Vrije Universiteit Brussel – Belgium

Faculty of Sciences

In Collaboration with Ecole des Mines de Nantes – France

and

Sodifrance

2005

Characterization and Detection of Concerns in Java Code

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

By: Harmin Parra Rueda

Promoter: Prof. Theo D'Hondt (Vrije Universiteit Brussel)
Co-Promoters: Hervé Albin–Amiot (Ecole des Mines de Nantes)
Erwan Breton (Sodifrance)

Abstract

Many software systems are subject to continual revisions and expansions as new requirements are discovered, bugs are found, or migrations to new technologies need to take place in order to improve software quality.

Many program evolution tasks involve the manipulation of source code in order to isolate and change the implementation of different concerns. However, such concerns may be scattered over the whole code because they cannot be easily modularized, they can be the result of inadequate design, the result of unanticipated changes, or they can be the result of a lack of expressiveness in the technology available to the original developer.

Before performing a modification to a software system, code relating to concerns has to be carefully identified. Finding and understanding concerns scattered in source code is a difficult task that takes a large proportion of the total effort required to perform software maintenance and evolution.

This thesis presents a model to characterize concerns, aimed to detect and to display an abstract view of concerns related code, easing the software comprehension and maintenance process.

A prototype was developed in order to validate the proposed models and some case studies are presented.

Contents

Abstract ii

Contents iii

List of Tables v

List of Figures vi

1 Introduction **1**

2 Motivation: An Example of Program Maintenance Involving Scattered Concerns **4**

3 State of the Art **8**

3.1 Non-automatic aspect mining tools 8

3.1.1 Sample code 8

3.1.2 Query–based search tools 10

3.1.2.1 Aspect Browser 10

3.1.2.2 AMT 12

3.1.2.3 The Eclipse Framework 15

3.1.2.4 PRISM 18

3.1.2.5 CME 22

3.1.3 Exploratory tools 26

3.1.3.1 JQuery 27

3.1.3.2 FEAT 32

3.1.4 Comparison of non-automatic aspect mining tools 38

3.2 Automatic aspect mining tools 41

3.2.1 Clone code detection 41

3.2.2 Fan-in analysis 41

3.2.3 Formal concept analysis 43

3.2.4 Execution trace analysis 44

3.2.5 Formal concept of execution traces 46

3.2.6 Comparison of automatic aspect mining tools 46

4 Thesis Contribution 48
4.1 Limitations of existing tools 48
4.1.1 False aspect candidates 48
4.1.2 Undetected aspects 48
4.1.3 Concern characterization constructs in multiple concern operations 48
4.1.4 No detection of concern instances 49
4.1.5 No abstraction of detected concerns 49
4.1.6 The user still needs to analyze the source code 50
4.2 Thesis objectives 50
4.2.1 A new concern characterization construct: Method Invocation Sequence 51
4.2.2 Abstract view of concern instances 51

5 Design considerations 52
5.1 Characterization of Concerns 52
5.2 Source code representation 54
5.3 Concern instance 54

6 Validation 59
6.1 First case study: persistence 59
6.2 Second study case: web sharing information 66
6.3 Limitations 69

7 Discussion 74
7.1 Type usage and Method invocations 74
7.2 Fields 74
7.3 Java artifact relationships 75
7.4 Method declarations and Aspect advices 76

8 Future Work 78

9 Conclusions 80

Bibliography **82**

Appendix A Glossary of Terms **87**

List of Tables

Table 3.1: The Eclipse *Java Search* option. 16
Table 3.2: Comparison between the CME *AspectJ Compatible Search* and the Eclipse Search Engine. 24
Table 3.3: JQuery predefined top-level queries. 28
Table 3.4: JQuery predefined queries. 29
Table 3.5: TyRuBa unary predicates. 30
Table 3.6: TyRuBa binary predicates. 31
Table 3.7: Some relationships in FEAT. 35
Table 3.8: FEAT supported queries. 37
Table 3.9: Comparison of non-automatic aspect mining tools. 39
Table 3.10: Comparison of automatic tools. 47
Table 7.1 : Java artifact relationships. 75

List of Figures

Figure 3.1: Aspect Browser screen shot. 13
Figure 3.2: AMT screen shot. 13
Figure 3.3: Eclipse screen shot. 17
Figure 3.4: Eclipse screen shot. 17
Figure 3.5: PRISM screen shot. 20
Figure 3.6: PRISM screen shot. 21
Figure 3.7: CME screen shot. 25
Figure 3.8: JQuery screenshot. 33
Figure 3.9: JQuery screen shot. 34
Figure 3.10: FEAT screen shot. 38
Figure 3.11: Various polymorphic method calls. 42
Figure 3.12: Fan-in values for code in figure 3.11. 42
Figure 5.1: Characterization of concerns. 55
Figure 5.2 : A simplified concern characterization instance. 56
Figure 5.3: The source code representation. 57
Figure 5.4 : The concern instance representation. 58
Figure 6.1: Persistence – one query and one update instance. 63
Figure 6.2: Persistence – one connection and one closing instance. 64
Figure 6.3: Persistence – two update instances. 66
Figure 6.4: Persistence – two connection, two update and two closing instances. 67
Figure 6.5: Web Sharing Information – one session, two writing and two reading instances. 70
Figure 6.6: The assignment problem. 72
Figure 6.7: The conditional statement problem. 72
Figure 6.8: The control flow problem. 73
Figure 7.1: Levels of source code granularity obtained through artifact relationships with FEAT. 77

1 Introduction

Corporate and commercial software systems keep changing and evolving. Software systems are subject to modifications as new requirements and bugs are discovered, and when a software migration is required to adapt a system into a new technology to improve software quality.

One of the primary goals of software engineering is to improve software quality and to facilitate maintenance and evolution by seeking technologies and methodologies that reduce software complexity, improve comprehensibility, promotes reusability and facilitate evolution [1].

One way to reduce complexity and improve comprehensibility can be attain trough decomposition of software systems into meaningful and manageable computation units called modules [2], and through composition of such units to make them work together.

The major advantages of modular programming are:

1) The ability to write one module with little (or none) knowledge of the code in another module.

2) The possibility to replace or to make dramatic changes to one module without changing other modules.

3) The improvement of the comprehensibility by making possible to study the system one module at a time.

Our ability to achieve the goals of software engineering depends fundamentally on our ability to keep separated all concerns. In the present thesis dissertation, we will use the terms “aspect” and “concern” indistinctly. We use the term “aspect” or “concern” to refer to any technical consideration a developer might have about the implementation of a system. Some examples of concerns include[32]: persistence, logging, caching, security, authentication, transactions, error handling, synchronization, debugging, assertions, metrics and web sharing information.

The modularization of concerns is helpful during software maintenance and evolution because developers do not have to deal with the entire program each time they want to make a change. They can focus on just the modules that implement the concern that needs to be changed.

Before performing a modification to a software system, developers must explore the system's source code to find and understand the code portion relevant to the change task. Software maintenance and evolution often becomes a challenging task to developers who are required to deal with large software systems, understand the source code, and to understand the structure and behavior of specific concerns.

Nevertheless, not all concerns are easily modularized and, on the contrary, might be scattered (spread) across many modules. Another inconvenient is the tangling of concerns, which are concerns that need to interact and cooperate with each other. The scattering and tangling of concerns in source code is the consequence of four principal causes [3]:

1) Inadequate design: a software system design can fails to create modules hiding implementation details associated with a concern.

2) Programming Language Limitations: Some times, even a well done software system design and implementation make it impossible to separate every concern with only the basic construct of a programming language.

Sometimes it is possible to overcome such limitation through the use of design patterns. For example, the Visitor design pattern [4] is a solution to separate structure from behavior in a hierarchical object collection. Design patterns can help address a small set of well-identified problems, but they cannot help to address the majority of modular decomposition problems.

Another proposed solution is provided by the Aspect-Oriented Programming paradigm [1, 5, 6, 7], which is an extension to Object-Oriented programming languages aimed to modularize scattered concerns in separate modules. Nevertheless it cannot help to address all possible causes of concern scattering, especially when two or more concerns are tangled and cannot be separated because they need to interact together.

3) Unanticipated changes: another cause of scattered and tangled concerns is the emerging of new requirements that did not exist during the development of a system, but needs to be considered for the evolution of the system.

4) The last principal reason for scattering and tangling of concerns is code decay, which refers to the fact that due to repeated maintenance, the time pressure and the difficulty to understand the whole program's source code, software modifications may violate design constrains and may introduce coupling between modules, generating further scattering and tangling of concerns in source code.

Before performing a modification to a software system, code relating to concerns has to be carefully identified. Due to the scattered nature of concerns, searching for them in existing code is a non-trivial task.

An example of scattered concerns is provided by the analysis of the Tomcat [42] source code undertook by the Palo Alto Research Center (PARC) [6]. Tomcat is an open source web server that implements the Java Servlet and the Java Server Page (JSP) specification. The objective of the analysis was to investigate how concerns were employed by Tomcat.

The study discovered some well modularized concerns like the XML parsing and the URL pattern matching. On the other hand, the Logging concern were scattered across several classes. Other scattered concerns were Session expiration and Session tracking.

Some problems and limitations posed by scattered concerns are:

- They produce redundant code.
- They are difficult to reason about.
- They are difficult to find.
- They are difficult to maintain.

Scattered and tangled concerns pose a challenge to developers who need to find the code involved across several classes without any help from Object-Oriented tools.

This thesis dissertation proposes a concern characterization aimed to detect scattered concerns in source code.

2 Motivation: An Example of Program Maintenance Involving Scattered Concerns

The need for maintaining and improving software and information systems has risen dramatically over the past decade [11, 12]. Corporate software systems are challenging because they are critical to the operation of companies, they contain important corporate knowledge and business rules, they represent a large investment and, as time pass by, their technology and architecture become obsolete.

Notwithstanding, around 50% to 62% of the time spent on software maintenance, is devoted to understanding the system being maintained [9]. This is due mainly because mission-critical systems might have been maintained for many years by different programmers, because supporting documentation may not be current, or because of the presence of scattered concerns throughout the code.

For instance, let's consider a case of software maintenance to change the persistence mechanism of a given system. Suppose that we have a system that use the Java JDBC API to do persistence and that we want to switch to Hibernate.

JDBC [14] is the built-in Java [15] API that allows users to execute common SQL statements within java programs. SQL is a standard language for accessing and manipulating data from database systems, and JDBC is the de-facto API for java applications to perform such SQL calls against a database. However, JDBC and SQL are not object-oriented. Even small projects might require a lot of SQL code, very few people are good to writing SQL code, writing JDBC/SQL code is tedious and error-prone (example: JDBC `Statement` and `ResultSet` objects should be closed manually. If they are not closed, we get a cursor leak). In addition, developers have to switch from the Object-Oriented language, to the "row" and "table" language to retrieve and manipulate data from databases.

On the other hand, Hibernate [16] is an Object-Relational mapping framework that allows transparent persistence for java objects against relational databases. It generates stub classes that carry out, behind the scene, the persistence object operations, and it offers the Hibernate Query Language (HQL), which is a language similar to SQL that allows developers to retrieve objects instead of rows and tables.

As an example, consider that we have the following `Person` class:

```
public class Person {

    private String id;
    private String name;
    private int age;
    private float weight;
}
```

And that we have the corresponding PERSON table:

PERSON	
PK	id
	name
	age
	weight

Now, we are going to compare how to retrieve an object of type `Person` with id = "12345" using JDBC and Hibernate.

1) JDBC solution:

```
1   import java.sql.*;

2   Class.forName("aDatabaseDriver");
3   Connection conn = DriverManager.getConnection.("aDatabaseUrl",
                                          "aUsername", "aPassword");

4   Statement st = conn.createStatement();
5   ResultSet rs = st.executeQuery("SELECT name, age, weight
                                FROM PERSON WHERE id = '12345'");
6   rs.next(); //get the first row of the query

7   String name = rs.getString("name");    //The name column
8   int age = rs.getInt("age");            //The age column
9   float weight = rs.getFloat("weight");  //The weight column

10  Person p = new Person("12345", name, age, weight);
```

```
11  rs.close();
12  st.close();
13  conn.close();
```

2) Hibernate Solution:

With Hibernate, it is necessary to define xml documents to provide the Hibernate configuration and a mapping configuration for each class. The Hibernate xml configuration file sets the properties that Hibernate uses to connect to the database and it looks like this:

```
<hibernate-configuration>
   <session-factory>
      <property name="hibernate.connection.driver_class">
         aDatabaseDriver</property>
      <property name="hibernate.connection.url">
         aDatabaseUrl</property>
      <property name="hibernate.connection.username">
         aUsername</property>
      <property name="hibernate.connection.password">
         aPassword</property>
      <property name="hibernate.dialect">
         org.hibernate.dialect.MySQLDialect</property>
   </session-factory>
</hibernate-configuration>
```

The first four lines are the same parameters used by the JDBC API to get a connection to the database. The last property defines the SQL dialect used when converting the Hibernate Query Language (HQL) into SQL.

The mapping xml documents define how the classes' fields are mapped to table columns in the database. For our example, this file looks like this:

```
<hibernate-mapping>
   <class name="Person" table="PERSON">
   <id name="id" type="string" unsaved-value="null" >
      <column name="ID" sql-type="char(32)" not-null="true"/>
   </id>
   <property name="name" column="name" type="string"/>
   <property name="age" column="age" type="int"/>
   <property name="weight" column="weight" type="float"/>
   </class>
</hibernate-mapping>
```

This xml file describes how class fields are mapped into tables. It specifies the name of the class field, the name of the column it is mapped to and the types of the fields. The id element block describes the primary key used by the persistence Person class (for a complete tutorial on Hibernate, please go to http://www.hibernate.org).

Finally, the java code to retrieve an object of type `Person` with id = "12345" will be:

```
1   import org.hibernate.*;

2   Session session = SessionFactory.openSession();
3   Person p = session.load(Person.class, "12345");
```

In both examples, the java code highlighted in yellow color, represent the code implementing the opening of a database connection/session (we can call it the *connection operation*), and the code highlighted in green represent the code implementing the database query (we can call it the *query operation*).

The Hibernate related code is shorter and simpler than the JDBC related and, therefore, a migration from JDBC to Hibernate may be highly desired since it may improves software quality, maintainability, comprehensibility, and reducing software complexity.

Nonetheless, in large Object-Oriented systems, the persistence might not be well modularized and may be scattered across several classes. Hence, the detection of the persistence related code could be a difficult task.

3 State of the Art

In this chapter we are going to describe the existing aspect mining tools for detection of concerns. We are going to consider a small example in order to illustrate exactly how they work and to discuss their advantages and limitations.

In the present thesis dissertation, we will use the terms "aspect" and "concern" indistinctly to refer to a technical consideration a developer may have about the implementation of a system.

There are two kinds of aspect mining tools: the non-automatic ones that require a seed from the user, and the automatic ones which do not. A seed is an input describing how an aspect may look like.

3.1 Non-automatic aspect mining tools

These aspect mining tools can be categorized as being either query-based or exploratory [49]. Both require a seed and depend on the user's understanding of the software to be analyzed. In order to evaluate these tools for detection of concerns, we are going to consider a sample code.

3.1.1 Sample code

Let's consider a class called `Person` with a method called `queryAndUpdate`, which makes a connection to a database, performs a query and an update, and lastly, closes the connection to the database.

```
public class Person {

  private String id, name;
  private int age;
  private float weight;

  public void queryAndUpdate(){
    try {
      /*Connection to the database*/
      Class.forName("aDatabaseDriver");
```

```
      Connection conn = DriverManager.getConnection("aDatabaseUrl",
                                                    "aUsername", "aPassword");
      /*an update*/
      Statement st1 = conn.createStatement();
      st1.executeUpdate("UPDATE ...);
      st1.close();

      /*a query*/
      Statement st2 = conn.createStatement();
      ResultSet rs = st2.executeQuery("SELECT ...);

      rs.next();
      this.name = rs.getString("name");    //The name column
      this.age = rs.getInt("age");         //The age column
      this.weight = rs.getFloat("weight"); //The weight column
      rs.close();
      st2.close();

      /*closing of the connection*/
      conn.close();
    } catch (Exception e) { }
  }
}
```

We are going to consider 4 operations:

1) Connection operation, defined by the following method invocations:

java.lang.Class.forName
java.sql.DriverManager.getConnection

2) Query operation, defined by:

java.sql.Connection.createStatement
java.sql.ResultSet.close
java.sql.ResultSet.getFloat
java.sql.ResultSet.getInt
java.sql.ResultSet.getString
java.sql.ResultSet.next
java.sql.Statement.close
java.sql.Statement.executeQuery

3) Update operation, defined by:

java.sql.Connection.createStatement
java.sql.Statement.close
java.sql.Statement.executeUpdate

4) Closing operation, defined by:

```
java.sql.Connection.close
```

It should be noted that some method invocations can be shared by two or more operations. In our example, the `java.sql.Connection.createStatement` and the `java.sql.Statement.close` methods are shared by the Query and the Update operations.

3.1.2 Query–based search tools

These kinds of tools receive a query as an input. The kind of analysis performed by these tools may include the search of text patterns, type usage, method invocations or predefined queries. Query–based tools are required to receive an input from the user, which can be either a regular expression or a string.

3.1.2.1 Aspect Browser

Overview

Aspect Browser [17, 18] is one of the first aspect mining approaches. Its main functionality is to search of textual-pattern matching using a like-Unix grep language and highlights matching text with a specific color. It extracts fragments of identifier names from source code according to a programmer-specified naming convention.

The results are reported as a list of aspect candidates. An aspect in Aspect Browser is defined as a pair of a regular expression and a color. This functionally is strongly dependant on the naming convention used in the source code, and assumes that implemented crosscutting concerns have a signature which can be identified by a textual regular convention.

When an aspect candidate is found, the matching text is highlighted in the source code, and the tool will indicate the match-count.

In addition, Aspect Browser provides a view in which each file is represented as vertical strips, where a row of pixels in the strip represents a line of code. Whenever a line of code is highlighted to indicate an aspect candidate, the corresponding row pixel of the files vertical strip representation is also highlighted with the same color, which makes easy to see how an aspect is

scattered. If more than one aspect appears in the same line of code, the view will display a red line indicating that there is an "aspect collision".

Aspect Browser has two more functionalities. The first one finds redundancies in the code, the second one reports any line that appears more than once. It is an effective approach to identify code written with copy-paste, which is a common technique used by software developers.

Evaluation

This tool does not perform any type analysis, so it was necessary to define our text patterns as the method names implementing the concerns (example: "`close`"). This is very inconvenient, since we want to distinguish invocations to the `close` method on objects of types `Connection`, `Statement` and `ResultSet`. Moreover, if there were a `close` method defined in another class, those method invocations would also be added to the result. Equally, if there were variables named `close`, or if such word were present in code comments, such occurrences would also be added to the result.

In the case of static method invocations, we were able to type both the type and the method name in the search input (example: "`Class.forName`" and "`DriverManager.getConnection`").

The tool automatically designates a color for each search and the corresponding source code will also be highlighted with that color. It is possible to enable and disables the colors by the user.

The tool allows users to create concern data structures called *groups*, and to add textual-regular expressions to them. It is possible to enable and disable the color of a single aspect candidate or a whole group of aspect candidates. Particularly, we were able to create groups called "Connection", "Query", "Update" and "Closing".

Regarding the case of the `close` method (defined in types: `java.sql.Connection`, `java.sql.Statement` and `java.sql.ResultSet`, and used in operations: Query, Update and Closing), the tool detected all method invocations, without performing any type distinction, and it only allowed us to add the results to only one group.

It is not possible to execute duplicate queries or to share one query result with more groups (i.e.: it is not possible to add the `close` method to the Query, Update and Closing operations; or the `createStatement` method in both Query and Update operations).

The tool identified two types of redundancies: redundant lines of codes and common identifiers. The common identifiers detected with this example include the following text occurrences: `age`, `name`, `rs`, `weight`, `id`, `close`, `a` and `conn`. The tool is also able to identify redundant lines of code but, because our sample class is too short, it didn't identify any.

The textual-pattern matching is heavily dependent on the programmer coding convention. It works well only if consistent naming conventions for types, methods, variables and classes are carefully followed. Conversely, this method will not work if naming conventions are not strictly followed.

Limitations

Does not perform type analysis, so in the case of method names, it does not distinguish the method declaring class/interface. If there were another type declaring also a method called `close`, Aspect Browser will include it, generating distracting results to the user. In addition, it does not allow the addition of a textual-pattern search result to more than one group.

Since this tool only performs textual-pattern searches, it does not distinguish between a package name, a type name, a variable name, a method name, or a code comment. The user needs to spend time to analyze and to filter the results.

Figure 3.1 offers a screen shot of the Aspect Browser tool.

3.1.2.2 AMT

Overview

The Aspect Mining Tool (AMT) [20, 21, 22] is a multi-modal analysis framework that combines text-based and type-based analysis.

The text-based analysis technique of AMT, similar to Aspect Browser, works best if consistent naming conventions for types, methods, variables and classes are carefully followed. However, it is not helpful if naming conventions are not followed, or are followed only partially.

Moreover, the results using this technique can be distracting because if the majority of the code adheres to naming conventions while the rest does not, the developer might be convinced with the results of the query and might forget to question himself/herself for possible lines of code that might not follow the

naming conventions. Unfortunately, legacy code might not follow naming conventions.

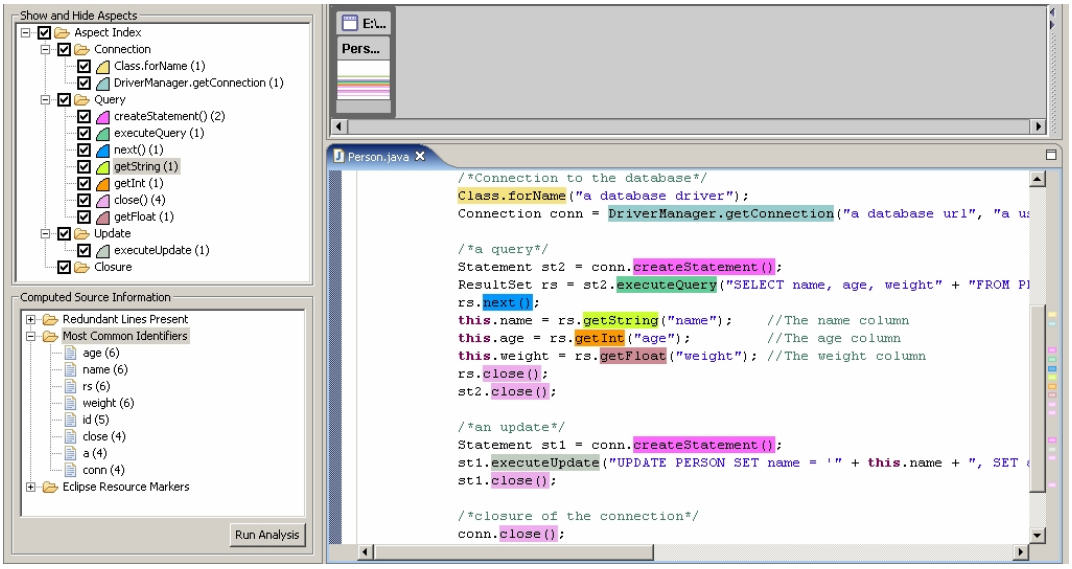

Figure 3.1: Aspect Browser screen shot.

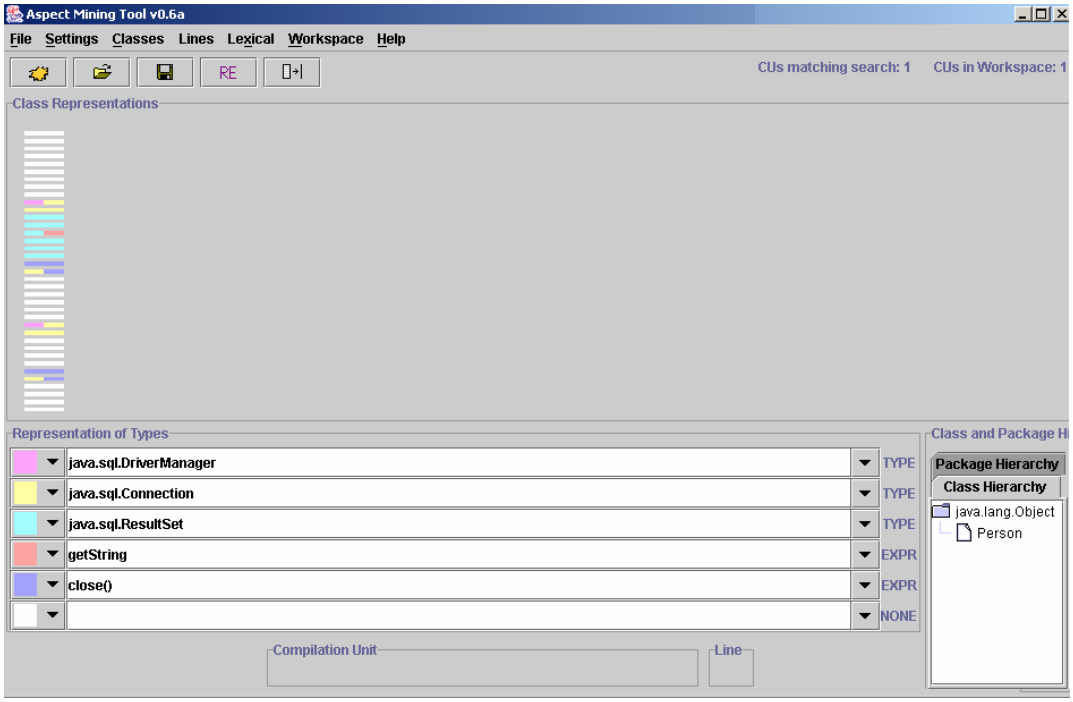

Figure 3.2: AMT screen shot.

The type-based analysis searches for instantiations and usages of types. The naming convention for objects and variables become irrelevant for this technique.

The type-based analysis has its limitations also, since it search for object references of the same type regardless their functionality. For instance, if we were to analyze the code implementing the GUI of a java application and we were to do a type-based analysis on objects of type `JButton`, the tools will identify all instantiations of the `JButton` class regardless their functionality (example: color settings, font settings, file browsing, etc.).

The tool also represents java files as a collection of horizontal strips, were each strip represents one line of code. The tool allows users to choose a color for each query result, and it will display that color in the matching line strip of the java file representation. If a line matches more than one search criteria, it will be separated into different colors. Unfortunately, the results are not linked to the source code.

The tool is quite old and requires Java 1.3. Moreover, it uses a modified version of the AspectJ [33] compiler, and the CLASSPATH variable should be carefully setup if AspectJ is already installed.

Evaluation

The type-based analysis works pretty well. We were able to do type-based search for `java.sql.DriverManager`, `java.sql.Connection`, `java.sql.ResultSet` and so on. It found all the instantiations and usages of such types. The type-based analysis works with objects and variables, but not with method invocation. That is, it does not analyze the declaring type, and arguments of method invocations.

The text-based analysis search for occurrences of a text, and does not support the use of wildcards (i.e. the "*" character). Since the type-based analysis does not work with method invocations, we had to detect them using text matching, as with Aspect Browser (example: to detect invocations of the method `java.sql.ResultSet.getString()`, we have to search the text "`getString`", "`getString(`", or "`.getString(`").

It was not possible to create concern data structures in order to store different query results.

Lastly, the line strips for the java file of our example were displayed with the corresponding colors specified in the search, but we were not able to see the corresponding source code for each search result.

Limitations

The AMT tool works best if naming conventions for types, methods, variables and classes are followed. The code that does not follow such naming conventions is not detected.

The type-based analysis does not work with method invocations. The tool doesn't find out the signatures of method invocations, they have to be detected with textual searches, and in this case, the same limitations of Aspect Browser apply to AMT. Furthermore, it is not possible to create a concern data structure in order to store different query results.

Finally, the tool is quite old and the results are not linked to the source code, making the tool almost useless.

Figure 3.2 presents a screen shot of the AMT tool.

3.1.2.3 The Eclipse Framework

Overview

The Eclipse framework [22] is not an aspect mining tool, but it comes with a pretty mature search engine that can be useful for detecting concerns in source code. It comes with two major search options, the first one is called *File Search*; the second, more complex, is called *Java Search*.

The *File Search* option comes with two sub-options. The first one is able to find text pattern-matching and can receive as input any text, including the following wildcards:

- "`*`", which denotes any string.
- "`?`", which denotes any character, and
- "`\`", which makes possible the search of escape characters (`\n`, `\t`, `\r`, `\\`, `\s`, etc.).

The second sub-option allows the search of Perl-like regular expressions [34, 35] (expressions like: `[a-zA-Z]`, `[^0-9]`, "`ba+`", "`gr(a|e)y`", etc.). In both options, the user can enable or disable the case sensitive constraint.

The *Java Search* option allows user to search java code artifacts (constituent java source code elements). Table 3.1 summaries the queries available under the *Java Search* option.

Element	Queries
Type	Declarations
	References
	Implementators (for interfaces and abstract classes)
Constructor	Declarations (definition)
	References (invocations)
Method	Declaration (definitions)
	References (invocations)
Field	Declaration
	References (both read and write access)
	Read access
	Write access
Package	Declarations (package clauses)
	References (import clauses)

Table 3.1: The Eclipse *Java Search* option.

In all the above options, the user can use the "*" and the "?" wildcards (not "\"). It is also possible to enable and disable the case sensitive constraint.

Another strength of the Eclipse search engine is that it performs super-type matching[1] on:

a) Types when it searches for types.
b) The declaring-type and on each argument of a method signature when it searches for methods.

Evaluation

In our example, the Eclipse search engine successfully found all method invocations for the method names given as an input (`java.sql.Connection.createStatement`, `java.sql.Statement.executeQuery`, `java.sql.Statement.executeUpdate`, `java.sql.ResultSet.getString`, etc.). It is also possible to search just by the method name, but in that case, the tool will not perform any type analysis.

1 Go to appendix A for a definition.

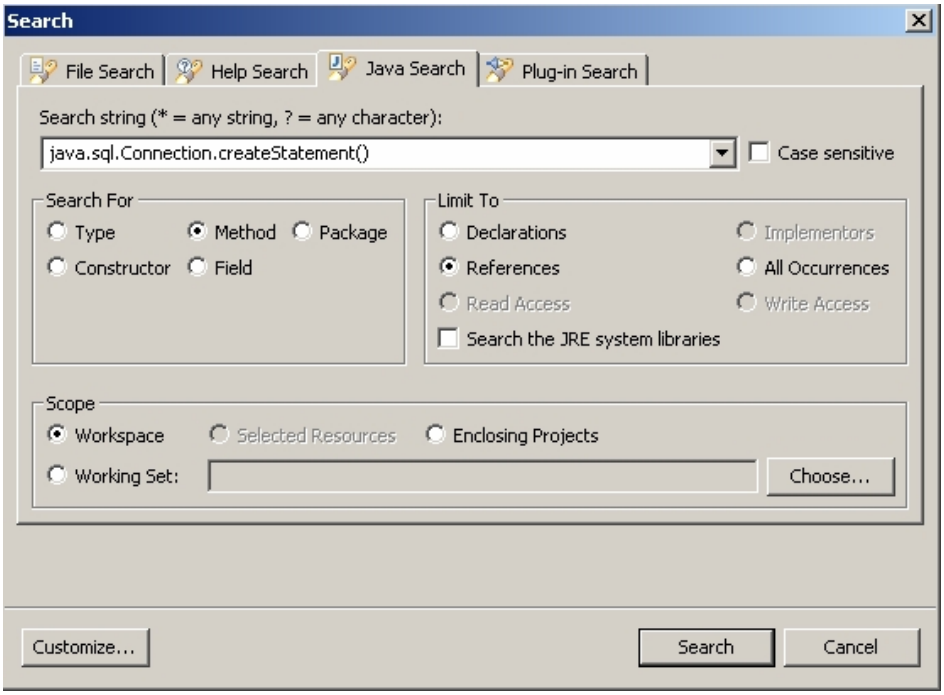

Figure 3.3: Eclipse screen shot.

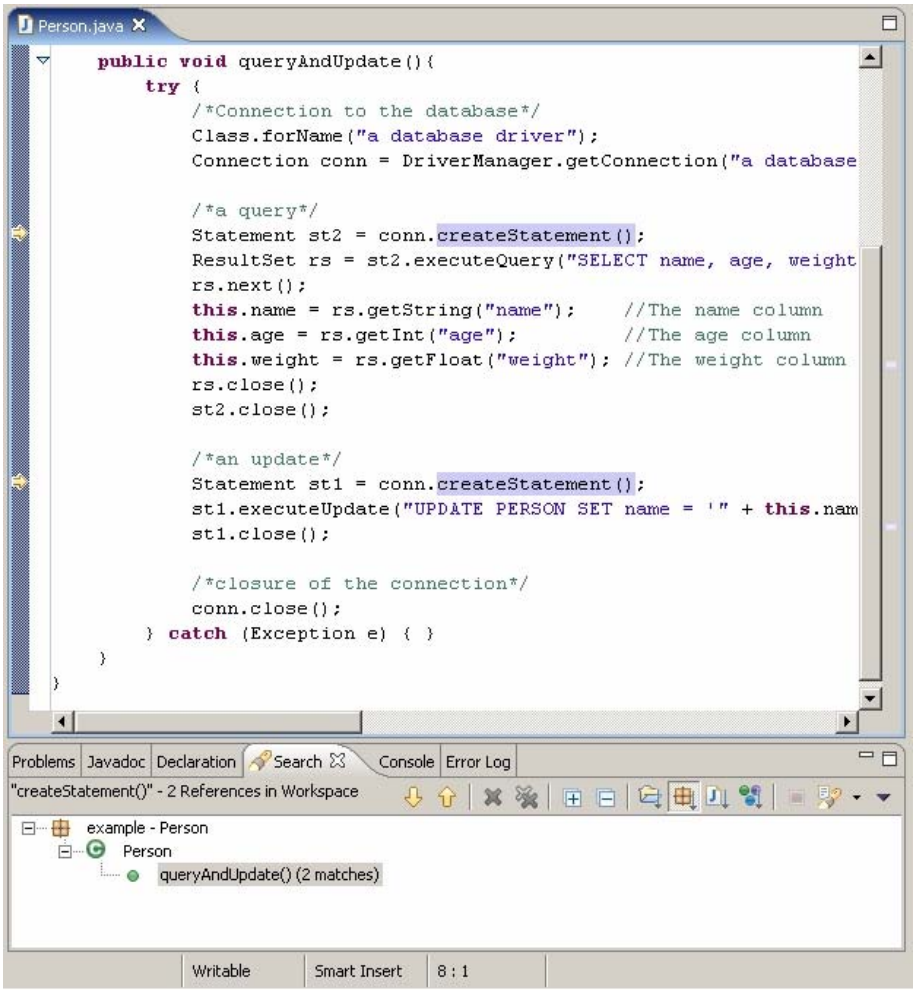

Figure 3.4: Eclipse screen shot.

The search was equally successful in the search of static method invocations (`java.lang.Class.forName` and `java.sql.DriverManager.getConnection`). In addition, the tool was able to do the searches with either the short name or the full qualified name of method classes (example: `java.lang.Class.forName` and `Class.forName`).

The search engine is also able to apply pattern-matching method signatures (examples: `java.sql.ResultSet.getInt(*)`, `java.sql.ResultSet.getInt(?)`, `java.sql.ResultSet.getInt(String)`, etc.).

The result of the search is displayed in a tree view. The top level nodes represent the project packages. In the next level nodes represent the classes and the third level nodes represent the classes' method declarations where a match is found. The results are linked to the source code, so when a user does double click on one of the nodes, the Eclipse framework will display the corresponding line of code.

Limitations

The Eclipse search engine is just that, a search engine. It does not allow users to create concern data structures and to add query results to them. Moreover, it only allows one search at a time. Two screen shots of the Eclipse Framework can be found in figures 3.3 and 3.4.

3.1.2.4 PRISM

Overview

The PRISM tool [23, 24] supports java code and partially supports C# code. Aspects are defined as method invocations and the user is allowed to input pattern-matching expressions to describe the method invocation signatures. In addition, it supports the "`*`" wildcard, which denotes any string, and "`(..)`" which denotes any number and type of arguments.

Some examples of valid inputs are:

```
className.methodName(..)
className.methodName
className.*
className.*(..)
*.methodName(..)
className.methodName(type1, type2, .. )
```

Nevertheless, it is not possible to describe arguments types as pattern-matching expressions. The user has to enter the complete arguments' type names.

It is possible to characterize class and method names with the following pattern-matching expressions:

```
fragmentName*
*fragmentName
*fragmentName*
```

Some invalid pattern-matching expressions are:

```
className
className.methodName()
className.methodName(*)
```

PRISM displays its results in a tree view. The top level nodes represent classes and the next level nodes represent the lines of code where each pattern-matching is found. The results are also linked to the source code, so it is possible to jump to the corresponding source code with a double click on a tree node.

The tool allows the creation of groups of pattern-matching expressions. In addition, it offers a ranking view which reports the most frequently used types across method invocations.

Evaluation

In our example, we were able to create data structures for our operations (Connection, Query, Update and Closing) and we were able to characterize our operations as a collection of method-name pattern-matching expressions. For instance, the Update operation was defined as the following collection of expressions:

```
java.sql.Connection.createStatement(..)
java.sql.Statement.executeUpdate(..)
java.sql.Statement.close(..)
```

The tool also allowed us to add an expression in more than one operation. For instance, the `Connection.createStatement` and `Statement.close` methods, which are shared by the *Query* and *Update* operations, were added to both of them. Nevertheless, the tool associates their method invocations simultaneously

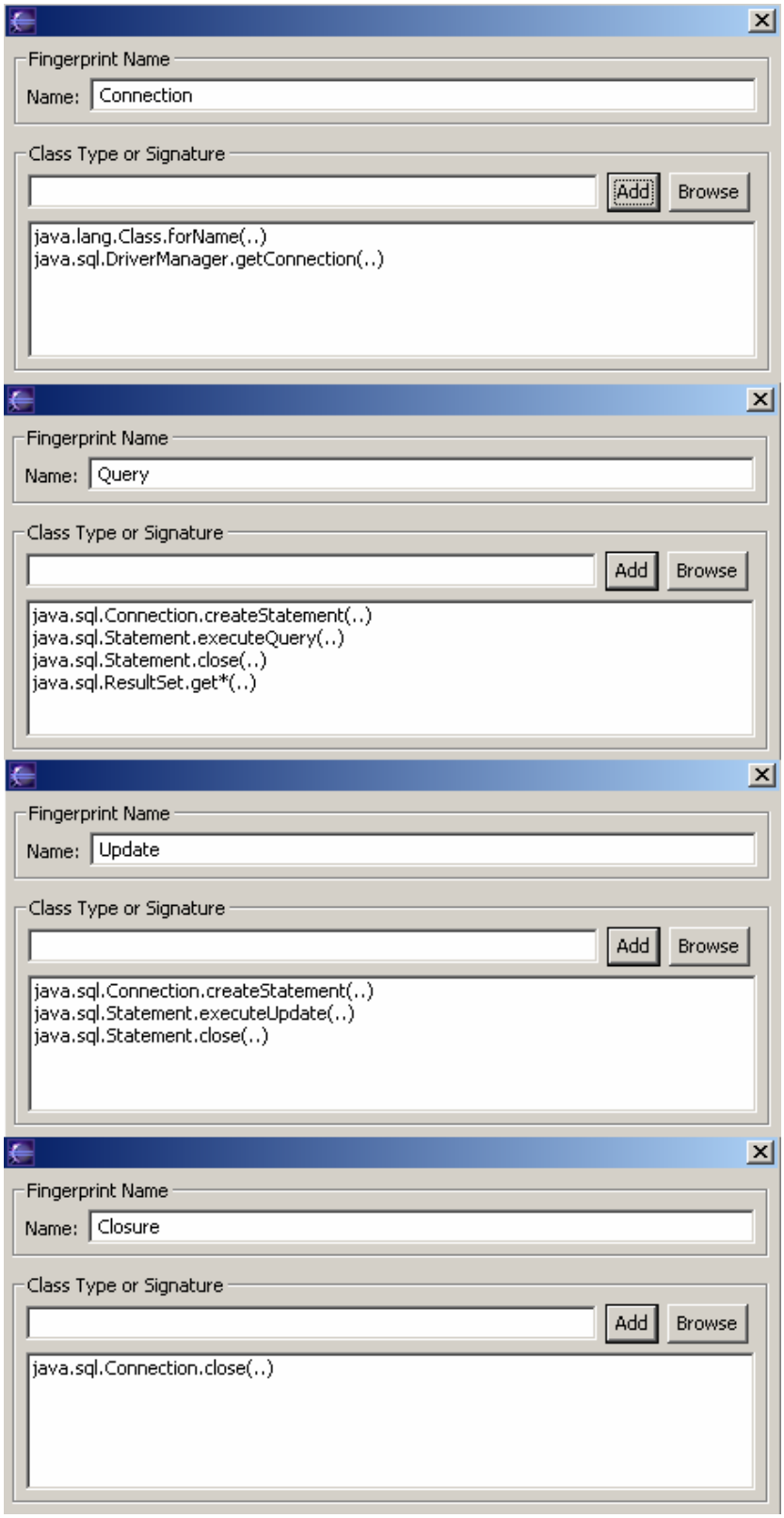

Figure 3.5: PRISM screen shot.

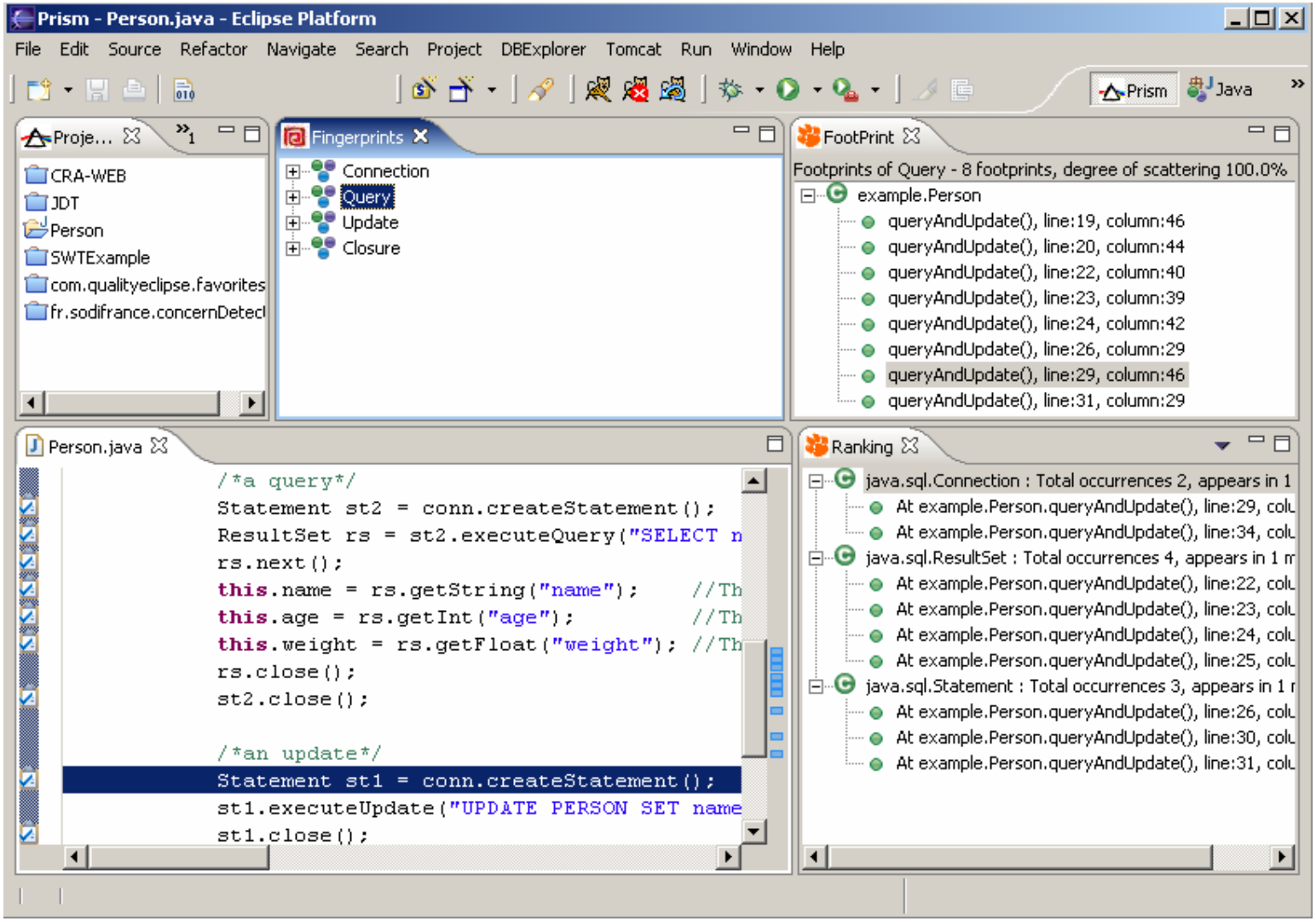

Figure 3.6: PRISM screen shot.

to both operations. It cannot distinguish when such method invocations are used to perform one or the other operation.

Limitations

In the case of methods invocations that are shared by more than one operation, the tool is unable to distinguish to which operation a specific method invocation is used to.

It does not perform a super-type matching on the method's declaring-type and on each of its arguments. For example: suppose we have the following type definitions:

```
interface superinterface
{
  public void methodA();
}
```

```
class subtype implements superinterface
{
  public void methodA() { /*some code*/ }
}
```

The expression `superinterface.methodA(..)` is unable to detect the method invocation in:

```
subtype sub = new subtype();
sub.methodA();
```

The expression `superinterface.methodA(..)` is unable to detect the method invocation in:

```
subtype sub = new subtype();
sub.methodA();
```

Figures 3.5 and 3.6 offer some screenshots of the PRISM tool.

3.1.2.5 CME

Overview

The Concern Manipulation Environment (CME) [36] is an Eclipse open-source project aimed to support the identification, extraction and composition of concerns. It is based on the premise that concerns can be encapsulated and that they should be treated as first-class entities [38, 39]. The CME allows developers to identify concerns in existing software, regardless whether they are implicit (scattered in the code) or if are already encapsulated with AspectJ [33].

In CME, concerns are modeled as elements and their relationships. Elements can be classes, interfaces, fields, methods, advices or pointcuts. However, the extraction and composition of concerns is not yet implemented.

CME provides a query capability [40] to help in the identification of concerns in existing software. The results of a query can be used to define a new query or to enlarge an existing one. The definition can be *extentional*, where actual elements found are added to the concern, or *intensional*, where the concern's content are defined by the query itself, and so the concern's elements are updated as the underlying software changes.

Since queries play a prominent role in the identification (detection) and the weaving (composition) of concerns, CME has adopted a uniform, shared query language. CME has also embedded some AspectJ pointcut–style queries in order to make this tool natural for AspectJ developers.

Pointcuts in AspectJ and related Aspect Oriented technologies are queries over runtime events where advice can be applied, such as object creation, method call or execution, or field access.

Queries are used to describe points of interest in software. The CMD offers three categories of searches: the *AspectJ Compatible Search*, the *Artifact Search* and the *Concern Model Search*.

The *AspectJ Compatible Search* category allows users to execute queries using an AspectJ-style language in order to make queries familiar to AspectJ developers. Table 3.2 offers a contrast between this search category and the Eclipse Search Engine.

Both tools receive a name characterization for the queries. The differences are present in the query against fields, in which CME may also receive a characterization of the field type. The search against types is also different in both tools. CME searches for static initializations and references in try-catch block, whereas the Eclipse Search engine searches for type declarations and references. Another feature in CME is that queries for methods may also include the returning type.

Element	**CME AspectJ Compatible Search**	**Eclipse equivalent query**
Method	Calls	Declarations
	Execution	References

Constructors	Calls	Declarations
	Execution	References
Fields	Get	Reads
	Set	Writes
Types	Static Initialization	n/a
	Try-Catch Handling	n/a

n/a: not available.

Table 3.2: Comparison between the CME *AspectJ Compatible Search* and the Eclipse Search Engine.

The *Artifact Search* category allows users to find artifacts (source code constituent parts), regardless whether they belong to a java type definition (in a *.java* file) or to an aspect definition (in a *.aj* file). The searchable elements or artifacts are:

- Type
  - Aspect
  - Class
  - Interface
- Member
  - Field
  - Operation
    - Advice
    - Method
  - Pointcut
- Project
- Package

Lastly, the *Concern Model Search* category allows users to query relationships among artifacts. Relationships are composed by a source, a target and a name. The source and the target could be any artifact, and the name of the relationship may be *dependsOn*, *extends*, *implements*, *invokes* and *refersTo*.

The allowed wildcards in CME are:

- "*", which denotes any string.
- "..", which denotes any package, or anything when describing method parameters.
- "+" which includes all subtypes in the query.

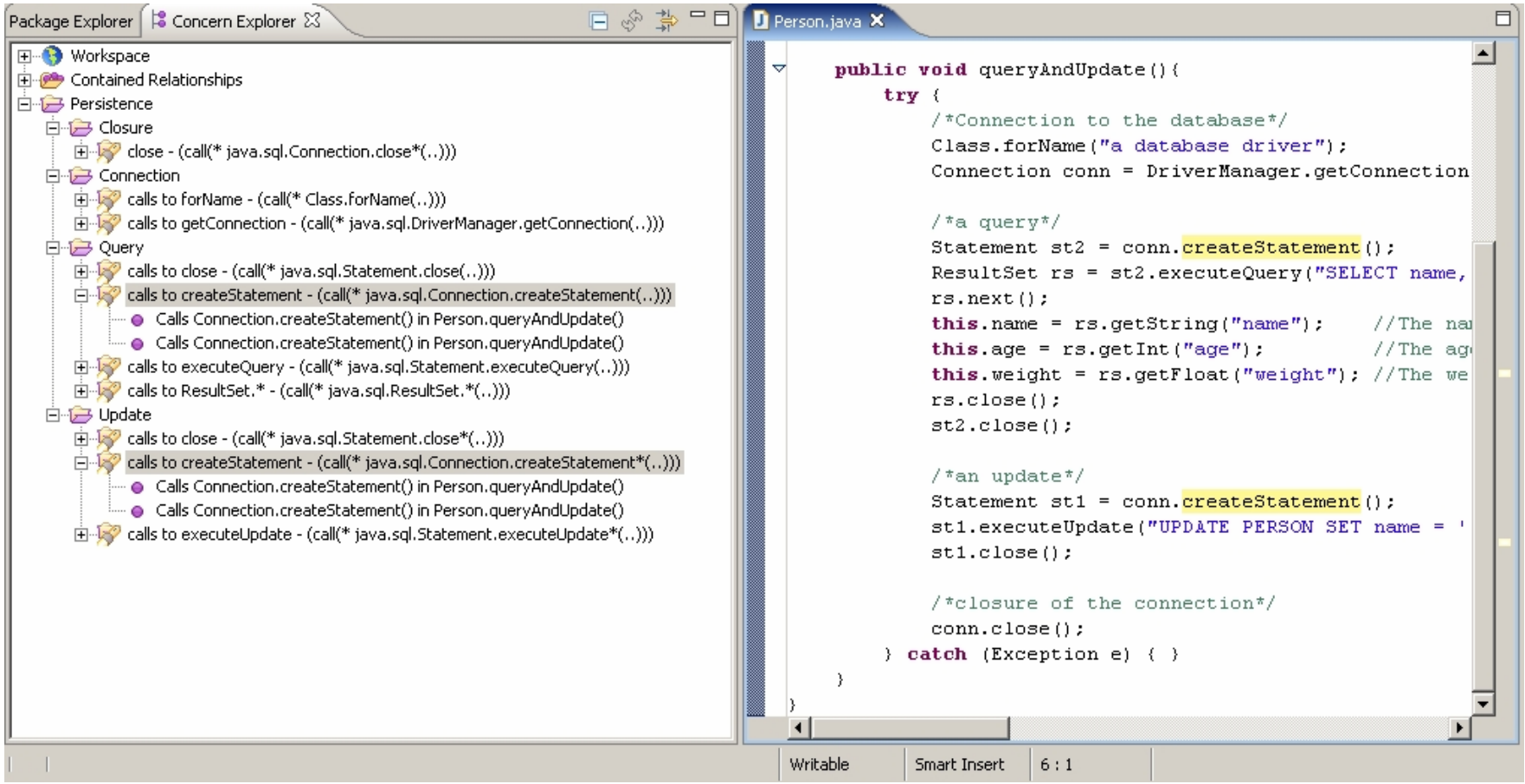

Figure 3.7: CME screen shot.

The tool can also include the logical operators "||" and "!", and queries may include modifiers (private, public, protected, final, static, etc.) and AspectJ keywords like `within` and `withincode`.

Lastly, some examples of the CME AspectJ pointcut-style queries are:

Searching for	**Query to use**
All public or protected methods. returning a `String`.	`(public || protected) method String *(..)`
All fields of type `int`.	`field int *`
All calls to any method named "`foo`", from classes or aspects matching `p.C*`.	`call(* foo(..)) && within(p.C*)`
All calls to any `foo` method occurring within any `bar` method.	`call(* foo(..)) && withincode(* bar(..))`

Evaluation

We were able to create data structures for the Connection, Query, Update and Closing operations.

As with PRISM, we defined each operation as method invocations to the methods of interest for each operation. To do that, we made `calls` queries to the method signatures we were interested.

A screenshot of CME can be found in figure 3.7.

Limitations

As with PRISM, the `java.sql.Connection.createSatement` and `java.sql.Statement.close` method calls are added to both Query and Update operations. The tool cannot determine to which operation the method invocation actually belongs. This produce distracting results and the user needs to expend time in analyzing not only the results, but also the source code, in order to determine to which operation each method call belongs.

3.1.3 Exploratory tools

Exploratory tools incorporate semantic information to navigate source code. They focus on providing intelligent exploratory capabilities, with the user controlling much of the function, in order to lead the user to the discovery of an

aspect. These tools give the user ways to navigate more quickly and intelligently around source code.

3.1.3.1 JQuery

Overview

JQuery [25, 26] is an exploratory tool that allows users to do hierarchical code browsing and query searches. It extends an earlier prototype called QJBrowser [27] and its query language is built on top of TyRuBa [28], which is an expressive logic programming language similar to Prolog[29].

JQuery provides a generic mechanism for constructing tree views from queries or code navigation based on particular kinds of relationships. Additionally, the tool allows users to incrementally extend these tree views using additional queries. JQuery performs an initial source code parsing to build a logic database of program information. Users can type queries, or select from a set of predefined queries, which JQuery will execute and then will construct a tree view from the query results. The tree views can be incrementally extended with further queries.

JQuery offers predefined top-level queries that serve as starting points for explorations. To support continued exploration, a JQuery tree can be incrementally refined by the developer. At each node in the tree the developer may wish to explore further and may choose to extend the current view with a new subtree. The subtree shows the results of a selected query that finds code units connected to the selected unit through some relationship of interest. Table 3.3 shows the JQuery predefined top-level queries.

Abstract Classes	Displays a tree containing all abstract classes.
Abstract Method Browser	Displays a tree containing all abstract methods.
Bookmarks	Displays a tree containing all the user defined bookmarks.
Class Creation	Displays a tree with all lines of code creating classes' instances through the `new` operator.
Compiler Errors	Displays a tree with all compilations errors.
Compiler Warnings	Displays a tree with all compilations warnings.
InstanceOf Testing	Displays a tree with all lines of code using the `instanceof` keyword.

Interface Implementation	Displays a tree with all interfaces implementations.
Java Structure Browser	Almost the same as the Eclipse Package Explorer view. It displays a tree with all packages, *.java* and *.class* files, classes, interfaces, fields and methods.
Method Browser	Displays a tree with all methods.
Package Browser	Displays a tree containing all packages, classes and interfaces.
Tasks	Displays a tree containing all the user defined tasks.

Table 3.3: JQuery predefined top-level queries.

In the exploration process, developers can expand a tree node with the help of a contextual menu offered by the tool. The contextual menu is specific for each type of node and contains a list of all the ways in which the tree can be extended at that node. An important difference between JQuery and the CME tool is that queries in CME need to be typed by the user, whereas queries in JQuery are offered in contextual menus, easing the navigation of code. Table 3.4 shows the predefined queries offered for each type of node.

Node	Category	Relationships
Package		Top-level classes
		Top-level interfaces
		All top-level types
Class	Members	Initializers
		Fields
		Constructors
		Methods
	Inheritance	Inherited methods
		Inherited fields
		Supertypes
		Subtypes
		Implemented interfaces
		Superclasses
		Subclasses

Method	Calls	Creates
		Incoming calls
		Outgoing calls
	Signature	Signature
		Modifiers
		Arguments
		Returns
		throws
	Field accesses	Reads fields
		Writes fields
		Reads/Writes fields
	Inheritance	Method hierarchy
		Inherited by
		Override
		Overridden by
Fields		Read by
		Written by
		Read/Written by
		Type of field

Table 3.4: JQuery predefined queries.

Users are also allowed to expand a tree node by doing queries using a logic language built on top of TyRuBa [28].

TyRuBa Language Overview

- Type Hierarchy

JQuery has the following TyRuBa type hierarchy:

- Element
  - Package
  - CU (Compilation Unit)
  - Field
  - Type
    - Primitive
    - RetType (interface or class)

- Block
  - Initializer
  - Callable
    - Method
    - Constructor
- Marker
  - Bookmark
  - warning
  - Error
  - Task

➢ Core Predicates

Unary predicates:

Predicate	Description
`cu`	`cu(?X)` means: "`?X` is a Compilation Unit (.class or .java file)"
`package`	`package(?X)` means: "`?X` is a package"
`class`	`class(?X)` means: "`?X` is a class"
`interface`	`interface(?X)` means: "`?X` is an interface"
`method`	`method(?X)` means: "`?X` is a method"
`constructor`	`constructor(?X)` means: "`?X` is a constructor"
`initializer`	`initializer(?X)` means: "`?X` is an initializer"
`field`	`field(?X)` means: "`?X` is a field"
`bookmark`	`bookmark(?X)` means: "`?X` is a bookmark"
`warning`	`warning(?X)` means: "`?X` is a compiler warning"
`error`	`error(?X)` means: "`?X` is a compiler error"
`task`	`task(?X)` means: "`?X` is a task"

Table 3.5: TyRuBa unary predicates.

Binary predicates:

Predicate Name	Argument Types	Description
`priority`	Task, String	`priority(?T,?P)` means: "Task `?T` has priority `?P`"

`name`	Element, String	`name(?E,?S)` means: "Element `?E` has name `?S`"
`child`	Element, Element	`child(?Sup,?Sub)` means: "Element `?Sup` has a child `?Sub`"
`extends`	RefType, RefType	`extends(?C1,?C2)` means: "Class (or Interface) `?C1` extends Class (or Interface) `?C2`"
`implements`	RefType, RefType	`implements(?C,?I)` means: "Class `?C` implements Interface `?I`"
`throws`	Callable, RefType	`Throws(?C,?T)` means: "Callable `?C` throws `?T`"
`type`	Field, Type	`type(?F,?T)` means: "Field `?F` is of type `?T`"
`modifier`	Element, String	`modifier(?E,?S)` means: "Element `?E` has modifier (i.e public, private, static, etc) `?S`"
`arg`	Callable, Type	`arg(?C,?T)` means: "Callable `?C` has an argument of type `?T`"
`returns`	Callable, Type	`returns(?C,?T)` means: "Callable `?C` returns Type `?T`"
`signature`	Callable, String	`signature(?C,?S)` means: "Callable `?C` has signature `?S`"

Table 3.6: TyRuBa binary predicates.

Some sample queries are:

- `class(?C), method(?C, ?M), name(?M, main)`. Returns all classes that have a method called `main`.
- `class(?C), field(?C, ?F), name(?F, id)`. Returns all classes that have an attribute called `id`.
- `class(?C), implements(?C, ?I), name(?I, persistent)`. Returns all classes that implements the interface called `persistent`.

Evaluation

It is necessary to indicate to the JQuery tool which files to parse. In our evaluation, we selected our own project and the `java.lang.Class`, `java.sql.Connection`, `java.sql.ResultSet` and `java.sql.Statement` classes from the `rt.jar` file of the Java Runtime Environment System Library.

After that, we could expand each class node by querying their methods, and then, from each method of interest, we expanded the method nodes by querying their incoming calls. The results of the queries are the lines of code that makes such methods calls.

The tool does not allow users to characterize aspects and to create concerns data structures, but it allows the creation of trees of navigation. In our example, those trees are the incoming calls for the methods that we are interested in.

Limitations

JQuery is a tool that is related to, but not explicitly built for aspect mining. It allows user to quickly and intelligently browse the source code, and it provides considerable help in developing an understanding of how a program works.

The user needs to be familiar with the Eclipse Framework capabilities in order to perform queries in built-in java classes.

It is not possible to create concerns data structures and to associate a query result to them. Some screenshots are shown in figures 3.8 and 3.9.

3.1.3.2 FEAT

Overview

The Feature Exploration and Analysis Tool (FEAT) [30, 31, 41] introduces the concept of Concern Graph. A Concern Graph abstracts the implementation details of a concern by storing the key structure implementing a concern. By storing structure, a Concern Graph documents explicitly the relationships between different elements of the concern. More precisely, a Concern Graph is a subset of a structural program model built by FEAT.

The program model represents the declaration and uses of various program elements of class-based object-oriented languages. Formally, a program is expressed as a graph $P = (V, E)$, where V is the set of vertices, and E is the set of labeled, directed edges.

A vertex in P can be one of three types.

- **Class vertex I** represents a global class or interface, without its members.
- **Field vertex (F)** represents a field member of a class.
- **Method vertex (M)** represents a method member of a class.

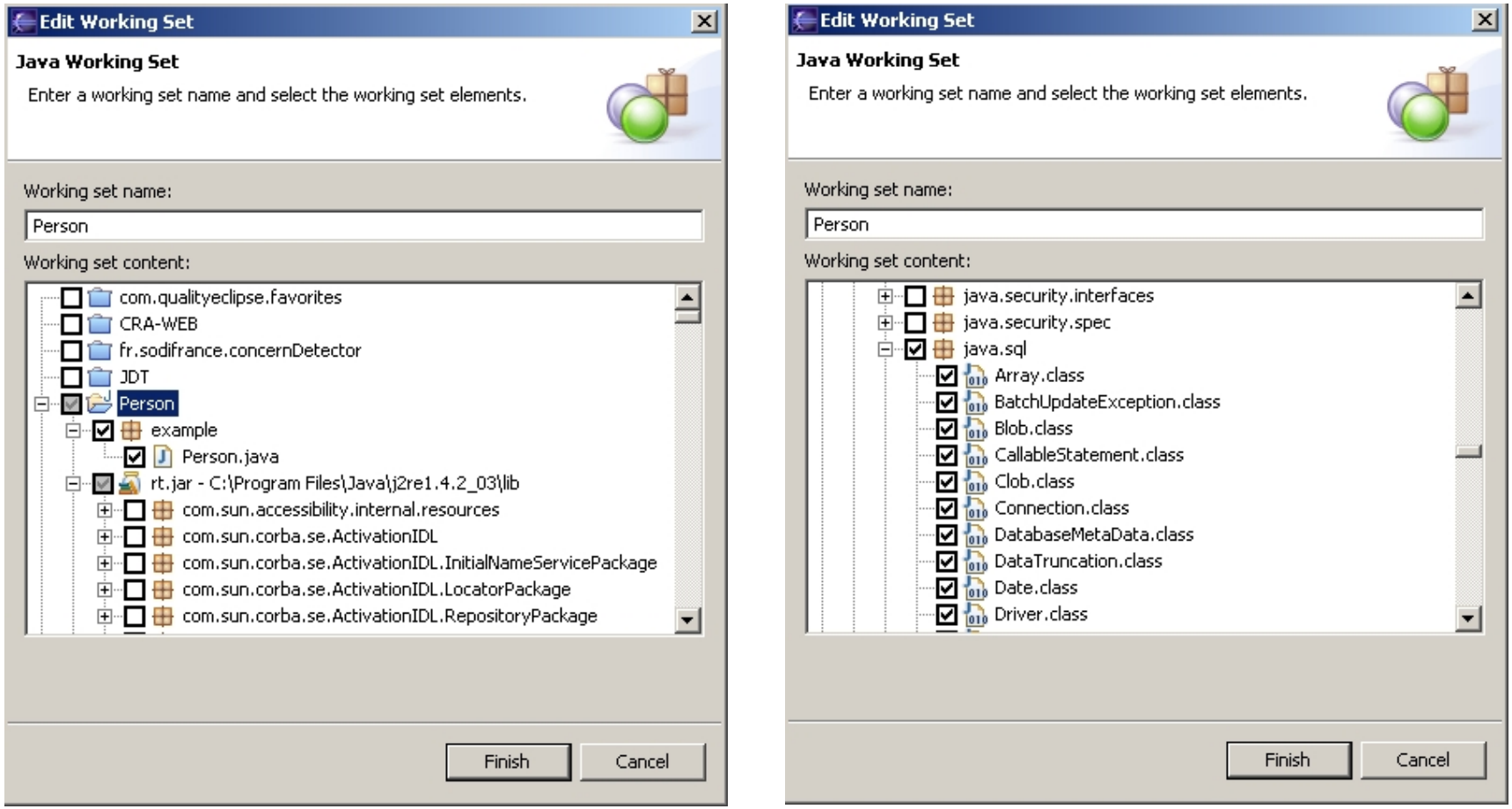

Figure 3.8: JQuery screenshot.

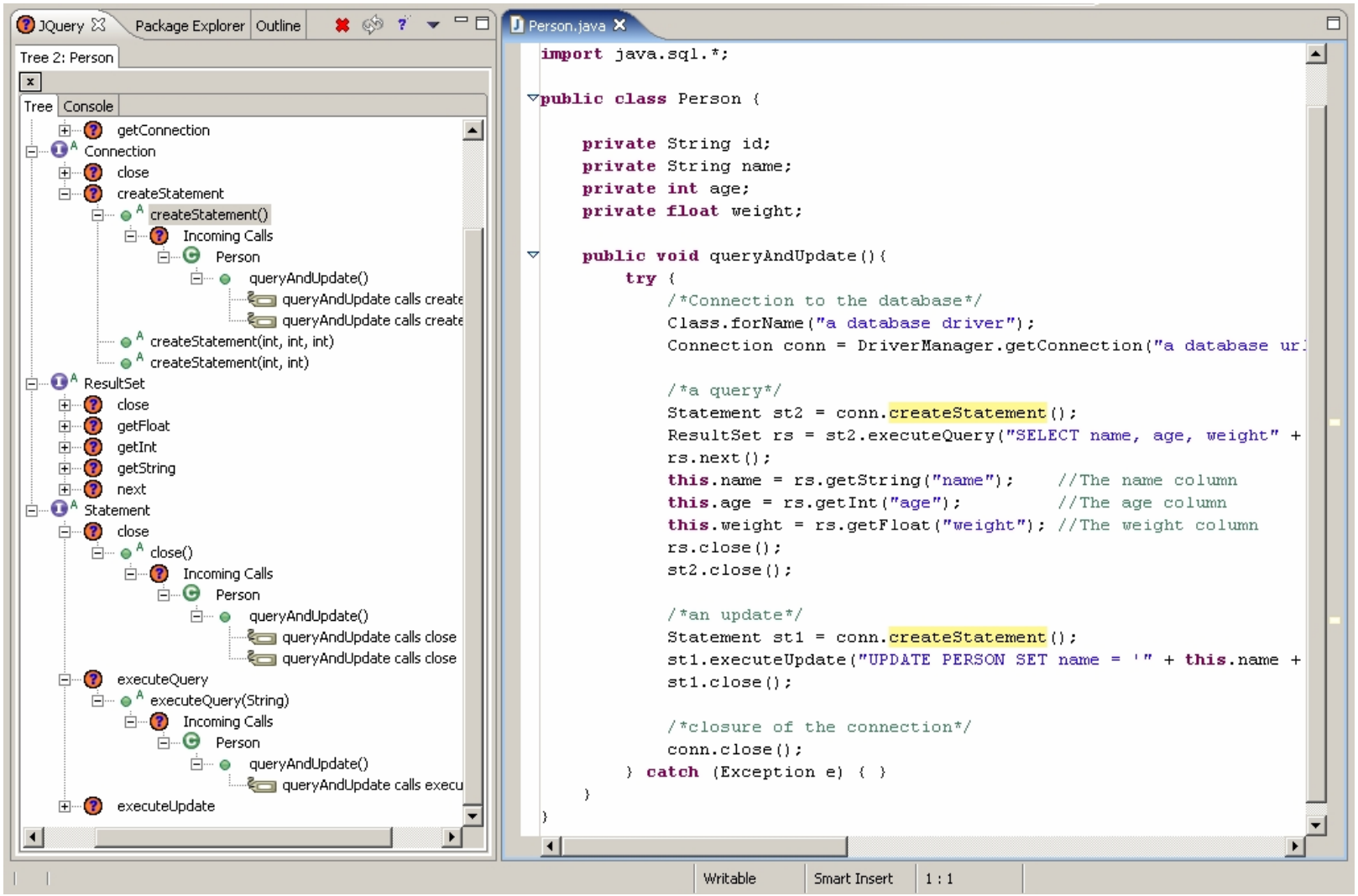

Figure 3.9: JQuery screen shot.

An edge in *P* can be one of six types, depending on the type of vertices it connects: (M, M), (M, F), (M, C), (C, C), (C, M), and (C, F). Edges are labeled with the semantic relationships they represent.

Some examples of edges that connect vertices of *P* are:

Name	Type	Description
(**calls**, *m1*, *m2*)	(M, M)	The body of *m1* contains a call that can bind (dynamically or statically) to *m2*.
(**reads**, *m*, *f*)	(M, F)	The body of method *m* contains an instruction that reads a value from field *f*.
(**writes**, *m*, *f*)	(M, F)	The body of method *m* contains an instruction that writes a value to field *f*.
(**checks**, *m*, *c*)	(M, C)	The body of method *m* checks the class of an object, or casts an object, to *c*.
(**creates**, *m*, *c*)	(M, C)	The body of method *m* creates an object of class *c*.
(**declares**, *c*, {*f* | *m*})	(C, F | M)	Class *c* declares method *m* or declares field *f*.
(**superclass**, *c1*, *c2*)	(C, C)	Class *c2* is the superclass of *c1*.

Table 3.7: Some relationships in FEAT.

For example, if a class called `Multiplier` has a method called `product(int, int)`, there will be an edge from `Multiplier` to `product(int, int)` called **declares**.

In FEAT, an aspect is defined as a subset of the graph *P* documenting the implementation of a concern in *P*, and it is stored in a structure called Concern Graph.

FEAT allows users to search the source code by performing queries of relations between different elements (such as fields and methods), and to keep track of elements and relations that are of interest. These elements and relations are saved in a Concern Graph.

A Concern Graph is display by FEAT as trees. The root of each tree is a class that contributes to the implementation of a concern. FEAT provides a set of queries to enable users to access vertices of the program model that are related to the vertices in the Concern Graph. A user can navigate the program model in both the direct and reverse directions of the edges emanating from the vertices.

There are two categories of queries in FEAT:

- Fan-in: returns all the vertices in the program model that depend on the selected class, field or method node.
- Fan-out: returns all the outgoings edges for the selected node. Fields do not have outgoing edges.

A complete list of queries supported by FEAT can be found in table 3.8.

Element	**Query Category**	**Relation**
Class/Interface	Fan-in	Checked by
		Created by
		Extended by
		i-extended by
		Implemented by
		Transitively extended by
		Transitively implemented by
		Referenced by
	Fan-out	Declaring
		Extending
		i-extending
		Implementing
		Transitively extending
		Transitively implementing
method	Fan-in	Called by
		Overridden by
		Referenced by
	Fan-out	Checking
		Creating
		Having p-types
		Having r-types

		Accessing
		Calling
		Overriding
		Using
Field	Fan-in	Accessed by
		Referenced by
	Fan-out	-----

Table 3.8: FEAT supported queries.

Evaluation

We were able to create a Concern Graph for each operation (i.e.: Connection, query, Update and Closing).

Just like the JQuery tool, with FEAT we had to use the Eclipse Search Engine in order to get the type-declaration reference of the `java.lang.Class`, `java.sql.DriverManager`, `java.sql.Connection`, `java.sql.Statement` and `java.sql.ResultSet` types.

Once got those references, we performed a *fan-out/declaring* query, and we added the appropriate methods to each Concern Graph. It was possible to add an element (in this case a method) to more than one Concern Graph.

Additionally, for each method in each Concern Graph, we perform a *fan-in/called-by* query to find out all invocations to our methods of interest. The result yielded the `queryAndUpdate()`method declared in the class `Person`, and we added that relation to our concerns. Since the method `Person.queryAndUpdate()` is calling all the methods of our concerns, it was added to all the Concern Graphs.

Finally, the FEAT tool is able to compare two Concern Graphs at a time, and discover any "collision" between them. Figure 3.9 displays the comparison between the Query and the Update Concern Graphs.

The methods `java.sql.Connection.createStatement()` and `java.sql.Statement.close()` are present in both Concern Graphs. This is what we call an aspect collision, and FEAT flagged those elements with a red diamond.

A screen shot of the FEAT tool is presented in figure 3.10.

Limitations

Similar to PRISM, FEAT is unable to distinguish whether a `java.sql.Connection.createStatement()` and a `java.sql.Statement.close()` method invocation is used to perform a Query or an Update operation.

The user also needs to be familiar with the Eclipse Framework capabilities in order to include built-in java classes in the exploration task.

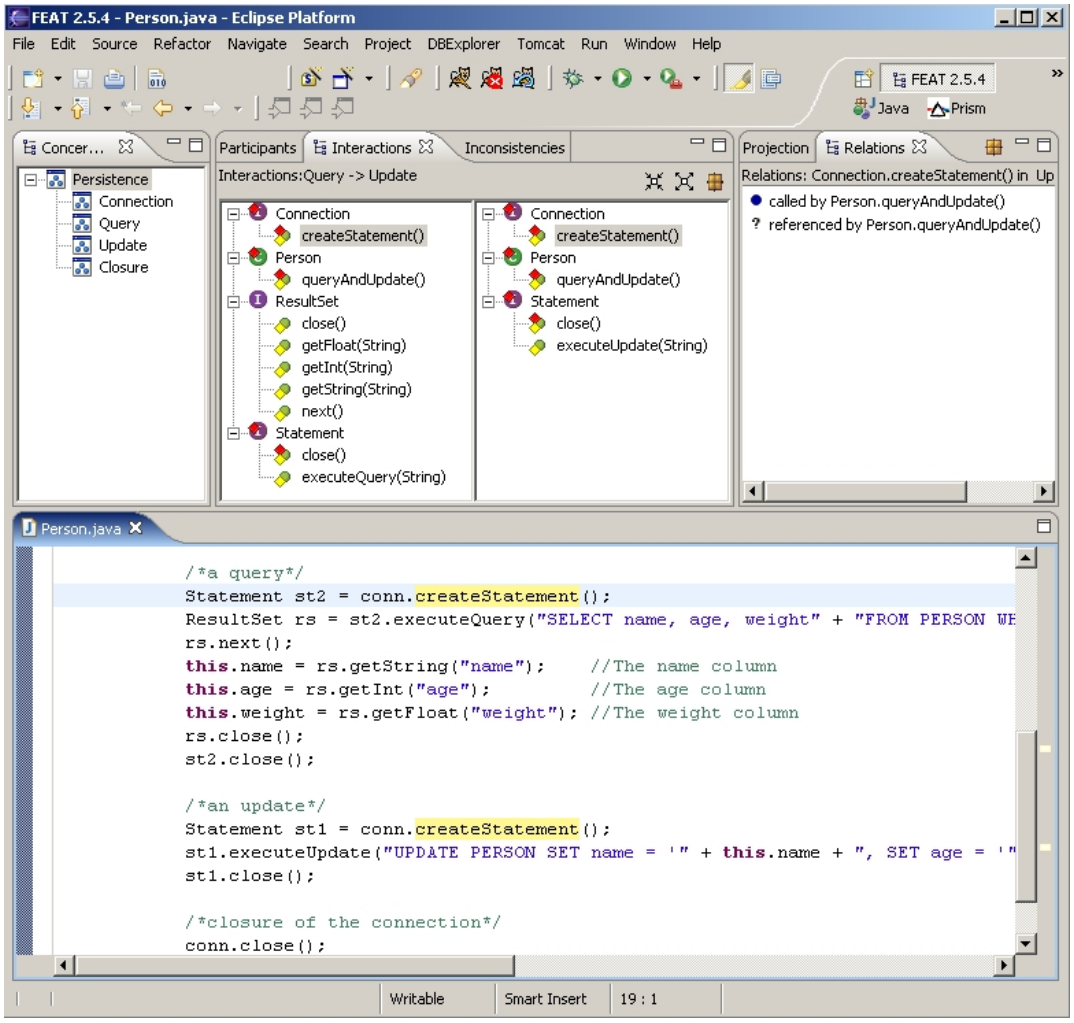

Figure 3.10: FEAT screen shot.

3.1.4 Comparison of non-automatic aspect mining tools

Table 3.9 makes a comparison of non-automatic aspect mining tools.

	Search capabilities						Browsing Capabilities
	Text pattern	**Type usage**	**Method invocation**	**Other**	**Super-type matching**	**Allowed wildcards**	
Aspect Browser	√	X	X		n/a	"*"	n/a
AMT	√	√	X		X	none	n/a
Eclipse	√	√	√	• *Java* artifacts[1] • Relationships[3]	√	"*","?", "\"	*Java* artifacts
PRISM	X	X	√		X	"*", ".."	n/a
CME	X	√	√	• *Java* artifacts • *AspectJ* artifacts[2] • Relationships	√	"*", "..", "+"	n/a
JQuery	n/a	n/a	n/a		n/a	n/a	• *Java* artifacts • Relationships
FEAT	n/a	n/a	n/a		n/a	n/a	• *Java* artifacts • Relationships

n/a: not applicable.
1 *Java* artifacts: type, method and field.
2 *Aspectj* artifacts: aspect, pointcut and advice.
3 Relationships: declares, declared by, calls, called by, reads, read by, etc.

Table 3.9: Comparison of non-automatic aspect mining tools.

	Allows creation of concern data structures	Allowed characterization constructs	Allows repeated constructs in different concerns	Extra analysis performed	Users need to explicitely include built-in class files
Aspect Browser	√	Text pattern	X	• Aspects match count • Most common identifiers • Redundant lines of code	n/a
AMT	X	•Type usage •Text pattern	n/a	X	X
Eclipse	X	n/a	n/a	Match count	n/a
PRISM	√	Method invocations	√	Ranking of type usage	X
CME	√	•*Java* artifacts •*AspectJ* artifacts •Relationships •*AspectJ*-like queries	√	X	X
JQuery	X	n/a	n/a	X	√
FEAT	√	•*Java* artifacts •Relationships	√	Collision between two concern definitions	√

n/a: not applicable.

Table 3.9: Comparison of non-automatic aspect mining tools. (continuation)

3.2 Automatic aspect mining tools

These tools do not require any input from the user and are intended to identify aspect candidates automatically.

3.2.1 Clone code detection

Overview

The objective for clone code detection is to factor out copy-paste-adapt code. It relies on the assumption that crosscutting code is typically duplicated over the entire application, and could be identified using clone detection algorithms.

Several clone detection techniques have been proposed:

- **Text-based** techniques [55, 56] perform little or no transformation to the 'raw' source code before attempting to detect identical or similar (sequences of) lines of code. Typically, white space and comments are ignored.
- **Token-based** techniques [57, 58] apply a lexical analysis (tokenization) to the source code, and subsequently use the tokens as a basis for clone detection.
- **AST-based** techniques [59] use parsers to first obtain a syntactical representation of the source code, typically an abstract syntax tree (AST). The clone detection algorithms then search for similar subtrees in this AST.
- **PDG-based** approaches [60, 61] go one step further in obtaining a source code representation of high abstraction. Program dependence graphs (PDGs) contain information of semantical nature, such as control and data flow of the program.

Limitations

These techniques usually suffer from long execution times.

3.2.2 Fan-in analysis

Overview

The fan-in analysis [62] search for methods that are called from many different places (and hence have a high fan-in) and whose functionality is needed across different methods, classes and packages.

This method relies on the observation that scattered and crosscutting functionality that largely affects the code modularity is likely to generate high fan-in values for key methods implementing this functionality. Some examples include: logging, tracing, pre and post-condition checks and exception handling.

A fan-in of a method *m* is defined as the number of distinct method bodies that can invoke *m*. Because of polymorphism, one method call can affect the fan-in of several other methods. An example is shown in figure 3.11. Three different calls to polymorphic method *m* are contained in class D. The resulting sets of callers and corresponding *fan-in* values are shown in Figure 3.12. Observe that the call in f2 to B's *m* contributes to the fan-in of m in B's supertypes (A) as well as its subclasses (C1 and C2).

```
interface A {
  public void m();
}
class B implements A {
  public void m() {};
}
class C1 extends B {
  public void m() {};
}
class C2 extends B {
  public void m() { super.m();}
}
class D {
  void f1(A a) { a.m(); }
  void f2(B b) { b.m(); }
  void f3(C1 c) { c.m(); }
}
```

Figure 3.11: Various polymorphic method calls.

Method	Caller set	Fan-in
A.m	D.f1, D.f2, D.f3	3
B.m	D.f1, D.f2, D.f3, C2.m	4
C1.m	D.f1, D.f2, D.f3	3
C2.m	D.f1, D.f2	2

Figure 3.12: Fan-in values for code in figure 3.11.

The fan-in analysis follows three consecutives steps:

Step1: Automatic computation of the fan-in metric for all methods in the source code.
Step2: Filtering the results:

- Restrict the set of methods to those having a fan-in above a certain threshold (for example 10).
- Filter getters and setters.
- Filter utility methods, like `toString()`, collection manipulation methods, etc.

Step 3: Manual analysis of the remaining set of methods.

Limitations

This technique may produce false positives (returned aspect candidates that are not crosscutting concerns). It may also miss some aspect candidates with low fan-in.

3.2.3 Formal concept analysis

Overview

Instead of being an aspect mining technique, formal concept analysis [63, 64] can be considered a software mining technique aimed to improve program understanding and maintenance. This technique identify meaningful grouping of elements that have common properties in a structure called *formal concept*.

A prototype called DelfSTof was implemented in Smalltalk. As elements, this prototype uses classes and methods; as properties, it uses the substrings appearing in their names. As a result, formal concepts will group classes and methods with similar names. The choice for these properties was motivated for the naming conventions that programmers usually employ.

The DelfSTof prototype is able to find:

- Polymorphic methods: methods that have exactly the same name, but do not belong necessarily to the same class hierarchy, since Smalltalk is dynamically typed, and it allows any class to be substituted for another one, as long as it defines the required method.

Polymorphic methods present in different class hierarchies are interesting to detect. This situation can be due to a case of a bad design, a case of bad naming, or a case of crosscutting behavior.

- Delegating method: methods that delegate responsibility by calling a method with the same name. The presence of many such delegating methods in a single class may indicate that the class is implementing the *decorator* design pattern [4].

- Code duplication: methods spread over different classes that not only have similar name, but a similar implementation as well. This case may indicate a crosscutting behavior.

- Design patterns: many design patterns use certain naming convention. For example, the *Visitor* design pattern [4] uses the convention that the `Visitor` class contains methods having the substring `Visit` in their names.

- Relevant domain concepts: frequently occurring properties (substrings in classes and method names) give a good idea of the important concepts in the application or problem domain.

Limitations

The formal concept analysis is not an aspect mining technique as such, although it improves program understanding and maintenance. The DelfSTof prototype is highly dependent on naming conventions. By considering substrings of class and method names, some elements (classes and methods) that actually belong together are divided over different formal concepts, simply because they do not share the same exact substring in their name.

The aspect mining contribution of this tool is the detection of polymorphic methods across class hierarchies, and methods spread over classes with similar names that could have similar implementation and could indicate a case of crosscutting behavior.

3.2.4 Execution trace analysis

Overview

This was the first aspect mining tool that detects crosscutting concerns based on dynamic analysis. The analysis uses program traces that are generated during program execution. These traces are then investigated for recurring execution patterns [8].

A program trace is a sequence of method invocation entries and exits. Aspect candidates are recurring execution relations. This technique distinguishes the following execution relations:

- Outside-Execution Relations
  - Outside-Before-Execution Relations:
    Method execution `u` before method execution `v`: `u(); v();`
  - Outside-After-Execution Relations:
    Method execution `u` after method execution `v`: `v(); u();`

- Inside-Execution Relations
  - Inside-First-Execution Relations:
    Method execution u first inside method execution v:
    ```
    v(){
      u();
      ...
    }
    ```
  - Inside-Last-Execution Relations:
    Method execution u last inside method execution v:
    ```
    v(){
      ...
      u();
    }
    ```

Recurring executions relations are considered aspect candidates. For example, in the following event trace:

```
a(){
  d();
  ...
}
...
b(){
  d();
  ...
}
...
c(){
  d();
  ...
}
```

There is a recurring inside-first-execution relation between the method `d()` and methods `a()`, `b()` and `c()`, so `d()` is considered an aspect candidate.

Limitations

While static analysis does not need any program execution, it is complete and input-insensitive. On the other hand, the dynamic analysis needs the execution of the program and is input-sensitive, and thus, a complete dynamic analysis is not applicable as it is impossible to execute all possible paths. Besides, it would slow down the software execution during the examination.

Moreover, Java API method executions do not appear in the program traces if the classes itself are not present as source code.

Furthermore, abstract and interface method traces are lost because of dynamic bindings at run-time. This fact can also produce false aspect candidates.

3.2.5 Formal concept of execution traces

Overview

This is another tool that performs dynamic code analysis, and it attempt to let requirements (use-cases) guide aspect identification by applying formal concept analysis to execution traces [65]. The relation between executed methods and use-cases is subject to concept analysis.

This technique is based on the assumption that a crosscutting functionality is implemented by code fragments spread across several classes.

The concept analysis will produce two categories of concepts:
- Use-specific concepts that groups methods that contribute to only one specific use case.
- Use-generic concepts that groups methods that contribute to more than one use-case.

The first case alone is not sufficient to identify crosscutting concerns, since it is possible that a given functionality might be decomposed into sub-functionalities assigned to distinct modules. The second case detects classes that contain methods that contribute to different functionalities. Such classes are considered aspect candidates.

Limitations

This technique not only requires the execution of the program being analyzed, but also requires users to define the use-cases for the main functionality. Users need to have a good understanding of the program functionalities in order to define the use-cases.

As with the *execution traces analysis* technique, this technique is input-sensitive and does not deal with code that is not executed, nor does it traces Java API methods for subsequence analysis.

3.2.6 Comparison of automatic aspect mining tools

Table 3.10 makes a comparison of automatic aspect mining tools. These tools actually use a hidden concern characterization.

Technique	Hidden concern characterization	Analysis includes Java API code
Code clone detection	Clone code. Source code can be represented as: • Text. • Lexical tokens. • AST nodes. • PDG nodes.	√
Fan-in analysis	Method invocations with a high fan-in.	√
Formal concept[1]	Groups of classes and methods with similar names.	X
Execution trace analysis	Method invocations having a recurring execution pattern. The execution patterns are: • Outside-Before. • Outside-After. • Inside-First. • Inside-Last.	X
Formal concept of execution traces	Classes having methods that contributes to more than one use-case.	X

1 The DelfSTof prototype.

Table 3.10: Comparison of automatic tools.

4 Thesis Contribution

4.1 Limitations of existing tools

From the analysis made in the preceding chapter, we can point out some limitations of the existing aspect mining tools.

4.1.1 False aspect candidates

The automatic tools may produce false aspect candidates (returned aspect candidates that are not crosscutting concerns). The user needs to analyze the results in order to filter out false aspect candidates.

4.1.2 Undetected aspects

The automatic tools usually use a hidden concern characterization. Aspects that do not conform to such hidden characterization are not detected.

4.1.3 Concern characterization constructs in multiple concern operations

The non-automatic tools add a detected characterization construct to all the concern operations whose definition contains such construct without applying any kind of filter.

Let's get back to the Update and the Query operations discussed in section 3.1. These two operations have the following method invocations in common:

```
java.sql.Connection.createStatement
java.sql.Statement.close
```

The PRISM, CME and the FEAT tool add all such method invocations to both operations. They cannot filter which of those method invocations are actually implementing a Query and which ones an Update operation.

This situation creates distracting results to the user, since `createsStatement` invocations implementing a Query, are added to the Update concern operation, and vice-versa.

The user needs to expend time analyzing the source code in order to filter the results.

4.1.4 No detection of concern instances

By concern instance we mean the occurrence of a concern operation. Let's consider the following example:

```
1       Connection conn = ...
2       ...
3       Statement st1 = conn.createStatement();
4       st1.executeUpdate("an sql statement");
5       st1.close();
6       ...
7       Statement st2 = conn.createStatement();
8       st2.executeUpdate("another sql statement");
9       st2.close();
```

In this example we have two occurrences of the Update operation, so we say that we have two concern instances. The first instance is located from lines 3 to 5; the second one is located from lines 7 to 9.

None of the tools is able to identify concern instances. They can detect the method invocations that form part of the update operation characterization, but they cannot detect how many concern instances are present.

Again, the user has to analyze the source code in order to find out concern instances.

4.1.5 No abstraction of detected concerns

The tools don't offer an abstract view of detected concern operations. The results that they produce are just the link to the source code of the detected constructs that form part of the characterization of concern operations.

The user needs to analyze the source code in order to extract operation properties, and operation relationships.

By property we mean information that is of interest for the user. Some examples are:

- For database connections: the database url, the username and the password.
- For caching and web sharing information: the key and the value of the data.
- For assertions: the boolean condition being tested.

In existing tools, the user is presented with the lines of code where the characterization constructs appear in code. Again, the user needs to analyze the results in order to extract information that may be of interest for him/her.

Additionally, we can have relationships between operations. For example, in persistence, the connection precedes queries, updates and closings, so these operations are related by the *precedence* relationship.

If we were to parse a source code using 2 connections and several queries and updates, the user will need to analyze the results and the source code, to figure out to which connection each query, update and closing is related to.

4.1.6 The user still needs to analyze the source code

The fact that the existing aspect mining tools produce false aspect candidates, do not detect concern instances, do not offer an abstract view of detected concerns, and do not filter the detected characterization constructs, forces the user to analyze the source code and to filter the results.

4.2 Thesis objectives

Summing up, the existing approaches for characterization and detection of concerns in source code, fail to produce accurate results and to produce an abstract view of detected concerns.

Users are faced with inaccurate and distracting results. Users need to filter the results and still need to study the code in order to have a good understanding of the concerns they wants to detect. Users need to discover properties and relationships by themselves.

The objective of this thesis dissertation is the proposal of:

1. A concern characterization construct aimed to detect concern instances.
2. A concern characterization model aimed to offer an abstract view of detected concern instances.

4.2.1 A new concern characterization construct: Method Invocation Sequence

In the Object-Oriented programming paradigm [7], programs are essentially defined as a set of objects collaborating with each other by sending messages. According to this definition, the two most important thinks in Object-Oriented programs are objects and messages. Moreover, Object-Oriented programs always encapsulate the logic business in methods.

Based on this reasoning, this thesis dissertation is proposing a new concern characterization construct aimed to improve the method invocation construct. The proposed construct is called *method invocation sequence* and represents a description of how method invocations are linked. A method link refers to the data flow between two method invocations.

Let's consider the following code:

```
Statement st1 = conn.createStatement();
st1.executeUpdate("an sql statement");
```

The return object of the first method invocation is also the invoked object of the second one. The notion of "method link" expresses the fact that the invoked object of the second invocation comes from the return object of the first invocation. It describes the data-flow between two method invocations.

By identifying method invocation sequences, we expect to associate a given method invocation to the right concern operation, solving the problem mentioned in section 4.1.3.

We will validate whether this new concern characterization construct can detect concern operation instances, and by this means, taking away from users the burden of analyzing the source code and filtering the results by themselves.

4.2.2 Abstract view of concern instances

We are also proposing a concern instance representation in order to present a user-friendly view of detected concern instances. Our representation will include operation properties (information that is of interest for the user) and operation relationships (example: precedence).

5 Design considerations

A prototype was implemented as a plug-in for Eclipse in order to validate our proposals. In this chapter we want to describe some design considerations of our prototype

5.1 Characterization of Concerns

The characterization of concerns will be the input of our prototype. In order to detect concern instances and to produce an abstract view of them, a new concern characterization is necessary. The characterization of concerns proposed in this dissertation is shown in figure 5.1.

In this characterization, a concern is defined as a set of operations. An operation may have associated several *concern operation instance* implementations. An implementation has a technology context associated to it, and it is described by concern characterization constructs.

A concern characterization construct represents a description of how an operation concern is implemented. This model includes the existing characterization constructs:

- Text matching
- Type usage
- Source code artifacts
- Source code artifact relationships
- Method invocations

This thesis dissertation is proposing a new characterization construct: the method invocation sequence. This new characterization construct is aimed to detect concern instances and therefore, will be the concern characterization constructor considered in our prototype. We will validate whether a concern instance implementation can be detected by this new characterization construct.

A method invocation sequence is defined as a set of method invocations, which in turn has two attributes. The attribute "obligatory" determines whether the presence of a method invocation in a concern instance is obligatory or optional.

The attribute "principal" is used to indicate that each detected invocation have to be placed in a different concern instance. In chapter 7 "Validation" we will see the utility of this attribute.

Additionally, a method invocation sequence contains a set of *method links*, which describes the data flow within the sequence. In order to define a method link, it is necessary to decompose a method invocation in its constituent parts: the returned value, the invoked object, and its parameters. We will use the term *Method Data* to refer to any of such constituent parts. A method link is defined as a link between two method data of two different method invocations.

Furthermore, an operation may have properties associated to it, and may be a participant in a relationship with another operation, where each operation is playing a specific role. A property value can be bound to a method data, and finally, an operation relationship is defined by a method link between two method invocations of two different operations.

In order to make this representation clearer, we will consider a small and hypothetical example. Let's consider a simplified persistence java code using the JDBC.

```
//Connection to the database
Class.forName("a database driver");
Connection conn = DriverManager.getConnection(...);

//more code

//an update
Statement st2 = conn.createStatement();
st2.executeUpdate("UPDATE ... FORM Person WHERE ...");
st2.close();

//more code
```

Figure 5.2 illustrates a **very simplified** instantiation the corresponding concern characterization. The boxes represent objects instead of classes. In this example, the *SQL Statement* property value is bound to the first parameter of the `executeUpdate()` method invocation. The precedence relationship between the Connection and the Query operation is defined by the method link between the `DriverManager.getConnection()` and the `createStatement()` method invocations.

5.2 Source code representation

We need to build a program model in order to be able to detect our new concern characterization construct in the source code. The source code representation used in this dissertation is shown in figure 5.3.

Our source code representation includes the declaration and the references of class fields and local method variables. This is necessary since we are interested in data flow analysis in order to detect method invocation sequences.

The model also includes the method invocations that are inside of method declarations, and for each invocation we include its corresponding method data.

A method data can be either an atomic data or a complex data. An atomic data is either a field/variable reference, or a literal (string literal, number, liberal, boolean literal, character literal or null literal). A complex data is an expression that depends on more than one atomic data.

5.3 Concern instance

The concern instances will be the output of our prototype. A concern instance representation, aimed to display a user-friendly view of detected concern instances, is shown in figure 5.4.

A concern instance belongs to a concern operation. It has an associated code source that is represented as a method invocation sequence. A concern instance may also have properties associated to it. A property has a name and value associated to it, which in turn is a method data of one method invocation of the method invocation sequence associated to the source code of the concern instance.

Finally, a concern instance may participate in several relationships with other concern instances. A relationship is represented by two concern instances, each one playing a role.

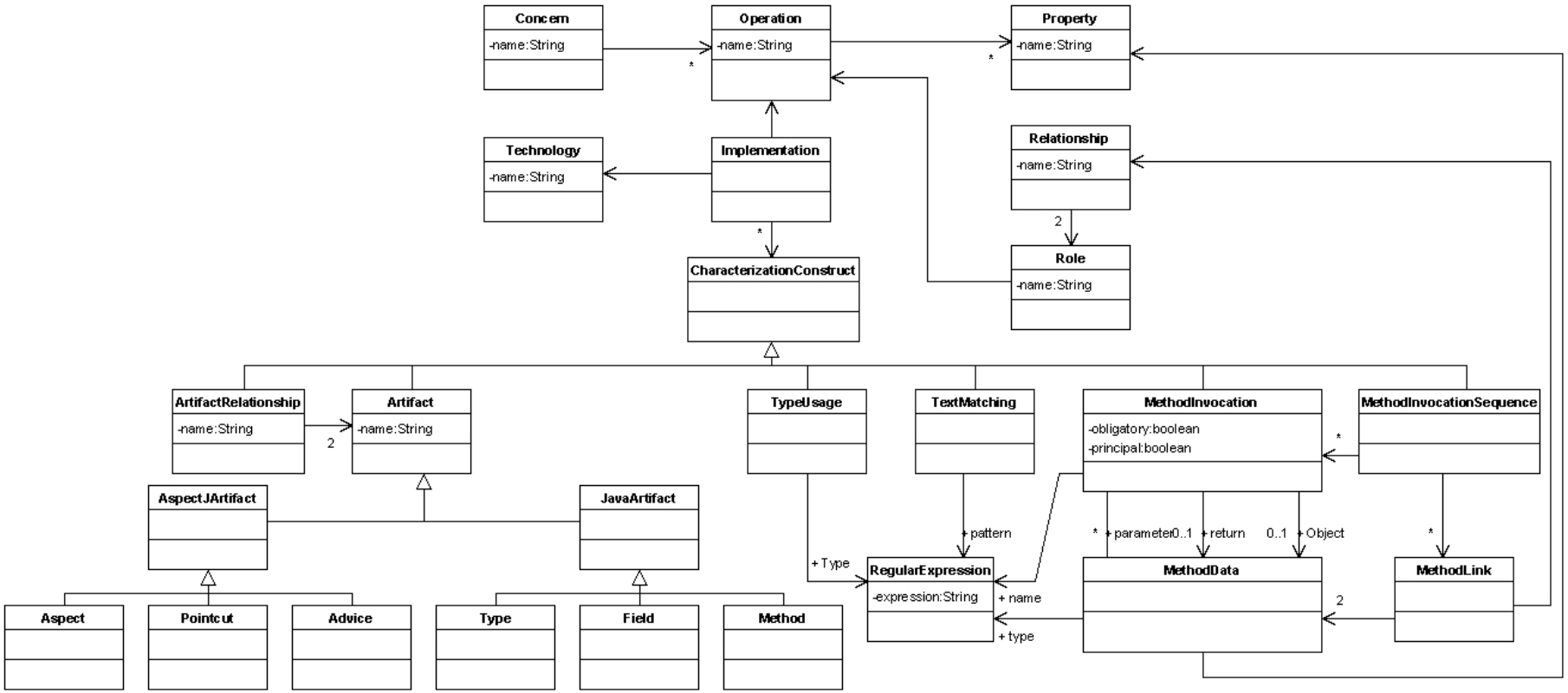

Figure 5.1: Characterization of concerns.

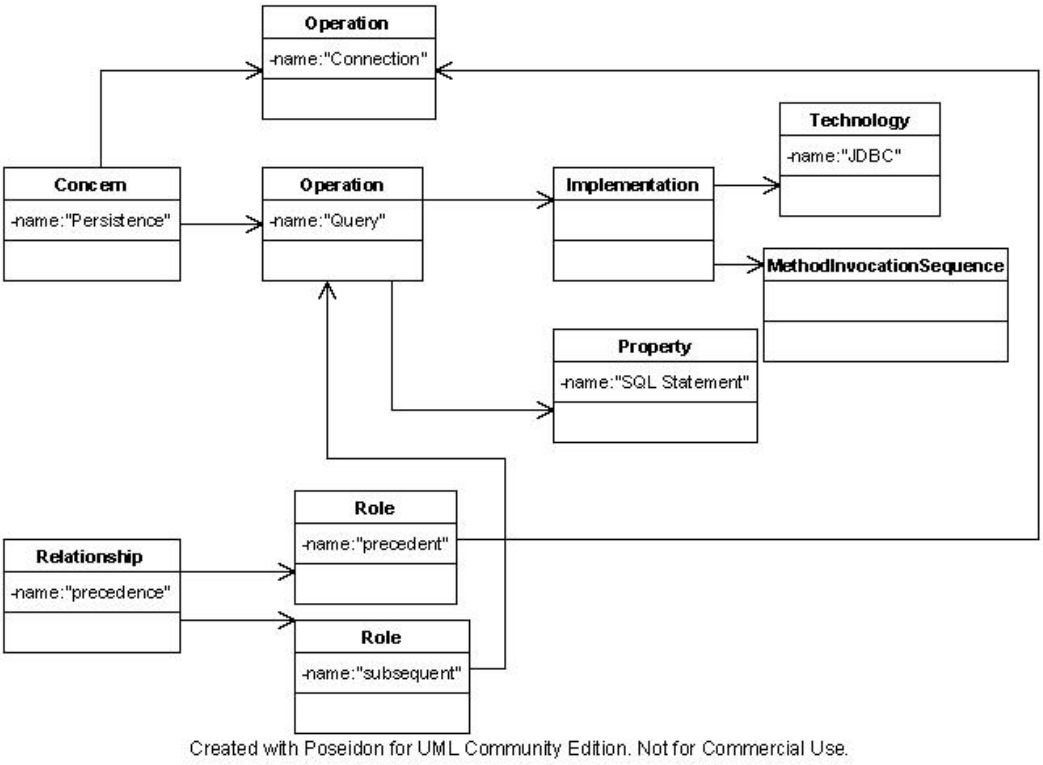

Figure 5.2 : A simplified concern characterization instance.

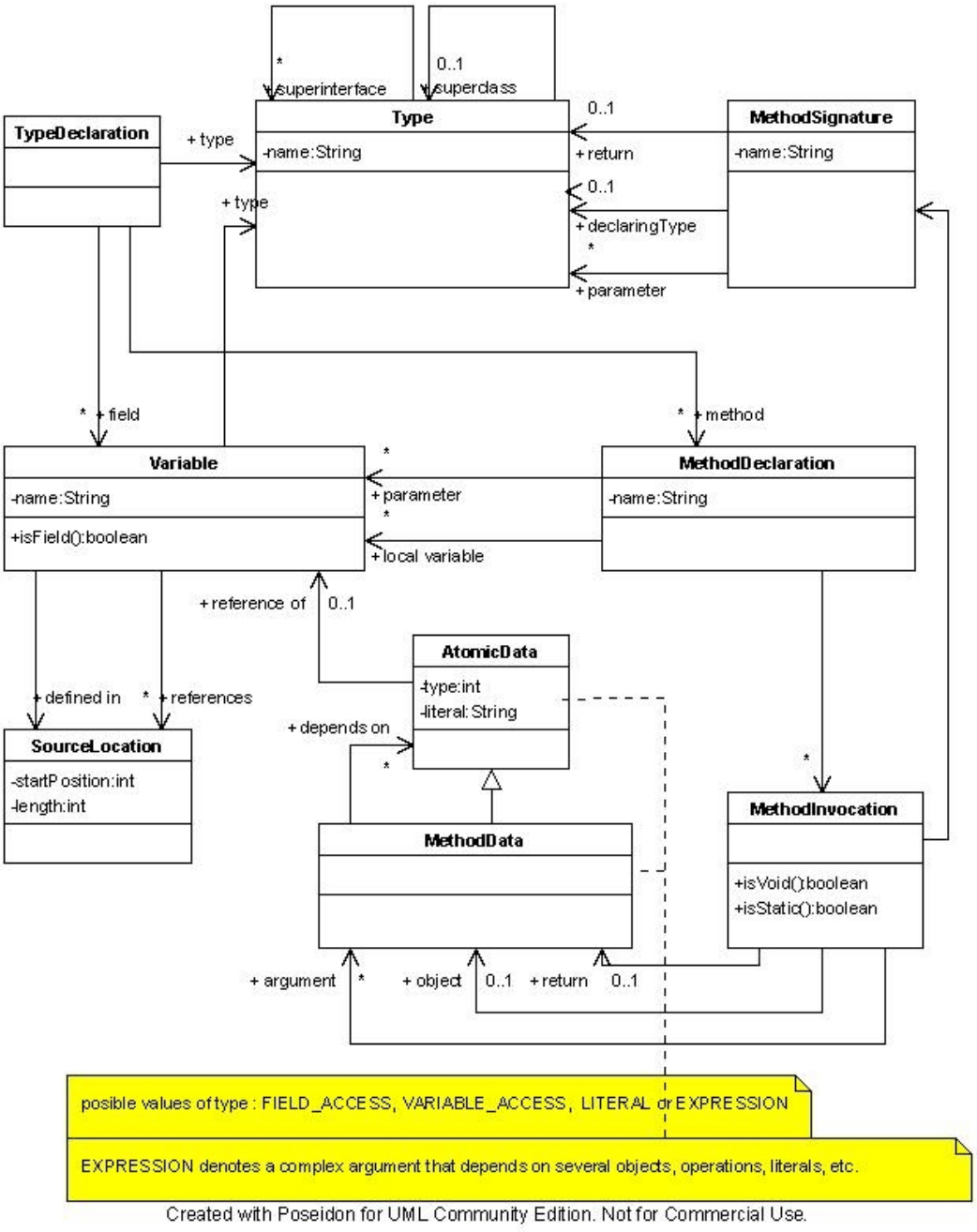

Figure 5.3: The source code representation.

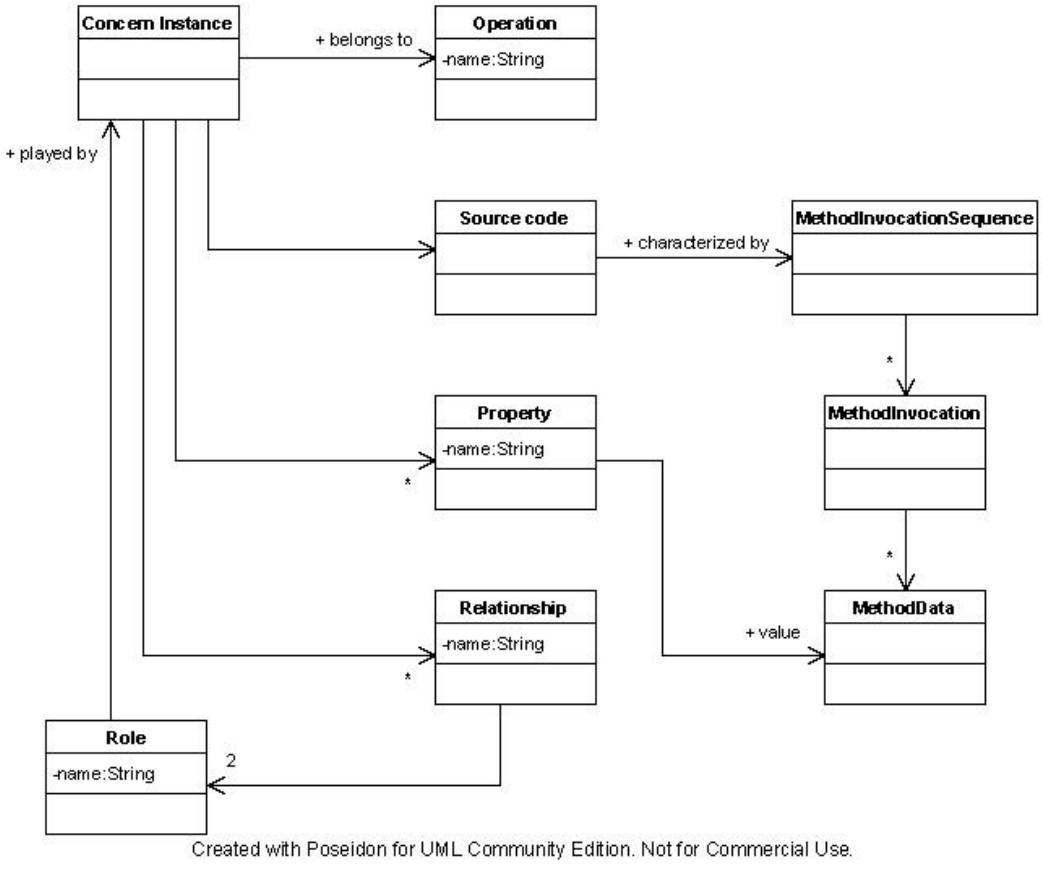

Figure 5.4 : The concern instance representation.

6 Validation

To investigate whether the *method invocation sequence* construct is useful to detect concern instances, and to investigate whether our concern characterization model provides an abstract view of detected instances, we conducted two case studies. The first case study deals with persistence and the second case study with web sharing information.

6.1 First case study: persistence

Persistence, the storage and retrieval of application data from secondary storage media, is often used as a classical example of a crosscutting concern [42]. We will consider several examples of persistence with JDBC [14].

Our persistence concern was defined as:

Concern: Persistence

Operation: Connection

- Properties: database url, username and password.
- Method invocations:

Obligatory	Principal	Method
false	false	`Class.forName`
true	true	`DriverManager.getConnection`

- Property bindings:

Property	Method Data	Method
database url	parameter1	`DriverManager.getConnection`
username	parameter2	`DriverManager.getConnection`
password	parameter3	`DriverManager.getConnection`

Operation: Query

- Properties: SQL Statement
- Method invocations:

Obligatory	Principal	Method
true	false	`Connection.createStatement`
false	false	`ResultSet.*`
true	false	`Statement.close`
true	true	`Statement.executeQuery`

- Method links:

Data	Method	Data	Method
object	`Statement.executeQuery`	return	`Connection.createStatement`
object	`ResultSet.*`	return	`Statement.executeQuery`
object	`Statement.close`	return	`Statement.executeQuery`

- Property bindings:

Property	Data	Method
SQL statement	parameter1	`Statement.executeQuery`

Operation: Update

- Properties: SQL statement
- Method invocations:

Obligatory	Principal	Method
true	false	`Connection.createStatement`
true	false	`Statement.close`
true	true	`Statement.executeUpdate`

- Method links:

Data	Method	Data	Method
object	`Statement.executeUpdate`	return	`Connection.createStatement`
object	`Statement.close`	return	`Statement.executeQuery`

- Property bindings:

Property	Data	Method
SQL statement	parameter1	`Statement.executeUpdate`

Operation: closing

- Method invocations

Obligatory	Principal	Method
true	false	`java.sql.Connection.close`

Operation relationships:

Name: precedence			
Operation1: Connection		Operation2: Query	
Role1: precedent		Role2: subsequent	
Method Link			
Data	Method	Data	Method
return	`DriverManager.getConnection`	object	`Connection.createStatement`

Name: precedence			
Operation1: Connection		Operation2: Update	
Role1: precedent		Role2: subsequent	
Method Link			
Data	Method	Data	Method
return	`DriverManager.getConnection`	object	`Connection.createStatement`

Name: precedence			
Operation1: Connection		Operation2: Closing	
Role1: precedent		Role2: subsequent	
Method Link			
Data	Method	Data	Method
return	`DriverManager.getConnection`	object	`Connection.close`

Example 1

```
//Connection to the database
Class.forName("a database driver");
Connection conn = DriverManager.getConnection("a database url",
                                    "a username", "a password");

//a query
Statement st1 = conn.createStatement();
ResultSet rs = st1.executeQuery("SELECT name, age, weight FROM
                                   PERSON WHERE id = " + id);
rs.next();
this.name = rs.getString("name");    //The name column
this.age = rs.getInt("age");         //The age column
this.weight = rs.getFloat("weight"); //The weight column
rs.close();
st1.close();

//an update
Statement st2 = conn.createStatement();
st2.executeUpdate("an update");
st2.close();

//closing of the connection
conn.close();
```

The methods `Connection.createStatement` and `Statement.close` are shared by both the Query and the Update operations, but our approach successfully filtered such method invocations and added them to the right operation instance.

Our prototype detected the concern instances and also detected their corresponding property values. For each detected property values, it displays the literals, fields and variables it depends on. For instance, for the query operation instance, it informs that the *SQL statement* property value depends on the string literal `"SELECT name, age, weight FROM PERSON WHERE id = "`, and on a variable named `id`.

The prototype also detected operation relationships. For instance, for the query operation instance, it detected the *precedent* relationship with the connection operation. For the *connection* instance, it detected three *precedent* relationships: one with the query, one with the update and one with the close instance.

Figure 6.1 offers a screen shot of the detected query and update instances, and figure 6.2 presents a screen shot of the connection and closing instances.

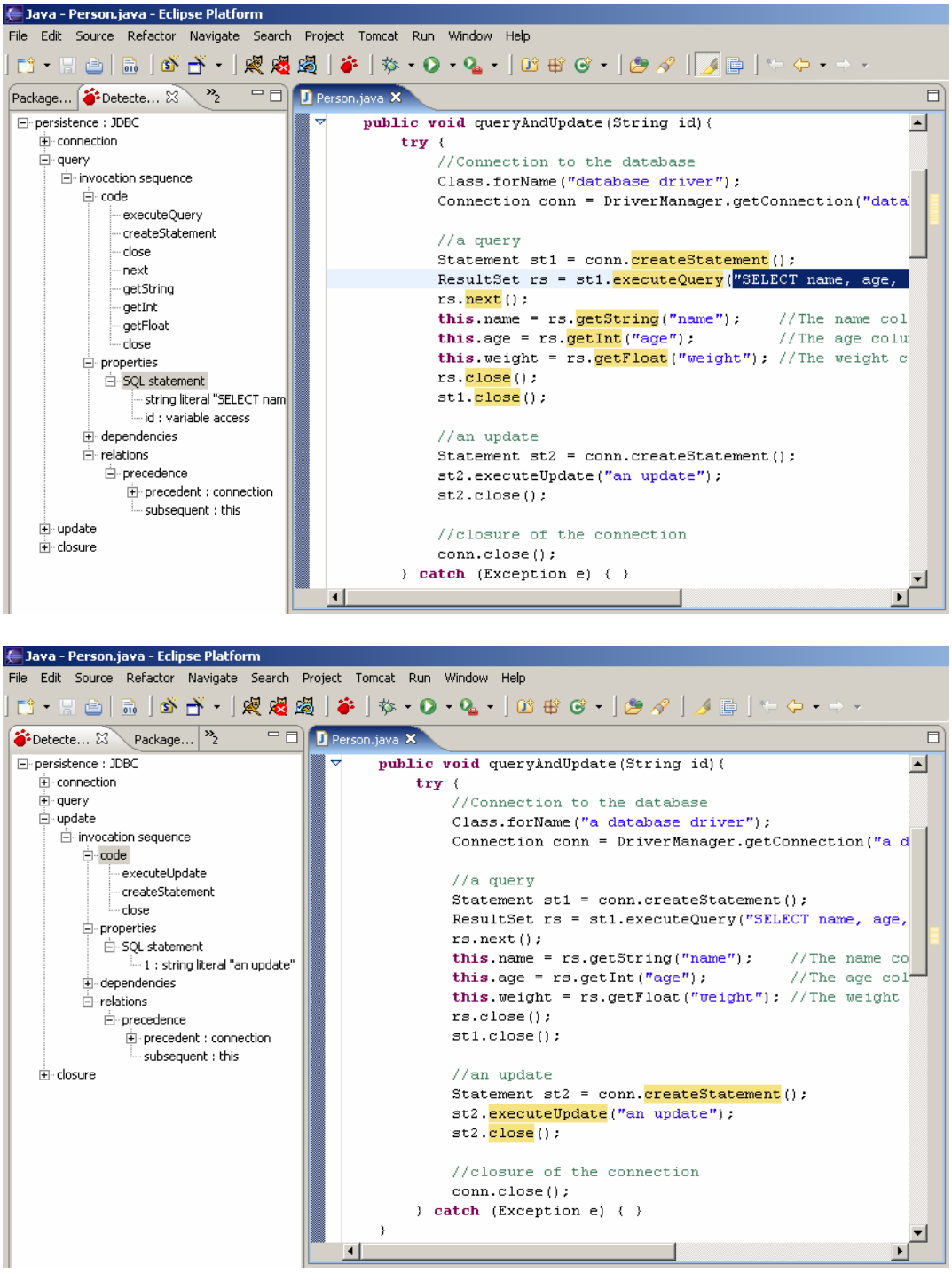

Figure 6.1: Persistence – one query and one update instance.

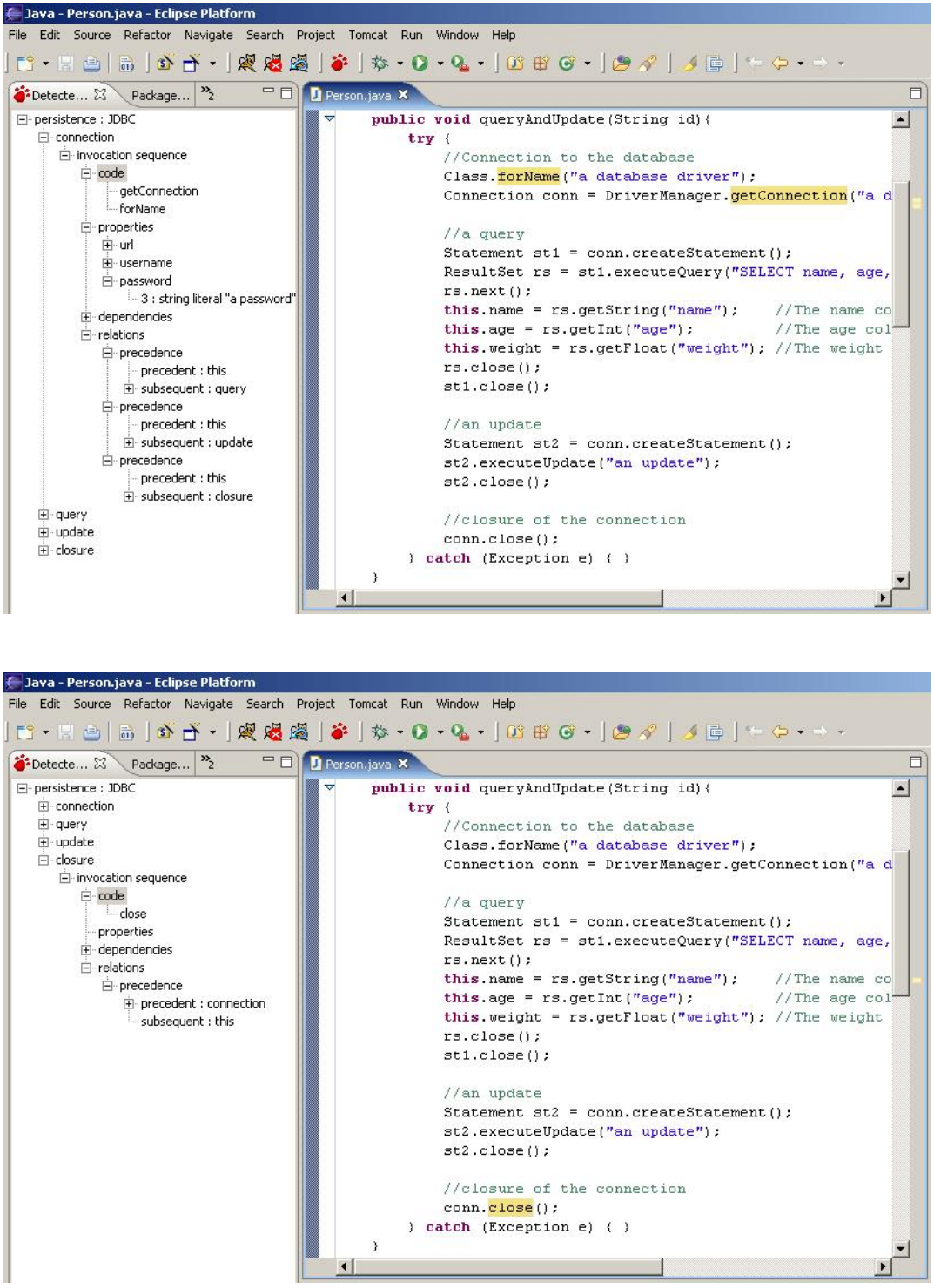

Figure 6.2: Persistence – one connection and one closing instance.

Example 2

```
//two updates
Statement st2 = conn.createStatement();
st2.executeUpdate("an update");
st2.executeUpdate("another update");
st2.close();
```

Our prototype successfully detected two update instances. In our concern characterization, we set the *principal* attribute of the `Statement.executeUpdate` method invocation to be true, so each detected invocation is placed in a separate instance. The methods `Connection.createStatement` and `Statement.close` are shared by both instances. Figure 6.3 offers a screen shot with this example.

Example 3

```
//Connection to the database
Class.forName("database driver");
Connection conn1 = DriverManager.getConnection(...);
Connection conn2 = DriverManager.getConnection(...);

//an update using conn1
Statement st1 = conn1.createStatement();
st1.executeUpdate("an update");
st1.close();

//an update using conn2
Statement st2 = conn2.createStatement();
st2.executeUpdate("an update");
st2.close();

//closing of the connections
conn1.close();
conn2.close();
```

In this example we have two connection, two update and two closing instances. The first query and the first closing instances have a precedent relationship with the first connection instance. Similarly, the second query and the second closing are related with the second connection instance.

Our prototype successfully identified each instance, and identified the relationships between those instances. Figure 6.4 shows that the second update instance is linked to the second connection instance by the precedence relationship.

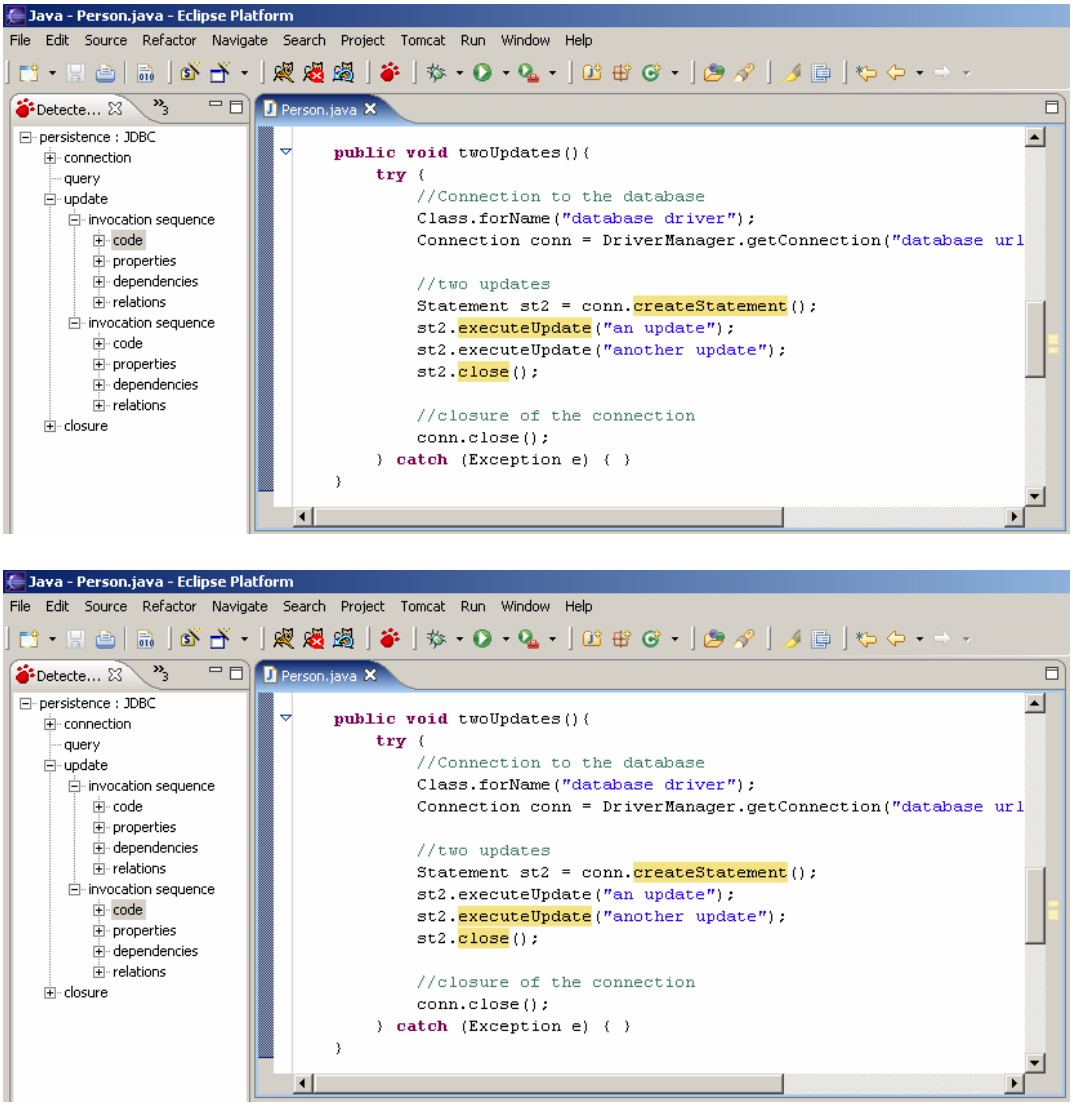

Figure 6.3: Persistence – two update instances.

6.2 Second study case: web sharing information

Web components, like most objects, usually work with other objects to accomplish their tasks [44]. There are several ways they can do this. They can use private helper objects (for example, JavaBeans [45] components), they can share objects that are attributes of a public scope, they can use a database, and they can invoke other web resources.

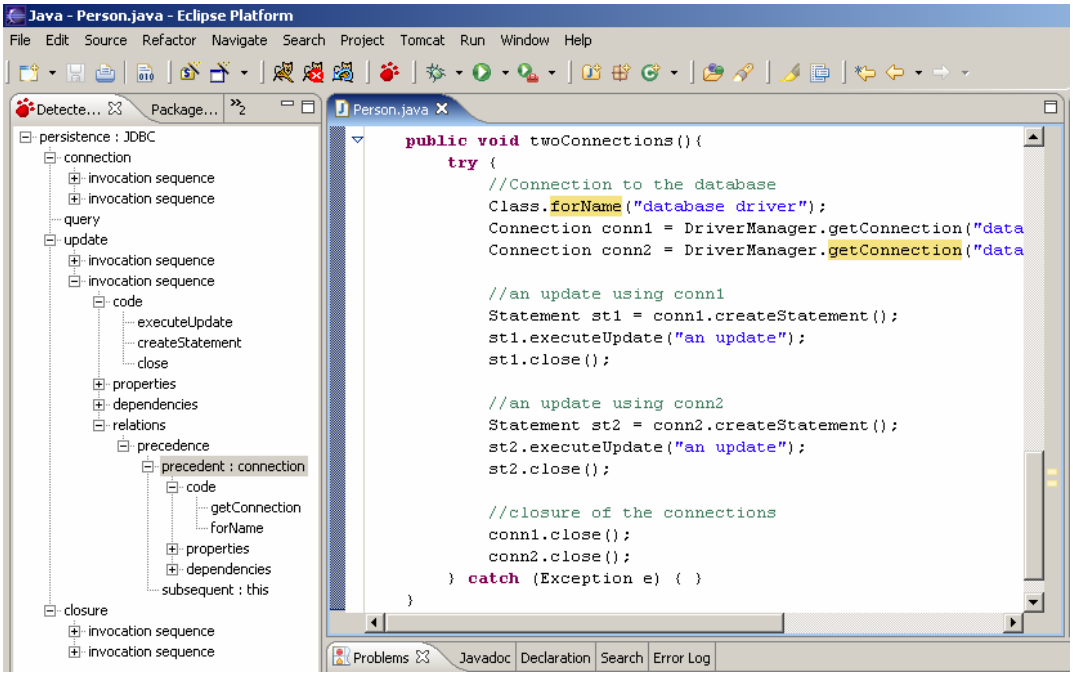

Figure 6.4: Persistence – two connection, two update and two closing instances.

We will consider an example using the Java Servlet [46] technology, in which web components share information via objects that are maintained as attributes of four scope objects: web context, session, request and page.

In our example, we are going to use a session scope object, where shared information is maintained as attributes of the session scope object. Attributes are represented as a key/value pair.

We have identified three operations: Session retrieval, Writing and Reading. The Writing and the Reading operations are linked to the Session retrieval operation through a precedence relationship, and the Writing and the Reading operations are linked through the producer-consumer relationship.

The web sharing information concern was defined as:

Concern: Web sharing information

Operation: Session retreival

- Method Invocations:

Obligatory	Principal	Method
true	true	`HttpServletRequest.getSession`

Operation: Writing

- Properties: key and value.
- Method invocations:

Obligatory	Principal	Method
true	true	`HttpSession. setAttribute`

- Property bindings:

Property	Data	Method
key	parameter1	`HttpSession. setAttribute`
value	parameter2	`HttpSession. setAttribute`

Operation: Reading

- Properties: key.
- Method invocations:

Obligatory	Principal	Method
true	true	`HttpSession. getAttribute`

- Property bindings:

Property	Data	Method
key	parameter1	`HttpSession. getAttribute`

Operation relationships:

Name: precedence			
Operation1: Session retrieval		Operation2: Writing	
Role1: precedent		Role2: subsequent	
Method Link			
Data	Method	Data	Method
return	`HttpServletRequest.getSession`	object	`HttpSession. setAttribute`

Name: precedence			
Operation1: Session retrieval		Operation2: Reading	
Role1: precedent		Role2: subsequent	
Method Link			
Data	Method	Data	Method
return	`HttpServletRequest.getSession`	object	`HttpSession. getAttribute`

Name: consumer-producer			
Operation1: Writing		Operation2: Reading	
Role1: producer		Role2: consumer	
Method Link			
Data	Method	Data	Method
parameter1	`HttpSession. setAttribute`	parameter1	`HttpSession. getAttribute`

Example 4

```
Person carlos = new Person("Carlos"), juan = new Person("Juan");

//Session retrieval
HttpSession session = request.getSession(false);

//writing of attributes
session.setAttribute("boss", carlos);
session.setAttribute("employee", juan);

//some code

//reading of attributes
Person boss = (Person)session.getAttribute("boss");
Person employee = (Person)session.getAttribute("employee");
```

Our prototype successfully found one session, two writing and two reading instances. Moreover, for the session instance it detected four precedence relationships, and for each writing and reading instance, it detected one precedence and one producer-consumer relationship. Figure 6.5 offers a screen shot with this example.

6.3 Limitations

Our prototype has some limitations due to the source code representation used for its implementation. We present some examples that illustrate such limitations.

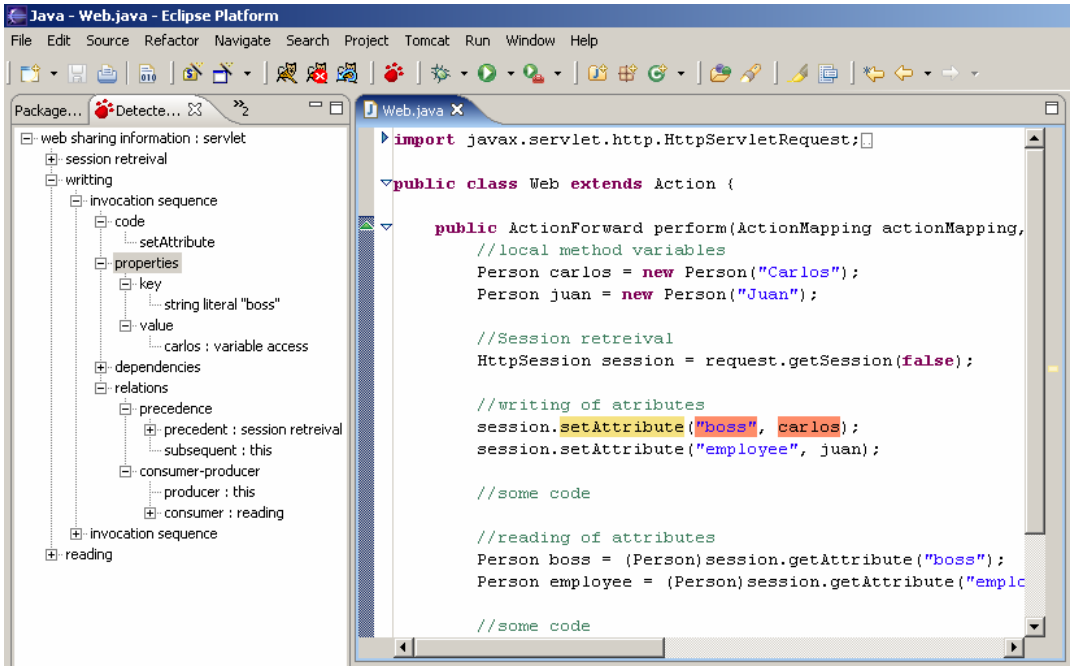

Figure 6.5: Web Sharing Information – one session, two writing and two reading instances.

Example 5

```
//first update
Statement st2 = conn.createStatement();
st2.executeUpdate("an update");
st2.close();

//second update
st2 = conn.createStatement();
st2.executeUpdate("another update");
st2.close();
```

This example shows two update instances. The first one begins with the first assignation of the `st2` variable, and the second instance begins with the second assignation of `st2`.

Our prototype places each `executeUpdate` invocation in a separate concern instance, but it also places the two `createStatement` and the two close invocations to both of them.

This situation could be solved if assignments are added to the source code representation, so the first instance could be found between the first and the

second assignment of the `st2` variable, and the second instance could be found after the second assignment. Figure 6.6 shows a screen shot of this example.

Example 6

```
//two updates
Statement st2 = conn.createStatement();
if(...)
      st2.executeUpdate("an update");
else
      st2.executeUpdate("another update");
st2.close();
```

This code has a conditional statement that will execute only one of two possible database updates. It is important to note that only one update will be executed, so both `executeUpdate` invocations can be added to the same concern instance.

Our source code representation does not include conditional statements, and our prototype places each `executeUpdate` invocation in a separate concern instance. Figure 6.7 shows a screen shot of this example.

Example 7

```
public void connection(){
      Connection conn = DriverManager.getConnection(...);
      update(conn);
      conn.close();
}

public void update(Connection conn){
      Statement st2 = conn.createStatement();
      st2.executeUpdate("an update");
      st2.close();
}
```

In this example, a database update in the `update(Connection)` method is taking place using a database connection created in the `connection()` method.

Our prototype identified the connection the closing and the update instances. It also identified the precedence relationship between the connection and the closing instances, but it was unable to identify the precedence relationship between the update and the connection instance.

In the source code parsing process, our prototype takes each method declaration as a unit of analysis. That means that each method declaration is analyzed separately, and the parsing process will yield the concern instances and the concern relationships found within each method declaration.

Currently, our prototype does not perform analysis control flow, and is unable to detect relationships between concern instances found in different method declarations.

Figure 6.8 shows a screen shot of this example. Both concerns are detected, but no relationship is detected between these two instances.

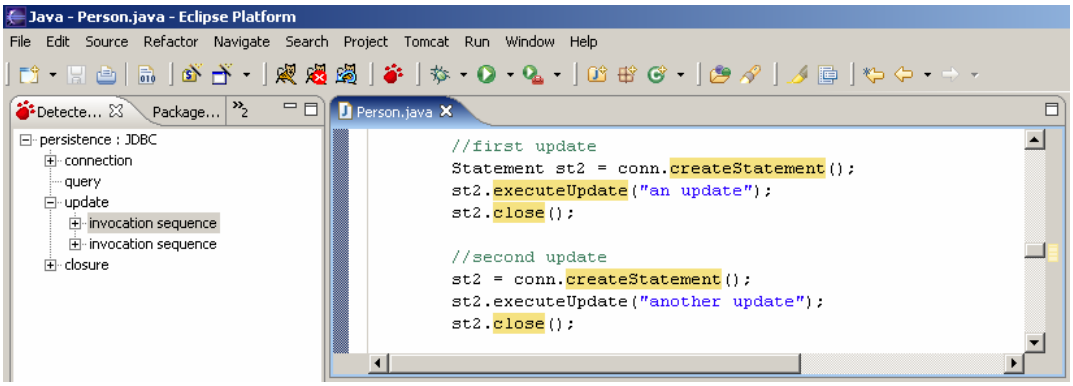

Figure 6.6: The assignment problem.

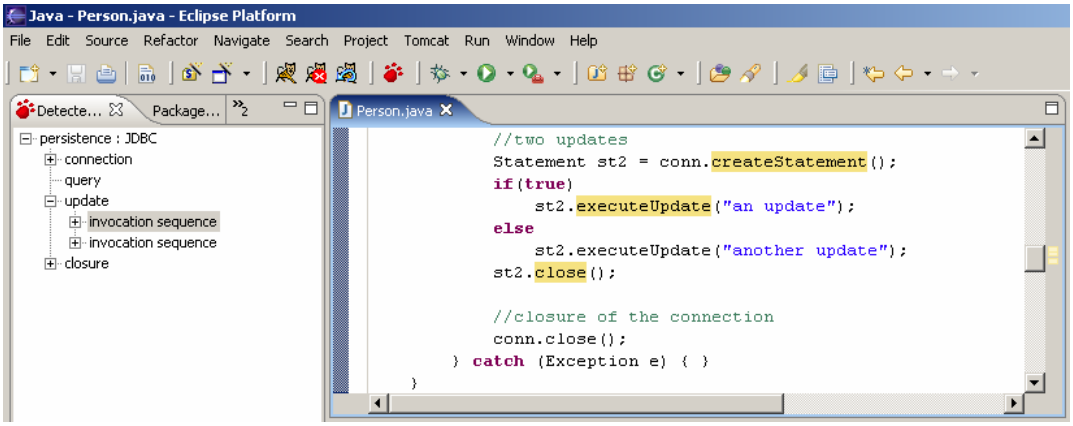

Figure 6.7: The conditional statement problem.

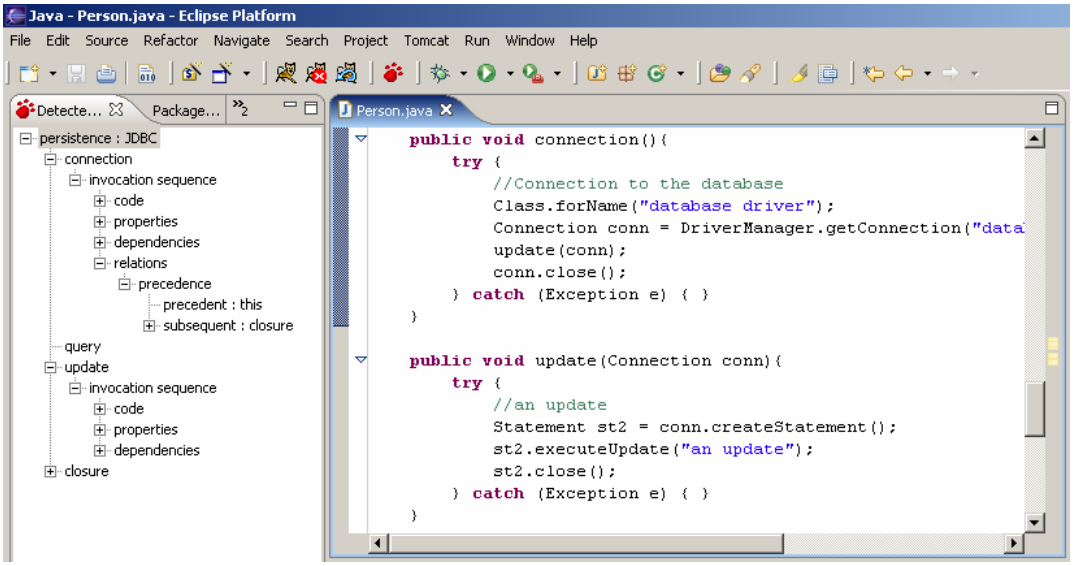

Figure 6.8: The control flow problem.

7 Discussion

In this chapter, we summarize our views on the different techniques for characterization of concerns that we found in the state of the art of existing aspect mining tools.

7.1 Type usage and Method invocations

The type usage was one of the first for concern characterization techniques, and was introduced by AMT [20]. As its name suggest, the type usage technique identifies the usages of instance objects of certain type. This includes declarations, assignations and method invocations on objects.

The method invocation is a straightforward technique to characterize and detect scattered concerns in source code. This technique is incorporated in the majority of the existing aspect mining tools.

Object-Oriented programs [7] are essentially defined as a set of objects collaborating with each other by sending messages. These two techniques are aimed to characterize and detect the essential elements of Object-Oriented programs: objects and messages passing.

7.2 Fields

One possible reason for considering fields as a characterization constructor could be the detection of read and write access of public fields. Nevertheless, it is generally consider a bad programming practice to have public fields in classes.

Notwithstanding, this technique can be simulated with the method invocation technique. It will suffice to add getter and setter methods, and to replace each field read access with a getter invocation, and each field write access with a setter invocation. In this way, the field read access can be simulated by detecting the getter method invocations and the write field access by detecting the setter method invocations.

7.3 Java artifact relationships

A Java artifact is defined as a constituent element in a *.java* file. The Java artifacts considered by FEAT and CME are: type declaration, method declaration and field. These two tools are also able to identify bidirectional relationships between those artifacts.

Some java artifact relationships are shown in table 7.1. The inverse relationships are displayed in parenthesis.

Source Artifact	Relationships	Target artifact
Type	created by (creates)	Method
	declares (declared by)	
	declares (declared by)	Field
Method	reads (read by)	Field
	writes (written by)	
Method	calls (called by)	Method

Table 7.1 : Java artifact relationships.

We have already talked about field read and writes access. Method calls are method invocations, and a type creation is the invocation of its constructor.

In this thesis dissertation we consider that the artifact relationships offers different levels of source code granularity for detected artifacts.

Let's consider the following code:

```
public class Person {

  public void queryAndUpdate(){
    /*Connection to the database*/
    Class.forName(...);
    Connection conn = DriverManager.getConnection(...);
    /*more code*/
  }
}
```

And let's consider the `DriverManager.getConnection()` method as an artifact. With the FEAT tool, we add the *called by* relationship, and we get the lines where such method is invoked. Additionally, since the source and the target of the *called by* relationship are methods, you will also get the method declarations who are invoking the `getConnection()` method. Furthermore, with the declared

by relationship between methods and types, you will get the type declarations who have method declarations that are invoking the `getConnection()` method.

Figure 7.1 shows a screen shot of this example, in which the line of code of the `getConnection()` invocation, the enclosing `queryAndUpdate()` method declaration, and the enclosing `Person` type declaration are detected.

In brief, for each detected artifact, you can have three levels of source code granularity:
1. The lines of code where the artifact is found.
2. The enclosing method declarations of those lines of code.
3. The enclosing type definitions of those method declarations.

7.4 Method declarations and Aspect advices

AspectJ [47] is an aspect-oriented extension to the *Java* programming language that enables the clean modularization of crosscutting concerns. In aspectJ, a well modularized crosscutting concern is implemented in a unit called *aspect*.

An *aspect* is a unit composed of *pointcuts*, *advice* and ordinary Java member declarations. An *advice* is a method-like constructs used to define additional behavior when its associated *pointcut* is reached during the program execution. *Pointcuts* are a means of referring to collections of join points and certain values at those join points; *Join points* are well-defined points in the execution of the program (i.e.: method call, method execution, constructor call, constructor execution, field get, field set, object pre-initialization, object initialization, class initialization, exception handler execution and advice execution) [48].

Aspects are already modularized concerns, and a method declaration as a concern characterization construct, can also be seen as a modularized concern. The execution of a concern modularized in a method can be found by detecting the invocations of such method; the execution of a concern modularized in an aspect advice takes places when the program execution reaches the joint points defined in the corresponding aspect pointcut.

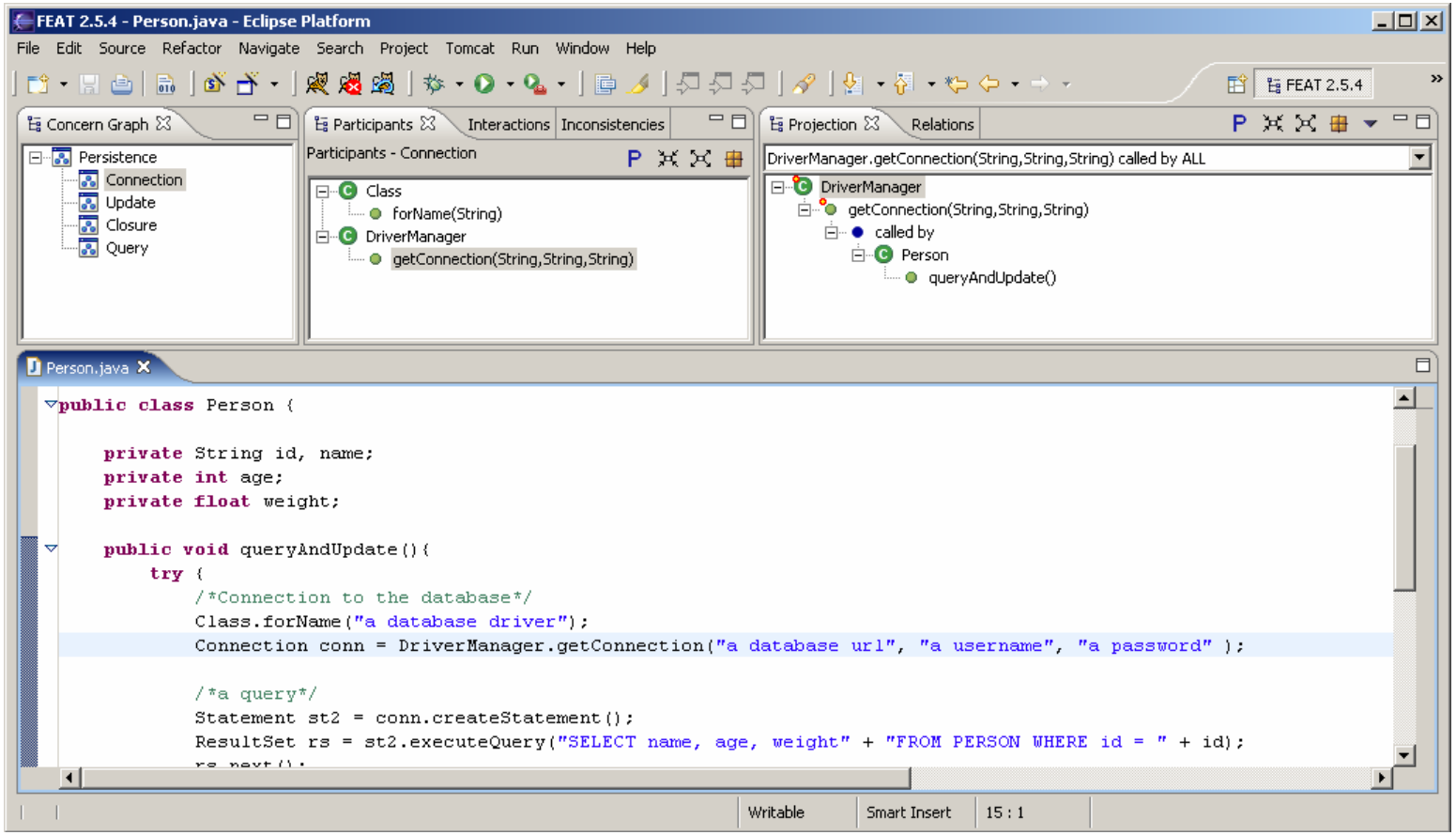

Figure 7.1: Levels of source code granularity obtained through artifact relationships with FEAT.

8 Future Work

By asking users to provide a concern characterization and by having a source code representation, we want to provide the basis for an automatic software refactoring and migration tool.

In order to support source code manipulation, we used the eclipse JDT plug-in for the implementation of our prototype. The JDT plug-in [54] provides the infrastructure for compiling and manipulating java source code.

The JDT plug-in is provided with a parser that allows users to obtain a source code representation called *abstract syntax tree* (AST). An AST node represents a java source code construct such as a name, type, expression, statement or declaration.

The source code representation used by our prototype is a subset of the abstract syntax tree that only considers the class declaration, field declaration, variable declaration, method declaration, simple name, and method invocation AST nodes.

In addition, the detected concern instances, which represent the detected method invocation sequences, keep the references to the corresponding method invocation AST nodes.

Some of the flaws detected in our tool during its evaluation (section 6.3) can be solved be including the assignment AST node and flow control related AST node statements (If-else, switch) in our source code representation.

By having the JDT plug-in facility for manipulating java source code, we propose an upgrade of our prototype in order to have two kind of automatic software manipulation tools.

The first proposed tool is an automatic refactoring tool to migrate detected concerns instances from OOP to AOP. Some interesting techniques can be found in [50, 51, 52, 53].

Nevertheless, there are cases where detected concerns related code is difficult, if not impossible, to be extracted to AOP code. This is especially true when the related code makes references to local and temporary variables [50, 51, 52].

To tackle this case, we are also proposing an automatic migration tool aimed to change the implementation of detected concerns without extracting them to AOP code. This can be accomplished with the JDT plug-in, which offers a mature API for creation and manipulation of AST source code representations.

9 Conclusions

Several aspect mining tools have been proposed in order to identify scattered code throughout a software system. These tools can be categorized as being automatic or non-automatic.

Non-automatic tools are either query–based or exploratory [49]. Query–based tools require a seed by the user and they may search for text patterns, type usage, method invocations, *Java* artifacts and artifact relationships. Exploratory tools allow users to navigate quickly and intelligently around the code, in order to lead the user to the discovery of scattered concerns in source code.

These tools require a seed from users. Query–based tools require the formulation of a query that will return meaningful results, and exploratory tools need a starting point to allow users the source code navigation.

The formulation of a seed is a non–trivial task that requires the user to have a good understanding of the source code. Users need to have an idea of how concerns are implemented in order to start the search query or navigation of the source code.

On the other hand, automatic tools do not require any seed. However, they use a hidden concern characterization construct.

Several constructs have been also proposed to characterized concerns. These constructs include text patterns, type usage, method invocations, *Java* artifacts and artifact relationships.

Despite the fact that several aspect mining tools and techniques have been proposed, they still fail to produce accurate results.

This thesis dissertation has proposed a new concern characterization construct called *method invocation sequence*, which is an extension of the method invocation construct that includes information about method linkage (data-flow between method invocations). This construct effectively detects concern instances (concern operation occurrences).

Moreover, a new concern characterization has been proposed in order to provide an abstract view of detected concern instances, which includes operation properties and operation relationships.

To validate our proposals, we implemented a prototype, and although some limitations were discovered during the validation, they can be solved by improving the source code representation used by the prototype.

Bibliography

[1] P. Tarr, H. Ossher, W. Harrison and S. M. Sutton. *N Degrees of Separation: Multi-Dimensional Separation of Concerns*. Proceedings, International Conference on Software Engineering, 1999, pp. 107-119.

[2] David L. Parnas. *On the criteria to be used in decomposing systems into modules*. Communications of the ACM, 15(12) : 1053–1058, December 1972.

[3] Martin P. Robillard. *Representing Concerns in Source Code*. Ph.D. Thesis. Department of Computer Science, University of British Columbia. November 2003.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison – Wesley Longman, Inc., Reading, MA, USA, 1995.

[5] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean – Marc Loingtier, and John Irwin. *Aspect-oriented programming*. In Proceedings of the 11th European Conference on Object-oriented Programming, volume 1241 of Lecture Notes in Computer Science, pages 220–242. Springer-Verlag, Heidelberg, Germany, June 1997.

[6] http://www.parc.com/research/csl/projects/aspectj/default.html

[7] T. Korson and J. McGregor, *Understanding Object-Oriented: A Unifying Paradigm*, Communications of the ACM, Vol. 33, No. 9, 1990.

[8] S. Breu and J. Krinke, *Aspect mining using event traces*, In IEEE International Conference on Automated Software Engineering, IEEE Press, 2004

[9] R. K. Fjeldstad and W. T. Hamlen. *Application Program Maintenance Study: Report to Our Respondents*. Proceedings GUIDE 48, Philadelphia, PA, April 1983.

[10] Swanson, E.B. *The Dimension of Maintenance*. Proc. 2nd Int. Conf. on Software Eng., Oct. 1976, pp. 492~197.

[11] V. Rajlich. *Comprehension and Evolution of Legacy Software*. 19th International Conference on Software Engineering. May 17 – 23, 1997. Boston, Massachusetts. P. 669.

[12] H.A. Müller. *Reverse Engineering Strategies for Software Migration*. Sixth European Software Engineering Conference and ACM Foundations of Software Engineering *(ESEC/FSE-97)*, Zürich, Switzerland, September 22-25, 1997.

[13] A. Taivalsaari, R. Trauter, E. Casais. *Workshop on object-oriented legacy systems and software evolution*. OOPS Messenger 6(4): 180-185 (1995).

[14] http://java.sun.com/docs/books/tutorial/jdbc/

[15] http://java.sun.com/

[16] http://www.hibernate.org/

[17] Griswold, W. G., Kato, Y. and Yuan, J. J. *Aspect Browser: Tool Support for Managing Dispersed Aspects*. Position paper for the First Workshop on Multi-Dimensional Separation of Concerns in Object-oriented Systems (OOPSLA '99).

[18] http://www-cse.ucsd.edu/users/wgg/Software/AB/

[19] J. Hannemann and G. Kiczales. *Overcoming the Prevalent Decomposition of Legacy Code*. Workshop on Advanced Separation of Concerns at the International Conference on Software Engineering (ICSE). Toronto, 2001.

[20] http://www.cs.ubc.ca/~jan/amt/

[21] http://www.eecg.utoronto.ca/~czhang/amtex/

[22] http://www.eclipse.org

[23] C. Zhang and H.-A. Jacobsen. *A Prism for Research in Software Modularization Through Aspect Mining*. Technical Communication, Middleware Systems Research Group, University of Toronto, September 2003.

[24] http://www.eecg.toronto.edu/~czhang/prism/

[25] Doug Janzen and Kris De Volder. *Navigating and querying code without getting lost*. Proceedings of the Conference on Aspect-Oriented Software Development. ACM Press, New York, NY, USA, March 2003.

[26] http://jquery.cs.ubc.ca/

[27] Rajeswari Rajagopalan and Kris De Volder, *QJBrowser: A Query-Based Browser Model*, submitted to ICSE.

[28] http://tyruba.sourceforge.net/

[29] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard*. Springer-Verlag, New York, 1996.

[30] Martin P. Robillard and Gail C. Murphy. *Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies*. In Proceedings of the 24th international conference on Software engineering (ICSE), pages 406–416. ACM Press, 2002.

[31] http://www.cs.ubc.ca/labs/spl/projects/feat/

[32] https://dynaop.dev.java.net/nonav/release/1.0-beta/manual/index.html

[33] http://www.eclipse.org/aspectj/

[34] http://www.english.uga.edu/humcomp/perl/regex2a.html

[35] http://www.regular-expressions.info/

[36] http://www.eclipse.org/cme

[37] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W.G. Griswold. *An Overview of AspectJ*. In Proc. 15th European Conference on Object-Oriented Programming, June 2001.

[38] W. Harrison, H. Ossher, S.M. Sutton, Jr. and P. Tarr. *Concern Modeling in the Concern Manipulation Environment*. IBM Research Report RC23344, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, September 2004.

[39] W. Harrison, H. Ossher and P. Tarr. *Concepts for Describing Composition of Software Artifacts*. IBM Research Report RC23345, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, September 2004.

[40] P. Tarr, W. Harrison, and H. Ossher, *Pervasive Query Support in the Concern Manipulation Environment*. IBM Research Report RC23343, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, September 2004.

[41] Martin P. Robillard and Gail C. Murphy. *Evolving Descriptions of Scattered Concerns*. Technical Report SOCS-TR-2005.1, McGill University, Canada, January 2005.

[42] http://jakarta.apache.org/tomcat/

[43] A. Rashid and R. Chitchyan, *Persistence as an Aspect*. 2nd International Conference on Aspect-Oriented Software Development. ACM, 2003.

[44] http://java.sun.com/j2ee/1.4/docs/tutorial/doc/Servlets5.html

[45] http://java.sun.com/products/javabeans/

[46] http://java.sun.com/products/servlet/

[47] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. *An overview of AspectJ*. In ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming, Springer-Verlag, 2001.

[48] R. Laddad, *AspectJ in Action: Practical Aspect-Oriented Programming*, Manning Publications Co.

[49] D. Shepherd, L. Pollock, and E. Gibson, *Design and Evaluation of an Automated Aspect Mining Tool*, *International* Conference on Software Engineering Research and Practice, June 2004.

[50] S. Hanenberg, C. Oberschulte, and R. Unland. *Refactoring of aspect-oriented software*. In Proceedings of the 4th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays), September 2003.

[51] D. Binkley, M. Ceccato, M. Harman, P. Tonella. *Automated Pointcut Extraction* Proceedings of the workshop on Linking Aspect Technology and Evolution workshop, Chicago, Illinois 23 March 2005.

[52] D. Binkley, M. Ceccato, M. Harman, F. Ricca and P. Tonella, *Automated Refactoring of Object Oriented Code into Aspects*, Proc. of ICSM2005, International Conference on Software Maintenance , Budapest, Hungary, 2005.

[53] M. Fowler. *Refactoring: Improving the design of existing code*. Addison-Wesley Publishing Company, Reading, MA, 1999.

[54] http://www.eclipse.org/jdt

[55] J.H. Johnson. *Identifying redundancy in source code using fingerprints*. In Proceedings of the IBM Centre for Advanced Studies Conference, 1993.

[56] S. Ducasse, M. Rieger, and S. Demeyer. *A language independent approach for detecting duplicated code*. In Proceedings of the International Conference on Software Maintenance (ICSM'99), September 1999.

[57] T. Kamiya, S. Kusumoto, and K. Inoue. *CCFinder: A multilinguistic token-based code clone detection system for large scale source code*. IEEE Transactions on Software Engineering, July 2002.

[58] B.S. Baker. *On finding duplication and near-duplication in large software systems*. In Second Working Conference on Reverse Engineering (WCRE'95), Los Alamitos, California, IEEE Computer Society Press, 1995.

[59] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. *Clone detection using abstract syntax trees*. In Proceedings of the International Conference on Software Maintenance (ICSM'98), IEEE Computer Society Press, 1998.

[60] R. Komondoor and S. Horwitz. *Using slicing to identify duplication in source code*. In Proceedings of the 8th International Symposium on Static Analysis, Springer-Verlag, 2001.

[61] J. Krinke. *Identifying similar code with program dependence graphs*. In Proceedings of the Eight Working Conference On Reverse Engineering (WCRE'01), IEEE Computer Society Press, October 2001.

[62] M. Marin, A. Deursen, and L. Moonen. *Identifying aspects using fan-in analysis*. In Proc. of the 11th IEEEWorking Conference on Reverse Engineering (WCRE 2004), Delft, IEEE Computer Society, November 2004.

[63] K. Mens and T. Tourwe. *Delving source-code with formal concept analysis*. Elsevier Journal on Computer Languages, Systems & Structures, 2005. To be published.

[64] T. Tourwe and K. Mens. *Mining aspectual views using formal concept analysis*. In Proc. of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2004). IEEE Computer Society, September 2004.

[65] P. Tonella and M. Ceccato. *Aspect mining through the formal concept analysis of execution traces*. In Proc. of the 11th IEEE Working Conference on Reverse Engineering (WCRE 2004), IEEE Computer Society, November 2004.

Appendix A

Glossary of terms

Term	Definition
Artifact relationship	A relationship between two artifacts. One plays the role of source, and the other plays the role of the target. They are usually bidirectional. Some examples: declares, calls, called by, reads, written by.
Aspect	A technical consideration a developer might have about the implementation of a system.
AspectJ artifact	A constituent element of an *.aj* file: an aspect, an advice or a pointcut declaration.
Characterization construct	An artifact used to define a concern operation. It can be a method invocation, a type usage, a text pattern, a Java element (type, field, method) or a relationship between elements (calls, called by, reads, writes …).
Concern	Same as an aspect.
Concern characterization	The definition or description of a concern, usually in terms of type usages and/or method invocations.
Concern collision	There is a collision between two concerns if they share at least one characterization construct in their characterization.
Concern instance	A portion of code containing the implementation of one concern operation occurrence.
Concern operation instance	Same as a concern instance.
Concern property	Information related to an operation that is of interest.
Declaring type	The declaring type of a method is the class where it is defined.
Java artifact	A constituent element of a *.java* file: type declaration, field, method declaration.

Method data	The data involved in a method invocation: the receiver, the invoked object and the arguments.
Method declaration	The implementation of a method.

Method invocation linkage	Two method invocations are linked if they share at least one data.
	For example, in this code
	`b = a.method1();`
	`c = b.method2();`
	The invoked object of `method2` is linked to the return value of `method1`.

Qualified name	The complete name for types and methods. The qualified name of types contains the name of its package. The qualified name of methods contains the qualified name of its declaring type, the name of the method, and the qualified name of each parameter.
Regular expression	A string describing the name of a type or a method. It may contain wildcards as "*" (any string), "?" (any character) or "\" (for escape characters like \n, \t, \r, \\, \s, etc).
Super-type	A type *T1* is super-type of type *T2* if one of the following conditions is fulfill: • *T1* is the super-class of *T2*. • *T1* belongs to the set of implemented interfaces of *T2*. • *T1* is a super-type of the super-class of *T2* (should it has one). • *T1* is a super-type of any of the implementing interfaces of *T2* (should it has any).
Super-type matching	The process to determine whether a regular expression describes the qualified name of a type or any of its super-types.
Type	A class, an interface or a primitive type.
Type declaration	The code implementing a type.
Type usage	References of objects of certain type.