's foundation classes. http://ring.aist.go.jp/openlab/kyasu/.

Annexe I

Exemple de clones

Cette annexe présente le code source avant reconception de deux méthodes clones extraites de JDK 1.1.5. Les clones sont les méthodes *setBit()* et *flipBit()* de la classe java.math.BigInteger. Afin de rester clairs, seuls les éléments pertinents de la classe sont présentés ci-dessous.

```
package java.math;

import java.util.Random;


public class BigInteger extends Number {


  ...

  public BigInteger setBit(int n) throws ArithmeticException {

    if (n<0)

      throw new ArithmeticException("Negative bit address");

    int byteNum = n/8;

    byte[] result = new byte[Math.max(byteLength(), byteNum+2)];

    for (int i=0; i<result.length; i++)

      result[result.length-i-1] = getByte(i);

    result[result.length-byteNum-1] |= (1 << (n%8));
```

```
  return valueOf(result);
}
...
public BigInteger flipBit(int n) throws ArithmeticException {
  if (n<0)
    throw new ArithmeticException("Negative bit address");
  int byteNum = n/8;
  byte[] result = new byte[Math.max(byteLength(), byteNum+2)];
  for (int i=0; i<result.length; i++)
    result[result.length-i-1] = getByte(i);
  result[result.length-byteNum-1] ^= (1 << (n%8));
  return valueOf(result);
}
...
private int byteLength() {
  return bitLength()/8 + 1;
}
...
private byte getByte(int n) {
  if (n >= magnitude.length)
    return signByte();
  byte magByte = magnitude[magnitude.length-n-1];
```

```
    return (byte) (signum >= 0 ? magByte :
                   (n <= firstNonzeroByteNum() ? -magByte : ~magByte));
  }
}
```

Annexe II

Reconception avec ***Strategy***

Cette annexe présente un exemple de reconception avec le patron de conception *Strategy*. Il s'agit du code source après transformation des clones présentés dans l'annexe I.

II.1 Composante générale

Le code source de la nouvelle composante configurable est le suivant:

```
package newH;
import java.util.Random;
import java.math.*;
public class BI_setBit_1_Handler {
  private newI.BI_setBit_1_InterfaceData _stratData;
  public BI_setBit_1_Handler(newI.BI_setBit_1_InterfaceData stratData) {
    _stratData = stratData;
  }
  public BigInteger setBit( int n, newI.BI_setBit_1_InterfaceDiffs stratDiffs)
                           throws ArithmeticException {
    if( n < 0)
      throw new ArithmeticException( "Negative bit address");
```

```
    int byteNum = n / 8;
    byte [ ] result = new byte [Math.max(_stratData.byteLength(),byteNum+2)];
    for( int i = 0; i < result.length; i ++)
      result [ result.length - i - 1 ] = _stratData.getByte( i);
    stratDiffs.StatementExpression_complexDiff0( result, byteNum, n);
    return java.math.BigInteger.valueOf( result);
 }
}
```

II.2 Classe originale transformée

La classe originale contient les méthodes clones transformées, une référence à une instance de la composante générale, et les classes contenant les méthodes reflétant les comportements particuliers de chaque clone original. Le nouveau code source de la classe est le suivant:

```
package java.math;
import java.util.Random;
public class BigInteger extends Number
                        implements newI.BI_setBit_1_InterfaceData {
  ...
  public BigInteger setBit( int n) throws ArithmeticException {
    return BI_setBit_1_handler.setBit( n, BI_setBit_1_BI_setBit_strategy);
  }
```

```
...
public BigInteger flipBit( int n) throws ArithmeticException {
  return BI_setBit_1_handler.setBit( n, BI_setBit_1_BI_flipBit_strategy);
}
...
public int byteLength() {
  return bitLength() / 8 + 1;
}
...
private byte getByte(int n) {
  if (n >= magnitude.length)
    return signByte();
  byte magByte = magnitude[magnitude.length-n-1];
  return (byte) (signum >= 0 ? magByte :
                (n <= firstNonzeroByteNum() ? -magByte : ~magByte));
}
...
private newH.BI_setBit_1_Handler BI_setBit_1_handler
        = new newH.BI_setBit_1_Handler( this);
private BI_setBit_1_BI_setBit BI_setBit_1_BI_setBit_strategy
        = new BI_setBit_1_BI_setBit();
private BI_setBit_1_BI_flipBit BI_setBit_1_BI_flipBit_strategy
```

```
        = new BI_setBit_1_BI_flipBit();
  class BI_setBit_1_BI_setBit implements newI.BI_setBit_1_InterfaceDiffs {
    public BI_setBit_1_BI_setBit() {
    }
    public void StatementExpression_complexDiff0( byte[] result,
                                                 int byteNum, int n) {
      result [ result.length - byteNum - 1 ] |=( 1 <<( n % 8));
    }
  }
  class BI_setBit_1_BI_flipBit implements newI.BI_setBit_1_InterfaceDiffs {
    public BI_setBit_1_BI_flipBit() {
    }
    public void StatementExpression_complexDiff0( byte[] result,
                                                 int byteNum, int n) {
      result [ result.length - byteNum - 1 ] ^=( 1 <<( n % 8));
    }
  }
}
```

II.3 Interfaces

Deux interfaces sont créées lors de cette transformation: une interface pour les différences, et une pour le contexte. Le code source de l'interface de contexte est le suivant:

```
package newI;

import java.util.Random;

import java.math.*;

public interface BI_setBit_1_InterfaceData  {

  public int byteLength();

  public byte getByte(int n);

}
```

Le code source de l'interface pour les différences est le suivant:

```
package newI;

import java.util.Random;

import java.math.*;

public interface BI_setBit_1_InterfaceDiffs  {

  public void StatementExpression_complexDiff0( byte[] result,

                                                   int byteNum, int n);

}
```

Annexe III

Reconception avec *Template method*

Cette annexe présente un exemple de reconception avec le patron de conception *Template method*. Il s'agit du code source après transformation des clones présentés dans l'annexe I.

III.1 Composante générale

```
package newH;
import java.util.Random;
import java.math.*;
public abstract class BI_setBit_1_Handler {
  private newI.BI_setBit_1_InterfaceData _stratData;
  public BI_setBit_1_Handler(newI.BI_setBit_1_InterfaceData stratData) {
    _stratData = stratData;
  }
  abstract public void StatementExpression_complexDiff0( byte[] result,
                                                          int byteNum, int n);
  public BigInteger setBit( int n) throws ArithmeticException {
    if( n < 0)
      throw new ArithmeticException( "Negative bit address");
```

```
    int byteNum = n / 8;
    byte [ ] result = new byte [Math.max(_stratData.byteLength(),byteNum+2)];
    for( int i = 0; i < result.length; i ++)
      result [ result.length - i - 1 ] = _stratData.getByte( i);
    StatementExpression_complexDiff0( result, byteNum, n);
    return java.math.BigInteger.valueOf( result);
  }
}
```

III.2 Comportements particuliers

La classe représentant le comportement particulier de la méthode *setBit()* est la suivante:

```
package newH;
import java.util.Random;
import java.math.*;
public class BI_setBit_1_BI_setBit extends newH.BI_setBit_1_Handler {
  private newI.BI_setBit_1_BI_setBit_IData _stratData;
  public BI_setBit_1_BI_setBit( newI.BI_setBit_1_BI_setBit_IData stratData) {
    super( stratData);
    _stratData = stratData;
  }
  public void StatementExpression_complexDiff0( byte[] result, int byteNum, int n) {
```

```
    result [ result.length - byteNum - 1 ] |=( 1 <<( n % 8));
  }
}
```

La classe représentant le comportement particulier de la méthode *flipBit()* est la suivante:

```
package newH;
import java.util.Random;
import java.math.*;
public class BI_setBit_1_BI_flipBit extends newH.BI_setBit_1_Handler {
  private newI.BI_setBit_1_BI_flipBit_IData _stratData;
  public BI_setBit_1_BI_flipBit( newI.BI_setBit_1_BI_flipBit_IData stratData) {
    super( stratData);
    _stratData = stratData;
  }
  public void StatementExpression_complexDiff0( byte[] result,
                                                int byteNum, int n) {
    result [ result.length - byteNum - 1 ] ^=( 1 <<( n % 8));
  }
}
```

III.3 Classe originale transformée

Le nouveau code source de la classe BigInteger est le suivant:

```
package java.math;
import java.util.Random;
public class BigInteger extends Number
       implements newI.BI_setBit_1_BI_setBit_IData,
                  newI.BI_setBit_1_BI_flipBit_IData {
  ...
  public BigInteger setBit( int n) throws ArithmeticException {
    return BI_setBit_1_BI_setBithandler.setBit( n);
  }
  ...
  public BigInteger flipBit( int n) throws ArithmeticException {
    return BI_setBit_1_BI_flipBithandler.setBit( n);
  }
  ...
  public int byteLength() {
    return bitLength() / 8 + 1;
  }
  ...
  public byte getByte( int n) {
    if( n >= magnitude.length)
      return signByte();
    byte magByte = magnitude [ magnitude.length - n - 1 ];
```

```
    return( byte)( signum >= 0 ? magByte :
                  ( n <= firstNonzeroByteNum() ? - magByte : ~ magByte));
  }
  ...
  private newH.BI_setBit_1_BI_setBit BI_setBit_1_BI_setBithandler
          = new newH.BI_setBit_1_BI_setBit( this);
  private newH.BI_setBit_1_BI_flipBit BI_setBit_1_BI_flipBithandler
          = new newH.BI_setBit_1_BI_flipBit( this);
}
```

III.4 Interfaces

L'interface contenant les signatures des opérations de contexte communes aux deux clones est la suivante:

```
package newI;
import java.util.Random;
import java.math.*;
public interface BI_setBit_1_InterfaceData  {
  public int byteLength();
  public byte getByte(int n);
}
```

L'interface contenant les signatures des opérations de contexte uniques à la méthode *setBit()* est vide puisqu'aucune opération de ce genre n'existe.

```
package newI;

import java.util.Random;

import java.math.*;

public interface BI_setBit_1_BI_setBit_IData

       extends newI.BI_setBit_1_InterfaceData {

}
```

Il en est de même pour l'interface des opérations de contexte uniques à *flipBit()*

```
package newI;

import java.util.Random;

import java.math.*;

public interface BI_setBit_1_BI_flipBit_IData

       extends newI.BI_setBit_1_InterfaceData {

}
```