Vrije Universiteit Brussel - Belgium
Faculty of Sciences
In Collaboration with Ecole des Mines de Nantes - France
2004

SOURCE CODE MINING FOR CODE DUPLICATION REFACTORINGS WITH FORMAL CONCEPT ANALYSIS

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange)

By: Roberto A. Riquelme Torres

Promoter: Prof. Dr. Theo D'Hondt (Vrije Universiteit Brussel)
Co-Promotor: Prof. Dr. Kim Mens (Université Catholique de Louvain) and
Dr. Tom Tourwé (Centrum voor Wiskunde en Informatica)

Abstract

Code duplication exists for many reasons, although it is known to be a bad practice. It may save costs on the development phase, but it surely has a higher cost of maintenance. What can we do to detect and eliminate those problems?

This document shows how to use the mathematical foundation of Formal Concept Analysis to detect code duplication.

Formal Concept Analysis allows us to identify concepts, or groupings of elements, that have common properties. We work with a particular kind of elements and properties: Methods containing similar Regular Parse Tree Expressions. We classify these concepts in a certain way to reveal candidates for code duplication refactorings and show them in an ordered way, so that the user can decide to apply the refactoring when needed.

This work is done as an extension of the existing Delfstof framework for VisualWorks Smalltalk, which is currently being developed by researchers at CWI [1] and UCL.[2]

[1] Centrum voor Wiskunde en Informatica, Amsterdam, Netherlands. http://www.cwi.nl
[2] Université Catholique de Louvain, Louvain-la-Neuve, Belgium. http://www.ucl.ac.be

Acknowledgments

There's so many people that I would like to thank for this wonderful, yet stressful year.

First of all, my family, specially my parents Roberto and Silvia who always believed in me, and have always supported me in all my life projects. Also to my best friend Claudio, who always gave me the energy to continue working.

I would like to thank also my advisors Kim Mens and Tom Tourwé, and specially Andy for all their help in this research, and all the people at PROG, who always made me feel at home.

And last, but not least, I would like to thank my girlfriend Karen, who has always supported me, giving me the energy and understanding needed to go along with this project.

Thank you all very much!

...we never end implementing our dreams...
Jo. Piquer - DCC - Universidad de Chile

Brussels, Belgium
August 23, 2004

Roberto Riquelme T.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

One of the factors that severely complicates the maintenance and evolution of software systems is *code duplication* or *cloning*. It is known to be a bad programming practice, and we should try to avoid it. Unfortunately this is not always a possibility. Instead, techniques are necessary to remove this duplication from existing code.

The best way of fixing this problem is proposing refactorings to remove the duplications. But, in software systems, it is difficult to detect those duplications, but yet more difficult is to remove them properly.

Our goal is to detect these duplications, and propose the possible refactorings to the user, and let him or her decide to apply the modifications.

1.2 Duplicated Code

The first question we should answer is: *What is duplicated code?*. Duplicated code is the result of copying a code fragment or the structure of a piece of code, and possibly, performing some minor changes on that copy. Those minor changes can be, for example: renaming variables, changing the order of execution of some statements or adding comments. In this way the developer may reuse an existing structure in the program or reuse already existing behavior.

Code duplication is also known as *cloning* [KG03]. This term raises the following question: which kinds of copies are considered clones?. In most of the cases we cannot consider the whole code fragments as a duplicate, but only those lines that match. We want both fragments to have a similar structure or have a sub-part of the code fragment in common.

Each detection technique is different. One of them considers that a piece of code is duplicated if they only differ in 10% of the lines. If two fragments of code are considered clones, we say that a *cloning relation* exists between them [DRD99].

1.3 Why does Code Duplication exist?

Code duplication is a widespread technique, although it is known to be a bad practice. But why does it still exist? There is not a unique answer to this question.

Code duplication generally happens to students or beginner programmers that are not used to the Object Oriented paradigm and do not know how to use polymorphism and other Object Oriented concepts in an optimal way. For instance, we may have duplicated code in a class hierarchy, because we did not know that the superclass already implemented something similar, or that one of the subclasses already has it implemented, and we may move it to the superclass.

Cloning also occurs with more experienced programmers, when they quickly need to implement some functionality, but do not consider creating the proper abstractions, or making use of functionalities already implemented.

Although code duplication is a bad practice, it still occurs in existing software development.

1.4 Impact of Duplicated Code

Code duplication can have a severe impact on the quality, reusability and maintainability of a software system.

When a programmer copies code which contains errors, the copy will also include the errors inserted in the code. As mentioned before, in a lot of cases, only the structure of the duplicated fragment can be re-used. The developer has to adapt the duplicated code. This process can be error prone and may introduce new bugs in the software.

Code duplication also indicates bad design, lack of a good inheritance structure or abstraction. This makes it very difficult to reuse part of the implementation in future projects. It also has an impact on the maintainability of the piece of software.

Cloning makes it more difficult to implement new functionalities in the system, or change existing ones, because it takes extra time to comprehend already existing implementation and concerns which has to be adapted and which are not necessarily implemented at one location in the code.

Although code duplication seems to simplify the initial development of a piece of software it has a large impact on the quality of the developed software. It increases the maintenance cost which is already the biggest cost factor. Code duplication handicaps the software engineer as well as the maintainer of the software.

1.5 Position of this Document

In this thesis we propose the use of the mathematical technique of Formal Concept Analysis in order to detect the *Code Duplication* bad smell using a special kind of Abstract

Syntax Tree, and propose a possible refactoring to eliminate the cloning from the code.

The idea of applying Formal Concept Analysis to source code is not new. Our contribution lies in the use of regular parse-tree expressions as properties, which will be used as input of the FCA algorithm, and classifying the discovered concepts in order to propose refactorings which may eliminate the duplicated code. In fact, this work is done as an extension of the Delfstof framework for VisualWorks Smalltalk currently under development by researchers at UCL and CWI.

Using this technique, we mine a system's source code to detect code duplication in a way that is independent of the actual system being analyzed. Depending on the characteristics of each code duplication, particular refactorings can be proposed and applied to eliminate this bad practice.

Although this approach can be improved in many ways, it allows us to mine Smalltalk source code to detect code duplication and discover the refactoring possibilities available.

1.6 Overview

The remainder of this thesis is structured as follows. Chapter 2 of this document gives us the general idea of the tools and theory behind refactorings, a brief introduction to Formal Concept Analysis, the Delfstof framework and our tree representation known as Regular Parse-Tree Expressions. In chapter 3 we explain our approach and how we used formal concept analysis to mine the source code to detect code duplication. Some experiments with this technique and their results are shown in chapter 4. We conclude this document in chapter 5, proposing ideas for future work.

Chapter 2

Preliminaries

2.1 Refactoring

2.1.1 What is Refactoring?

Refactoring [FBB+99, MT04b] is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written.

This last sentence seems odd at first glance. Programmers are used to "if it works, don't fix it". In our current understanding of software development we believe that we design and then we code. A good design comes first, and the coding comes second. Over time the code will be modified, and the integrity of the system, its structure according to that design, gradually fades. The code slowly sinks from engineering to hacking.

Refactoring is the opposite of this practice. With refactoring you can take a bad design, and rework it into well-designed code. Each step is simple, even simplistic. You move a field from one class to another, pull some code out of a method to make into its own method, and push some code up or down a hierarchy. Yet the cumulative effect of these small changes can radically improve the design. It is the exact reverse of the normal notion of software decay.

2.1.2 Brief History

Refactoring was invented in Smalltalk circles in the 1980's, and the book by Martin Fowler [FBB+99] is the classic reference. Although refactoring code has been done informally for years, William F. Opdyke's 1992 PhD dissertation [Opd92] is the first known paper to specifically examine refactoring.

Refactoring is such an important concept that it has been identified as one of the most

important software innovations by David Wheeler [1]

2.1.3 Unit Testing

Unit testing is a method of testing the correctness of a particular module of source code, generally all methods of a class that do non-trivial things. It is not component testing, which is interested whether the application behaves generally the way it was intended to.

Because the changes refactoring introduces to the source code by definition do not affect its behavior, the tests are used to continually ensure the proper working of the target program.

2.1.4 Bad Smells in Code

Refactoring is not only about esthetics or beautiful code. We have to be clear on *when* refactorings should be applied. In [FBB+99], Martin Fowler and Kent Beck enumerate 22 situations where one should refactor. The most important and common ones are:

Duplicated Code. Having the same code in different places is the most common error found in applications. We can use *Extract Method* to unify the code in one place. If however the duplicated code is in two sibling subclasses, we can use *Extract Method* and *Pull Up Method*. In cases when the code in the extracted method has nothing to do with the class, you may use *Extract Class*.

Long Method and Large Class. Because complexity increases with every line we add to a method, the code will be easier to test and maintain if we consider using *Extract Method*. Similarly, if a class is too large, first avoid duplicated code. If the class has too many instance variables, *Extract Class* can be used to group variables belonging together in their own class.

Long Parameter List. Long parameter lists are hard to understand and usually deliver too much (and not necessarily the right) information to the method. With objects, we can always ask another object to get the needed information. Using *Replace Parameter with Method* may be a solution to separate a parameter logically from the others. Alternatively we could pass a special parameter object (a Value Object or Transfer Object).

Feature Envy. A method more interested in another class that the one it is in may indicate feature envy. By using *Move Method* we can move the method to the class where the most data used by the method is.

Data Clumps. It is frequent that the same set of variables are spotted together in various places of the code. Use *Extract Class* to group together those variables in their own class.

[1] http://www.dwheeler.com/innovation/innovation.html

Speculative Generality. Sometimes handling for special cases are implemented, but never used. The results are methods which are only called by their own tests and thus only add to the complexity of maintenance and understanding. Replace unnecessary delegation with the code of the delegate (*Inline Class*), *Collapse the Hierarchy* if there are abstract classes not doing much. Purge unused parameters, rename methods, fields and classes to achieve sensible names.

Temporary Field. Temporary variables can be replaced with the expression used to set the variable (*Inline Temp*). We could also use *Extract Method* and move the expression to a new method, the replace all references to the temporary variable with a call to the new method *Replace Temp with Query*. If however we have a very complex expression, having one or multiple temporary variables is sensible to ease understanding and maintenance.

Message Chains. If we have code like:

```
foo.getBar().getFoo().getThis().getThat().doSomething()
```

we have to consider accessing the target object directly.

Inappropriate Intimacy. Some classes know too much about other classes and should therefore be broken up (use *Move Method* and *Move Field*, or *Extract Class* to extract the common code into another class). Subclasses are an example for this misbehavior, because they always know too much about their parent. In this case, we can replace inheritance with delegation.

Comments. Comments are not evil by themselves, but are sometimes used as a deodorant to cover bad code. Often we can remove the comments after doing some refactorings, because they are not needed anymore. The code should be self-explaining. However, there are cases when we still should use comments: If we are unsure on what to do or to explain *why* we did something.

These are only the most important bad smells that we can find in source code. This gives us the idea of how badly the coding can be done. The best way to avoid refactorings is to correctly design and extend your software project, having in mind that changes will occur.

Planet	Size			Distance to the Sun		Has a moon	
	small	medium	large	near	far	moon	no moon
Mercury	x			x			x
Venus	x			x			x
Earth	x			x		x	
Mars	x			x		x	
Jupiter			x		x	x	
Saturn			x		x	x	
Uranus		x			x	x	
Neptune		x			x	x	
Pluto	x				x	x	

Table 2.1: Solar System example. Table T of binary relations

2.2 Formal Concept Analysis

This section provides an introduction to Formal Concept Analysis [GW99] theory by means of a practical example.

2.2.1 Overview

Formal Concept Analysis (FCA) [GW99] is a branch of lattice theory that allows us to identify significant groups of *elements* (denoted as *objects* in FCA literature) that have common *properties* (denoted as *attributes* in FCA literature)[2].

We demonstrate the concepts and theory of FCA by means of a simple example: the solar system. In this case, our elements will be the planets (*Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune, Pluto*) and we consider 7 possible characteristics that identify each one of the elements divided in 3 groups: size (*small, medium, large*); distance to the sun (*near, far*); and possession of moons (*moon, no moon*). Table 2.1 shows the relation between the planets and each property.

2.2.2 Context

In order to work, FCA needs a *context* setup. A **context** is defined as a triple containing a set of *elements (E)*, a set of *properties (P)* and a binary relation R between them (to check if an element does or does not have a certain property).

In table T, the first column (the names of the planets) is the finite set of *elements*, and the first row (the characteristics) is the finite set of *properties*. The binary relation R is usually represented as a table T. Here we see, for instance, that *Pluto* is a *small* planet,

[2]In this document we use the terms *element* and *property* instead of *object* and *attribute* to avoid the confusion of terms with object-oriented software development

Planet	Size			Distance to the Sun		Has a moon	
	small	medium	large	near	far	moon	no moon
Mercury	x			x			x
Venus	x			x			x
Earth	**X**			**X**		**X**	
Mars	**X**			**X**		**X**	
Jupiter			x		x	x	
Saturn			x		x	x	
Uranus		x			x	x	
Neptune		x			x	x	
Pluto	x				x	x	

Table 2.2: Solar System example. A concept example in Table T

top	({*Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune, Pluto*},∅)
c_{10}	({*Earth, Mars, Jupiter, Saturn, Uranus, Neptune, Pluto*},{*moon*})
c_9	({*Mercury, Venus, Earth, Mars, Pluto*},{*small*})
c_8	({*Jupiter, Saturn, Uranus, Neptune, Pluto*},{*far, moon*})
c_7	({*Earth, Mars, Pluto*},{*small, moon*})
c_6	({*Mercury, Venus, Earth, Mars*},{*small, near*})
c_5	({*Jupiter, Saturn*},{*large, far, moon*})
c_4	({*Uranus, Neptune*},{*medium, far, moon*})
c_3	({*Pluto*},{*small, far, moon*})
c_2	({*Earth, Mars*},{*small, near, moon*})
c_1	({*Mercury, Venus*},{*small, near, no moon*})
bottom	(∅,{*small, medium, large, near, far, moon, no moon*})

Table 2.3: Concepts of the Solar System example

is *far* from the sun and *has a moon* [3].

2.2.3 Concepts

A **concept** is defined as a pair of sets of elements and properties such that they have a maximal collection of elements sharing common properties. Graphically, this can be represented as a maximal rectangle in the cross-table T (2.1). For example ({*Earth, Mars*},{*small, near, moon*}) is a concept, as we can see in table 2.2, whereas ({*Earth, Mars*},{*small, near*}) is not a concept, because it may contain more elements for these properties, or more properties for these planets (as seen before).

Table 2.3 shows the complete list of concepts. It is important to note that concepts are invariant against row or column permutations in the cross-table T.

[3] in fact, it is called Charon

2.2.4 Concept Lattice

The set of all the concepts of a given context forms a *complete partial order*. With this partial order, we can define that a concept (X_0, Y_0) is a **subconcept** of concept (X_1, Y_1) if $X_0 \subseteq X_1$ (or, equivalently, $Y_0 \subseteq Y_1$)

For example: ({*Uranus, Neptune*},{*medium, far, moon*}) is a subconcept of ({*Jupiter, Saturn, Uranus, Neptune, Pluto*},{*far, moon*}).

If we order all the concepts this way, we build the *Concept Lattice*. Figure 2.1 shows the Concept Lattice for this example.

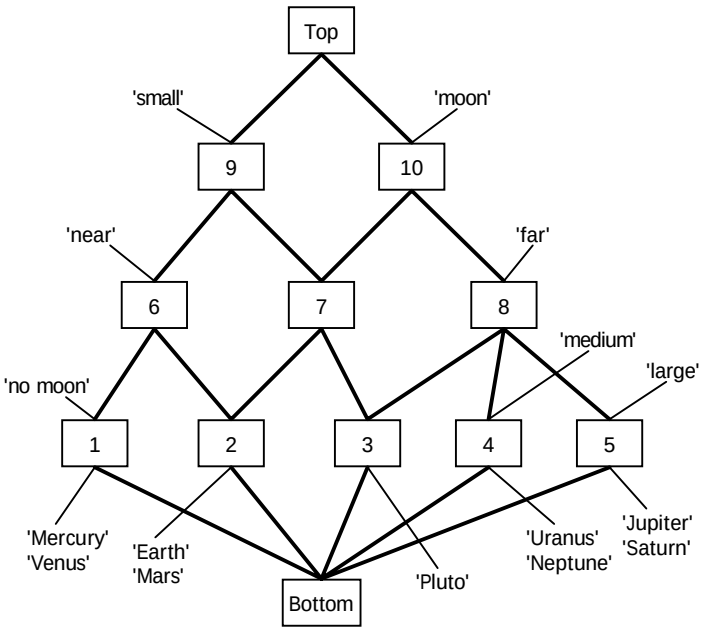

Figure 2.1: Concept Lattice of the Solar system example

Applying the concept-subconcept relation to all the concepts, we note that a most general (concept without any properties) and a most specific (concept with all attributes) concepts appear. These are called *top* and *bottom* concepts. The *top concept* represents the Properties that *all* elements have, and the *bottom concept* represents the Elements that have *all* the properties.

These superconcepts are generally empty. This means there is no element that has all the properties and no property that all elements have (depending on the characteristics chosen), or as we can see in table 2.1, in the case of *top concept*, there is no column with marks for all the elements and in the case of *bottom concept*, there is no row with marks for all the properties in the table.

2.3 Delfstof

2.3.1 Overview

Delfstof [4] [MT04a] is a framework for VisualWorks Smalltalk that allows the creation of tools to mine a system's source code automatically and efficiently for relevant concepts of interest, that are called *source-code regularities*. These regularities can be concerns addresses in the code, design patterns, programming idioms, conventions adopted and where and how they are implemented.

To realize this source code mining, Formal Concept Analysis is used. After executing the mining, the results are filtered, classified and combined to present them to the user in a more convenient way.

2.3.2 Mining for Source-Code Regularities

The mining is done following these steps:

1. Choose the elements and properties to compute the concept lattice
2. Compute the lattice
3. Filter irrelevant and redundant concepts
4. Classify the concepts
5. Combine and annotate concepts

The important contribution of this approach is the particular choice of elements, properties, filters and analyzers, and how they allow us to discover interesting regularities in source code, independent of the application.

Choose elements and properties

To detect regularities, source-code entities such as classes, methods and method parameters are chosen as the elements.

As properties, simple substrings of the names of the source-code entities are generated. Therefore, the concepts will group entities with similar names [5].

To limit the number of generated properties, not all the possible substrings are considered. The names of the classes, methods and parameters are split in substrings according

[4] Delfstof is a Dutch word which designates the result of a delving process. In English, the verb "to delve" means "to make careful investigation for facts, knowledge, etc.". Coincidentally, the pronunciation sounds like the English "delve stuff" which is what the framework does.

[5] this approach relies firmly on naming conventions

to the capital leters and other separators occurring in them. Also, substrings with minimal conceptual meaning are discarded, such as: '`with`', '`from`', '`the`', '`object`', as well as substrings that are too small (less than 3 characters). Colons, plurals and case difference are also ignored when comparing substrings. For example, the following code:

```
#unifyWithDelayedVariable: inEnv: myIndex: hisIndex: inSource:
```

the substrings generated are: '`unify`', '`delayed`', '`variable`', '`env`', '`index`', and '`source`'.

Compute the lattice

Applying the FCA algorithm to the elements and properties described before, generates a very large concept lattice which groups elements that have the same substrings in the names of their classes, methods and parameters.

Filter the concepts

The number of concepts found by FCA is of the same order of magnitude as the number of considered elements. Here the number of concepts is reduced eliminating redundancy and the irrelevant ones applying some *filters*.

The first one ignores all the concepts that have two or less elements (these concepts are too small and do not provide relevant information).

Another filter ignores all concepts that share only one property (i.e. substring). This filter discards some interesting concepts, but it eliminates many more irrelevant ones.

The third filter is more specific. It discards concepts that contain only classes (with similar name) in the same hierarchy. This filter is based on the convention that classes belonging to a same hierarchy, often have similar names.

Classifying the concepts

The concepts that remain after the filtering are rather unstructured. In this part, automatic reorganization of the concepts is done for easier understanding and interpretation. For better comprehension and visualization of the classifications, the remaining concept lattice is flattened and shown directly in the StarBrowser [WD03], as a tree structure, as we can see in figure 2.2.

Some groups of concepts are:

1. *Single class concepts*: groups concepts of which all elements are methods (or parameters of those methods) belonging to a single class;

2. *Hierarchy concepts*: groups classes, methods and parameters that belong to a same class hierarchy;

Case	#elements	#properties	#raw	#filtered	time (sec)
Delfstof	756 (135)	237	617	126	5
StarBrowser	731 (52)	352	740	115	7
SOUL	1469 (111)	434	1188	281	22
CodeCrawler	1370 (93)	477	1419	327	24
Refactoring Browser	4779 (271)	729	4179	1234	414

Table 2.4: Results of Delfstof applied to Smalltalk source code

3. *Crosscutting concepts*: groups concepts of at least two different class hierarchies (different from `Object`).

These classifications help us to have at first glance a general idea of the structure of the application analyzed.

Combining and annotating concepts

By organizing the concepts, the structure of the original lattice is lost. Since there is a lot of overlap between concepts that are nearby in the lattice, when reorganizing the concepts this may lead to redundancy among concepts that get classified into different classifications. That is why highly overlapping concepts are recombined into a single nested one.

In addition, automatic regrouping and annotation is done to present the classifications in a clearer way: different concepts related to the same class(es) are combined, methods are annotated to the classes they belong to, and concepts are annotated with their properties.

2.3.3 Test Cases and Results

This particular application of the framework was applied to five different cases to detect source code regularities: Delfstof, StarBrowser [WD03], SOUL [MBM03], CodeCrawler [LD03] and Refactoring Browser [RBJ97], getting the results summarized in table 2.4.

SOUL is an interpreter for a Prolog-like language. *Delfstof* is the framwork itself. *StarBrowser* and *Refactoring Browser* are advanced Smalltalk browsers and *CodeCrawler* is a language-independent reverse engineering tool, which combines metrics and software visualization.

The columns *#elements* gives the size of each case, i.e. the number of classes and methods of each case. The number in parenthesis is the number of classes.

The column *#properties* is the number of substrings generated from the elements.

The column *#raw* shows the total number of concepts discovered by FCA, and column *#filtered* shows the remaining concepts after applying the filters explained in 2.3.2.

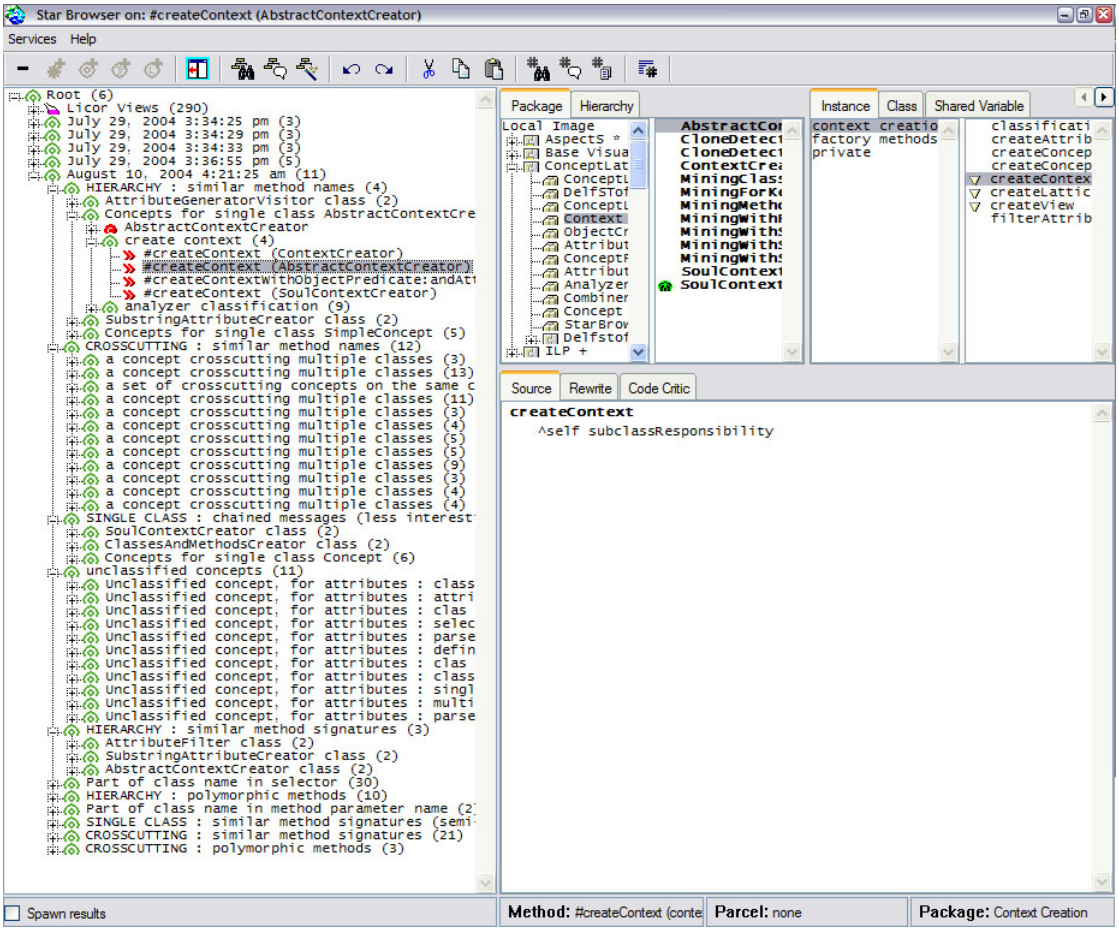

Figure 2.2: Results of Delfstof applied to Delfstof with Substrings

Results

Delfstof applied to the cases shown before, discovered *programming idioms*, a basic level of *code duplication* (and refactoring opportunities), and some *design patterns* [GHJV95].

The various programming idioms found were:

- **Accessing methods**, discovered thanks to the naming conventions. Although most of them were discarded due to the application of the first filter.
- **Polymorphic methods**, that have the same name, in different classes, but in the same hierarchy.
- **Chained messages**, group methods with its auxiliary methods in the same concept. These chains are recognized by FCA since the auxiliary methods often have a name that is similar to that of the originating method.
- **Delegating methods**, which delegate responsibility by calling a method with the same name, even in the same class.

Some code duplication was also detected: several concepts contained methods that not only have a similar name, but also a similar implementation. These are typical cases of copy & paste code reuse. The way a concept containing duplicated code is classified, can provide useful hints about which refactorings to apply. For example, for the concepts classified as a *single class concept*, the duplication occurred in a single class, and an *extract method* refactoring is appropriate; for the concepts classified in the *hierarchy concepts*, a combination of *extract method* and *pullup method* refactorings seems more suitable.

And finally, also thanks to naming conventions, some design patterns [GHJV95] were found. For example, the *Visitor* pattern uses the convention that each *visit* method defined by a *visitor* class encodes the name of the class being visited. As several other patterns use similar naming conventions, Delfstof also detected occurrences of the *Abstract Factory*, *Builder*, *Observer*, and *Decorator* design patterns.

2.4 Parse-Tree Expressions

The process whereby a computer compiler takes parts of human written source code and produces an internal structure that represents the correct parts of a program that can be executed is what we call *Parsing* [Ken03]. We can represent this parsing in infinite ways. In the following sections the most common will be discussed.

2.4.1 Parse Trees

A **Parse Tree** is a *grammatical structure* represented as a *tree data structure*.

The grammatical structure of a language [6] is a set of rules governing the use of the language. Specifically, programming languages are considered *formal grammars* which conform precisely to a grammar generated by a pushdown automaton with arbitrarily complex commands. Note that there are an infinite number of grammars for any single language and hence, every grammar will result in a different parse tree from a given input sentence because of all the intermediate rules.

A tree data structure [7]emulates a tree structure with a set of linked nodes. Each node has zero or more child nodes. The node of which a node is a child is called its parent node. A child has at most one parent; a node without a parent is called the root node. Nodes with no children are called leaf nodes.

Parse trees contain the full input stream (i.e. code and comments), and they can be navigated as a normal tree data structure to extract information about the source code.

Figure 2.3 is an example of a very simple grammar and its parse tree representation. And in figure 2.4 we have an example of a parse tree representation of piece of pseudo-code.

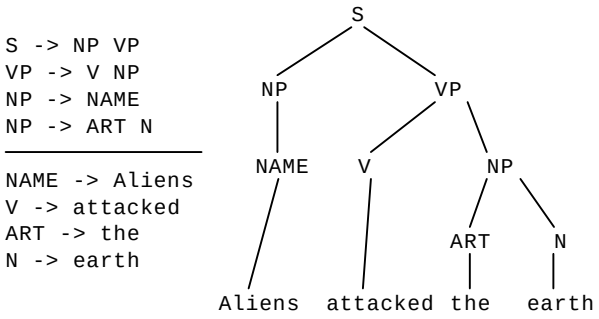

Figure 2.3: A very simple grammar and parse tree example

[6] Any language: human, non-human and/or formal
[7] In graph theory, an *acyclic graph*

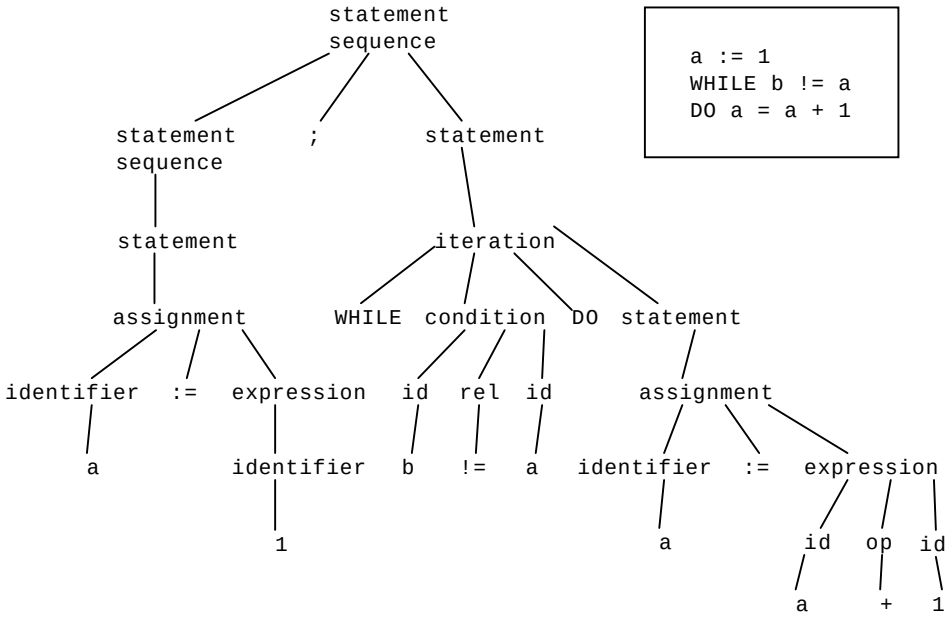

Figure 2.4: Pseudo-code Parse Tree Example

2.4.2 Abstract Syntax Trees

An *Abstract Syntax Tree* is a data structure representing the structure of a parsed sentence in some language, often used as a compiler or interpreter's internal representation of a computer program while it is being optimized and from which code generation is performed. The range of all possible such structures is described by the abstract syntax [BYM+98].

This kind of tree is a far superior intermediate form precisely because of its insensitivity to the grammar that produced it and because it highlights the structure of the language, not the grammar. For example, it does not contain layout information or code comments (as does the parse tree).

We can see the differences between these two parse trees in the example of figure 2.5.

2.4.3 Regular Parse-Tree Expressions

The *Regular Parse-Tree Expressions* are our own kind of Abstract Syntax Tree, to work with code duplication detection.

These *expressions* are an adaptation of the parse trees that Smalltalk's Refactoring Browser [RBJ97] generates to work with Smalltalk source code.

Using the visitor design pattern, we can create parse trees that accommodate to our needs. This personalization includes our own kinds of assignments, blocks, cascade mes-

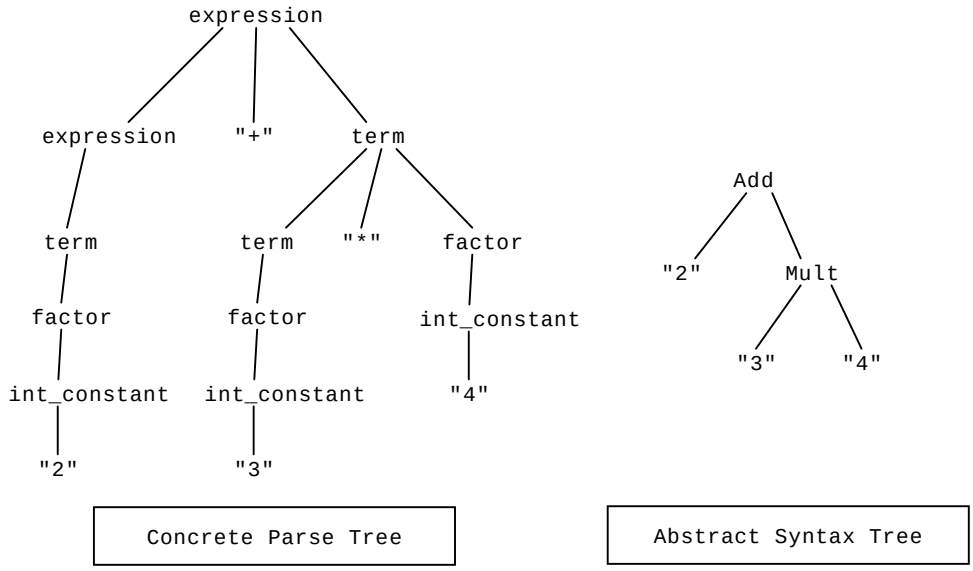

Figure 2.5: Difference between a Concrete Parse Tree and an Abstract Syntax Tree

sages, arrays, literals, messages, methods, returns, sequences and variables.

We are going to build standard method trees, but with some internal changes, such as leaving the *variable*, the *literal* and the *statement* nodes of the tree as wildcards instead of giving them fixed values as in a regular AST. This is because to detect code duplication, we need to compare structure of the methods, not the values. These wildcards have a special generic syntax used by the refactoring browser code matcher, in order to match the regular parse-tree expression nodes with the actual parse-tree nodes from the source code.

All the varying parts of the patterns (the wildcards) will be converted to meta-variables. Each meta-variable must begin with a ` character. Immediately following the ` character, other character can be inserted to specify what type of node this meta-variable can match. After all the special characters have been introduced, you must give a valid variable name. The special characters currently supported are listed in table 2.5.

When matching a statement or statements, it will be necessary to match a whole sequence node that includes both the temporaries in that sequence node as well as the other statement nodes.

For example, if we want to parameterize the following code:

```
aDictionary at: key ifAbsent: [aDictionary at: key put: value]
```

we will need to make meta-variables for the dictionary, key, and the object added to dictionary, since these are the only parts that can vary. Since all of these objects can be a single variable or a sequence of messages sent to a variable, we need to use the @ character in their names. Also, since we should look for more matches inside each node,

Character	Type	Comment	Examples
`	recurse into	whenever a match is found, look inside this matched node for more matches	"` `@object foo" matches foo sent to any object, plus for each match found look for more matches in the ` `@object part
@	list	when applied to a variable node, this will match a literal, variable, or a sequence of messages sent to a literal or variable when applied to a keyword in a message, it will match a list of keyword messages (i.e. any message send) when applied with a statement character, it will match a list of statements	"| `@Temps | ..." matches a list of temps  "`@.Statements" matches a list of statements "`@object" matches any message node, literal node of block node  "foo `@message:`@args" matches any message sent to foo
.	statement	matches a statement in a sequence node	"` .Statement" matches a single statement
#	literal	matches only literal objects	"` #literal" matches any literal (#(), #foo, 1, etc.)

Table 2.5: Special characters supported by the meta-variables.

we also need to use the ` character. This results in the ``@ prefix added to each name (the first ` means a meta-variable). Besides these conditions, we also have to remember that the `at: ifAbsent:` message may have temporary variables, and other statements in the `ifAbsent:` block. Using this prefixes, we get the following expression:

```
``@aDictionary at: ``@key
            ifAbsent:
                    [| `@Temps |
                    ``@.Statements.
                    ``@aDictionary at ``@key put:``@value]
```

Some example representations of these trees is shown in figure 2.6 with some respective matches for each pattern.

```
RBMessageNode('@x0 isNil ifTrue: ['@x10 := '@x2 defaultStringValue])
      Matches:
      affiliation1 isNil ifTrue: [affiliation1 := self defaultStringValue]
      and
      author isNil ifTrue: [author := self defaultStringValue]

RBAssignmentNode('@x1 := '@x2 parseStatements: '@x3)
      Matches:
      node := self parseStatements: false
      and
      leaf := self parseStatements: true
```

Figure 2.6: Example of Regular Parse-Tree Expression with wildcard nodes and its matches

This parametrization is used because it has all the necessary elements to detect basic code duplication cases:

- Extract Method,
- Pullup Method,
- Extract Class in Same Hierarchy, and
- Extract Crosscutting Class

We should note that this is not the only parametrization possible. An even more advanced one is possible, that can give us a finer grained detection of code duplication. It includes parametrization not only of the statements, variables and literals, but also of the messages, assignments, blocks, and method nodes available in the Refactoring Browser.

2.5 Related Work

In the last few years, many researchers have been working with the subjects that we mention: Code Duplication Detection, Refactorings and Formal Concept Analysis applied to Software Engineering. We will reference some of the work already done.

First, let's mention the work done in the Refactoring area. The first reference that we should mention when talking about Refactorings is the book written by Martin Fowler [FBB+99], which is considered as the most important book in the area, because it shows almost all the possible bad smells available and the best way to refactor them out, but always at a theoretical level, with some examples. All the work related to refactoring includes this reference, including the important survey [MT04b].

In this subject, one of the most important tools available is the Smalltalk's Refactoring Browser [RBJ97], that already has the ability of refactoring Smalltalk source code, if you know what to do and where to do it.

Our refactoring work starts with the overview of existing research in this field, and the existing work on code duplication detection, such as [KG03] that gives a general taxonomy on clone detection, but does not give any detail on what kind of duplication was detected or how to refactor it out. The same happens in [Rys02] which compares clone detection techniques with fingerprints[8].

One approach tries to detect clones in a way that is independent from the language in which we are working [DRD99], not depending on parsers, only using strings.

We know that there's no unique way to detect clones. So the question on which one is better arises. This document [RC03] compares the techniques *simple line matching*, *parameterized matching*, and *metric fingerprints*, getting the following results:

- Simple line matching is best suited for a first crude overview of the duplicated code
- Metric fingerprints work best in combination with a refactoring tool that is able to remove duplicated subroutines
- Parameterized matching works best in combination with more fine-grained refactoring tools in the statement level

The paper [BYM+98] shows us how to detect exact and near miss clones over arbitrary program fragments in program source code using abstract syntax trees, but do not suggest practical means to remove the detected clones. They claim that since their method operates in term of the program structure, clones could be removed by mechanical methods producing in-lined procedures or standard macros.

In the Formal Concept Analysis field, an application described in [Buc03] uses FCA to detect software patterns, which may help to rediscover design patterns and allow redocumentation of existing systems. This approach is similar to the one in [MT04a],

[8] Fingerprints are unique identifiers for each method. The more similar the identifiers are, the more duplicated code exists between them

where FCA is used to detect source code regularities: concerns addresses in the code, design patterns, programming idioms and coding conventions, and where they have been implemented.

This last document is the base to our work, since it introduces the Delfstof framework, that we extend to work with regular parse-tree expressions, and detect code duplication.

We may note that despite all the research available in the different areas (Refactoring, FCA and Code Duplication), none of them includes all three.

Our approach takes the best of FCA, to detect code duplication and propose possible refactorings, as we will see in the following sections.

Chapter 3

Source Code Mining for Refactorings

This section will explain step by step the process of customizing the Delfstof framework for our purpose. And also, it will explain step by step how this extension works. As an extension, we have to follow the predefined steps that the framework executes to create the Elements and the Properties, calculate the Concept Lattice; Filter, Classify and Combine the concepts, and accommodate them to achieve our goal.

3.1 Elements and Properties

3.1.1 Overview

To execute the Formal Concept Analysis algorithm and calculate the lattice, we need to create *Elements* and *Properties* of some kind.

Since our goal is to detect code duplication in Smalltalk source code, we will work with Regular Parse-Tree expressions, as seen in section 2.4.3.

As *elements* we choose all the methods of all the classes in a certain software project, and as *properties* we choose the regular parse-tree expressions of those methods.

The idea is to group in a certain concept all the methods that have similar regular parse-tree expressions. Our motivation to use this kind of technique is because Smalltalk has a very good management of its structure, specially the parse trees of the Refactoring Browser which we can reuse and accommodate to our needs.

We may note the first difference with the other existing application of the Delftsof framework, our approach does not rely on any naming convention, since the comparison takes place at structure level, and not by names and strings.

3.1.2 The Parse-Tree Nodes

The elements that we can create are of only one kind. The methods are always the same in the respective software project, and have the same fixed parse-trees. Our variable part are the properties that we generate. Depending on the type of generic parse-tree expression generated, the results will be different in some way. The Refactoring Browser allows us to use the Smalltalk tree structure through a visitor pattern, that gives us the possibility to personalize various types of nodes of the parse-trees. They are:

Assignment Node. This is the particular node of an assignment:

```
identifier := expression
```

where the identifier can be an instance variable, class variable, temporary variable that will refer to the object answered by the expression.

Variable Node is the identifier part of the Assignment node. In other words, the left part of an assignment, which can be of the instance, the class or temporary.

Block Node is the tree representation of Smalltalk's blocks of code between square brackets '`[ ]`' which are used to build control structures.

Cascade Node expressions are a series messages separated by *semicolons* (;), all of which are sent to the "receiver"

Literal Nodes are constant expressions such as numbers, characters, strings and symbols.

Literal Array Node is the abstraction of an array of literals, represented by `#(literal1 literal2 ... literalN)`.

Message Nodes are the nodes of Unary (no arguments), Binary (one argument) or Keyword (one or more arguments and use keywords followed by colon before each argument) messages sent to Smalltalk objects.

Method Node is the most generic node [1]. It contains the nodes for the entire method definitions.

Optimized Nodes are some control-flow messages which match certain patterns. those are:

- `... ifTrue: [...]`
- `... ifFalse: [...]`
- `... ifTrue: [...] ifFalse:[]`
- `... ifFalse: [...] ifTrue:[]`
- `... and: [...]`

[1] In our case, the parent node of each tree

- `... or: [...]`
- `[...] repeat`
- `... timesRepeat: [...]`
- `... to: ... do: [:index | ...]`
- `... to: ... by: ... do: [:index | ...]`
- `[...] whileTrue: [...]`
- `[...] whileFalse: [...]`

Return Node is the message that returns a value to the expression that called it. By default it is *self*, but it can be overridden by placing a caret (^) symbol in front of a statement.

Sequence Node are expressions separated by *periods* which are executed in sequence. They are usually used to mark end of statements in Smalltalk. Its representation is:

$$\texttt{expressionSequence ::= expression(.expression)}^{*}\texttt{(.)}^{opt}$$

The names of the nodes give us the first idea of tree representation. A node may contain zero, one or more children nodes. For example, a method node can contain a variable node, which can be an assignment node and a block node, which can also have a block, a variable, and a return node.

Our personalization includes making some of these nodes wildcards to match them with any piece of code that has a similar structure, but that has any string in the respective node. The Refactoring Browser has its own representation of these wildcards, as we saw on section 2.4.3, and in table 2.5.

Of the available nodes from the Refactoring Browser, we see that most of them refer to structure, which is exactly our point of comparison. The nodes that we will make wildcards are the `Variable`, and `Literal` nodes, leaving all the others fixed. We want the fragments of code to be similar, except for the literals and identifiers used. Also, we do not consider the non-functional information, such as comments, for the comparisons. Note that this is our approach for the comparison. More advanced comparisons can be made leaving other nodes of the tree variable and/or fixed.

3.1.3 Generation of Elements and Properties

In Delfstof, everything starts with a *context*. Before the execution you set all the parameters where you wish to execute the source code mining. They include the source code itself, the *elements*, the *properties*, some attribute filters, concept filters, basic analyzers, and classifications.

The first step we need to do is collect all the methods of all the classes of the particular Smalltalk source code that we want to mine. We give as a parameter to the framework

the name of the respective Package or Bundle to be analyzed. Given this parameter, the framework starts initializing the *elements* and the *properties* we will use in our mining process.

Smalltalk has a straightforward way of collecting this information. The `createObjects`[2] function of the `AttributeCreation`[3] class does all the work:

```
createObjects
      ^ (OrderedCollection withAll:  self consideredClasses)
            addAll:  self consideredMethodDefinitions;
            yourself
```

Here `self` refers to the *package* or *bundle* in which we are working. We collect all the classes, and of all those classes we take its methods, and we add them to the `OrderedCollection`.

Once all the methods have been collected, we go over all of them one more time and build the possible parse-trees for these methods. These parse trees will contain the wildcard nodes with the notation mentioned in section 2.4.3, and in table 2.5. To avoid confusion, every wildcard node has a number assigned incrementally, to differentiate them from each other. For example

```
RBAssignmentNode(`@x1 := `@x2 implementingClass parseTreeFor:  `@x3 selector)
```

and

```
RBReturnNode(^ `@x4 traverseNode:  `@x5 inClass:  `@x6 implementingClass)
```

are representations of two statements. The first one is an assignment, and the second a Return statement, each one with three variable parts.

The process putting together all the *elements* and *properties* ends when all the methods and its respective regular parse-tree expressions have been generated.

3.1.4 A First Filter

As you may notice, if we consider all the methods of all the classes of a certain software project, and of all those methods we generate all the possible parse-tree expressions, we get an extremely big collection of *elements*, and an even bigger collection of *properties*.

The first filter that we implement in our application is that we do not generate all the possible parse trees. We consider that only methods with two or more statements are worthy of having their parse-tree built. This filter leaves out some interesting parse-trees, but we consider that they do not interfere with our goal of code duplication detection.

[2]Object refers to the Elements
[3]Attribute refers to the Properties

	RPTE-1	RPTE-2	···	RPTE-N
Method1				
Method2				
⋮				
MethodN				

Table 3.1: FCA table to be calculated.

For example, it leaves out *accessing methods*, that only initialize a variable or return its value.

Our tradeoff is that this filter eliminates more useless things than it leaves useful things, and our number of *properties* considerably decreases, making our lattice smaller and decreasing the time that takes to calculate it, as we will see in the following section.

3.2 Concept Lattice

3.2.1 Building the Lattice

Once we have created the *elements* and the *properties*, we use them as input for the Formal Concept Analysis algorithm in VisualWorks Smalltalk[4]. In one hand, we have the methods of the classes of the respective software project, and on the other hand we have a collection of regular parse-tree expressions (or RTPE) which we have to check with the available methods.

The algorithm will fill table of binary relations similar to table 3.1. In an efficient way, what this algorithm does is check every tree with every available method and see if they match in the abstract structure of the regular parse-tree expression.

For example, the following methods:

```
displayOverridden MethodsConceptsForDrawingEditor
    | ctx |
    ctx := ConceptContext fromBlocksForObjects:  [self applicationClassesFrom:
    Refactory.HotDraw.DrawingEditor]
                relations:  [:class | self overriddenMethodsIn:  class].
    SCG.Classifications.ExtentionalClassification root
                add:  ctx createConcepts createView

displayOverriddenMethodsConceptsForFigure
    | ctx |
    ctx := ConceptContext fromBlocksForObjects:  [self applicationClassesFrom:
    Refactory.HotDraw.Figure]
                 relations:  [:class | self overriddenMethodsIn:  class].
```

[4]Implemented by Fred Spiessens: fsp@info.ucl.ac.be

	RPTE-1	RPTE-2	···	RPTE-N
Method1	X	X		
Method2				
⋮				X
MethodN	X			X

Table 3.2: FCA table with matched methods and trees.

```
    SCG.Classifications.ExtentionalClassification root
                add:  ctx createConcepts createView

displayOverriddenMethodsConceptsForDrawing
    | ctx |
    ctx := ConceptContext fromBlocksForObjects:  [self applicationClassesFrom:
    Refactory.HotDraw.Drawing]
                relations:  [:class | self overriddenMethodsIn:  class].
    SCG.Classifications.ExtentionalClassification root
                add:  ctx createConcepts createView
```

match the following regular parse-tree expressions:

```
RBMessageNode(`@x8 root add:  `@x9 createConcepts createView)

RBAssignmentNode(`@x1 := `@x2 fromBlocksForObjects:
.            [`@x3 applicationClassesFrom:  `@x4]
.            relations:  [:`@x5 | `@x6 overriddenMethodsIn:  `@x7])
```

and it will mark the field where the method and its abstract representation meet.

The first result that the algorithm will give us is a table similar to 3.2, where we have all the methods with their matching regular parse-tree expressions.

3.2.2 Building the Concepts

After filling the table with all the matches between methods and trees, the algorithm builds the maximal groups of elements and properties, known as concepts.

In a group, every element of the concept has a particular property, and every property of a concept belongs to a certain kind of elements. We are looking for *maximal* groups of elements. This means that in a concept, no other element (outside the concept) has those same properties, and that no other property (outside the concept) is shared by all the elements.

As we can imagine, this process gives us a very large number of concepts. The next step is to filter them to start the classification.

3.3 Concept Filtering

As we will see in our tests of section 4, the number of concepts discovered by the Formal Concept Analysis algorithm, before applying any filtering, is of the same order of magnitude as the number of considered elements. This means, that very few elements share the same properties at first, and if we gave the information like that, the software engineer would have to look at a significant number of concepts in order to try and understand the source code.

Fortunately, there is a lot of redundancy and useless concepts, or 'noise'. To reduce some of this noise, and reduce the number of useless concepts, we apply some simple *filters*.

The first filter we apply to the concept lattice generated ignores all the concepts that contain one or zero elements, since these concepts are generally too small to provide relevant information. For code duplication, we need concepts that have at least two elements or methods sharing some parse-tree. It is useless to have a concept with one method, since it is not duplicated with any other one.

A second filter applied ignores all the concepts that share only one property (in our case, one regular parse-tree expression). If we are interested in detecting code duplication, we are looking for code with more than one parse-tree structure duplicated, not only one statement, which can be duplicated by coincidence, like a variable assignation, a message or a return statement.

Notice that these two filters are independent of the kinds of elements being analyzed. We customized them as a first step to getting the best results possible in our goal of discovering code duplication.

3.4 Concept Classification

After building the elements (all the methods), the properties (the regular parse-tree expressions), and applying the first filters, we get simple sets of rather unstructured concepts. Therefore, we need to reorganize these remaining concepts automatically in a way that is easier for the software engineer to interpret and analyze.

The idea of *classifying* the concepts is similar to flattening the remaining concept lattice, so it can make more sense to the person analyzing it.

The concept classifications are shown represented as a tree in the StarBrowser[WD03].

Our classifications are based on four main possible code duplication refactorings, arranging together all the concepts that can be refactored in the same way. They are:

Extract Method. This classification groups all the concepts which have all the similar parse-tree expressions in different elements or methods, and belong to the same class. These concepts can be fixed with the *Extract Method* refactoring.

PullUp Method. Here we group the concepts in which we have a similar method, with

the same name, belonging to different classes but in the same hierarchy, and that does not exist in the respective superclass. This is the case we can fix with the *PullUp Method* refactoring.

Extract Class in Hierarchy. This concept is similar to the *PullUp Method* refactoring, with the difference that the method already exists in the superclass, and it cannot be pulled up. In this case, we can refactor it out using *Extract Class in Hierarchy*, where we create an auxiliary class containing the duplicated methods, and adding the calls to the methods using it, avoiding the duplication.

Extract Crosscutting Class. In this classification, we group together all the concepts that have similar methods, but belong to two different hierarchies. We explicitly require two or more different hierarchies to be involved. We verify this requirement checking that the most specific common superclass of the considered classes is `Object` and that none of the methods in the concept are defined on the `Object` class itself (or it would be included in the *Extract Class in Hierarchy* classification).

This taxonomy allows the software engineer to understand and clearly see what kinds of code duplications exist in the current software system, and better yet, know what kind of refactoring to apply to them, so he or she can get a better and more understandable software system.

Chapter 4

Experiments

This section will show some practical applications of our tool, explained in section 3.

We will analyze two experiments made with existing Smalltalk source code: the Delfstof framework and the Refactoring Browser. After analyzing the results, we will discuss the steps to follow to apply the refactorings.

4.1 Delfstof Experiments

This experiment will analyze the Delfstof framework itself, that is still under development, looking for code duplication to improve its design and understandability.

4.1.1 Setup of the Experiment

As we saw in chapter 3, we have to create a context where we will execute our mining. In this example, the only difference with the general code duplication mining context explained in section 3.1.1 is the package of the Smalltalk image where we will work. The name of the package in this case is *ConceptLattices*, which includes all the Delfstof framework and the Formal Concept Analysis source code, and is considered to be a small/medium sized package.

4.1.2 Results for Delfstof

The results of executing the mining in the Delfstof framework, are the following:

- Number of Elements: 663 methods (in 163 classes)
- Number of Properties: 241 parse-trees
- Number of raw Concepts found: 156
- Number of Filtered Concepts found: 6

- Time of execution: 1.6 seconds

We can see the results tree show in StarBrowser in figure 4.1

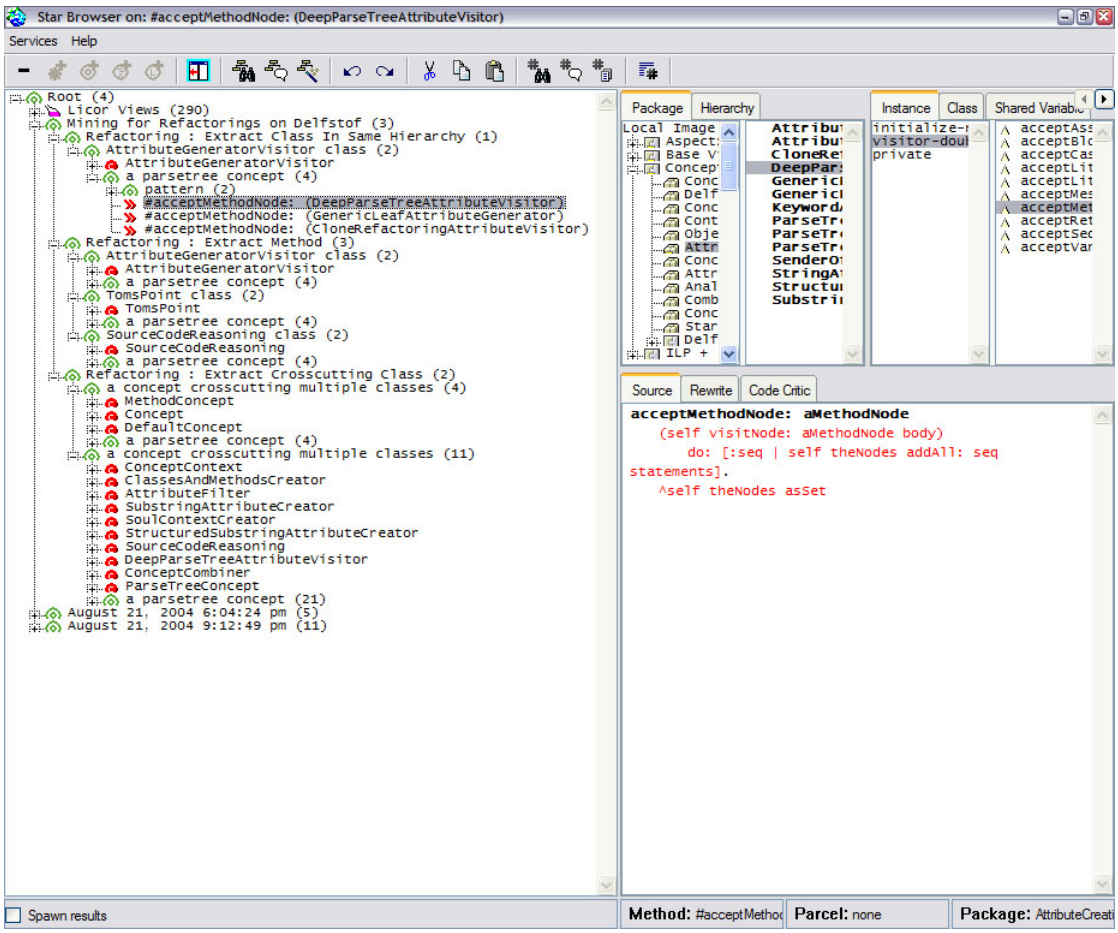

Figure 4.1: Results of Delfstof mining in StarBrowser

In this case, the tool found code duplication that fitted in three of the four classifications available for code duplication refactoring possibilities:

- Extract Class In Same Hierarchy (1 concept)
- Extract Method (3 concepts), and
- Extract Crosscutting Class (2 concepts)

Looking inside these classifications, we get the results described as follows.

Extract Class In Same Hierarchy. Here we find a hierarchy of three subclasses and a superclass that have the same method definition `#acceptMethodNode:` which

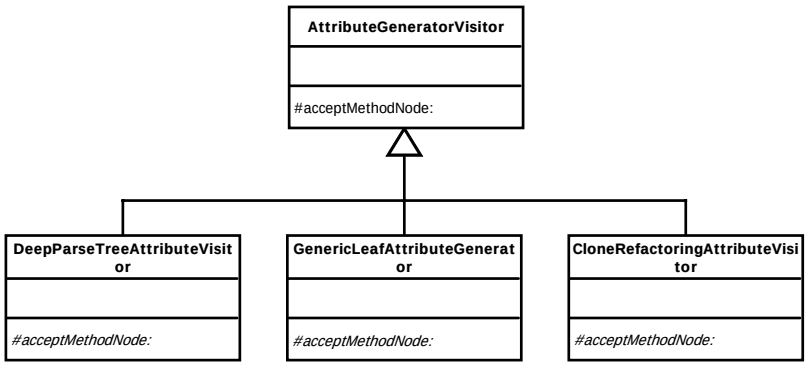

Figure 4.2: Existing Class structure in Delfstof

belongs to the visitor we use to create the parse trees. We can see the representation in figure 4.2.

The implementation of the method in the subclasses is exactly the same, so we can think about applying the *PullUp Method* refactoring, but the method already exists in the superclass `AttributeGeneratorVisitor` with an implementation different from the subclasses.

The *Class Extraction in Same Hierarchy* refactoring proposed in this case, is intended to create an auxiliary class, so we can move the implementation to it and make the calls (send the messages) to the new implemented method, as we see in figure 4.3. This class contains the implementation of the methods, and the subclasses will have to change to call the unique method implementation, to avoid the duplication of code.

In this first case, we successfully detected the code duplication, and proposed a refactoring that could be applied at any time to eliminate the existing duplication

Extract Method. The second classification detected three cases of possible method extraction.

In the first case, also in the `AttributeGeneratorVisitor` class, the algorithm found three methods that have a similar implementation, only differing in one parameter. The refactoring proposed by [FBB+99] in this case is *Extract Method* with the different parts of the differing statements passed to the new method as a parameter, and referenced by the existing ones, as we can see in figure 4.4

This is one of the most common refactorings, and also one of the easiest to implement.

The case detected in the `SourceCodeReasoning` class is different to the one just mentioned. The methods found, despite having the same structure, each one of them depend on different instance variables, which would make the *Extract Method* refactoring inappropriate. This is the reason why we show the refactorings in this

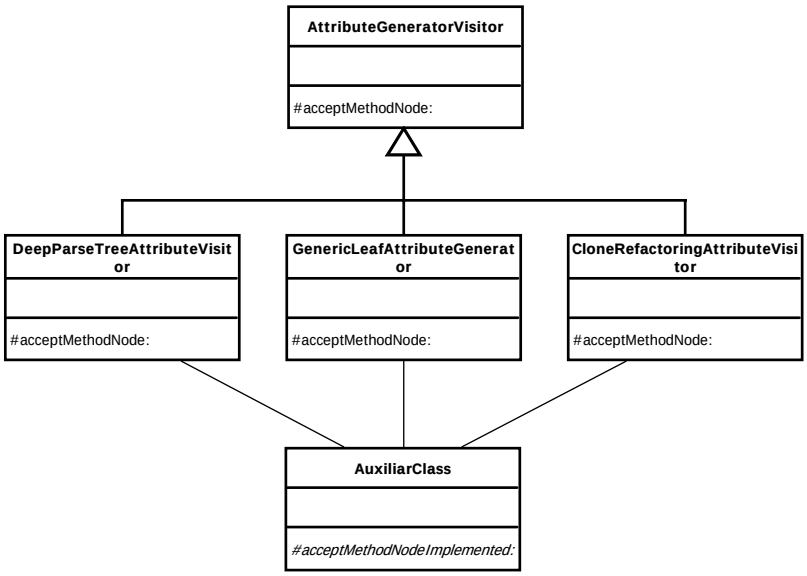

Figure 4.3: Proposed refactoring Delfstof

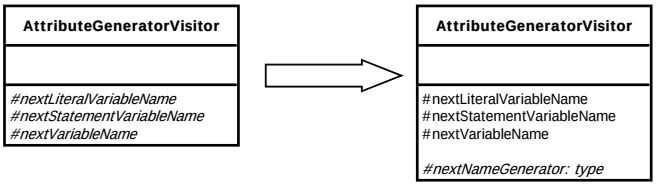

Figure 4.4: Extract Method refactoring

way, to see that code duplication may exist, but not always it is possible to refactor it out.

Extract Crosscutting Class. This classification includes two of the found concepts.

One of the concepts found duplicated code from twenty methods in ten different classes, and the other found three methods from three different classes. In section 3.1.1, we mentioned that we only built the regular parse-tree expressions for those methods with two or more statements. This is the case with two statements that would need a finer filtering. The algorithm detected all these classes that had two statements in common, but all where lazy initializations, and a return statement at the end of the methods.

The first twenty methods had the `RBAssignmentNode(` `@x10 := ` `@x11 new)` and the `RBReturnNode(^` `@x16)` nodes in common, meanwhile the other three had the `RBAssignmentNode(` `@x1 := ` `#x2)` and the `RBReturnNode(^` `@x16)` nodes in common.

In this case, refactoring is not a good idea, but at least we found the duplications as an idea on how to make finer filtering in the future.

4.2 Refactoring Browser Experiments

This experiment will analyze the Refactoring Browser source code, looking for code duplication, to see that even in a tool that has been designed to work with refactorings, has its own problems of code duplication.

4.2.1 Setup of the Experiment

In this experiment, our setup includes a context that is similar to the one seen in chapter 3, but including the source code of the Refactoring Browser. The name of the package is *RefactoringBrowser*, and includes all the source code of this application, which is considered to be of large size.

4.2.2 Results for Refactoring Browser

The results of executing this application in the Refactoring Browser, are the following:

- Number of Elements: 4503 methods (in 271 classes)
- Number of Properties: 3718 parse-trees
- Number of raw Concepts found: 2499
- Number of Filtered Concepts found: 189

- Time of execution: 2 minutes and 37 seconds

We can see the results tree show in StarBrowser in figure 4.5

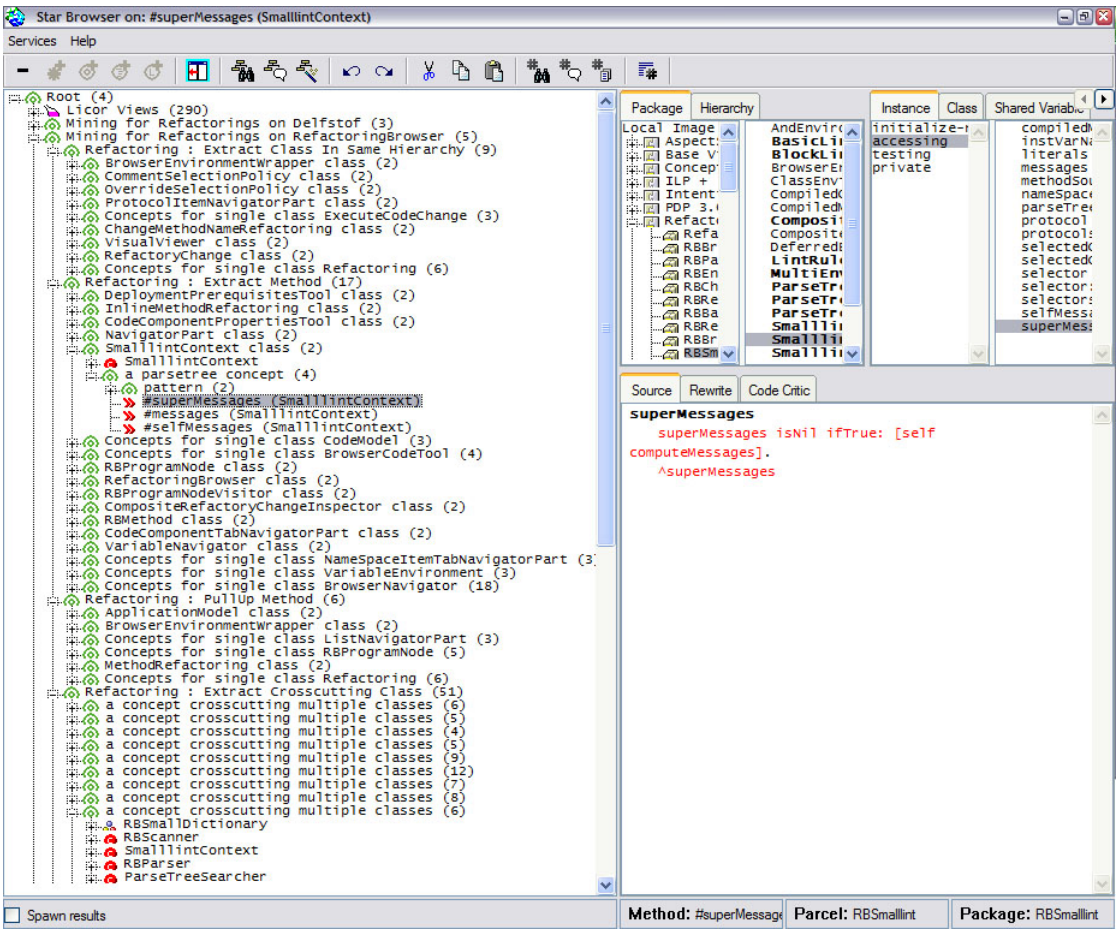

Figure 4.5: Results of RefactoringBrowser mining in StarBrowser

In this experiment, the tool found cases of all four possible code duplication classifications:

- Extract Class In Same Hierarchy (17)
- Extract Method (17)
- PullUp Method (6), and
- Extract Crosscutting Class(51)

Since the Refactoring Browser is a very large application, we will explain some of the discovered and classified concepts, as well as the possible refactorings applicable to improve the design of the application.

Extract Class In Same Hierarchy. This classification found some code duplication in different classes in the same hierarchy, which cannot be pulled-up, but can be extracted to an auxiliary class. An example is the method `#editorClass` of the classes `MenuViewer`, `IconViewer`, and `CanvasViewer` as we can see in figure 4.6. They share the same implementation, differentiating only in one parameter. This difference can be passed as a parameter to the new function called, similar to the diagram 4.3 shown in the previous example, but with a parameter of the different part.

```
editorClass
    | class |
    class := #{MenuEditor} valueOrDo: [nil].
    class isNil
        ifTrue:
            [self warn: (#UIPainterNotLoaded << #browser >> 'UIPainter not loaded.').
            ^nil].
    ^class

editorClass
    | class |
    class := #{UIMaskEditor} valueOrDo: [nil].
    class isNil
        ifTrue:
            [self warn: (#UIPainterNotLoaded << #browser >> 'UIPainter not loaded.').
            ^nil].
    ^class

editorClass
    | class |
    class := #{UIPainter} valueOrDo: [nil].
    class isNil
        ifTrue:
            [self warn: (#UIPainterNotLoaded << #browser >> 'UIPainter not loaded.').
            ^nil].
    ^class
```

Figure 4.6: Duplicated code in class MenuViewer, IconViewer and CanvasViewer respectively

Extract Method. In this bigger project we find two different possibilities of applying the *Extract Method* refactoring. We have classes with methods that are exactly the same, and classes with methods that share some of its statements.

Both cases can be refactored extracting an auxiliary method including the duplicated code and adding a call to that method in the right position of the existing method, where the cloning occurs.

In the class `VariableEnvironment`, we have three methods that are exactly the same, maybe because they have similar behavior, but in that case, we should centralize the implementation. These methods are: `#addClass: instanceVariable:`, `#addClass: instanceVariableReader:`, and `#addClass: instanceWriter:`, as

we can see in figure 4.7. The bodies of these methods, as in the previous experiment, can be replaced by a call to another method that contains the full implementation, avoiding the duplication of code.

```
addClass: aClass instanceVariable: aString
    (instanceVariables at: aClass fullRootName ifAbsentPut: [Set new])
        add: aString.
    self flushCachesFor: aClass.
    self addSearchString: aString

addClass: aClass instanceVariableReader: aString
    (instanceVariableReaders at: aClass fullRootName ifAbsentPut: [Set new])
        add: aString.
    self flushCachesFor: aClass.
    self addSearchString: aString

addClass: aClass instanceVariableWriter: aString
    (instanceVariableWriters at: aClass fullRootName ifAbsentPut: [Set new])
        add: aString.
    self flushCachesFor: aClass.
    self addSearchString: aString
```

Figure 4.7: Duplicated code in methods of VariableEnvorinment class

Something a little different occurs in class `BrowserCodeTool`, where the methods `#createInstVar`, `#inlineParameter`, `#renameTemporary`, `#removeParameter`, and `#bindTight`, where the methods share two large statements, which can be perfectly refactored in another method. We can see them in figure 4.8.

Note that these methods depend on a temporary variable *node*, that we can move together with the statements to another method, since it is only used between them.

PullUp Method. As we saw in the previous example, these classification includes similar method of in different classes in the same hierarchy, and also the method does not exist in the super class. That is the case for the method `#postcopy`, shown in figure 4.9. This methods exists in the classes `RBMessageNode`, `RBBlockNode`, and `RBMethodNode`, in the hierarchy of class `RBMethodNode`. This method could be moved to this last class, and when called, it will look in the hierarchy of the classes, finding it in the superclass, avoiding the existing copy.

Extract Crosscutting Class. For this classification, something particular happened. All the possible methods that could be extracted in crosscutting classes, first had to be refactored using *ExtractMethod* in all the classes involved, then the method could be extracted in an auxiliary class which would replace the duplicated code.

This behavior is common in almost all the classifications. We find code that is not completely duplicated between different methods in different classes of the same or different hierarchies. According to Martin Fowler in [FBB+99], the correct way to refactor the code is by steps. First extract the duplicated code in an auxiliary method, then extract that method to the respective auxiliary class, which can be called by all the others that need the extracted implementation.

```
createInstVar
    | node |
    node := self findNode.
    (node isNil or: [node isVariable not])
       ifTrue: [^self warn: (#CouldNotFindTheNode << #browser >> 'Could not find the node')].
    self performRefactoring: (TemporaryToInstanceVariableRefactoring
              class: self methodClass
              selector: self selector
              variable: node name)

 inlineParameter
     | node |
     node := self findNode.
     (node isNil or: [node isVariable not])
        ifTrue: [^self warn: (#CouldNotFindTheNode << #browser >> 'Could not find the node')].
     self handleError:
            [| ref |
            ref := InlineParameterRefactoring
                      inlineParameter: node name
                      in: self methodClass
                      selector: self selector.
            ref execute.
            codeModel setSelector: ref newSelector]

renameTemporary
    | newName node |
    node := self findNode.
    (node isNil or: [node isVariable not])
       ifTrue: [^self warn: (#CouldNotFindTheNode << #browser >> 'Could not find the node')].
    newName := self request: (#EnterNewNameC << #browser >> 'Enter new name:') initialAnswer: node
name.
    newName isEmpty ifTrue: [^self].
    self performRefactoring: (RenameTemporaryRefactoring
              renameTemporaryFrom: (self convertToSourceInterval: node sourceInterval)
              to: newName
              in: self methodClass
              selector: self selector)

 removeParameter
     | node |
     node := self findNode.
     (node isNil or: [node isVariable not])
        ifTrue: [^self warn: (#CouldNotFindTheNode << #browser >> 'Could not find the node')].
     self handleError:
            [| ref |
            ref := RemoveParameterRefactoring
                      removeParameter: node name
                      in: self methodClass
                      selector: self selector.
            ref execute.
            codeModel setSelector: ref newSelector]

 bindTight
     | node |
     node := self findNode.
     (node isNil or: [node isVariable not])
        ifTrue: [^self warn: (#CouldNotFindTheNode << #browser >> 'Could not find the node')].
     self performRefactoring: (MoveVariableDefinitionRefactoring
              bindTight: (self convertToSourceInterval: node sourceInterval)
              in: self methodClass
              selector: self selector)
```

Figure 4.8: Duplicated code in methods of BrowserCodeTool class

```
postCopy
    super postCopy.
    arguments := arguments collect: [:each | each copy].
    body := body copy
```

Figure 4.9: Code duplication in the hierarchy of class RBProgramNode

We group the code duplications directly in the ultimate classifications, to clearly have the idea of what is that we want to refactor, and to see the goal of applying the refactorings.

Chapter 5

Conclusion

In this chapter, an overview of this document and the future work is provided. The main investigation questions are revisited, and the methodology used is discussed, analyzing the contribution of this approach. Finally, directions for future work are presented.

5.1 Summary

In this document, an approach for code duplication detection and classification for proposing refactorings is given and demonstrated with examples.

This work is done as an extension of the existing Delfstof framework, for VisualWorks Smalltalk, which uses the theory of Formal Concept Analysis for code duplication detection.

This formalism groups sets of elements according to some kinds of properties. In our case the *elements* are the methods of a particular software project, meanwhile for the *properties* we choose a special kind abstract syntax tree with the *identifiers* and *literals* nodes as wildcards. We call it *regular parse-tree expression*.

The regular parse-tree expressions are matched to the methods, looking for similar structures. If a method matches a particular parse-tree expression, we see that it has the same structure, and can be grouped in a single concept by the formal concept analysis algorithm.

After executing the formal concept analysis algorithm, a finer filtering is necessary to group similar concepts together. What we looked for is grouping together the different concepts that share the same possible kind of code duplication refactorings, such as *Extract Class in Same Hierarchy*, *Extract Crosscutting Class*, *PullUp Method*, and *Extract Method*. These concept classifications can be shown flattened as a tree structure in VisualWorks Smalltalk's StarBrowser.

This approach was tested on two software projects: Delfstof itself and the Refactoring Browser. With the filters implemented, interesting code duplications of the four mentioned kinds were found. They were not applied, just proposed in case that some software

engineer would like to improve the design and understandability of them.

Although the filtering and the property generation can be improved in many ways, this gives us a very good first look at duplicated code in existing software projects, which we can improve by applying refactorings to eliminate the code duplication bad smells.

5.2 Contribution

Software engineering and Formal Concept Analysis have been working together since a long time. We have seen it applied to reengineer class hierarchies, to analyze object-oriented framework reuse,to support software maintenance and object-oriented class identification tasks, and to detect regularities in source code. In this last application, some of the regularities are a basic level of code duplication.

Our contribution applies the mathematical technique of Formal Concept Analysis to software engineering, using it to detect code duplication not at the usual string level, but at a higher structural level. We have a very special and particular way in which we create the elements and the properties to use as input for the formal concept analysis algorithm, and also a unique way to filter the concepts and group them together automatically to show them ordered in a way that is easier for the software engineer to see where the duplications take place, and also facilitate the possibility of applying the proposed refactorings.

This approach is better than other we have seen, such as the other tool made for the delfstof framework, which looks for regularities in source code. That approach depends on strings and coding conventions, and in general, *string based* techniques, and ours depends only on comparing similar structures, even using the same framework to do the process. Other approaches have also tried to detect code duplication using as a parameter the Duplicated Lines of Code (DUPLOC), but since they work with *Lines of Code*, they can be tested only in languages that save their source code in text files, such as Java, C and C++, and not in languages like Smalltalk that works with a full environment image.

As we can see, our approach does the work it should, and it does it very well automatically finding things that to a programmer could take many hours or even days to detect

5.3 Future Work

The important aspects where this approach can be improved are improving the filtering of the concept classifications found, make it more fine grained so that we can get even better results and maybe apply them automatically, without asking the user to execute it.

We also see that is perfectly possible to apply this approach to other programming languages, such as Java, C or C++, since they also have working environments that can be read from Smalltalk and analyzed to detect duplicated code.

A very interesting possible future work application that has to do with code duplication is the detection and refactoring of Aspects [KLM+97] as we can see in [Hir02], in [Lad03] and in [vDMM03].

Bibliography

[Buc03] Frank Buchli. Detecting software patterns using formal concept analysis. Master's thesis, Institut für Informatik und angewandte Mathematik, September 2003. 20

[BYM+98] Ira Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of ICSM'98*. IEEE, November 1998. 16, 20

[DG03] Uri Dekel and Yossi Gil. Revealing java class structure with concept lattices. Master's thesis, Israel Institute of Technology, February 2003.

[DRD99] Stéphane Ducase, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. *IEEE International Conference on Software Maintenance*, page 109, September 1999. 1, 20

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Object Technology. Addison-Wesley, 1999. 4, 5, 20, 32, 37

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1st edition, 1995. 14

[GW99] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag, January 1999. 7

[Hir02] Robert Hirschfeld. Aspect-oriented programming with aspects. Technical report, DoCoMo Communications Laboratories Europe, April 2002. 42

[Ken03] John Kennedy. Parse trees. Technical report, Mathematics Department, Santa Monica College, 2003. 15

[KG03] Cory Kapser and Michael W. Godfrey. Toward a taxonomy of clones in source code: A case study. Technical report, Software Architecture Group (SWAG), School of Computer Science, Univertisy of Waterloo, 2003. 1, 20

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European*

Conference on Object-Oriented Programming, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997. 42

[Lad03] Ramnivas Laddad. Aspect oriented refactoring series. http://www.theserverside.com/articles/, December 2003. 42

[LD03] Michele Lanza and Stephane Ducasse. Codecrawler - lessons learned in building a software visualization tool. In *7th European Conference on Software Maintenance and Reengineering*, pages 409 – 418. IEEE Computer Society, 2003. 12

[MBM03] Wolfgang De Meuter, Johan Brichau, and Kim Mens. *SOUL Manual*. Programming Technology Lab, Vrije Universiteit Brussel, draft edition, December 2003. 12

[MT04a] Kim Mens and Tom Tourwé. Conceptual code mining - mining for source-code regularities with formal concept analysis. to be published in ESUG 2004 Journal, 2004. 10, 20

[MT04b] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30, February 2004. 4, 20

[Opd92] William Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urban-Champaign (UIUC), 1992. 4

[RBJ97] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. In *Theory and Practice of Object Systems, vol. 3, no. 4*, 1997. 12, 16, 20

[RC03] Filip Van Rysselberghe and Serge Cemeyer. Evaluating clone detection techniques. Technical report, Lab On Re-Engineering, University of Antwerp, 2003. 20

[Rys02] Filip Van Rysselberghe. Detecting duplicated code using metric fingerprints. Master's thesis, University of Antwerp, 2002. 20

[TCBE03] T. Tilley, R. Cole, P. Becker, and P. Eklund. A survey of formal concept analysis support for software engineering activities. In Gerd Stumme, editor, *Proceedings of the First International Conference on Formal Concept Analysis - ICFCA'03*. Springer-Verlag, February 2003. to appear.

[Ton04] Paolo Tonella. Formal concept analysis in software engineering. In *26th International Conference on Software Engineering (ICSE)*, pages 743–744. IEEE, May 2004.

[vDMM03] Arie van Deursen, Marius Marin, and Leon Moonen. Aspect mining and refactoring. Technical report, Software Evolution Research Lab - CWI and Delft Univ. of Technology, September 2003. 42

[WD03] Roel Wuyts and Stéphane Ducasse. Unanticipated integration of development tools using the classification model. In *Journal of Computer Languages, Systems and Structures*, pages 63–77, 2003. 11, 12, 28