Aspect Mining using Clone Detection

by:
Orlando Alejo Méndez Morales

A Thesis presented for the degree of
Master of Science

Department of Information, Systems and Algorithms
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
The Netherlands
August 2004

Aspect Mining using Clone Detection

By:

Orlando Mendez

Thesis Advisor:

Prof. Dr. Arie van Deursen

Thesis Committee:

Prof. Dr. Arie van Deursen

Dr. Ir. Leon Moonen

Dr. Ir. Arjan J.C. van Gemund

Ir. Frans Ververs

Ir. Marius Marin

For where your treasure is, there will be your heart also.
Want waar uw schat is, daar zal ook uw hart zijn.
Porque donde esté tu corazón, ahí estará también tu tesoro.

Lucas 12:34

Abstract

Any software system in usage is required to be adapted in order to meet the ever changing and increasingly demanding requirements of its users. This adaption does not come for free and moreover, as the system evolves, in numerous situations it demands a considerable effort from developers to comprehend what the code is actually doing, before they can carry out the pertinent maintenance to the system.

Software exploration is one of the software engineering areas meant to support the evolution of systems. Since software exploration provides techniques and tools to reveal the "anatomy" of the system, it can be used to discover features that might be hidden or tangled with the main concern(s) the system was created for.

Decomposition is a technique used in software engineering with the purpose of manage the complexity, development and maintenance of software. Object oriented software is decomposed in packages, classes and methods, forming a class hierarchy. However, this existing decomposition will not always have a clear separation or modularization of the concerns required for the business logic of the system, revealing the necessity to develop methods and tools able to find these cross-cutting concerns (also known as "aspects" for short).

Because aspect oriented programming prevents tangling or scattering of cross-cutting concerns throughout the system, the resulting system is expected to be more understandable, easier to maintain, and potentially easy to reuse. Aspect mining and refactoring are needed to identify and extract aspects from an object oriented system to turn it into an aspect oriented one. In this thesis, we will concentrate on aspect mining for object oriented systems written in the Java language.

The purpose of this research project addresses the analysis of the current principles, methods and tools supporting the identification of aspects in object-oriented systems, and investigates clone detection as a mean to achieve (automated) aspect mining.

Acknowledgements

This thesis is the result of my graduation project on aspect mining at the Software Evolution Research Laboratory (SWERL) in Delft University of Technology, The Netherlands.

If I had to describe all the experiences -both good and bad- that I have accumulated in these 25 months of my stay in Holland, which were financially supported by NUFFIC (the Netherlands Organization for International Cooperation in Higher Education), it seems I could write down an entire book. Luckily, I do not need to do so, and only this thesis suffices :-) Anyway, I want to say that it has been worthwhile to come here, despite all the difficult moments and situations I have faced.

My acknowledgement goes to professor Dr. Arie van Deursen for his wise guidance through the whole project. *Eerlijk gezegd* (I like this expression in Dutch), if I can say I have learnt what to carry out research means, it is because Arie has taught me a lot about it. Moreover Arie, thank you for listening and pointing me back to the right direction when I felt uncertainty. May our friendship endure throughout the years.

I want to thank also Marius Marin, Ph.D. student of the Aspect Mining and Refactoring project at SWERL. Marius, your insights and questions more than once made me reflect about what and why was I doing things; thanks. Dr. Leon Moonen has contributed effectively to this document, specially the revisions and comments in the last draft(s) were in my opinion key to refine ideas and concepts otherwise vague. Leon, I also would like to be present to see you in your kayak in a rapid class V ;-)

My thankfulness goes also to the lecturers and personnel of TUDelft who put the best of their efforts in the educative process I begun in 2002. Special mention is to Ir. Frans Ververs and Dr. Ir. Arjan van Gemund, members of my thesis committee, together with Arie, Marius and Leon. I would like to thank as well to Frank v.d. Vlugt, former coordinator for international Computer Science students, whom hereby I take advantage to congratulate: we made it guys!

At home, I want to thank the support and company of my 2 countrymen: Oswaldo & Leonardo. Guys, I have learnt with you what *to reach concensus* means. The people at Levend Water gemeente have received me and helped me to grow as human being, specially Klaas-Jan, Gerdine, Leon, Mylène and my cunning young "nephews", Daniël and Joran ;-) *Mi amiga Nancy,* you also deserve a few lines in this place: thank you for listening (without judging) me saying things that most of those who know me could not believe I went through being here.

Lastly, I want to dedicate this work:
To my parents Marcelo & Graciela for their continual support during all these years.
To the God of the Bible for showing me that "all things are possible for those who believe".
Lord, I'm learning to walk upon the waters holding Your hand.

Orlando Mendez. *Delft, The Netherlands; August 2004.*

Contents

List of Figures

CHAPTER 1

ASPECT ORIENTED PROGRAMMING AND ASPECT MINING

1.1. Introduction to Aspect Oriented Programming

Given the complexity in size and in understanding of modern software systems, the necessity arises of developing tools that may aid in the comprehension, maintenance and in general, in contributing to the evolution of such systems.

An important paradigm to address this problem is **Aspect Oriented Programming** (AOP), which aims to implement cross-cutting concerns (also called aspects) in software systems. Concerns are the way humans separate or modularize the functionality of a system. An aspect is a feature of the system which cannot be encapsulated in the current decomposition (i.e., methods, classes, packages, etc.) used in object oriented systems. Aspects are properties that affect the maintenability and evolvability of the system's components.

In order to put in practice AOP, the following implementation elements are required:

- *A component language.* The language in which the existing system is implemented.
- *An aspect language.* A language that allows us to capture the cross-cutting features of the system.
- *A weaver.* A special language processor that weaves both languages.
- *A component program.* The system itself; which in the interest of this thesis, is implemented in an object oriented way.
- *An aspect program.* The program(s) necessary to capture the cross-cutting behaviour of the system.

1.1.1. General idea of AOP. Aspect Oriented Programming [KLM+97] nowadays is an increasingly popular programming discipline based on the concept of cross-cutting concerns[1]. In the software development context, a feature of a system is "an aspect, if it cannot be cleanly encapsulated in a generalized procedure (i.e., object, method, procedure, etc.)" [Kea97, p. 7]. By adopting AOP in the most often used programming paradigms (i.e., Object Oriented Programming, Procedural Programming), users are able to develop systems with the following advantages:

- Obtaining better modularized and concise code
- Making explicit the way concerns cross cut one another
- Assisting developers navigate, and
- Improving understanding about the structure of their code.

Before Aspect Oriented Programming came to be what it is now, there have been other ideas that have contributed to its conception. The first idea we can mention is the one given by E.W. Dijkstra, who in his book "A discipline of Programming" [Dij76], stated that the best way to deal with a complex computational task, is to

[1] In the rest of this document, we may use the terms cross-cutting concern or aspect indistinctly.

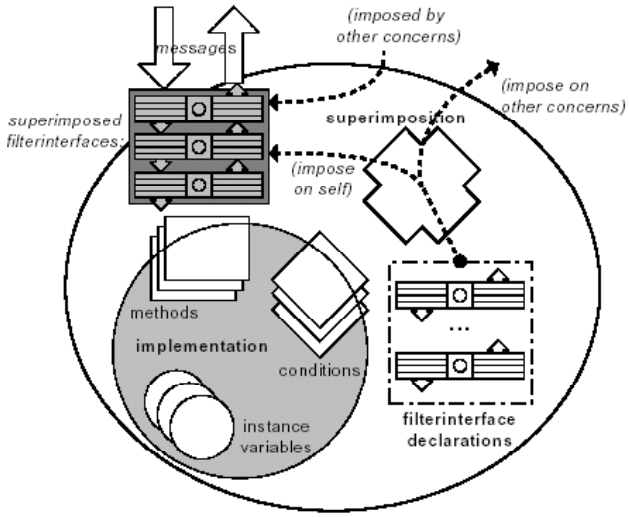

FIGURE 1.1.1. The CF Model with superimposition (Id. as in [BA01])

focus on one specific concern at a time. This basic idea has evolved and improved during time as we will see next.

1.1.2. Composition Filters Model. In the early 90's, Mehmet Aksit from Twente University, has created the **Composition Filters** (CFs) Model [MWB+93]. Composition Filters is an approach in which AOP can be achieved, by providing control over the messages sent and received by objects[2]. The Composition Filters mechanism provides an aspect language, that is able to manipulate aspects through the notion of superimposition. Superimposition "means that one abstraction can enhance other abstractions with additional concerns by decorating ('superimposing') concern specifications" [BA01]. Superimposition is depicted in figure 1.1.1.

1.1.3. Adaptive Programming. Also by the early 90's, **Adaptive Programming**(AP) came to be known. In AP, programs are decomposed into several crosscutting building blocks. In order to write code that follows the law of Demeter(which states: "Only talk to your immediate friends") in an ideal way, concerns whose implementation is scattered across several classes, should be cleanly localized. Clean localization means a clean separation of various behavioral concerns from the structural information, through a class graph[3]. A class graph describes some classes and the existing relationships among them from a particular point of view. Adaptive programming therefore can be considered as a (sub)set of AOP in the sense that it also provides modularity in another way than object-oriented (de)composition [Bre02].

1.1.4. Subject-Oriented Programming(Hyperspaces). Subject-oriented programming [HO93] is a technology that supports building object-oriented systems as compositions of subjects. The concept of subject is "a collection of classes or class fragments whose hierarchy models its domain in its own way"[4]. A subject may be

[2]See: `http://www.parc.xerox.com/research/csl/projects/aspectj/downloads/ISPJ-2002-keynote-1.ppt`

[3]See: `http://www.ccs.neu.edu/home/lieber/connection-to-aop.html`

[4]Subject-oriented programming: overview of concepts; `http://www.research.ibm.com/sop/sopoverv.htm`

a complete application in itself, or it may be an incomplete fragment that must be composed with other subjects to produce a complete application. Subject-oriented programming thus supports building object-oriented systems as compositions of subjects, extending and integrating them with other (new) subjects.

Subject-oriented programming involves determining how to subdivide a system into subjects, and writing the composition rules needed to compose them correctly [IBM98]. Subject-oriented programming has been expanded in the Multi-dimensional separation of concerns as explained by Tarr et al. [TOHJ99]. Multidimensional separation of concerns denotes a separation considering:

- Multiple, arbitrary kinds (dimensions) of concerns.
- Overlapping or interacting concerns.

Tarr and Ossher extended further the concept of achieving multiple separation of concerns, and developed at IBM the concept of hyperspace(s) [Res99], which are structures that group concerns into dimensions, being these dimensions sets of concerns that have no units (either classes, modules, procedures, etc.) in common, i.e, dimensions are sets of disjoint concerns.

Hyperspaces contain also the following elements:

- Hypermodules, which are composed from hyperslices, i.e., collections of units specified according to the hyperspace definition. Hypermodules are building blocks of the systems, and (commonly) they are not executable programs
- A composition rule. This rule specifies how such hyperslices should be integrated.

1.1.5. The AspectJ Language. AspectJ [KHH+01] is also one of the origins of the Aspect Oriented Programming paradigm. AspectJ is aimed to implement cross-cutting concerns, and it is based in the Java programming language, .

AspectJ allows to write programs including both aspect and non-aspect (i.e., object oriented) code. To make possible coordination between aspect and non-aspect code existing in a program, AspectJ based its design in the join point model. A *join point* is a "well-defined point within the execution of a program" [KHH+01, p. 3], such as a method call.

To pick out join points during execution, AspectJ provides *point-cuts*, which can be defined in classes or in aspects (cross-cutting concern code). When a program reaches a join point, code called *advice* is executed. This code is the one that represents the cross-cutting behavior desired in an application, and can be executed before, after or around (in practice instead of) a join point.

1.2. Software exploration

Software engineering is a broad concept that comprises the activities related to creating and maintaining software applications by applying computer science, project management, domain knowledge and other technologies and practices[5]. It is precisely in the maintenance area where considerable effort is spent in order to extend the life of software. The purpose that **software evolution** pursues is to keep systems in synchrony with ever-changing needs.

An evolving system will tend to show two properties [Moo02, p. 4]:

(1) Continuing change. A software system is an entity that needs to be adjusted in order to meet the needs of its users.

[5]Source: `http://en.wikipedia.org/wiki/Software_engineering`

(2) Increasing complexity. An effect of the previous property is that the complexity of the system will increase unless something is done to prevent it.

Because of these two properties, it is important in order to reach software evolution's goal, to know what does a software system contain. We say then that we need to **explore** the system. By exploring a software system we are able to understand it better, and a considerable amount of research has been done in order to come up with a theory on **program comprehension** to aid software engineers in this task [Moo02, p. 9].

1.2.1. Reverse engineering. The set of techniques meant to support program comprehension is known as **(software) reverse engineering.** The goal of reverse engineering is, given a system, to derive the *design artifacts* that gave origin to it. In the context of software this may consist of obtaining the requirements, specifications and the architecture of the system [Moo02, p. 11].

1.3. Aspect mining

Among the current practices being developed in the context of software exploration in order to improve software evolvability, a recent technique is **aspect mining**, which consists in the search and isolation of aspects which might be considered (good) candidates for refactoring in the system [vDMM03].

Refactoring [Fow99] aims to preserve the original functionality of software systems yet improving its internal structure. Thus, refactoring is a technique that helps in:

- Improving software design
- Understanding software
- Detecting bugs
- Accelerating the software development.

Nevertheless, a prior requirement to refactoring an object oriented system into an aspect oriented one is to detect portions of code that are amendable to be improved (i.e., aspects). In this thesis, we carry out such detection by means of aspect mining. Detection of cross-cutting concerns is therefore very important to improve the maintainability, comprehension and evolvability of information systems.

Next, we present some examples of the aspects we want to mine.

1.3.1. Examples of aspects. Since Aspect Oriented Programming is a developing discipline, certain themes are in the maturating process. One central theme is the one of cross-cutting concerns and their categorization.

A first categorization about aspects is provided in the AspectJ programming guide [Tea02], where aspects are divided into *development* aspects and *production* aspects. These aspects mentioned next can be considered typical examples of cross-cutting behaviour in software systems, whose implementation has been successfully implemented in the AspectJ language, and that is the reason we include them here.

However, we propose the following outline as a template that might be useful as a first step towards a more extensive and comprehensive categorization of aspects, looking forward for an aspect taxonomy [MvDM04, p. 2].

1.3.1.1. *Aspects in AspectJ* . Creators of AspectJ [Tea02] have categorized aspects as follows:

- Development aspects

Name: Tracing
Description:

When debugging, programmers often invest considerable effort in figuring out a good set of trace points to use when looking for a particular kind of problem. This aspect shows the internal workings of a program.

Join-point(s):
Method calls specified by the user.

Sample code:

```
aspect
SimpleTracing {
 pointcut tracedCall():call(void   FigureElement.draw(GraphicsContext));
 before(): tracedCall() {
   System.out.println("Entering: " + thisJoinPoint); }
}
```

Name: Logging
Description:
Our second example shows how to log specific behavior. Logging reveals the behavior and certain features of the elements of a program.

Join-point(s):
As example, the following aspect counts the number of calls to the rotate method on a Line object and the number of calls to the set* methods of a Point object that happen within the control flow of those calls to rotate.

Sample code:

```
aspect PersistenceCounting {
 int FigureCount = 0; int ContribCount = 0;
 before():  { call(void *.Contrib.*.write*(StorableOutput)) &&
 cflow(call(Storable.writeStorable(..)))
    ContribCount++;
}
 before(): call(void *.Figures.*.write*(StorableOutput)) &&
 cflow(call(Storable.writeStorable(..))) {
    FigureCount++;
 }
}
```

Name: pre and post-conditions
Description:
In the design by contract style of programming, pre-conditions test that callers of a method call it properly and post-conditions test that methods properly do the work they are supposed to. This aspect aims to implement this programming style. This aspect identifies also method calls that should always be valid or hold in a correct program.

Join-point(s) for pre-conditions:

- assignment statements for parameters passed to method calls

For post-conditions:

- values returned to the caller,
- defined values for local variables or class attributes within local computations

Sample code:

```
aspect PointBoundsChecking {
 pointcut setX(int x): (call(void FigureElement.setXY(int,int))&&
 args(x, *)) ||  (call(void Point.setX(int)) && args(x));
```

```
 pointcut setY(int y): (call(void FigureElement.setXY(int, int))  &&
 args(*, y)) ||
 (call(void Point.setY(int)) &&    args(y));
 before(int x): setX(x) {
   if ( x < MIN_X || x > MAX_X ) throw new
     IllegalArgumentException("x is out of bounds.");
  }
 before(int y): setY(y) {
     if ( y < MIN_Y || y > MAX_Y ) throw
      new IllegalArgumentException("y is out of bounds.");
  }
}
```

- PRODUCTION ASPECTS

Name: Change monitoring.

Description:

When explicit implementation of a functionality turns out to be problematic, the use of an aspect can make it easier. More explicitly, it is characteristic of this aspect to contain code meant to indicate changes on an observed entity(object, method or field) in our code.

Join-point(s):

Method calls that are spread across different classes whose result is the change we desire to monitor.

Sample code:

```
aspect MoveTracking {
 private static boolean changed = false;
 public static boolean testAndClear() {
   boolean result = changed;
   changed = false;
   return result;
 }
pointcut move(): call(changed()) && within(*.Poly*Figure) || within(*.Text*Figure)
              || within(CompositeFigure)|| within(AbstractFigure);
after() returning: move() { changed = true; }
```

}

Name: Context passing.

Description:

When certain context information has to be transferred from one object to another, instead of having to re-implement the necessary methods with (an) extra parameter(s), it is a cleaner solution to define an aspect that captures this information flow.

Join-point(s):

Method calls along the flow stream that pass context information from one object to another. For example, the context information when changing the color value of a Figure object from a figure editor client to the method responsible for doing the change of color in a Figure.

Sample code:

```
aspect ColorControl {
  pointcut CCClientCflow(ColorControllingClient client):
cflow(call(* * (..)) && target(client));
  pointcut make(): call(FigureElement Figure.make*(..));
  after (ColorControllingClient c) returning (FigureElement fe):
  make() && CCClientCflow(c) {
      fe.setColor(c.colorFor(fe));
  }
```

```
}
```

Name: Providing consistent behavior.
Description:

These are aspects meant to implement methods of a given class -or package- whose behavior is shared. As example, consider the methods that handle different kinds of elements to be parsed. Such methods are declared in a JavaParser class, and they are:

```
parseMethodDec, parseThrows, and parseExpr
```

join points:

Methods returning the same type of objects, thus feasible to be captured by a common point-cut, as long as the behaviour of the methods is commonly shared as well.

Sample code:

```
aspect ContextFilling {
  pointcut parse(JavaParser jp):
      call(* JavaParser.parse*(..)) && target(jp)&&
     !call(Stmt parseVarDec(boolean)); // var decs  are tricky
    around(JavaParser jp) returns ASTObject: parse(jp) {
     Token beginToken = jp.peekToken();
     ASTObject ret = proceed(jp);
     if (ret != null) jp.addContext(ret, beginToken);
    return ret;
   }
}
```

1.4. Existing aspect mining tools

In this section we present an overview of currently available tools to assist us in the aspect mining task.

1.4.1. Aspect Browser. The Aspect Browser (AB) is a text-matching based program for finding cross-cutting concerns in code [GKY00]. It allows to visualize the results of a search via its Nebulous interface, which is its visualization module. In order to perform a search, the user types the expression he/she is interested in, and this expression is added to a list called aspect index. The representation of files matching the search expression is done in a window, where each source file is represented as a column (representing a directory) made out of color strips. Once a matching pattern is found, AB assigns a color to distinguish it from other cross-cutting concerns that the user may define. AB allows then to navigate through the source files in order to assist the user in the visualization of the scattered code. By double clicking on the color strips, a window with source code colored is displayed so one is able to see in what line of code the aspect in question appears.

As a step towards an AOP (re)implementation, AB provides a command to find all the redundant lines in the source code. Once such a list is displayed, we can select them and add these lines as aspects if we want. The result of this action is that the selected expression is inserted in the aspect index list.

1.4.2. The Aspect Mining Tool (AMT) . Hannemann and Kiczales [HK01] created another tool for aspect mining that performs text based and type based analysis. The extraction of aspects is divided in two phases:

(1) *Data Extraction (AMTAnalyzer).*To generate the type information for the source code to examine, it is necessary that the whole project is analyzed

simultaneously. The extracted data will be written into a single file. The following step is to call the visualizer to display the source code:

(2) *Data Visualization (AMTVisualizer).* The representation of the extracted information is similar to presented in the Aspect Browser tool (see section 1.4.1). A small difference regarding AB, is that when the user defines several expressions for queries over the source code, the relevant lines of code matching those criteria will have two or more colored parts, each color corresponding to each query expression.

From the results of their research, Hannemann and Kiczales pointed the following advantages regarding the mining methods they employed in the AMT tool:

- *Text based analysis* works better for many instances of the same types, where strict naming conventions are used. A disadvantage in this technique occurs when such naming conventions are not -completely- observed.
- *Type based analysis* yields better results with many similarly name objects of different types, neglecting naming conventions. Its corresponding disadvantage is when there are many instances of the same type used for different purposes.

1.4.3. Multi-Visualizer (AMTEX). Multi-visualizer [ZJ03] was developed at the University of Toronto, and is an extension of the visualization functionality of AMT (see section 1.4.2). It is designed to provide more powerful and easy-to-use management functionality for aspect mining of large projects, typically consisting of hundreds or thousands of classes.

The mechanism of the multi-Visualizer is profile-based mining. A mining profile can be considered as an analysis task for a certain interest. It consists of a target, a pattern and a metric. By further defining those sub-elements of a profile, it is quite flexible to perform a large variety of mining tasks. The following paragraphs explains those sub-elements in detail.

- *Target*

A target is a subset of all classes that a software system under investigation contains. A profile can use the concept of target to divide the large code space into smaller, more relevant groups to better capture results. In Java, natural lines of such divisions are packages. In AMTEX, a target is defined as an inclusion of the set packages or classes that are under inspection, together with an exclusion, a set of packages or classes that are not of interest. By default, the inclusion contains all the classes in the application.

- *Pattern*

A typical mining task in the context of AMTEX is to find recurrences of a particular usage of certain class types or regular expressions. Those class types and regular expressions can be grouped together to form a particular usage pattern such that, if any of the types or regular expressions is found in a class implementation, that class is added to a results collection.

- *Metric*

A metric in AMTEX defines the purpose of the mining activity. Three types of metrics are supported:

(1) Collection of pattern matching class types,
(2) Coupling index: Coupling indicates how much the types in the system are related to each other. Good architecture is always less coupled; and,
(3) Ranking of indicated number of classes that are most heavily used in the application. All these metrics can be collected separately or together in one profile.

1.4.4. Feature Exploration and Analysis Tool (FEAT) . Robillard and Murphy created a tool based upon the ideas of structural program model and concern graph [RM02]. The authors mention in [RM02, p.2.] that scattered code in a software system can be identified with the aid of a representation of the program (the structural program model) and derive from it concern graphs. The program model represents the declaration and usage of various program elements (such as classes, methods, and fields) of object-oriented languages; such representation consists of a graph where vertexes can be of any of the following types.

(1) *Class vertex,* representing a class, considered without its members;
(2) *Field vertex*, a field member of a class; and
(3) *Method vertex*, which represents a member method of a class.

Concern graphs are "abstract representations of the implementation details of a concern and makes explicit the relationship between different parts of the concern"[RM02].

Both structural program model and concern graphs are introduced arguing that the existing helps (like lexical analysis tools, code browsers, slicers, etcetera) for identifying scattered concern code are difficult to use as a basis for reasoning and analysis of such concerns. Therefore, FEAT has been designed and implemented to address the previously mentioned difficulties, with the following functionalities:

- *Displaying a concern graph:* Graphs are visualized as tree views. The reasons to represent them in this way, is that trees are easier to layout than graphs, and that the tree nodes can be collapsed to abstract information.
- *Accessing the program model:* FEAT provides a set of queries to enable users to access vertexes of the program model that are related to the vertexes in the Concern Graph. A user can navigate the program model in both the direct and reverse directions of the edges emanating from the vertexes.
- *Mapping to source code:* FEAT permits a user to view the source code corresponding to a concern graph element. To access the source code, the user selects an element in the tree view and uses a pop-up menu to request visualization of the code.
- *Manipulating the program model:* FEAT also allows to manipulate elements of the concern graph or query results. As example, a user can remove an element from a tree view, move an element from a query result window to the main concern window, or highlight an element. A user may also compare the results of a query to determine which elements are already part of the (cross-cutting) concern.

There are however, certain limitations when using FEAT: the first is that analysis of exception handling is not supported [RM02, p. 4]. Moreover, FEAT does not support the mining of cross-cutting concerns like object protocols, or pre- and post-conditions. These drawbacks are compensated with direct access to the source code implementing a cross-cutting concern. Lastly, FEAT cannot extract *call* edges implemented through reflection nor detect relationships present in the source code, which are lost due to compiler optimizations (like method in-lining).

1.4.5. QJBrowser. QJBrowser [RV02] was developed by Rajagopalan and De Volder at the University of British Columbia. It is a query-based code browser tool, and is claimed it can contribute to find aspects, because its design allows the creation of queries whose structure permit to define relationships among specific elements in Java code. Creators of QJBrowser argue that presenting existing source code in different views to developers, can support them indirectly to identify and manipulate cross-cutting concerns.

QJBrowser includes Tyruba or Sicstus Prolog as the query languages in which queries to the source code can be performed. Results of the queries are displayed in hierarchically organized views. In order to analyze and browse the code under examination, it is necessary to define a meta-model, that consists of the following parts:

- *a selection criterion:* which determines what elements will form the browser's view. QJBrowser uses a source model, that consists of logic facts extracted via static analysis. Static analysis is done by QJBrowser using a modified version of the AspectJ type checker. This static analysis is performed once we indicate to QJBrowser where the source code in question is.
- *an organization criterion:* which specifies how the result of the query is represented in the browser. The organization criterion defines a way to project the (multi)dimensional space that the selection criterion produces onto a tree.

1.4.6. JQuery. JQuery [Vol02] is the direct successor of the QJBrowser, discussed in section 1.4.5. Like its predecessor, JQuery builds a tree structure based on the logical queries made to the source code under exploration. The novelty is that the obtained tree serves as starting point from which new trees can be created and refined to extend the exploration process. This feature allows the flexibility to avoid switch views across different units of the code base.

The graphic user interface of JQuery is implemented as a plug-in for the Eclipse© platform. Like in QJBrowser, JQuery uses as query language TyRuBa, which includes a library with predefined predicates to support users unfamiliar with the language or unwilling to compose (complicated) queries. Another convenient feature for the user who gets expertise in the use of JQuery, is the possibility to customize it. This is possible by a configuration mechanism that allows to define new queries to express different relationships between the units of code under examination. For further details about the customization of JQuery, we refer the interested reader to section 5.1 of [Vol02].

1.4.7. Ongoing related work in aspect mining.

1.4.7.1. *Concern Manipulation Environment.* The Eclipse project has become a huge development platform with many extensions (many of them known as plug-ins) that suit for diverse development needs.

A special extension to support aspect-oriented software development (AOSD) is the Concern Manipulation Environment[6] (CME), which is an open suite of tools meant to support the full software lifecycle, from inception until release(s).

The CME is at the same time a framework to integrate and build AOSD tools and technologies. Precisely in this context, CME wants to provide support for (cross-cutting) concerns extraction from existing object-oriented Java software.

At the moment of writing this document, there are not available releases of the CME framework yet.

1.4.7.2. *Ophir.* Another recent development is the approach presented by Shepherd and Pollock in the Ophir project [SP03]. Having as aim the automated detection of aspects, the code to be examined is processed into a program dependence graph representation in order to detect code clones. Once the aspect candidates are identified and selected, they are merged (i.e., coalesced) into classes, with the purpose of consider such classes for eventual (automatic) refactoring of the source code at hand. Promising as it looks, by now Ophir's results are in the validation

[6] `http://www.eclipse.org/cme/`

phase and therefore is not included as an already available tool, like those explained in the previous subsections of this chapter.

1.4.7.3. *Fan-in analysis.* Fan-in analysis is an aspect mining technique proposed by Marin et al. [MvDM04], at the Software Evolution Research Lab in Delft University of Technology[7]. A more detailed explanation of fan-in and the benefits of combining it with clone detection are discussed in chapter 4.

1.4.8. Other approaches. Besides the tools discussed in the previous subsections, there are other methods related to (automated) extraction of aspects. One of these methods is the one investigated by Breu and Krinke [BK03], in which they use dynamic analysis to find aspects; aspects are represented as sequences of method calls that exist in the same order but in different contexts. The specific technique used to achieve aspect identification is tracing.

Lastly, there are other general tools available for Java code analysis, which to some extent provide help towards aspect detection: Jlint, Shimba, RevJava [8], the Eclipse-Hyades project[9] and QAStudio[10]. Here we must consider also other tools like Jcosmo[11], which is a tool created to detect code smells. Smells in source code are those fragments of code that could benefit of refactoring, mentioned in section 1.2.

1.5. Outline of this document

1.5.1. Problem statement. In this first chapter we have covered the details about software evolution, aspect oriented programming, software exploration and its relationship with aspect mining, together with an overview of the most representative existing aspect mining techniques and tools.

As we noticed in the section 1.4, existing techniques for aspect mining require from their users to be highly acquainted with the software in which they want to find aspects. This situation is undesirable if we consider the fact that software systems evolve increasing at the same time the complexity to maintain them -recall sub-section 1.2.

A better solution should aim to reduce the cognitive burden of users regarding the software they want to explore by providing them with a technique or method which would allow to obtain meaningful information in the search for aspects, with a minimal of effort.

Thus, summarising, *in order to achieve software evolution's goal, the migration from an object oriented system into an aspect oriented one, appears to be a good solution. Such migration however, requires finding first those features in the system that are (inherently) spread and/or tangled with the core concern of the system, and are therefore difficult to encapsulate in an object oriented manner. To find those cross-cutting concerns in a way as automated as possible, is the purpose of this research project.*

In order to solve this problem, there are therefore questions whose answers we believe worthy to investigate:

(1) Which are suitable candidates for automated aspect detection?
(2) What principles, rules or techniques can we apply for an automated detection of aspects?

[7]SWERL homepage: `http://swerl.tudelft.nl`
[8]`http://www.serc.nl/people/florijn/work/designchecking/RevJava.htm`
[9]http://www.eclipse.org/hyades/
[10]http://www.qa-systems.com/products/qstudioforjava/index.html
[11]`www.cwi.nl/projects/renovate/javaQA/intro.html`

(3) Once successfully answered the previous question, can these principles, rules or techniques be implemented in a tool?
(4) How can mined aspects be represented in an useful manner for the users interested in them?
(5) Could we apply with satisfactory results these aspect mining methods in software systems regardless of their size?

1.5.2. Proposed approach. In this thesis, we investigate the use of clone detection as a technique to achieve (automatic) aspect mining.

As clone detection will be the direction in which we will concentrate efforts, the techniques related to it are discussed in detail in chapter two. In chapter two we will present also some of the tools available for clone detection, together with the description (and justification) of the tool we selected to aid us in our cross-cutting concern mining.

1.5.3. Validation. We will present in chapter three the method we applied to search for aspects, and to validate our results, we will use as case study JHotDraw, a graphical application framework. In chapter four we will introduce another mining technique: fan-in analysis, and the benefits of combining it with clone detection. Finally, in chapter five we will present conclusions about this project and suggestions regarding future work in the interesting subject of aspect mining .

CHAPTER 2

CLONE DETECTION TECHNIQUES

2.1. Introduction

Duplicated code exists due to the habit of developers to repeat the solution they have implemented elsewhere when programming a system. The result of repeating code is most of the times adverse because we can adapt the copied code to meet our needs at hand, but due to many factors (e.g., pressure to meet a deadline, bad programming habits, mere carelessness when adapting the code, etc.) the maintenability and evolvability of the system is negatively affected.

We have then, that negative consequences of duplicating code are:

- Errors are duplicated together with the duplicated code, causing thus double amount of effort to correct code [Kri01].
- The software engineering principle of *encapsulation* is broken [BYM+98].

Clone detection can be used as a technique for aspect mining, since (portions of) cloned code can be considered as seeds to mine candidate aspects. A seed in the context of aspect mining consists then of "*The identification of [...] a method, interface or group of statements [that are] part of the concern's implementation*" [MvDM04, p. 2]. In the remainder of this chapter we will discuss how different approaches for clone detection work.

2.2. Clone detection techniques

Clone detection is a two phases process:

(1) *Transformation phase:* Here the clone detector takes care to adapt the input source code into an internal format for further treatment. This transformation may vary from simple white space and line break removing to the construction of a parse tree.

(2) *Comparison phase:* Comparison consists of matching the previously transformed entities to yield detected pairs of cloned code. The output of this matching phase consists normally of a file listing the clone pairs detected.

The coming subsections describe in more detail the way some clone detection approaches work.

2.2.1. String based. In [RD03], string based detection is classified in two variants:

(1) *Simple line matching.* This variant removes empty lines and white spaces during its transformation phase and during comparison all lines are compared with each other using a string comparison algorithm. To avoid intractable calculations, strings are hashed into buckets, and afterward, pairs in the same bucket are compared.

(2) *Parameterized line matching.* In this approach, elements in code like identifiers might be different in a clone pair and therefore they are considered as *changeable parameters*, hence the name of the approach: these changeable parameters are normally replaced by a common symbol identifier, during the transformation phase.

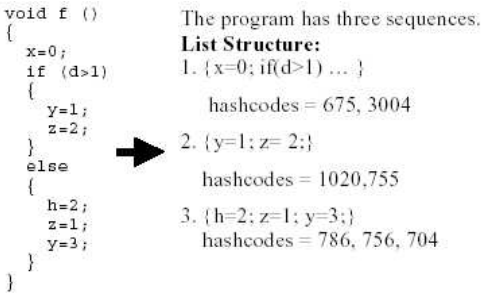

FIGURE 2.2.1. List structure example (Id. as figure 3 of [BYM+98])

2.2.2. Token based. Another approach for clone detection is based on tokens [KKI02]. The technique is as follows:

(1) The entire code base is converted into tokens according to the lexical rules of the programming language. This step is done by using a lexical analyzer.
(2) In the transformation phase, identifiers (such as variables, constants and types) are replaced by a special token (say '$p'). Also transformation rules specific to the programming language are applied; as an example in section 2.3 of [KKI02], a transformation rule is given to remove accessibility keywords:
    `protected Polygon getPolygon()`
    is transformed into:
    `Polygon getPolygon()`
    This yields a transformed token sequence of the code.
(3) From all the sub-strings that form the transformed token sequence, matching pairs are detected and each clone pair's location is converted into line numbers on the original source files.

2.2.3. Parse-tree based. Parsing is a powerful technique for clone detection in as much as it provides a complete syntactical analysis of the source code to explore.

The procedure according to [BYM+98] consists of the following steps:

(1) *Parse the source code.* The result of this is an abstract syntax tree (AST) with the syntactical representation of the program at hand.
(2) *Detect sub-tree clones.* To achieve this, sub-trees are categorized with hash values. Thus, a sub-tree A is considered to be a clone of a sub-tree B if they have the same hash value.
(3) Finally, another algorithm detects statement and declaration sequence clones. To achieve this detection, a list associated with a sequence in the program stores the previously computed hash values for each subtree member of the corresponding sequence, as illustrated in figure 2.2.1

    The result of this last algorithm is that larger clones subsume smaller ones, and atomic cloned statements are grouped in sequences.

2.2.4. Program dependency graph based. In [KH01] a tool is implemented to find clones in C code using the program dependency graph (PDG) of each procedure. A PDG is a representation of the program where nodes represent program statements and predicates, and edges represent data and control flow. The

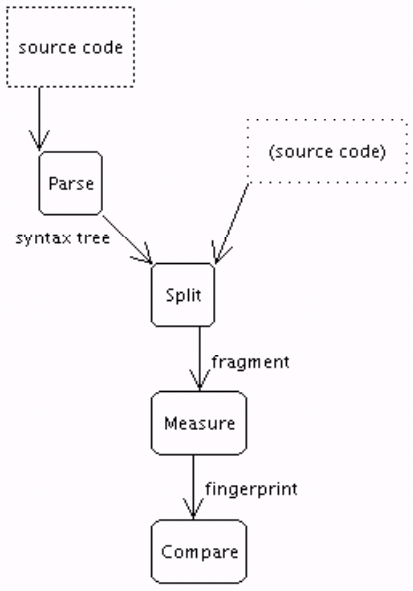

FIGURE 2.2.2. Clone detection based on fingerprint metrics (Cf. Figure 1 of [RD03])

advantage of clone detection based on PDG's is that clones with reordered statements or interwined with each other can be detected; such a type of clones are generally known as near-miss - as in section 1 of [BYM+98] - or simply near clones.

Another example of clone detection using PDG's is shown in [Kri01], where a specialized version of PDG's called fine-grained PDG is introduced. Besides the data flow (also known as value dependence) edges, a graph also contains edges mapping the control dependence and the reference dependence (i.e., computed values stored in variables) among nodes; all these nodes represent expression components.

2.2.5. Fingerprint metrics based. This approach uses software metrics in source code as criteria to detect whether two pieces of code are clones or not. Figure 2.2.2 shows the procedure which can be divided in these steps:

(1) *Parsing.* Source code is parsed and a big AST (as in subsection 2.2.3) is obtained.
(2) *Splitting.* The AST needs to be split into fragments; typically such fragments are methods or scope blocks since these can be easily extracted from the AST.
(3) *Measuring.* A predefined set of metrics (lines of code, cyclomatic complexity, etc.) are used to measure the previously obtained AST fragments, obtaining thus a fingerprint measure per fragment.
(4) *Comparing.* An approach that can be used to compare a pair of fingerprints is to calculate their Euclidean distance, and fragments with zero distance are then considered as clones.

2.2.6. Evaluation of the discussed techniques. The previously exposed detection techniques have their corresponding advantages and disadvantages. To try to cover all these issues here would require a thorough analysis which for the sake of conciseness, we will not do in this thesis. However, figure 2.2.3 summarises an evaluation - based on [RD03], [BYM+98] and [KH01] - of the clone detection

Criteria	Technique				
	String based	**Token Based**	**Parse Tree**	**Program dependency**	**Fingerprint metrics**
Language Independence	truly language independent	possible replacing the lexer	possible replacing the parser	possible replacing the parser and semantic analyzer (but difficult)	possible replacing the parser (but difficult)
Precision **1)Number of false positives** **2) Number of useless matches** **3) Number of recognisable matches**	1) null 2)many 3)lowest	1)low 2)low 3)lower than fingerprint based	1) 2)null/low 3)high	1) 2)low 3)high	1) depends on the way expressions are characterised and the length of fragments 2) more than token based 3) high
Granularity	equal continuous lines	duplicated symbol/lines neglecting functional blocks	AST nodes	PDG nodes	functional (methods or basic blocks)
Performance	See figure 2.2.4				

Figure 2.2.3. Summarised evaluation of clone detection techniques

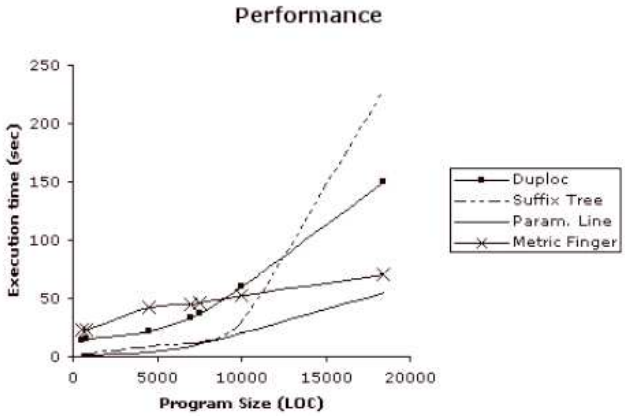

Figure 2.2.4. Performance of the different clone detection techniques (Id. as Figure 4 of [RD03])

techniques previously described, in terms of their performance, granularity, precision and language independence. Precision is explained as follows:

(1) **Number of false positives**. The number of matches incorrectly identified as a clone.
(2) **Useless matches.** The number of matches which are not worth to refactor, typically depending on the length of a match.

(3) **Recognisable matches**. The number of matches that are recognised as interesting, for instance, matches suitable for refactoring.

These three elements of precision in a clone detection technique, have direct impact on the precision and recall used as evaluation criteria for a clone detection tool.

However, as suggested by [WJL+03] (in section 9), a clone detection tool can also be evaluated in terms of human expertise, by judging the specific and overall reliability of clone detection tools with the aid of reference sets. These reference sets are basically data sets created by human experts (in this case, experts in cloned code identification) by which automated (clone detection) tool results can be evaluated. To generate reference sets for evaluation of clone detection tools is not easy though.

2.3. Clone detection tools

2.3.1. Overview. Several clone detection tools are available in order to perform our search of clones. The reason to use an existing clone detector rather than build one from scratch is the fact that we can take advantage of what others have implemented provided it fits our research purposes. Here we will only consider those tools able to process Java code.

2.3.1.1. *Duploc.* Duploc[1] is a clone detector based on the VisualWorks[2] framework. It is a lexical-based detector, which compares source code line-by-line and displays a two-dimensional comparison matrix. Because of the intrinsic limitations of its underlying detection technique many clones are not detected, specially if these are near-miss type (mentioned in sub-section 2.2.4).

2.3.1.2. *JPlag.* *JPlag*[3] is a tokenised substring matching detector with the purpose of detecting plagiarism in programs. Tokenised (sub)strings are compared to detect the percentage of matching tokens and based upon such percentage, a similarity value is determined.

Since JPlag is implemented with plagiarism detection in mind, it does not detect clones within files, in the assumption that the author of a single file of code is expected to be the same. Therefore, clones may remain undetected in the source code at hand.

2.3.1.3. *Moss.* *Moss* is an on-line service to detect clones intended also to detect program plagiarism. The technique used in Moss is not described by its creator A. Aiken, as cheating detection techniques "can be fooled if one knows how they work"[4]. However, the same as JPlag, Moss does not detect clones within files, potentially leaving clones undetected and reducing the overall accuracy of the detection process. The same as with *Jplag,* it remains to be seen whether this missing detection inside single files, will be indeed a problem for aspect mining using clone detection.

2.3.1.4. *CCFinder.* *CCFinder*[5] uses a transformation followed by token matching algorithm to detect clones. It is specially intended to be used with industrial-size source code projects. The user can specify the length of clones to detect (in terms of tokens) and besides the text file produced by the detection process, a graphical

[1] `http://www.iam.unibe.ch/~rieger/duploc/`
[2] `http://www.parcplace.com/vwnc/`
[3] `http://www.jplag.de/`
[4] section "How Does it Work?" in Moss site `http://www.cs.berkeley.edu/~moss/general/moss.html`
[5] `http://sel.ics.es.osaka-u.ac.jp/cdtools/index.html.en`

representation is available. Such graphical representation might be useful to map clone location, provided the size of the code at hand does not exceed 20 000 lines of code (LOC) for a better visualization. More detail of CCFinder is provided in sub-section 2.4.

2.3.1.5. *CloneDr.* CloneDr is a commercial clone detector using abstract syntax tree technology (described in section 2.2.3). Its precision (see section 2.4) is very high since all the clones it detects can be (automatically) factored out in a macro or subroutine. It seems however to be more oriented towards C code and we acknowledge not to have a hands-on experience with the tool.

2.4. The elected tool

Because there is a lack of a precise definition about what differentiates a clone from a non-clone, this lack can also cause trouble to evaluate the suitability of a given clone detector [WJL+03]. In our particular situation however, we want to choose a clone detector for the specific context of aspect mining.

Although in practice it is impossible (up to now) that a single tool can detect all clones in a software project, two important features are desirable in a clone detector, if we were to use it for aspect mining:

(1) *Precision*: defined in [BB02], precision is "the measure to the extent to which the number of clones identified by the tool are extraneous or irrelevant".
(2) *Recall*: also as defined in [BB02], it is "the measure to the extent to which the clones identified by the tool matches the clone base". A clone base is regarded as the total number of clones in the application being analysed.

In the coming sub-section we explain based upon these two previous features, why we have chosen to use CCFinder in our research project.

2.4.1. CCFinder's features. CCFinder [KKI02] fits to our purpose of detecting clones due to its satisfactory results in recall and precision (as shown in [BB02, section 4]), and its availability. One more attractive feature in CCFinder is its capacity to handle industrial-size source code, having in mind that we would like to have an aspect mining process/tool able to handle software projects regardless their size (as stated in section 1.5.1). Furthermore, another evident factor to choose CCFinder is that it detects clones in the language subject of our research, namely Java.

Important concepts to define in order to understand and use CCFinder's results are:

(1) **Clone relation.** A clone relation is defined as an equivalence relation (i.e., reflexive, transitive, and symmetric relation) on code portions. A clone relation holds between two code portions if (and only if) they are the same sequences.
(2) **Clone pair.** For a given clone relation, a pair of code portions is called clone pair if the clone relation holds between the portions. An example of clone pair is depicted in figure 2.4.2.
(3) **Clone class.** A clone class is a maximal set of code portions in which a clone relation holds between any pair of code portions. For example, suppose a file has the following 12 tokens: *a x y z b x y z c x y d*: We get the following three clone classes:
    C1. a x y z b x y z c x y d.
    C2. a x y z b x y z c x y d.
    C3. a x y z b x y z c x y d.

2.4.2. Clone detection process. The entire process of CCFinder's clone detecting technique consists of four steps (Summarized from [KKI02, section 2.3]):

(1) ***Lexical Analysis***. Each line of source files is divided into tokens corresponding to a lexical rule of the programming language. White spaces (line breaks, tabs and comments) are removed and sent to the formatting step to reconstruct the original source files.
(2) ***Transformation***. The token sequence is transformed with two subprocesses:
   (a) *Transformation by the transformation rules.* The token sequence is transformed, i.e., tokens are added, removed, or changed according to the transformation rules specified in figure 2.4.1.
   (b) *Parameter Replacement.* After the previous sub-step, each identifier related to types, variables, and constants is replaced with a special token. This replacement makes code portions with different variable names to become clone pairs.
(3) ***Match Detection***. From all the substrings on the transformed token sequence, equivalent pairs are detected as clone pairs. Each clone pair is represented as a quadruplet as shown in figure 2.4.2.
(4) ***Formatting***. Each location of clone pair is converted into line numbers on the original source files.

2.4.2.1. *Output results.* The output of CCFinder is a text file with the following sections:

- *Option section:* Including the version number of CCFinder, the language specification, etc.
- *Input files section:* Including the paths of the input source files.
- *Errors section:* Including locations at which the lexical analyzer reports some errors.
- *Clones section:* Including the maximal clone pairs (as shown in figure 2.4.2).

CCFinder results can be visualized through a graphical interface called Gemini [UKKI02], whose appearance is depicted in figure 2.4.3: Clones are shown as dots in a symmetric scatter plot -right side of the picture-, where every dot represents matched tokens whose clone counterpart is symmetrically located on the other side of the diagonal line. The left side of the picture depicts a list with the pair clones, whose notation is already shown in detail in picture 2.4.2.

2.5. Summary

In this chapter, we have looked at clone detection techniques and their functioning. Clone detection techniques vary in complexity and effectivity as they gather more information about the code they analyze: the simplest technique (line matching) only performs analysis based on lexical information and the more sophisticated ones (PDG or finger metrics based) are language semantics dependent and consume more computational resources (memory, time, etc.).

We have also made our election regarding a clone detection tool that may aid us in the mining of aspects. The tool, CCFinder, has provided satisfactory results and we consider it to be suitable for our needs. Its output format, however, a text file explained in sub-section 2.3.1.4 , still needs to be adapted so that we can usefully manipulate the obtained information in the next phases of the aspect mining process.

In the coming chapter, we will present our mining method applied to a case study.

Rule Number	Rule Description

RJ1 Remove package descriptions	**(PackageName'.')+ClassName → ClassName**
	e.g.,:
	`java.lang.Math.PI`
	is transformed into:
	`Math.PI`

RJ2 Supplement callees	**NDotOrNew NClassName'(' →NDotOrNew CalleeIdentifier'.'NClassName'('** By language specification a method is either an instance method or a class method. Therefore, if an instance calls a method without a callee instance or class then the omitted callee is the instance itself or a class of it.

RJ3 Remove initialization lists	'=' '{'**InitializationList**,'}' → '='{"}'
	']' '{'**InitializationList**,'}' → ']'{"}'
	e.g.,
	`return new int[]{1,2,3}`
	is transformed into:
	`return new int[]{}`

RJ4 Separate class definitions	**Insert UniqueIdentifier at each end of the top-level definitions and declaration.** This rule prevents extracting clone pairs of the code portions that begin at the middle of a class and end at the middle of another class definition.

RJ5 Remove accessibility keywords	**AccessibilityKeyword** →ϕ. Here,ϕ is a null sequence.
	e.g.,
	`protected void foo()`
	is transformed into:
	`void foo()`

RJ6 Convert to compound block	**Each single statement after *if ()*, *do*, *else*, *for ()*, *and while ()* is transformed into a compound block.** e.g.,
	`if (a==1) b=2;`
	is transformed into:
	`if (a==1) {b=2;}`

FIGURE 2.4.1. Transformation rules of CCFinder for Java code (Cf. table 2 of [KKI02])

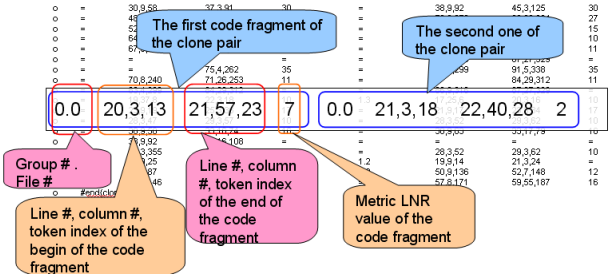

FIGURE 2.4.2. Clone pair code

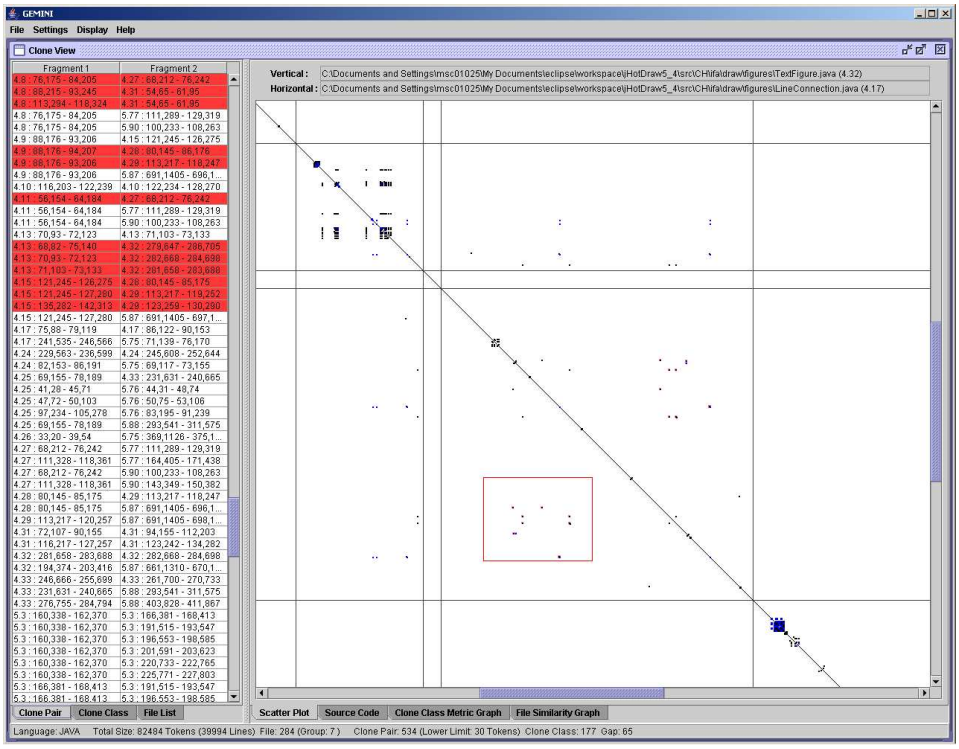

FIGURE 2.4.3. CCFinder's graphical interface: Gemini

CHAPTER 3

CASE STUDY: JHOTDRAW

3.1. Introduction

In section 1.5 of this thesis document we stated that in order to validate our research results, we will use open source systems. These suit well in our project, since they are freely available and therefore the discovered results can be shared and publicly debated.

The case study is JHotDraw, a graphic application framework. In the coming sections of this chapter we will discuss the results of aspect mining applied to JHotDraw, but before we would like to recall the notion of what an aspect is.

3.1.1. What is an aspect? (Revisited). Based upon the definition given in section 1.1.1, we can rephrase the notion of an aspect (or *cross-cutting concern*) as:

> A FEATURE OR PROPERTY OF THE SYSTEM WHICH CANNOT BE IMPLEMENTED IN A SINGLE COMPOSITIONAL UNIT, E.G., A PACKAGE, CLASS, OR METHOD.

In this section, in order to determine how to use clone detection to detect (candidate) aspects, we elaborate more the previous definition by posing the following question:

HOW CAN WE DETERMINE WHETHER A CLONE (CLASS) IS SEED OF A (CANDIDATE) ASPECT?

Any of the following criteria suffice to define whether a given clone (class) can be considered the seed of an aspect:

(1) Its code implements a functionality that does not fit well within the behavior (business logic) of the component it is in (i.e., tangled code). In many cases, such a clone is code that even can be extracted from the system, and yet the system is expected to work (e.g., logging), which suggests an aspect is *unpluggable* with regards to the system.
(2) As part of a (candidate) aspect, it *cross-cuts* the object oriented hierarchy in a system, i.e., it cannot be implemented in a single compositional unit (e.g., exception handling).
(3) In principle, a candidate aspect should be amendable to refactoring (preferably) into an aspect-oriented language. In this thesis document, we focus mainly but are not restricted to, aspects that can be expressed in the AspectJ language.

3.1.2. Aspect mining method. The mining method was set-up in the following way:

(1) Run CCFinder taking as input a list containing the names of source code files we are to analyze. We used CCFinder's default configuration of code clones with a length of 30 tokens (5 lines of source code in average) since a smaller number of tokens would produce an intractable amount of clones as observed in [BEvDT04].

(2) Select those clone classes (introduced in subsection 2.4.1, they are sets of cloned code with the same structure) with the highest values in metrics like class population (number of clones from the same type) and spreading ratio of the clones (i.e., in which files, directories or packages are the clones located). This step is discussed in more detail in section 3.2.3. Furthermore, it is desirable to have clone pairs grouped into classes since by doing so, we can identify easier the code that is a good seed to mine (candidate) aspects as these clone classes refer to code implementing in most of the cases the same functionality.
(3) Further inspect the code fragments fulfilling the previous features to find out what their function is within the code and what their dependencies are with regards to the elements (methods, attributes, classes, etc.) they interact with. This step is done with the aid of the FEAT tool described in sub-section 1.4.4.
Once selected the clone classes considered as most representative to contain seeds for cross-cutting concerns, for every clone class we have done manual inspection of the code by means of the Eclipse platform. We do so, because an important part of the analysis consists in finding the interactions between the code that the clone depends on (i.e., methods called within the clone's body) and the code depending on the clone (i.e., code that invokes the cloned code). These kinds of interaction in the source code are mainly explored using FEAT, and also by running JHotDraw within Eclipse in debug mode, so that we could see through use cases, how a given code clone was interacting with the rest of the code.
(4) Finally, having analyzed the code in the previous step, decide whether we found an aspect or not using our criteria discussed in the previous sub-section 3.1.1.

3.2. JHotDraw

JHotDraw[1] was setup as an exercise in the implementation of design patterns. In the coming subsections, we explain in more detail the architecture and aspects found in the JHotDraw project.

3.2.1. JHotDraw's architecture. Figure 3.2.1 shows the basic architecture of JHotDraw; a JHotDraw window is composed of a drawing view containing drawing(s) and the tools to create them. Each drawing is composed of one or more figures, each one owning handles that store information about the position of the owner figure.

We based our study case in JHotDraw version 5.4. This version contains 284 classes contained in 8 packages, 47 interfaces and a number of lines of code (LOC) above 18, 000.

3.2.2. Running CCFinder on JHotDraw. As we mentioned in subsection 3.1.2, we set the parameters of clone detection in CCFinder to 30 tokens as minimal length, and these are the results yielded when running CCFinder with JHotDraw's source code:

(1) *Execution time:* 30 seconds on a Pentium IV processor with 256 MB RAM (in graphical mode; this required more time for sorting/merging the pairs in the graphical layout). In a text mode, the execution takes no more than 2 seconds.

[1] `http://www.jhotdraw.org/`

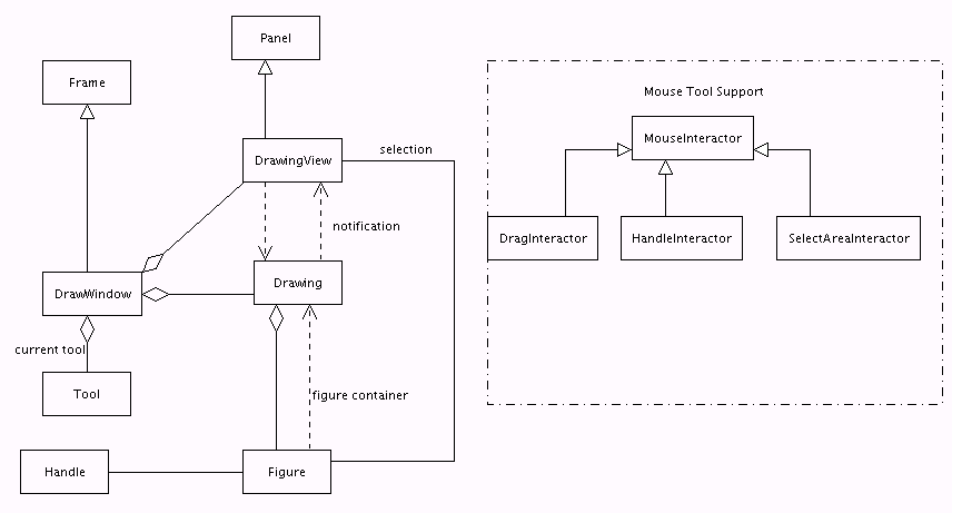

FIGURE 3.2.1. Architecture of JHotDraw version 5.4 (Cf. with `http://jhotdraw.sourceforge.net/online-docs/documentation/classdiagram.html`)

(2) *Longest pair clone detected:* **Bounds.java** (lines 388-417 : 418-447). This is a class with geometrical methods related to rectangles. Basically, this class is like java.awt.geom.Rectangle2D with some extra functionality.
(3) *Clone pairs:* 519.
(4) *Clone classes:* 174.
(5) *Largest population in a clone class*: 18. The largest clone class contained code duplicated is in the *createTools* method, discussed in subsection 3.2.6.3 which in the end did not lead to any aspect.
(6) *Highest Spread ratio*: 4. This metric value is observed for clone classes involving the cloned methods *undo* and *handles*.

3.2.3. Processing CCFinder's results. After running CCFinder on JHotDraw's source code, the step 2 mentioned in subsection 3.1.2, is mainly manual analysis performed as follows:

(1) We select first the clone classes fulfilling these two properties:
   (a) The population of the clone class is bigger or equal to 5, as we saw that a smaller number of elements in the clone class, did not profit in results interesting enough to lead us to find cross-cutting concerns. In most cases, clone classes with a small population, were subsumed by other larger clone classes. Figure 3.2.2 shows a distribution of the clone classes detected in JHotDraw.
   (b) The clone class has a spreading ratio of 2 or higher: by selecting the already established package division of the source code, we give priority to clone classes whose spreading ratio is "high", since spreading ratio of clones is a good indicator of a certain common functionality scattered and is also a distinctive characteristic of cross-cutting concerns. 3.2.2

   A visual aid to select the clone classes we are interested in is the clone class metrics graph generated by Gemini (the graphical interface of CCFinder), which is illustrated in figure 3.2.3, where the legends "RAD" and "POP" correspond to the spreading ratio and population respectively of the clone

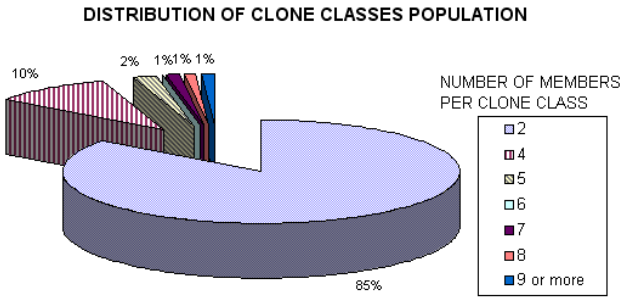

FIGURE 3.2.2. Distribution of clone classes population in JHotDraw

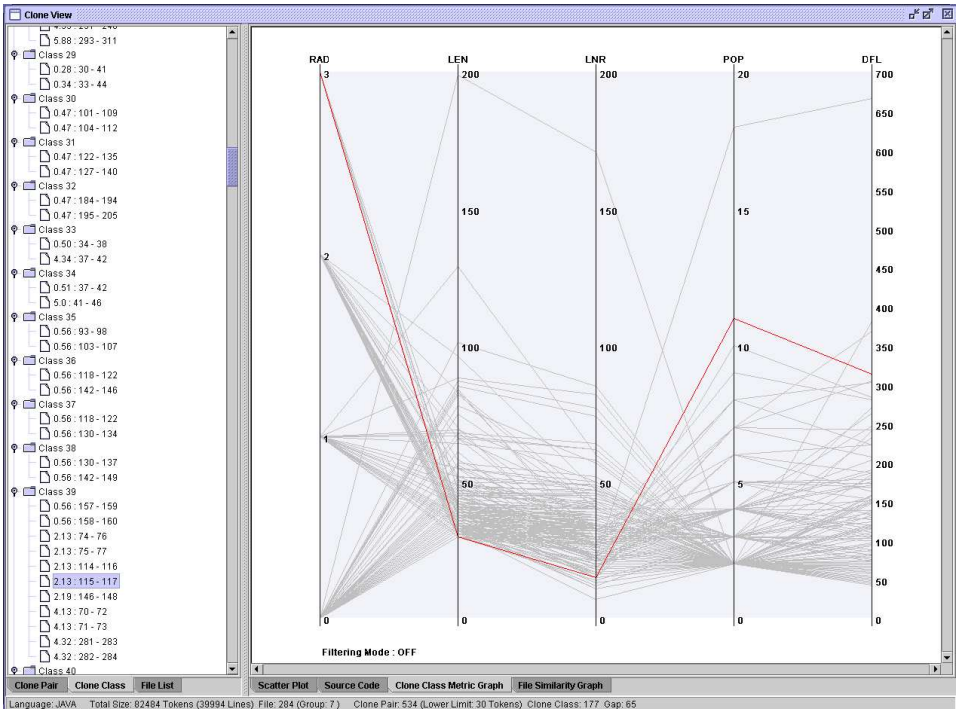

FIGURE 3.2.3. Clone class metric graph of JHotDraw source code

classes . Here we must to mention that because of the way CCFinder groups cloned code into classes (according to the clone class definition given in sub-section 2.4.2), there were many classes that subsumed or overlapped with others as we examined manually the code.

(2) For the rest of clone classes that were below the thresholds we set up in step 1, still we did the manual inspection of code for the sake of completion, but doing so proved to contribute in most of times to nothing new or relevant to what we already had found with the clone classes selected in the first step.
In this step we acknowledge that our methodology might be humanly error-prone, but there is a trade-off to make between investing more time in analysing code that is minimally duplicated (in the case of clone classes

package(s)	method	total number of declarations	occurrences detected as clones
figures, standard, contrib	draw	47	8
figure, samples, contrib	write	37	5
figures, contrib, samples, standard	handles	23	9
contrib, figures, standard, util	undo	27	5
application, contrib, samples	createtools	6	4

FIGURE 3.2.4. Methods in JHotDraw containing cloned used code as seed for aspect mining

with small populations), or go further in exploration of the clone sets that according to our criteria are more likely to lead us to aspects.

3.2.4. Redundant clone classes: the need of filtering. As mentioned in step 1 of the previous sub-section 3.2.3, we found that many of the clone classes created by CCFinder, were referencing to the same lines of code (i.e., these classes were overlapping or subsuming one another). Therefore, in order to make more efficient our code analysis, we need to filter the text file output from CCFinder: first using CCReformer, a program that does conversions or filtering for the clone data generated by CCFinder.

CCReformer is part of the set of utility programs that CCFinder comes with, and takes a clone data input file, converts it into a (clone) class format, and writes the result to an output file. However, this change in the format presentation of clone pairs *does not* remove clones referencing to the same lines of code.

The fact that we have (many) clone classes referencing to the same lines of code, complicates the mining task as the number of clone classes increases with the size of the software being mined. In this sub-section we introduce a transformation algorithm in order to reduce the number of clone classes yielded by CCReformer. The idea is to write into a file only non-redundant clone classes, since this is not done by CCFinder nor CCReformer. Specially, we have the case where a clone from class A is smaller (in terms of tokens) than a clone from class B, and both refer to the same lines of code :

$$\texttt{[clone x} \in \texttt{class A]} \subseteq \texttt{[clone y} \in \texttt{class B]}$$

Algorithm 1 shows the way we have dealt with this clone class redundancy. The total number of clone classes in JHotDraw given a minimum clone size of 30 tokens, was of 174, and by applying our filter, we were able to reduce the number of clone classes to analyze to 150, showing that 14% of the clone classes reported are redundant. The difference in this case is certainly small, but the utility of this filter became more evident in a larger system, namely the Catalina Server, which is part of the Tomcat[2] project, and whose size is around 90,000 LOC: from a total of 1034 clone classes detected, we were able to reduce the number of them to 707, thus reducing in 32% the number of clone classes to analyze.

3.2.5. Aspects found in JHotDraw. Figure 3.2.4 shows a summary of clones that were used as seeds to perform the aspect mining process. The analysis of the JHotDraw source code led us to the following results in our quest for aspects:

[2]Tomcat's website `http://jakarta.apache.org/tomcat/`

Algorithm 1 Transformation algorithm to remove redundant clone classes.

```
for index(0.. # clone classes){
 select beginning of clone class_i
 if (moreFound){
    selectClass();
    writeInfile ();
    deleteClass();
  }
 else{
   writeInFile(beginning position);
   deleteClass(beginning position);
  }
}
sub moreFound{
  for i (current_position .. #clone_classes) {
  if  (beginclone == clone_classes[i]){
     insert i in positions;
     }
 }
 return positions;
}
sub deleteClass{
 my pos;
 for pos (position_to_delete ..  # clone_classes){
  fileGroups[position_to_delete ] = ∅;
  if  (clone_classes [pos] eq "\#begin{set}")
     break;
  }
}
sub selectClass{
 for i ( 0 .. #main::positions ){
      count elements per class;
      insert count_class_i into sortedvalues;
 }
 sort  sortedvalues;
 writeInFile(sortedvalues[sortedvalues.length / 2 - 1]);
 deleteClass(begPos);
 for i( 0 .. sortedvalues.length)
     deleteClass(sortedvalues_i );
}
```

3.2.5.1. *Persistence.* Persistence is the property by which objects created by an application continue to exist and retain their values between one application execution and the next. Thus, persistence is a concern that is not necessarily vital for plotting drawings, which is the main concern of JHotDraw.

Persistence as an aspect in JHotDraw's code consists in the portions of code dealing with storage of drawings when using one of the possible JHotDraw's sample applications (mentioned in subsection 3.2.6.3). What is worth to mention is the fact that there is a *Storable* interface aimed "to flatten and resurrect objects"[3], and that the methods defined in this interface are declared across 36 different classes, which is a noticeable characteristic to claim that persistence is an aspect indeed.

EXAMPLE. A better way to explain how persistence is implemented, is describing what occurs when the user is saving a drawing containing among others, a *text area* type figure:

(1) A *save as* dialog box is opened to prompt the user to select the location and file name where she wants to store her drawing.

[3]JHotDraw API documentation

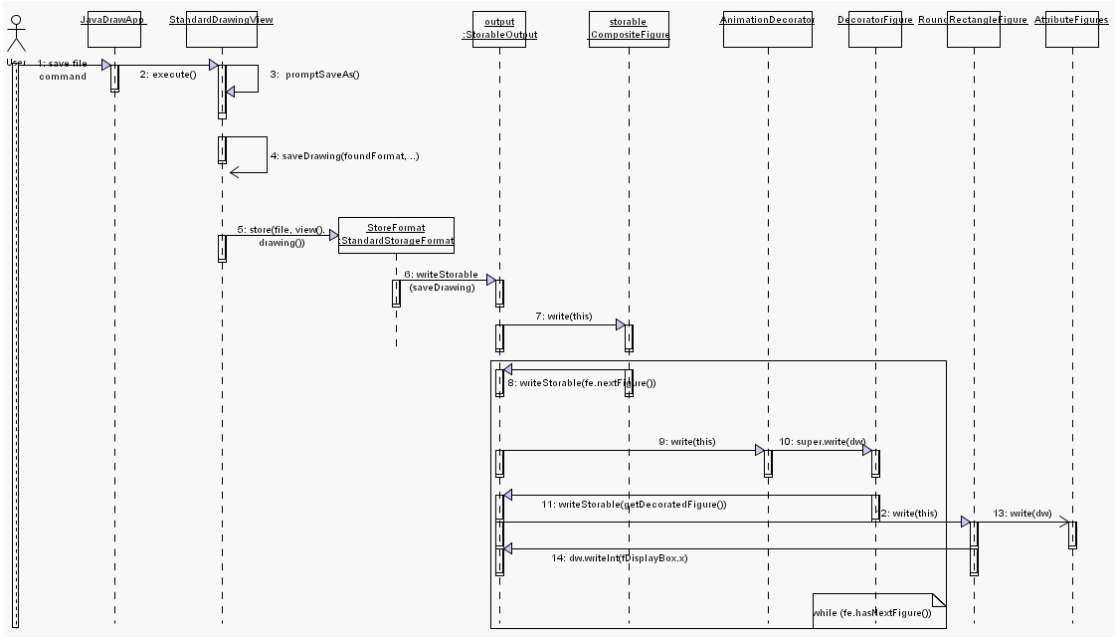

FIGURE 3.2.5. Saving a drawing in JHotDraw

Internally, the application is formatting the drawing by creating a *StandardStorageFormat* object, which contains both the name of the destination file and the drawing object to be saved. This format defines its own syntax and writes figure objects as plain text in a file with extension "draw".

(2) For every figure part the drawing (if the drawing is composed of multiple figures), a *StorableOutput* object writes its owner figure in the destination file. *StorableOutput* is a stream object that "flattens" *Storable* objects. Flattening means to simplify the structural information about something, in our case figures, preserving only their vital information.

In the particular case of a *text area* figure, the following information is stored:

- Its display box, which is a *Rectangle* object used as display for the figure. **This is the portion of code that was detected by CCFinder.**
- The text itself contained in the figure.
- Information about other figures connecting it, whether is a read-only figure, and its description.

(3) After saving all the elements of the drawing and closing the destination file, if there were no exceptions, the event sequence returns the control to the main application's window. The command "save as" that activated the storing process, returns the control to the user, and she can continue using the application.

END OF EXAMPLE.

The previous example is further illustrated with the sequence diagram of figure 3.2.5 . Summarising, when a drawing is to be saved in a file, the figures composing it are stored in terms of their attributes such as name, dimensions, and contents (in the case of text or image figures). The method identified by the clone detector

PertFigure.java

```
public void write(StorableOutput dw){
super.write(dw);
  dw.writeInt(fDisplayBox.x);
  dw.writeInt(fDisplayBox.y);
  dw.writeInt(fDisplayBox.width);
  dw.writeInt(fDisplayBox.height);
  writeTasks(dw,fPreTasks);
  writeTasks(dw, fPostTasks);
}
```

RectangleFigure.java

```
public void write(StorableOutput dw){
super.write(dw);
  dw.writeInt(fDisplayBox.x);
  dw.writeInt(fDisplayBox.y);
  dw.writeInt(fDisplayBox.width);
  dw.writeInt(fDisplayBox.height);
}
```

FIGURE 3.2.6. Seed method of the persistency aspect, in a clone pair view

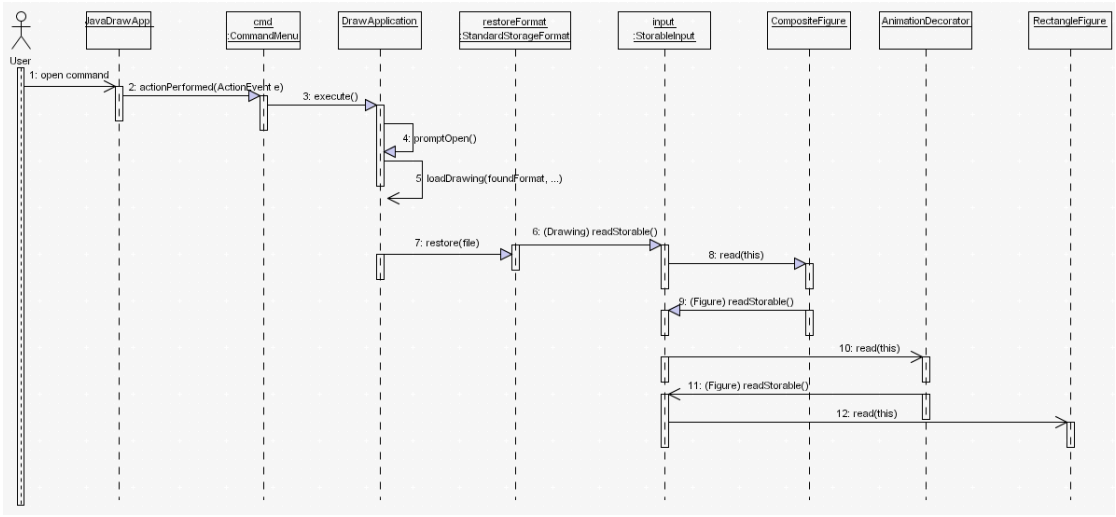

FIGURE 3.2.7. Opening a drawing in JHotDraw

```
public void read(StorableInput dr) throws IOException {
  super.read(dr);
  fDisplayBox = new Rectangle( dr.readInt(), dr.readInt(), dr.readInt(), dr.readInt());
  fFileName = dr.readString();
  Iconkit.instance().registerImage(fFileName);
}
```

FIGURE 3.2.8. Seed method of the resurrection aspect

is *public void write(StorableOutput dw)*, which writes the contained figure through a *StorableOutput* object. A sample of this method is in figure 3.2.6.

3.2.5.2. *Resurrection.* Like the previous aspect, when a drawing is stored in a file (files with extension .draw), code related to the concern of retrieving the figures is involved in the implementation of several classes. This aspect was found by inspecting the code related to the previous one (i.e., persistency), and deducing that any drawing made persistent, eventually will be opened again by the user. The *read(StorableInput dr)* method implementing part of this concern, had also 2 duplicated occurrences detected by CCFinder as shown in figure 3.2.8 and figure 3.2.7 shows a sequence diagram when opening a drawing .

CutCommand.java

```
public static class UndoActivity
  extends UndoableAdapter {
   private FigureTransferCommand myCommand;
   public UndoActivity(FigureTransferCommand
   newCommand) {
    super(newCommand.view());
    myCommand = newCommand;
    setUndoable(true);
    setRedoable(true);
}

public boolean undo() {
   if (super.undo() &&
   getAffectedFigures().hasNextFigure()) {
     getDrawingView().clearSelection();
     setAffectedFigures(myCommand.
     insertFigures(getAffectedFigures(),
                   0, 0));
     return true;
  }
  return false;
}
...
```

DeleteCommand.java

```
public static class UndoActivity
  extends UndoableAdapter {
   private FigureTransferCommand myCommand;
   public UndoActivity(FigureTransferCommand
   newCommand) {
    super(newCommand.view());
    myCommand = newCommand;
    setUndoable(true);
    setRedoable(true);
}

public boolean undo() {
   if (super.undo() &&
   getAffectedFigures().hasNextFigure()) {
     getDrawingView().clearSelection();
     setAffectedFigures(myCommand.
     insertFigures(getAffectedFigures(),
                   0, 0));
     return true;
  }
  return false;
}
...
```

FIGURE 3.2.9. Seed methods of the undoing aspect, in a clone pair view

3.2.5.3. *Undoing.* Twenty and four editing actions in JHotDraw can be undone, e.g., deleting a figure, grouping figures, etc. This functionality is expressed in the source code as a set of internal classes called *UndoActivity* which are spread through different classes involved with manipulating and drawing of figures. Figure 3.2.9 shows the constructor of *UndoActivity* and the *undo* method seen in a clone view.

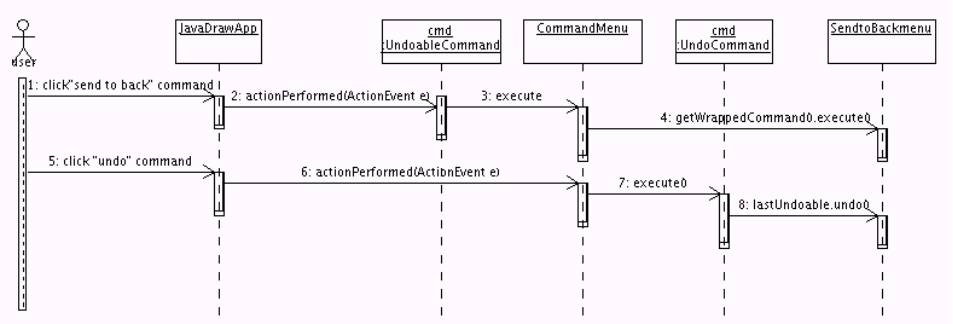

FIGURE 3.2.10. Undoing a *send to back* command in JHotDraw

UndoActivity is a class implemented in 24 classes:

- 7 related to figure handles (for triangle rotation, font size, figure resizing, etc.)
- 11 related to commands (paste, cut, delete, etc.), and
- 6 related to drawing tools (text, figure connection, figure border, etc.).

The superclass of all these declarations, *UndoableAdapter*, represents the most basic implementation for an undoable activity. *UndoActivity* declarations override methods to provide specialized behavior when necessary. For example, when the user performs a certain command editing the figures (e.g., change the size of a figure), the *UndoableAdapter* class will implement the behavior corresponding to an

undoable action for the user's command. So, if the user decides to undo her last action, the undo method in the affected object will return the drawing to the previous state of the drawing. Figure 3.2.10 illustrates the sequence of actions occurred when the user undoes a *send to back* command (i.e., when we want to send to the back a figure which is superimposed on another) in a figure in JHotDraw.

3.2.6. Candidate Aspects. The next subsections discuss two seed methods for which we do not feel confident enough to claim they have led us to find (new) aspects; however, we mention them here as candidates so that further research on aspect mining can either confirm or reject them as true aspects.

3.2.6.1. *Handling.* Handling is important in JHotDraw, since handles know their owning figure and they provide methods to locate the handle on the figure and to track changes. Seen as an aspect, the implementation of handling can be distinguished in 2 parts:

(1) **Its graphical representation**: When a figure is drawn, handles associated with it are created, which graphically are represented as a set of small squares, ovals or rectangles put normally in the corners of a figure selected by the user. In fact, this graphical representation was detected as cloned in the (candidate) aspect discussed next (i.e., drawing), as shown in figure 3.2.14.
(2) **Its functionality:** All the existing *handles()* methods in JHotDraw's code return a type-safe iterator of handles, and there is an important correlation between drawing and handling, since every time a figure is drawn or changed -enlarged, rotated, etc.-, its handles are refreshed (i.e., its information is updated).

We want to emphasize with these two features that their implementation is scattered across different figure classes, and it is noticeable that the behaviour implemented in the *handles()* method is to a great extent the same, regardless the figure it is declared in. A description of how handling operates is described next.

EXAMPLE. When the user selects a *TextFigure* object and moves it, a *StandardDrawingView* object responsible for rendering a drawing receives the user input and delegates it to the current tool, an *AnimationDecorator* object.

An update of the entire drawing occurs, by redrawing each of the figures contained in it. When our *TextFigure* is being re-drawn, because it is the current active figure, also its handles are re-drawn and next, information about it is updated:

- *AnimationDecorator* forwards a *handles()* call to *TextFigure.* A list is then created to store the figure's handles.
- For each of the figure's handles -in this example, four- added to the previously created list , a *NullHandle* object is created, holding a locator that specifies a point relative to the bounds of a figure. **These lines of the *handles()* method are the cloned code detected by CCFinder.** The relative points are the corners where the handles are graphically represented. It must be clear however, than depending on the type of figure currently active, different types of handles are created (e.g., radius handles for round rectangles, or group handles when several figures are grouped).
- After executing *TextFigure's handles()* method, the list containing the handles is returned to the *AnimatorDecorator* object, which in turn flows control back to *StandardDrawingView,* finalizing thus the updating strategy shown in this example.

END OF EXAMPLE.

Of course, in other types of figures, storing extra information in their handles is required. An example of this situation are *PertFigure* objects, where an extra handle is needed when two Pert diagrams are connected to one another. The extra handle will contain (besides the location data) the connection information to the other diagram, expressed in a new *PertDependency* object.

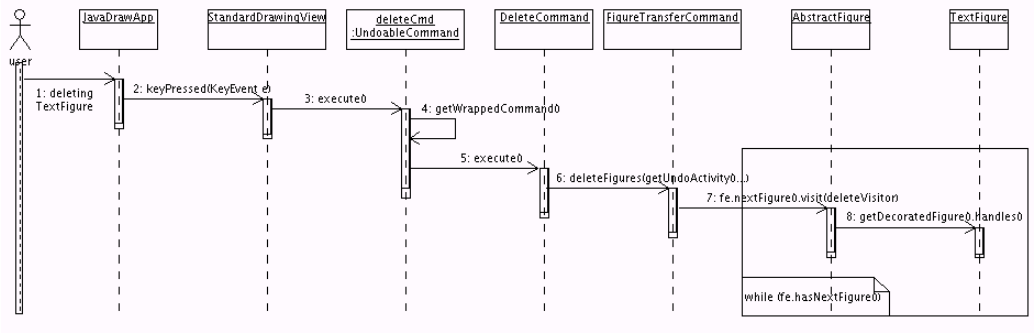

FIGURE 3.2.11. Handling in a sequence diagram

A clone pair of the method used as seed for the handling aspect is shown in figure 3.2.13, and figure 3.2.12 shows a qualitative analysis to consider handling as an aspect. Finally, figure 3.2.11 shows an example of handling when a user selects a text figure.

3.2.6.2. *Drawing.* In JHotDraw, the *public void draw(Graphics g)* method is responsible for the graphical representation of figures. This method is overridden according to the specific features of each figure (polygons, lines, text, etc.). An instance of the draw method used as seed is displayed in figure 3.2.14.

Drawing is the core concern of an application like JHotDraw, andbase as many as 8 declarations of the *draw*( ) method are cloned, which suggests it can be refactored. However, as described in another case study(es)[4] presentation, the benefit of refactoring would be little, since the methods involved in this aspect are well

[4]http:// `plg.uwaterloo.ca/~migod/846/project/ZhangDaiHolder-slides.ppt`

pros	*cons*
• The *handles* method is declared in a total of 23 classes • 9 (39%) of those declarations are cloned	• The classes containing this method are implementing interfaces whose functionality has 100% to deal with graphic manipulation. (Figure and more important, Handle).
• The *handles* method is called by the same methods in other classes.	• The kind of handles added to the iterator returned by the method are different: TriangleRotationHandle, NullHandle, etc.

FIGURE 3.2.12. Comparison table of handling as an aspect

TextFigure.java

```
public HandleEnumeration handles(){
 List handles = CollectionsFactory.
  current().createList();
 handles.add(new NullHandle(this,
 RelativeLocator.northWest()));
 handles.add(new NullHandle(this,
 RelativeLocator.northEast()));
 handles.add(new NullHandle(this,
 RelativeLocator.southEast()));
 handles.add(new FontSizeHandle(this,
 RelativeLocator.southWest()));
 return new HandleEnumerator(handles);
}
```

NodeFigure.java

```
public HandleEnumeration handles() {
   ...
 List handles = CollectionsFactory.
  current().createList();
 handles.add(new ConnectionHandle(this,
  RelativeLocator.east(), prototype));
 handles.add(new ConnectionHandle(this,
  RelativeLocator.west(), prototype));
   ...
 handles.add(new NullHandle(this,
  RelativeLocator.southEast()));
  handles.add(new NullHandle(this,
 RelativeLocator.southWest()));
 handles.add(new NullHandle(this,
  RelativeLocator.northEast()));
 handles.add(new NullHandle(this,
  RelativeLocator.northWest()));
 return new HandleEnumerator(handles);
}
```

FIGURE 3.2.13. Seed method of the handling aspect, in a clone pair view

RadiusHandle.java

```
public void draw(Graphics g) {
 Rectangle r = displayBox();
 g.setColor(Color.yellow);
 g.fillOval(r.x, r.y, r.width, r.height);
 g.setColor(Color.black);
 g.drawOval(r.x, r.y, r.width, r.height);
}
```

PolygonScaleHandle.java

```
public void draw(Graphics g) {
 Rectangle r =displayBox();
 g.setColor(Color.yellow);
 g.fillOval(r.x, r.y,r.width,r.height);
 g.setColor(Color.black);
 g.drawOval(r.x, r.y, r.width,r.height);
...
}
```

FIGURE 3.2.14. Seed method of the drawing aspect, in a clone pair view

pros	*cons*
• The method *draw(Graphics g)* is widely spread across packages and files(47 declarations)	• A draw method is considered to be well encapsulated within classes implementing a graphical functionality.
• 8 of those 47 declarations (17%) are cloned • The methods calling *draw(Graphics g)* are most of times the same.	• With exception of the clones, all the rest of methods are different among themselves according to the kind of figure they depict.

FIGURE 3.2.15. Comparison table of drawing as an aspect.

composed and encapsulated. In figure 3.2.15 we present a qualitative analysis of reasons to consider drawing as an aspect and in figure 3.2.16 we show how the

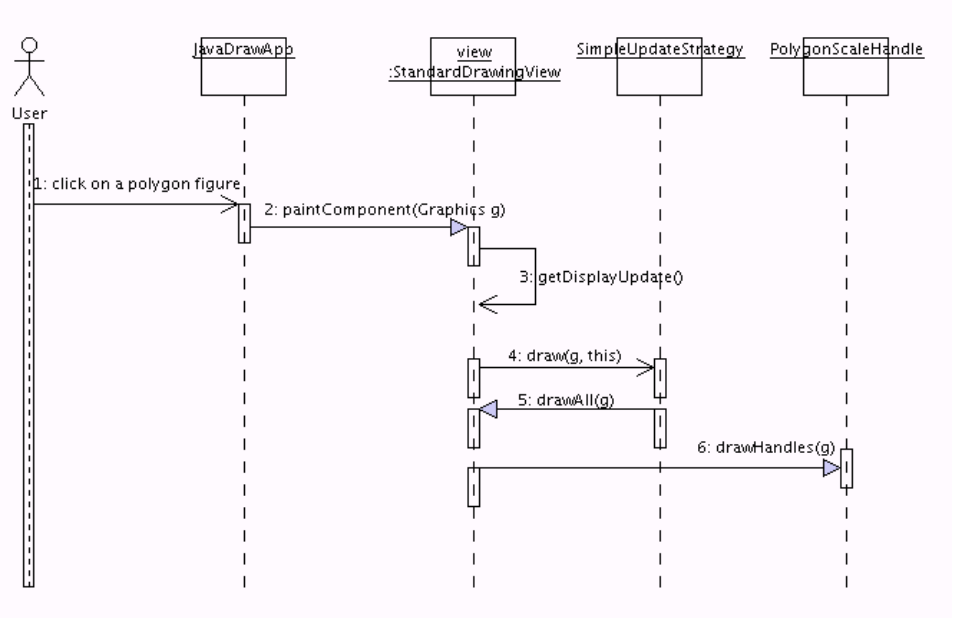

FIGURE 3.2.16. Sequence diagram when drawing a polygon's handles

graphical representation of a polygon's handles is done.

3.2.6.3. *A false positive.* JHotDraw comes with a set of sample applications to demonstrate its performance: three applets(like the one depicted in figure 3.2.17) and four stand-alone applications , and there is a *createTools(JPanel)* method which is invoked every time a sample application of these is to be created. Besides this specific case, *createTools* will not be declared elsewhere in the system. Based on this observation, we can say that this method does not lead to a specific aspect within the JHotDraw code.

In the particular case of the method *createTools*, such method is responsible for creating drawing tools within a Draw application, .i.e., tools to draw objects (like text, squares, polygons, etc.), and because of the several callings *createTools* does to the method *createToolButton(String, String, Tool)* when creating each type of tool, it was detected by the clone detector several times -see code snippet in figure 3.2.18. Although the refactoring of this method is feasible, it would not reduce the duplication caused by the invocation of *createToolButton* in each tool button created.

3.3. Towards refactoring

A way to ensure that we indeed have found an aspect, is to find out whether the cross-cutting concern code discovered with the help of seed can be refactored. In this section, we show to what extent some of the (candidate) aspects that were found, could be refactored in an AspectJ fashion.

3.3.1. Persistence. Based upon our observations of the control flow of methods implementing persistence, we notice that everytime a drawing is made persistent in JHotDraw, a call to the *StorableOutput.write()* method will take place.

Having this as starting point, we could write an aspect like the following to capture the behaviour expressed in the cloned code of the *write()* method, via inter-type declarations of the `fDisplayBox` field and a special method `write( )`:

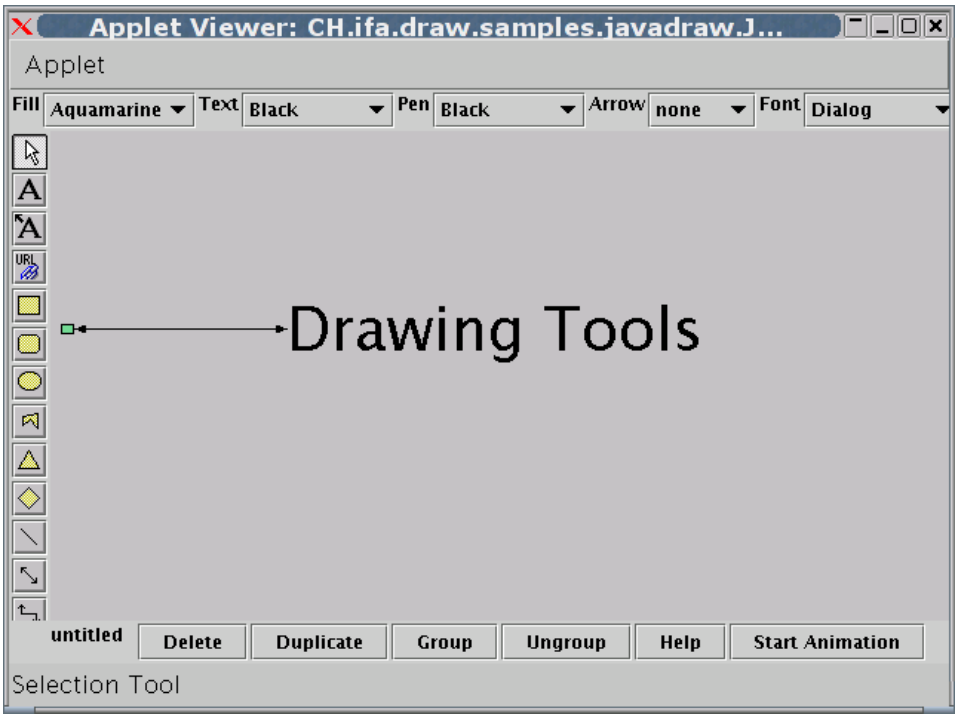

FIGURE 3.2.17. JHotDraw application in an applet version

```
protected void createTools(JPanel palette) {
super.createTools(palette);
Tool tool = new TextTool(this, new TextFigure());
palette.add(createToolButton(IMAGES + "TEXT", "Text Tool", tool));
tool = new ConnectedTextTool(this, new TextFigure());
palette.add(createToolButton(IMAGES + "ATEXT", "Connected Text Tool", tool));
tool = new URLTool(this);
palette.add(createToolButton(IMAGES + "URL", "URL Tool", tool));
}
```

FIGURE 3.2.18. Code portion of the *createTools* method

```
package CH.ifa.draw.contrib;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.figures.*;
import java.awt.*;
import CH.ifa.draw.util.*;
public interface Persistence {
   public void writes(StorableOutput dw);
  }
public aspect Rewriting {

   //declare an inter-type interface members
   private Rectangle Persistence.fDisplayBox;
   public void Persistence.writes(StorableOutput dw) {
       dw.writeInt(fDisplayBox.x);
       dw.writeInt(fDisplayBox.y);
       dw.writeInt(fDisplayBox.width);
       dw.writeInt(fDisplayBox.height);
   }
  declare parents : (RectangleFigure || RoundRectangleFigure || PertFigure)
```

```
        implements Persistence;
   pointcut WriteInFigures(Persistence po) : within(CH.ifa.draw.figures.*)
        && target(po)           && call(*  StorableOutput.writeStorable(*))
        && args(StorableOutput de) ;
  after(Persistence po, StorableOutput de) : WriteInFigures(po) {
       po.writes(de);
   }
}
```

An important comment here is that this is a (very) partial implementation, in the sense that there are many other elements in the code implementing persistence we are not taking into consideration. For example, here we are refactoring only cloned code common among RectangleFigure, RoundRectangleFigure and PertFigure classes, but there are other kinds of figures whose *write(*StorableOutput dw*)* methods is not cloned.

3.3.2. Handling. For handling, there is a special method called *selectionHandles(),* which depending on the active figure the user is interacting with, gets an enumeration of its handles.

Refactoring is possible if we declare an inter-type interface whose method crosscuts the *figure* classes using null-handle type (which can be of type *pert*, *text* or *node* figure), declaring a pointcut where the targets to be assigned are:

a) the current figure, and

b) the *handleEnumeration* object containing the handles.

The following section of code illustrates how we have implemented the refactoring of *null* handles.

```
public interface HandlingInterface extends Figure {
    public HandleEnumeration handles();
}
package CH.ifa.draw.contrib;
import CH.ifa.draw.framework.*;
import CH.ifa.draw.util.*;
import java.util.*;
import java.util.List;
import CH.ifa.draw.standard.*;
import java.io.*;
import org.aspectj.lang.Signature;
public aspect NullHandling {
   public static HandleEnumeration HandlingInterface.handles(Figure f) {
   List handles =  CollectionsFactory.current().createList();
   handles.add(new NullHandle(f, RelativeLocator.northEast()));
   handles.add(new NullHandle(f, RelativeLocator.southWest()));
   handles.add(new NullHandle(f, RelativeLocator.southEast()));
   handles.add(new NullHandle(f, RelativeLocator.northWest()));
   return new HandleEnumerator(handles);
}
   declare parents : (*.samples.net.NodeFigure || *.samples.pert.PertFigure ||
     *.figures.TextFigure)
       implements HandlingInterface;
   pointcut llamadas(HandlingInterface hi): (call(* *.samples.net.NodeFigure.handles()) ||
   call(* *.samples.pert.PertFigure.handles()) ||
   call(* *.figures.TextFigure.handles())) &&
   withincode (* StandardDrawingView.selectionHandles()) &&   this(hi);
   after(HandlingInterface hi) returning(HandleEnumeration he) : llamadas(hi) {
      he = HandlingInterface.handles((Figure) hi);
  }
}
```

Although these two examples could provide a modest gain in the number of calls saved with regards to the original implementation, such gain enforces our arguments to consider such examples as (candidate) aspects, because their refactoring is feasible.

3.4. Summary

In this chapter, we presented one case study to mine aspects. JHotDraw is an example found in several aspect mining research projects (e.g., [SP03], [Vol02], [RV02]), and is an interesting case study for aspect mining because:

- It is a project that follows a design patterns implementation and thus its code is well designed.
- Despite its good design, it contains enough complexity level to understand the relationships among the different elements in the code, causing the mining of aspects to be a non trivial task.

As expected, some (candidate) aspects were found, although it is difficult to say whether there were many or few, since previous studies do not discuss concretely which nor how many aspects did they find, but rather focus more on presenting how their exploring/mining approach works.

In the coming chapter, we introduce another mining technique called fan-in analysis, and the useful points in common from which clone detection and fan-in analysis can mutually benefit.

CHAPTER 4

CLONE DETECTION AND FAN-IN ANALYSIS IN ASPECT MINING

4.1. Introduction

In this chapter, we will present another aspect mining technique named fan-in analysis proposed by Marin et al. [MvDM04] at the Software Evolution Research Lab (SWERL)[1] in Delft University of Technology, and we will discuss how clone detection and fan-in can benefit each other in order to mine cross-cutting concerns.

We have chosen again the JHotDraw case study for this chapter in order to expose in detail what kind of seeds fan-in analysis finds. The next section introduces fan-in analysis.

4.2. Fan-in analysis

The fan-in analysis technique is an approach based upon the observation that code implementing a cross-cutting concern is (very) often called from different places throughout the software system at hand, thus revealing there is a scattered similar functionality, which is at the same time tangled with the main concern(s) of the system.

This method calling situation is known as **fan-in metric,** and is defined in [MvDM04, p. 3] as:

> "[..]the number of distinct method bodies that can invoke [a method] *m*. "

An important consideration with the previous definition, is that because of polymorphism, one method call can affect the fan-in of several other methods. A call to method *m* contributes to the fan-in of all methods refined by *m* as well as to all methods that are refining *m*. We use the code of figure 4.2.1 as example to illustrate better this situation: The fan-in value of B's *m* method contributes to the fan-in value of *m* in B's supertype (Interface A, with a value of 3) as well as its subclasses C1 and C2 (with a value of 3 and 2 respectively).

The implementation to obtain the fan-in metric is done inside the Eclipse platform as a plug-in that uses the Eclipse's *search for references* feature. The plug-in besides is able to report the list of the callers for all the methods in the selected source code and its output allows visualization of the callers and statistical reports.

4.2.1. The fan-in identification process. Fan-in analysis performs its mining process in the following steps:

(1) Automatic *computation of the fan-in metric* for all the methods in the system's source code, resulting in the set (sorted by fan-in value) of method-callers structures. For the experiment described in [MvDM04], a threshold of 10 was used.
(2) *Filtration of the first step's results*

[1] `http://swerl.tudelft.nl/`

```
interface A {
//caller set of A.m = { Example.foo, Example.bar, Example.zoo }; thus fan-in= 3
  public void m();
}
class B implements A {
//caller set of B.m = {Example.foo, Example.bar, Example.zoo, C2.m}; thus fan-in= 4
  public void m() {};
}
class C1 extends B {
//caller set of C1.m = {Example.foo, Example.bar, Example.zoo}; thus fan-in= 3
  public void m() {};
}
class C2 extends B {
//caller set of C2.m = {Example.foo, Example.bar}; thus fan-in= 2
  public void m() { super.m();};
}
class Example {
  void foo(A a) { a.m(); }
  void bar(B b) { b.m(); }
  void zoo(C1 c) { c.m(); }
}
```

FIGURE 4.2.1. Polymorphic method calls in class *Example* (Cf. figure 1 of [MvDM04])

(a) Select the restricted set of the methods with the top fan-in values -to be set in the case study example.
(b) Filter the getters and setters methods from the restricted set, based on the method's signature, in a first iteration, and its implementation, in a second iteration. Getters and setters were generally considered to provide little information for a subsequent investigation. For the experiment described in [MvDM04], the filter eliminated all the methods with names matching the get* / set* pattern. As an exception from the two iterations, the getters for static members references were reintroduced back into the candidates set, because such getters are considered as possible seeds for the *singleton* design pattern.
(c) Filter the utility methods, like *toString()*, collections manipulation methods, etc.

(3) Lastly, a *manual analysis* of the remaining set:
(a) the callers and call sites,
(b) the method names and implementation, and
(c) the comments in the source code.

4.3. Mining with Fan-in

Applying fan-in analysis to the JHotDraw's source code produced seeds for the following aspects:

4.3.1. Persistence and resurrection. As we saw earlier in the sub-sections 3.2.5.1 and 3.2.5.2, the persistence and resurrection of drawings is performed in JHotDraw by classes implementing the *Storable* interface. Persistence and resurrection are concerns that complement one another: classes of the *Figure* kind contain methods to write and read themselves to and from a *StorableOutput/Input* object, which is essentially a specialized type of output/input stream meant to send information into a storing device (e.g., hard disk).

With fan-in analysis we observe that methods implementing the *persistency/ resurrection* behavior report also a high fan-in value for some members of the *StorableOutput/Input* classes, as illustrated in figure 4.3.1.

Method	Fan-in value
readInt/writeInt	22/21
readStorable/writeStorable	20/18
readString	10

FIGURE 4.3.1. Fan-in values for the *StorableInput/StorableOutput* classes

method	fan-in value
CH.ifa.draw.util.Undoable.isRedoable	26
CH.ifa.draw.util.UndoableAdapter.undo	24
CH.ifa.draw.util.Undoable.isUndoable	11

FIGURE 4.3.2. Fan-in values for the methods of the *Undoable* interface

Method	Fan-in value
changed()	37
addFigureChangeListener(FigureChangeListener)	11
removeFigureChangeListener(FigureChangeListener)	10

FIGURE 4.3.3. Key methods implementing the Observer design pattern in JHotDraw

4.3.2. Undoing. The possibility to undo and redo editing actions is an important feature in applications dealing with text or graphics manipulation. In the specific context of JHotDraw, we have seen this concern being implemented across several activities, not only in editing actions directly accessible to the user through the graphical user interface, but also present in internal undoings when for example, the information contained in handles needs to be restored when attributes in the figures are changed (we refer the reader to sub-sections 3.2.5.3 and 3.2.6.1).

Because of the high number of activities that can be undone in JHotDraw, *undoing* is a concern widely scattered, which is also reflected in the high fan-in of methods belonging for instance, to the *Undoable* interface. A summary of the *Undoable* interface methods that participated in the detection of this aspect and their corresponding fan-in values is shown in figure 4.3.2.

4.3.3. The Observer design pattern. This design pattern is considered as a good candidate for refactoring into an aspect oriented way [KH02], due to the roles defined in the pattern's context that cross-cut the participant classes' logic. The elements that reflect the presence of the Observer design pattern are the methods used to attach/detach the observers to the subject role, to notify the observers of the subject's changes and to update them. The implementation of this pattern in JHotDraw (an instance of which is depicted in figure 4.3.4[2]) is suitable for fan-in recognition because its key methods have high fan-in values as shown in figure 4.3.3.

The *Observer* design pattern and the other two patterns discussed in the coming sub-section were selected among many others existing in the code base of JHotDraw since they have methods with fan-in values noticeably surpassing the threshold defined in sub-section 4.2.1. Furthermore, such methods strongly connote the relationship with the pattern in question : *attach/removeFigureChangeListener* in the

[2]Based on `http://www.dofactory.com/Patterns/PatternObserver.aspx`

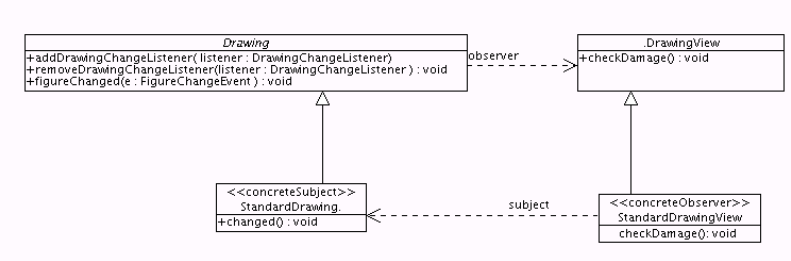

FIGURE 4.3.4. Instance of the Observer design pattern in JHotDraw

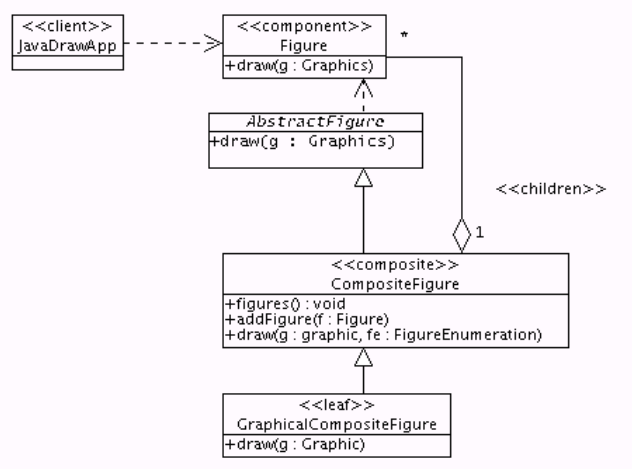

FIGURE 4.3.5. The *Composite* design pattern in JHotDraw

subject's role for *Observer, add(Figure)* in the *composite's* role for *Composite*, and *execute()* in the *Command* role of the *Command* pattern.

4.3.4. More aspects.

4.3.4.1. *The Composite design pattern.* The *Composite* design pattern -shown in figure 4.3.5[3]- contains methods whose functionality is invoked several times. Among the most noticeable methods because of their fan-in metric, are *add(Figure), includes(Figure)* and *figures(),* with values of 24, 14 and 13 respectively. The Composite design pattern qualifies as an aspect in JHotDraw because drawings are made out of (several) figures -as shown in figure 3.2.1-, thus a drawing can be seen as a *composite* figure, and hence we can notice that *figure* classes participate of this cross-cutting behaviour.

4.3.4.2. *Contract enforcement and consistent behaviour.* As mentioned in subsection 3.2.5.3, in JHotDraw there are 10 different editing commands (plus the possibility to undo and redo them) that the user can apply as depicted in figure 4.3.6. The *execute()* method of *Command* classes is invoked frequently by multiple invokers to carry out the request. This is a key method in the context of the *Command* design pattern (an instance of which is shown in figure 4.3.7[4]) and is a seed for the following cross-cutting concerns -highlighted in the comments of figure 4.3.8:

[3]Based on: `http://www.dofactory.com/Patterns/PatternComposite.aspx`
[4]Based on `http://www.dofactory.com/Patterns/PatternCommand.aspx`

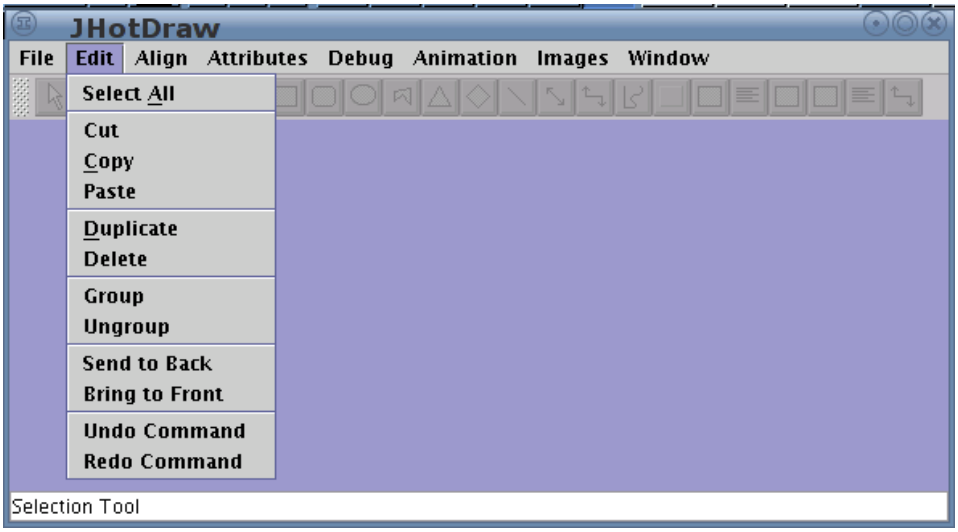

FIGURE 4.3.6. Editing commands in JHotDraw

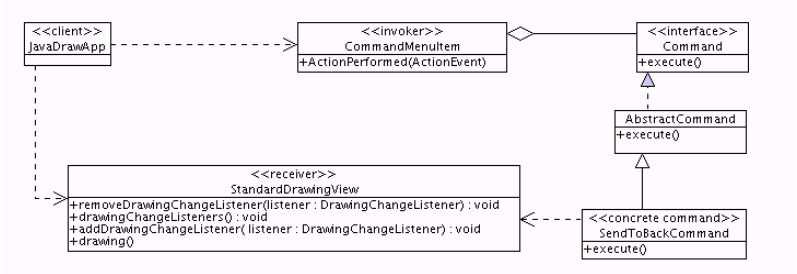

FIGURE 4.3.7. The *Command* design pattern in JHotDraw

```
public void execute() {
//perform checking whether view is not null
  super.execute();
  setUndoActivity(createUndoActivity());
  getUndoActivity().setAffectedFigures(view().selectionZOrdered());
  FigureEnumeration fe = getUndoActivity().getAffectedFigures();
  while (fe.hasNextFigure()) {
       view().drawing().sendToBack(fe.nextFigure());
   }
//refresh view if necessary
  view().checkDamage();
}
```

FIGURE 4.3.8. Declaration of *execute()* method in *SendToBack-Command* class (cf. Figure 8 of [MvDM04])

(1) It begins with a checking precondition, throwing an exception if it does not hold. This behaviour is regarded in literature as a *contract enforcement* aspect, mentioned in sub-section 1.3.1.1.
(2) Majority of the *execute()* implementations check in the end whether the figure has been changed and invoke a refreshing in that case, which is a way to provide *consistent behaviour* (also introduced in sub-section 1.3.1.1) among the classes implementing *execute()*.

Finally, we remark that the *AbstractCommand* class reported a fan-in value of 24 for its *execute* method.

4.4. Commonalities in the results produced by both techniques

Having discussed the previous sections, we can observe that both clone detection and fan-in analysis provide results which converge to a good extent in some aspects in JHotDraw's case. Among the commonalities, we highlight:

(1) THE UNDOING ASPECT. The *undoing* aspect resulted easily recognizable by both techniques due to the high values that overpass the thresholds set up in both techniques (defined in sub-sections 3.2.3 and 4.2.1, respectively) when selecting the code that is worth to analyse. Moreover, is evident that the functionality present in this concern is of an *unpluggable* and *cross-cutting* nature -recall criteria of sub-section 3.1.1-, as the core concern of JHotDraw is of graphical nature.

(2) PERSISTENCE AND RESURRECTION. Both techniques report high values in the number of method calls implementing the *Storable* interface (as shown in figure 4.3.1) and in terms of duplicated code (shown in figure 3.2.4) that reflect the extent to which these two concerns are scattered and tangled in the system.

4.4.1. Benefits of collaboration. Based on the commonalities discovered, we can discuss two issues regarding a collaborative use of both techniques:

(1) *The strength of each technique (or what cannot be found by the other)*

Fan-in analysis. A good reason to do a combined use of clone detection and fan-in analysis is that if we apply only clone detection, we leave out the importance of calls to key methods that belong to a specific cross-cutting concern.

An example illustrative of this situation, are the *changed()* and *add/removeFigureChangeListener (..)* methods that reported a high fan-in value (37, 21 and 21, respectively) and play an important role within the *Observer* design pattern discussed in sub-section 4.3.3, but that from the clone point of view would remain undetected, because these methods represent in total 7 declarations with a duplication level below the threshold set in sub-section 3.2.3, and have a small body.

Clone detection. On the other hand, by applying only fan-in analysis, we can remain unaware of the duplication in the code, which is also a good indication of common functionality that is spread and/or tangled in the main business logic of the system at hand. An example of this is the (candidate) aspect *handling* discussed in sub-section 3.2.6.1, whose seed code (written in the *handles()* method) has a low fan-in value: 4 in average for each of the 23 different declarations of *handles()*.

(2) *How to combine their usage*

The combination of both techniques can be done if once applied fan-in analysis, we discover with clone detection that the calling sites of these methods with high fan-in value, are duplicated code, which reveals scattered similar functionality.

The other way around consists into applying first clone detection, followed by fan-in analysis, to find methods included in the clone's body and therefore also related to a cross-cutting concern, but not large enough in terms of tokens to be reported by the clone detector. The idea is then to be flexible regarding the thresholds set up in both techniques, in the sense that if for example, a clone class with 4 elements has method calls within its clones with high fan-in values, then we can decide to explore more in detail this clone class, since it might be part of a cross-cutting concern.

4.5. Summary

In this chapter, we have discussed fan-in analysis as another technique for aspect mining, applied to JHotDraw's source code. Fan-in analysis is based "*in the simple rule that a method with high fan-in is likely to represent a functionality that is required across the application*" ([MvDM04, p. 10]). We have seen in sub-section 4.4, that both fan-in and clone detection techniques can complement one another finding aspects, in the sense that the individual results of one technique can be confirmed with the results of the other.

The combination of both techniques is promising in the eventual full automation of both of them, because the analysis of (seed) methods with a high fan-in metric can be aided by determining the places where these seeds are being invoked more frequently, which might be the case of duplicated portions of code, whose presence will be pointed out by clone detection.

CHAPTER 5

CONCLUSIONS

5.1. Introduction

In this last chapter we are going to summarize the results obtained when applying clone detection in aspect mining. The insights and lessons learnt are illustrative of how useful is to search for cross-cutting concerns in existing object oriented software. Once aspects are identified, existing Java code can be (preferably) refactored in the aspect oriented programming paradigm.

Full automatic mining of aspects is at the moment an ambitious aim, but we can consider the results of this research project as first steps in the sense that the procedure we used and the results we have obtained in our chosen case study are left as starting points for further research.

5.2. Contributions

Our research project has provided the following contributions:

(1) A SYSTEMATIC METHOD TO EXTRACT ASPECTS BASED ON A CLONE DETECTION TECHNIQUE.

As we saw in section 3.1.2, our approach to mine aspects although is yet mainly manual, involves the usage of different tools, namely CCFinder and FEAT (discussed in sub-sections 2.3.1.4 and 1.4.4, respectively). These tools when used together, give an insight of how the system under exploration is composed, and how cross-cutting behaviour is present. Therefore, we believe the method we applied is an important step towards full automation of mining as the seeds for aspects are provided semi-automatically by using clone detection.

(2) DETECTION OF ASPECTS IN THE JHOTDRAW PROJECT

We have presented an explicit set of aspects that were detected in the JHotDraw project. The implications of this detection are relevant because our results can be used as reference for the evaluation of other mining techniques (as we saw for instance, using clone detection and fan-in analysis to corroborate results).

5.3. Future work

Although the developed methodology has yielded satisfactory results in the case study addressed in this document, we acknowledge that our developed method lacks certain functionality, thus leaving space for others to extend the work that has been done so far. Improvements can be made in the areas of:

5.3.1. Further automation of the identification process. As we saw in chapter 3, the manual analysis done on the code still can be simplified if we knew before diving into the code exactly what clone classes are satisfying the criteria set up in sub-section 3.2.3. The algorithm presented in sub-section 3.2.4 deals partially with the problem, but a next step would consist into add functionality to select only those clone classes satisfying the criteria given in step 1 of sub-section 3.2.3.

5.3.2. Interaction with an specific IDE. In order to provide a useful representation of the seeds of aspects and of the rest of code interacting with them, it would be desirable to be able to visualize the (aspect) seeds without having to switch between the Gemini graphical interface (discussed in 2.4.1), and the Eclipse platform, where the rest of the code was mainly browsed.

A more complete solution would aim then at implementing the clone detection technique used -or another one, which are mentioned in [RD03]- as a plug-in for the Eclipse platform, for example.

5.4. Summary

To finalize, based on our experiences with clone detection as aspect mining technique, we can provide the answers to the questions posed in section 1.5.1:

(1) *Which are suitable candidates for automated aspect detection?*
As we saw in the results obtained during our case study, because of the clone detection technology used to mine aspects, we think that aspects with an important amount of duplicated code are more susceptible to be discovered. This of course does not imply that coverage of aspects by means of clone detection is extensive or exhaustive, but an important feature of cross-cutting concerns is the similarity of pieces of code scattered and tangled with the business logic code in the explored system.

(2) *What principles, rules or techniques can we apply for an automated detection of aspects*?
After carrying out the case study mentioned in this thesis, we have concluded that aspect mining needs to be a combination of techniques and tools, because there's no complete method that guarantees a full aspect mining coverage.
By using clone detection and combining the obtained results with fan-in analysis, the outcome has been relevant enough to believe that both techniques can complement each other. A more complete mining framework would aim to include other powerful techniques like tracing and/or slicing to analyse programs in a dynamic way.

(3) *Once successfully answered the previous question, can these principles, rules or techniques be implemented in a tool?*
As we have seen, the mining method we applied has some elements already automated; however, these are present in different tools. For clone detection we have relied in an already existing clone detector, but an better solution should aim to integrate clone detection (having cross-cutting concerns in mind), into a more complete environment, as mentioned in sub-section 5.3.2.

(4) *How can mined aspects be represented in an useful manner for the users interested in them?*
A useful way of representing aspects can be achieved by integrating a single graphical environment, that may display where:
 (a) the aspect seeds, and
 (b) the rest of the aspect
are located.

Complementing the answer of the previous question, we found highly useful the Gemini graphical interface to visualize the clone pairs (representing the seeds only), but also the representation of scattered functionality by using the Aspect Browser tool (both discussed in sub-sections 2.4.1 and 1.4.1, respectively) which allows to visualize in a more complete way the entire cross-cutting concern, provided Aspect Browser is fed with the necessary seeds.

(5) *Could we apply with satisfactory results these aspect mining methods in software systems regardless of their size?*
As the main tool we have used in our mining procedure has proved to be capable to handle industrial size software projects (i.e., CCFinder), we argue that by extension the mining technique can also manage large projects.

Bibliography

[BA01] L. Bergmans and M. Aksit. Composing multiple concerns using composition filters. 2001. University of Twente. http://trese.cs.utwente.nl.

[BB02] E. Burd and J. Bailey. Evaluating clone detection tools for use during preventative maintenance. In IEEE, editor, *Proceedings. Second IEEE International Workshop on Source Code Analysis and Manipulation*, pages 36 – 43. IEEE, October 2002.

[BEvDT04] M. Bruntink, R.v. Engelen, Arie v. Deursen, and T. Tourwe. An evaluation of clone detection techniques for identifying cross-cutting c oncerns. In *Proceedings of the 20th International Conference on Software Maintenance*. IEEE Computer Society, September 2004.

[BK03] S. Breu and J. Krinke. Aspect mining using dynamic analysis. In *Workshop on Software-Reengineering*, Bad Honnef, 2003.

[Bre02] S. Breu. Aspect oriented programming. University of Passau, January 2002. http://www.infosun.fmi.uni-passau.de/st/staff/breu/studium/seminar/AOP.pdf.

[BYM+98] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *International Conference on Software Maintenance*, pages 368–378. IEEE Computer Society, March 1998.

[Dij76] E.W. Dijkstra. *A DISCIPLINE OF PROGRAMMING*. Englewood Cliffs : Prentice-Hall, 1976. pp. 209-217.

[Fow99] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[GKY00] W. G. Griswold, Y. Kato, and J. J. Yuan. Aspectbrowser: Tool support for managing dispersed aspects. Technical Report CS1999-0640, March 2000.

[HK01] J. Hannemann and G. Kiczales. Overcoming the prevalent decomposition of legacy code. In *Proc. Workshop on Advanced Separation of Concerns*. IEEE, 2001.

[HO93] W. Harrison and H. Ossher. Subject-oriented programming: a critique of pure objects. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 411 – 428. ACM Press New York, NY, USA, 1993.

[IBM98] IBM. http://www.research.ibm.com/sop/sopoverv.htm. internet, December 1998.

[Kea97] G. Kiczales and J. Lamping ; et al. Aspect-oriented programming. In *Lecture Notes in Computer Science*. Springer-Verlag Heidelberg, 1997.

[KH01] R. Komondoor and S. Horwitz. Tool demonstration: Finding duplicated code using program dependences. In *Lecture Notes in Computer Science*, pages 383–386. 10th European Symposium on Programming, ESOP 2001, Springer-Verlag Heidelberg, April 2001.

[KH02] G. Kiczales and J. Hannemann. Design pattern implementation in java and aspectj. In *Proceedings of the 17th Annual ACM conference on*

Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pages 161–173. ACM, November 2002.

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object Oriented Programming*. Springer Verlag, 2001.

[KKI02] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–660, July 2002.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *11th European Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[Kri01] J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings Eighth Working Conference on Reverse Engineering*, pages 301 – 309. IEEE, October 2001.

[Moo02] L. Moonen. *Exploring Software Systems*. PhD thesis, Faculty of Natural Sciences, Mathematics, and Computer Science, University of Amsterdam, December 2002.

[MvDM04] M. Marin, A. van Deursen, and L. Moonen. Identifying aspects using fan-in analysis. In *11th Working Conference on Reverse Engineering*. IEEE Computer Society, November 2004. (to appear).

[MWB+93] M.Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object interactions using composition filters. In *Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming*, 1993.

[RD03] Filip Van Rysselberghe and Serge Demeyer. Evaluating clone detection techniques. In Michael W. Godfrey Tom Mens, Juan F. Ramil and Brian Down, editors, *Proceedings ELISA'03 (International Workshop on Evolution of Large-scale Industrial Software Applications)*, pages 25–36, September 2003.

[Res99] IBM Research. The software engineer's guide to hyperspace. http://www.research.ibm.com/hyperspace/GuideToHyperspace.htm, July 1999.

[RM02] M.P. Robillard and G.C. Murphy. Concern graphs: Finding and describing concerns. In *Proc. Int. Conf. on Software Engineering (ICSE)*. IEEE, 2002.

[RV02] R. Rajagopalan and K. De Volder. Qjbrowser: A query-based browser model. Master's thesis, University of British Columbia, July 2002.

[SP03] D. Shepherd and L. Pollock. Ophir: A framework for automatic mining and refactoring of aspects. Technical Report 2004-03, University of Delaware, October 2003.

[Tea02] The AspectJ Team. *The AspectJ Programming Guide*. Xerox Corporation, Palo Alto Research Center, Incorporated, 2002.

[TOHJ99] P. Tarr, H. Ossher, W. Harrison, and S.M. Sutton Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the International Conference on Software Engineering (ICSE'99)*, pages 107 – 119. IEEE Computer Society Press Los Alamitos, CA, USA, May 1999.

[UKKI02] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. Gemini: maintenance support environment based on code clone analysis. In *Proceedings. Eighth IEEE Symposium on Software Metrics*, pages 67–76. IEEE

Computer Society, June 2002.

[vDMM03] A. van Deursen, M. Marin, and L. Moonen. Aspect mining and refactoring. In University of Waterloo, editor, *First International Workshop on REFactoring: Achievements, Challenges, Effects*. University of Waterloo, November 2003.

[Vol02] K. De Volder. The jquery tool: A generic query-based code browser for eclipse. Presentation at Eclipse BoF at OOPSLA 2002, 2002.

[WJL+03] A. Walenstein, N. Jyoti, J. Li, Y.Yang, and A. Lakhotia. Problems creating task-relevant clone detection reference data. In *10th Working Conference on Reverse Engineering (WCRE 2003), Proceedings of*, pages 285–295. IEEE Computer Society, November 2003.

[ZJ03] C. Zhang and H-A. Jacobsen. Quantifying aspects in middleware platforms. In *Proc. 2nd Int. Conf. on Aspect-Oriented Software Development (AOSD-2003)*, pages 130–139. ACM Press, March 2003.

A. Script to eliminate duplicated clone classes from CCReformer's output

```
#!/usr/bin/perl
use strict;
my ($fileNo, $thisPos);
my @where;
#open my classes file
 open FILEHANDLE, "salidaClasses.txt" or die;
#create a file where to send the refined classes open
 OUTPUT, ">refinedClasses.txt" or die;
 while (<FILEHANDLE>){
   last if /^\#begin\{file description\}/;
 }
#COPYING THE FILE DESCRIPTION SECTION
 while (<FILEHANDLE>){
   print OUTPUT $_; last if /^\#end\{file description\}/;
 }
#POSITION AT THE CLONE SECTION
 while (<FILEHANDLE>){
   last if /^\#end\{syntax error\}/;
 }
 print OUTPUT "#begin{clone}\n";
#SLURP THE REST OF THE FILE
 my (@string, @string2);
 while (<FILEHANDLE>){
   @string2= split(/\s+/, $_);
   push @string, @string2
 }
 my $begPos;
 while ( @string){
   $fileNo = shift @string ;
   if ($fileNo=~ /\./) {
       $thisPos= shift @string;
       @where = busca ($fileNo, $thisPos);
       if ($#where > 0){
             selectClass(); }
       else{
             $begPos= 0;
             escribe($begPos);
             borra($begPos);
           }
     }
 }
sub selectClass{
 my ($i,  $noElem, $dummy);
 my %classSet;
 my (@sortedvalues, @temp);
 #FIND THE BEGINNING OF THE BLOCK CORRESPONDING TO OUR CURRENT
 #CLONE
 for $i ( 0 .. $#where ) {
   $begPos = @where[$i];
   while ($begPos > 0){
       last if $string[$begPos] eq "\#begin{set}" or $string[$begPos] eq undef;
       $begPos -=1;
   }
   if ($begPos == 0){
       $noElem +=1
   }
  #COUNT THE ELEMENTS OF THE CLASS
  for $i ($begPos .. $#string ) {
    #COUNT IF WE HAVE A ".", WHICH MEANS ONE CLASS MEMBER
    $noElem +=1 if $string[$i]=~ /\./;
    last if $string[$i] eq "\#end{set}";
  }
```

```
  #HASH THE BEGINNING POSITION AND THE NUMBER OF ELEMENTS PER CLASS
  $classSet{$begPos} = $noElem;
  $noElem = undef;
 }
 #keys in     %classSet  are the beginning positions per class
 @sortedvalues  = sort (keys %classSet);
 #ONCE ORDERED THE HASH LIST, GET THE PAIR IN THE MIDDLE.  THIS
 #REPRESENTS THE HEURISTIC TO GET THE MOST REPRESENTATIVE CLASS OF ALL
 $begPos =  @sortedvalues[$#sortedvalues/2 ];
 #WRITE THE ELECTED CLASS TO FILE AND DELETE IT FROM @string
 escribe($begPos);                          borra($begPos);
 #AND DELETE ITS REFERENCE FROM THE HASH SET
 delete  $classSet{$begPos};
 #AFTER WRITTEN THE ELECTED CLASS DELETE THE REST OF CLASSES
 #FROM  @string
 while (($begPos, $dummy) = each (%classSet)){
     @temp  = busca ($fileNo,   $thisPos);
     if (@string[1] ne  $fileNo or @string[2] ne $thisPos){
         shift @temp
     }
     $begPos = shift @temp;
     borra($begPos);
 }
}
sub busca{
 my $i;
 my @positions;
 @positions= (@positions ,0);
 for $i ( 0 .. $#string  ) {
    if  ($fileNo eq @string[$i] and $thisPos  eq  @string[$i + 1]){
             @positions= (@positions, $i + 1);
    }
 }
return @positions
}
sub borra{
 my $pos= 0;
 #SEARCH THE BEGINNING OF THE CLASS
 if ($string[$begPos] ne "\#begin{set}"){
 while ($begPos > 0){
    last if $string[$begPos] eq "\#begin{set}" or $string[$begPos] eq undef;
    $begPos -=1;
  }
 }
 $pos= $begPos;
 while (@string[$pos] ne "\#end{set}" and @string){
   splice(@string, $pos, 1);
 }
 splice(@string, $pos, 1);
}
sub escribe{
 my ($i, $j);
 $j = $begPos;
 if ($begPos == 0){
     print OUTPUT "\n#begin{set}\n $fileNo $thisPos ";
  }
 else{#IF WE DID THE CLASS SELECTION PROCEDURE
    print OUTPUT "\n#begin{set}\n ";
 }
 for $i ( $j .. $#string ) {
   print OUTPUT " ${string[$i]}";
   if ($string[$i + 1]=~ /\./ or $string[$i + 1]=~ /end/)
    {
      print OUTPUT "\n"
   }
 last if $string[$i + 1] eq "\#begin{set}"
 }
}
```

B. Script to invoke the clone detection and clone classes redundancy removal processes

```
#!usr/bin/perl
use strict;
my $command;
print "Introduce the  input file (with complete path) for CCfinder? ";
my $path = <STDIN>;
my $i;
my $name;
chomp ($path);
my $new_path= split (/\\/, $path);
for $i ( 0 .. $#main::new_path) {
$path = $main::new_path[i] . "/" }
open FILEHANDLE, $path or die;
open INPUT,
">inputFiles.txt" or die;
$name= "ccfinder java -i \"" . $path . "\" -o clonesOutput.txt";
print "$name \n"; print "executing CCfinder...\n"; $command= `$name`;
print "executing CCReformer...\n";
$command= `ccreformer class clonesOutput.txt -o classesOutput.txt`;
print "executing my script...\n";
$command= `perl version2.pl`;
```