Diploma Thesis

May 12, 2006

Detecting Design Violations and Code Smells by Bug-Impact Analysis

Dominik Schaffhauser

of Basel, Switzerland (00-907-188)

supervised by

Harald Gall

Martin Pinzger, Beat Fluri

University of Zurich

Department of Informatics

s.e.a.l.

software evolution & architecture lab

Diploma Thesis

Detecting Design Violations and Code Smells by Bug-Impact Analysis

Dominik Schaffhauser

University of Zurich
Department of Informatics

s.e.a.l.
software evolution & architecture lab

Diploma Thesis

Author: Dominik Schaffhauser, dominisc@access.unizh.ch

Project period: November 14, 2005 - May 14, 2006

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

First of all I would like to thank my girlfriend Michelle Ackermann for her great support and patience during my studies and for proof-reading this thesis.

My parents, Ursula and Anton Schaffhauser for providing the financial support that allowed me to rent an appartment in Zurich. They were always there when I asked for something, for which I am very greatful.

Beat Fluri for his support of this thesis and his work on file comparison to analyze fine grained changes between two revisions of a class.

Special thanks go to Michael Wuersch for his time and support with Hibernate and his work with the Evolizer Base plugin.

Last but not least I would like to thank Martin Pinzger for the idea of this diploma thesis and his great support during that time.

Abstract

By analyzing software architectures it is possible to detect risky weaknesses. There are many different ways to analyze software. One such method is to combine the collected data of a Bugzilla repository with the history information from the changed files of a CVS repository. This allows to detect bugs which affect unexpected source code entities. Furthermore it is possible to find bugs that lead to new bugs after being fixed. Integrated into an Eclipse plugin, the Class Evolution plugin provides the developers and designers with specific information on code that is good, or on code that needs a major refactoring.

The goal of this thesis was to develop an Eclipse plugin capable of telling the software engineer about weaknesses in classes and methods.

Zusammenfassung

Wenn Software Architekturen analysiert werden, koennen oft kritische Schwaechen gefunden wurden. Es gibt viele verschiedene Methoden, um Software zu analysieren. Eine Moeglichkeit ist das Kombinieren von gesammelten Daten von einer Bugzilla Repository mit den wegen Bugs geaenderten Daten aus dem CVS Repository. So ist es moeglich, Fehler zu entdecken, die zum Beispiel die Aenderung vieler Quell-Dateien zur Folge haben. Weiter koennen Fehler gefunden werden, die nie richtig behoben werden und immer wieder zu neuen Fehlern fuehren. Eingebunden als Eclipse-Plugin hilft es den Entwicklern und Designern mit Informationen zu gutem oder schlechtem Code, bei dem eine Umstrukturierung noetig waere.

Das Ziel dieser Diplomarbei war es, ein Eclipse-Plugin zu schreiben, das einem Software Entwickler die Moeglichkeit gibt, ein Software Projekt zu analysieren und dabei Schwachstellen in Klassen und Methoden zu entdecken.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Software systems are in a constant state of change. If a software system does not change, it dies. With the changes made to a software system it evolves. It gets new functions to satisfy clients using the system. Not all changes are good. Shortly after changes problems, so called bugs, can appear. A change almost always make the source code worse. It begins to *smell bad*. Especially if there are many people working on a system, the code rots faster. But there are places where the code is fine. Those places were not changed.

In this thesis we want to implement an application which helps a developer or a team of developers to detect critical parts of the system which are under constant change and where problems arises. We want to develop a plugin for an *integrated development environment* (IDE) that helps a developer to detect problematic areas in the system. The risky parts in the software system can be detected with a combination of data from a bug tracking system and data from a version history. The data from the bug tracking system helps us to identify the parts of the system that are problematic. With this data alone we know which areas of the system we need to change, but we do not know which sources will be affected and if the problem affects other unexpected sources too. For this reason we also want to investigate the data from the past, from the version history. After a bug was detected and fixed, there must be something mentioned in the version history. If we can link the bug data with the data from the version history, we get a strong source of information.

To use the combined data we apply generally accepted heuristics on the parts that involve a bug. Some bugs are easy to fix. One location needs a change and the problem is gone. This is fine. But there are bugs which involve several classes. Several of these classes might not have a conceptual relationship with the problem. This is bad. By applying those heuristics we want to know if risky parts can be detected.

Michael Wuersch and Andreas Jetter layed the core for the development of the plugin to analyze bugs and relate them to a version control. In their work on the Evolizer plugin they created mapping functionality for the *Concurrent Versions System* (CVS) repository.

Dane Marjanovic implemented a parser that maps the bug data from the Bugzilla repository into a MySQL database. The parser parses the XML files generated from Bugzilla.

Beat Fluri works on fine-grained change analysis. CVS only tracks changes on a text basis. Detailed information about changes in a class, method or field is not stored. Because of this lack of information he developed an Eclipse plugin that uses the structure comparison function provided by Eclipse to retrieve fine-grained change information.

In this thesis we investigate which of the previous work fits into the new plugin, search for heuristics that claim to ensure software quality and implement those heuristics in a plugin for an IDE. Finally, we make an evaluation with a medium-sized Java project to evaluate our results.

1.1 Overview

The thesis is structured as follows. Chapter 2 gives a brief overview on other work done in this area. Chapter 3 provides the reader with helpful background information on object oriented design principles. In chapter 4, the heuristics presented in chapter 3 are examined and implementation possibilities are discussed. Chapter 5 explains detail information about the implementation of the plugin. The results from the evaluation are presented in chapter 6. Chapter 7 concludes this thesis.

Chapter 2

Related Work

This section covers the fields that are related to the topic of this thesis. The three main fields are information visualization, fine-grained analysis of change couplings and the combination of a bug tracking system with a version control.

2.1 Information Visualization

This section covers other implementations of prototypes that visualize software evolution information. An existing prototype closest to the idea of this thesis is HATARI [SZZ05].

2.1.1 HATARI: Raising Risk Awareness

The HATARI[1] prototype computes the individual risk for all code locations by examining if already earlier changes have caused a risk. It relates a version history to a bug database to detect source entities that have been risky to changes in the past.

The goal of the HATARI prototype is to examine all code locations and identify their individual risk. HATARI starts on the bug side, taking a bug report and extracting the associated change from the version archive. In a next step, earlier changes at this location are determined that were applied before the problem was reported. The last change before the bug was fixed is responsible for that bug. In a last step, HATARI determines for every location all changes that were ever applied and computes the individual risk of change as a percentage of fix-inducing changes. Fix-inducing is the change that was applied before a bug appeared inducing a later fix.

The HATARI prototype has been developed for Eclipse (Fig. 2.1). It uses Bugzilla as the bug database and CVS as the versioning system.

2.2 Combining Version Control and Bug Tracking Systems

In [FPG03] Fischer et al analyzed the Mozilla [Moz] project using CVS as the version control system and Bugzilla as the bug tracking system. They imported the CVS data and the Bugzilla data into a *release history database* (RHDB) and evaluated the Mozilla project later on. The import process starts with checking out the project from the CVS server. A shell script is used to traverse through the source tree structure and retrieve the modification reports. The log information is parsed by a Perl script and stored in the database. Bug report identifiers are extracted from the

[1] Hatari is the Swahili word for ”risk” or ”danger”

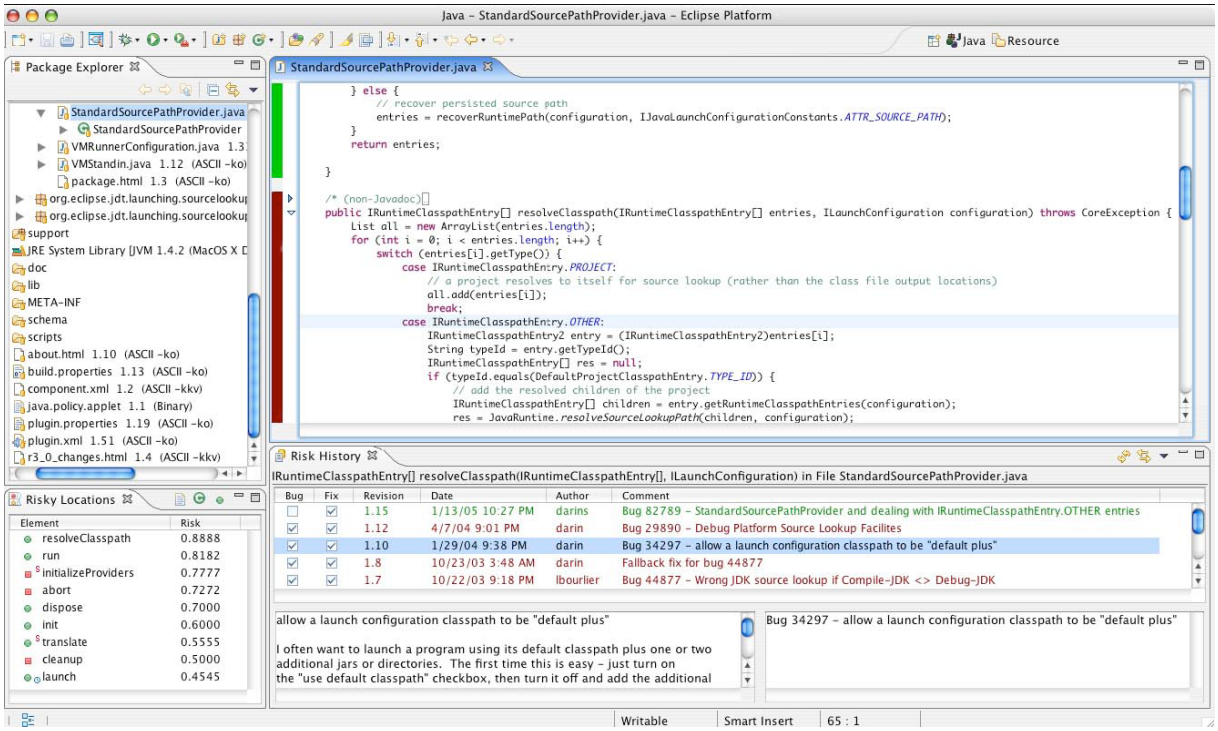

Figure 2.1: The HATARI plugin for Eclipse. The plugin extends Eclipse with colored bars in the Java editor that indicates risky locations in the code. The values of the risky locations are in the Risky Locations view. In another view there is the Risk History where the revisions of a Java source file are listed.

parsed reports using Perl regular expressions. Bug identifiers found in this step are used to retrieve the bug report descriptions via HTTP from the Bugzilla database. In the last step, the bug reports are parsed and stored into the RHDB.

The evaluation of the Mozilla project was done according to import, time-scale, historical, and coupling aspects.

2.3 Fine-Grained Change Analysis

In [FGP05] Fluri et al implemented an Eclipse plugin which uses the structure comparison function that comes with Eclipse. CVS tracks changes only on a text basis. Detailed information about changes in a class, method or field is not stored. The result is that change couplings also include couplings that are due to non-structural changes.

In their approach they used the source comparison facilities of the Eclipse IDE. They retrieve fine-grained change information, such as which method was changed, added or deleted. The validation of their approach revealed that lots of change couplings are not the cause of source code changes.

Chapter 3

Background

This chapter provides the theoretical background for this thesis. We will have a look at agile software development, a relatively new approach on how to manage software systems. Agile software development is chosen because it has the goal to minimize the risks involved in a project. In this thesis, we try to minimize the risk by learning from the past. We want to implement principles used in agile software development which claim to ensure the quality of a software system. If we apply those principles, we hope to detect weaknesses in existing projects.

3.1 Agile Software Development

As stated above, agile software development is a conceptual framework that supports managing software engineering projects. The goal is to minimize risks by using small timeframes for the release of new parts of the system. Iterations are always running versions and each iteration is a small project of its own. It is just the same with solving a complicated problem. We divide the big problem into smaller problems until we can solve the smaller problems. Piece by piece is then put together and the big problem is solved. Instead of doing a project in one gigantic step, it is done with small iterations that are put together piece by piece until the goal has been reached.

Agile methods do not put as much weight on documents as on face-to-face communication. Usually all people working on a project are in the same room. This also includes the customer. Customers normally do not know exactly what the outcome of the project should be. Therefore it is necessary that the customer is always present.

For agile software development there is a range of many different methodologies. For this thesis we are interested in principles that should lead to a good software quality. Good software quality can be achieved by many different means. We focus on the principles that support a developer in writing qualitative source code.

3.2 Object Oriented Design Principles

The list of principles we found is summarized here. An evaluation of the principles will be done in chapter 4.

3.2.1 The Single Responsibility Principle

"A class should have only one reason to change." [Mar02]

The idea of the Single Responsibility Principle is that classes have only one responsibility, and this responsibility is handled in only one class. In many cases, classes have many different responsibilities they actually should not have.

To illustrate this, lets have a class that handles many different responsibilities. For any reason, some clients have rising requirements to the system and we need to change something. The responsibilities are unnecessarily coupled in this class, so that changes on one responsibility might have an impact on the other responsibilities and vice versa.

On the other side, if we have several classes handling only one responsibility, every class with a part of the responsibility needs to be changed. This is very bad and makes the system rigid.

Responsibilities are a reason to change. Therefore if a class has only one responsibility, there is only one reason to change. The Single Responsibility Principle focuses on decoupling the different responsibilities of one class into separate classes.

3.2.2 The Open-Closed Principle

"Modules should be open for extension, but closed for modification." [Mey97]

The Open-Closed Principle states that existing code, which is running perfectly, must not be changed anymore. New functionality is added in new code, which does not need old code to be changed. The goal is to create software constructs which allow extension without modification. Plugins are a good example. The core of the product is not changed, while new functionality is added with plugins.

Entities which are created using the Open-Closed Principle are easy to maintain and reuse. The risk of destroying old, working code by implementing new functionality into the existing entity converges to zero.

Always sticking to this principle may lead to an overhead of design for the future. Maybe there is never the intention of extending a part of the software with new functionality. Then the developer did complicated work that could be avoided.

In Agile Software Development code is written to cover the demands of the clients. There is no build for the future necessary. If a client wants to have new functionality later, we may have to refactor our existing code. Now, after we needed to rebuild our code to match the new functionality, we create the code that applies to the Open-Closed Principle. If the client comes a third and fourth time, we do not have to worry anymore about more functionality - we just add it.

3.2.3 The Liskov Substitution Principle

"Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it." [Lis88]

In other words, any subtype must be able to substitute its base type. The violation of this principle leads directly to a violation of the Open-Closed Principle. Since the function that violates the Liskov Substitution Principle must know about all the derivatives of the base class, it has to be modified whenever a new derivative of the base class is created.

To avoid the problem of a derivative that can not substitute its base class, let us make the following example: Having a rectangle and a square. A square is a rectangle, mathematically,

there is no mistake about it. But a square behaves differently than a rectangle. If the width of a square is changed, its height needs also to be changed. The same does not apply to the rectangle. If the width is changed, the height remains the same as before. So if we use the `setWidth()` function of the base class, the square gets corrupted, since the height is not changed. Therefore the square violates the Liskov Substitution Principle.

To conform to the Liskov Substitution Principle, checking the is-A dependency on behavior is the way to go. A square object is not a rectangle object. It behaves different.

3.2.4 The Dependency Inversion Principle

"High-level modules shouldn't depend on low-level modules. Both should depend on abstractions. Abstractions shouldn't depend on details. Details should depend on abstractions." [Mar02]

A high-level module should communicate with its low-level modules through abstractions. The reason is illustrated in the following example: We implement a copy machine that reads from one specific source and writes to another specific source. The reader is a `TextReader` and the writer is a `TextWriter`. The copy machine contains a `TextReader` and a `TextWriter`, since it must know the classes to read from and write to. If we decide to change the reader from a `TextReader` to an `ImageReader`, we breach the Open-Closed Principle because we need to modify the existing working code in the `Copy` class to get the new functionality up and running.
Figure 3.1 illustrates an example where both, the high-level and low-level modules, depend on abstractions. New functionality has no effect on the `Copy` class.

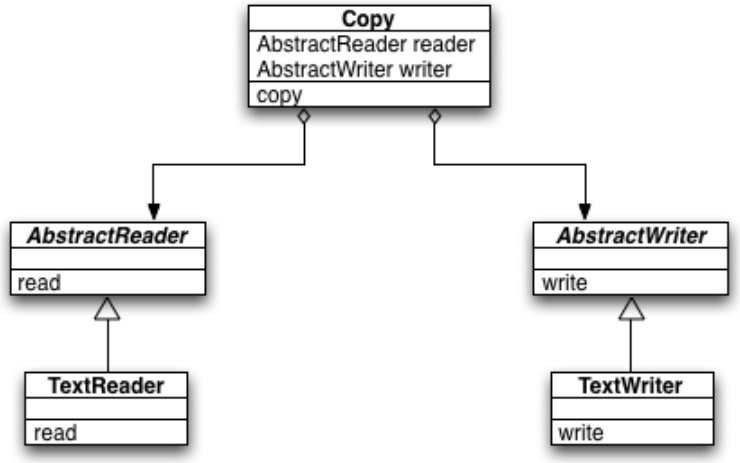

Figure 3.1: Illustration of the `Copy` Class. In this example, new functionality is added by adding a new concrete reader or writer class. The `Copy` class and the abstract classes need not be changed.

3.2.5 The Interface Segregation Principle

"Clients should not be forced to depend on methods that they do not use." [Mar02]

In many cases several methods are added to an abstract base class that only few of the subclasses actually use. This fattens an interface unnecessarily, since many of the subclasses do not use all the implemented methods. The problem may not be obvious if we do not look at it in detail. Every

subclass depends on the base class. If there is a change in the interface, all the subclasses need to change. Therefore it is necessary to keep the base classes small and implement only methods that every subclass actually uses. If there is the need for other methods that only some classes implement, we can use Java interfaces or build adapter classes. Adapter classes and other Design Patterns are illustrated in the book Design Patterns [GHJV97].

3.2.6 The Reuse/Release Equivalency Principle

"The granule of reuse is the same as the granule of release. Only components that are released through a tracking system can be effectively reused. This granule is the package." [Mar02]

In this principle, reusing and releasing packages are discussed. If we reuse classes of a released library, we are a client of the whole library and not only of the classes we reused. Reusing classes means, that the developer who reuses the classes never has to look at the source code.

3.2.7 The Common Closure Principle

"The classes in a package should be closed together against the same kinds of changes. A change that affects a package affects all the classes in that package." [Mar02]

The Common Closure Principle is tightly coupled to the Reuse/Release Equivalency Principle. Classes that change together should be located in the same package. It is inconvenient for the releaser and the reuser if there are classes in different packages that always change together. The reuser needs to upgrade more packages than he actually wants and the releaser can not release single packages.

3.2.8 The Common Reuse Principle

"The classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all." [Mar02]

The Common Reuse Principle helps a releaser to decide which classes belong to which package. A releaser that releases a package only because a single class changed does not make the reusers very happy. They have to revalidate their application only because of a class they actually do not care about. That is why classes that are reused together should be placed in a package.

3.2.9 The Acyclic Dependencies Principle

"The dependency structure between packages must be a directed acyclic graph. That is, there must be no cycles in the dependency structure." [Mar02]

In developing an application, there can be several people involved. It is therefore necessary to build teams that work on a certain part of the system. A team can be responsible for one or more packages, but it is important that only one team works on one package. If there are two teams working on one package, this can lead to problems if one team changes classes the other team needs and vice versa.

The same idea works for the whole project. Package dependencies should be a directed graph (Fig. 3.2). If there is a cycle in the package dependencies, this means that all packages in the cycle and their dependent packages need to be released at the same time. They all depend from each

other. Releasing a single package that another team uses becomes difficult. A cycle can emerge when new functionality is added. In this case, a solution to the problem is to create a new package that contains the classes the packages depend on (Fig. 3.3).

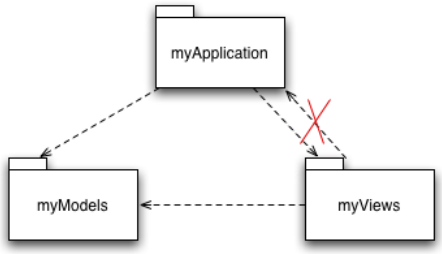

Figure 3.2: The dependencies between packages should form an acyclic graph. If there is a cycle, the packages can not be treated as a single unit anymore. Instead the packages in the cycle form a big unit, which is difficult to maintain.

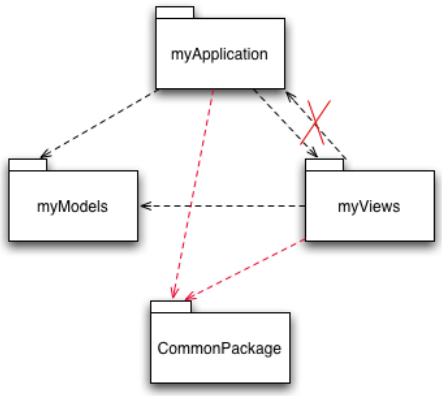

Figure 3.3: The dependency circle of `myViews` on `myApplication` can be broken if we create a new package that contains all the classes both packages depend on.

3.3 Code Smells

A code smell is bad code. This is the feeling someone gets when reading the source-code and thinks, that there is something wrong. Something smells here. That is what code smells are about. Instead of detecting principles that were breached, we can try to detect code smells. Since we are working with a version history and a bug tracking system, we pay special interest in this part of object oriented design heuristics.

Again we list some of the code smells Robert Martin describes in his book about Agile Software Development and add our own implementation ideas in chapter 4.

3.3.1 Rigidity

Rigidity is the opposite of flexibility. A stiff neck that has the cause due to muscular rigidity is difficult to fix. There are a lot of muscles depending on each other. Each one of them needs to be

made flexible in order for the neck to move properly again.

The same applies to a rigid software system. Changes inflicts cascades of subsequent changes. Until the rigidity is fixed, there must be changed many parts of the system.

3.3.2 Fragility

Fragility is the opposite of robustness. When we touch a drinking glass and that glass breaks, it is fragile. Even if we put our fingers only on a small part of the glass, the whole glass collapsed.

A system that is fragile behaves similar. After a simple change has been made to the system by adding new functionality or fixing a problem, the whole system breaks. Often, breakages occur in many places that have no conceptual relationship to the changed part.

3.3.3 Immobility

Immobility is the opposite of mobility. A laptop is a mobile system, which we can take to wherever we go. A desktop computer is rather immobile. There are a lot of dependencies for the desktop computer to work. We need additional power supply, we need an additional monitor, mouse and keyboard.

In an immobile software project, there are parts of the system that can not be moved to another part, because there are too many dependencies. Immobile systems are very bad for reuse purposes. Everything is coupled tightly together and no small updates can be applied to the system. There are only big modules with lots of dependencies.

3.3.4 Viscosity

Viscosity is the opposite of fluidity. A fluid mass that is not changed is in a stable state. By throwing something into the mass, the system is shortly in an unstable state, but recovers very quickly. A viscose mass is also in a stable state as long as it is not changed. In contrast to the fluid mass, the viscose mass needs a lot of time to recover from a change.

Viscosity in a software project means that it is difficult to get the system working correctly. After a change has been induced, the system is in an unstable state and needs many attempts to make it stable again.

3.3.5 Needless Complexity

Needless Complexity refers to systems that have infrastructure that is probably never used. Building infrastructure where not needed leads to more time spent in the development process and therefore higher costs. Additionally, the time to maintain the complicated parts of the system is increased, because it is hard to understand. Another term for Needless Complexity is *Building for the future*. Clients may never require new functionality that justifies building for the future.

3.3.6 Needless Repetition

Needless repetitions are code fragments that are repeated over and over again. The most dangerous repetitions are those that are slightly modified to satisfy the needs of the developer. If a code fragment has been copied and afterwards pasted and modified in several parts of the system, every single code fragment needs to be changed when a problem is detected. Many of the parts that need changing have no conceptual relationship and are therefore hard to locate.

3.3.7 Opacity

Opacity is the opposite of transparency. If we look through a window and see lightning and afterwards the noise of thunder, we know where the noise is coming from and understand the problem. If the window is too dirty to see through, we hear the noise and can only guess what the problem is. There might be many different possibilities where that noise came from and why it was there.

In an application, opacity refers to pieces of code that are hard to understand. If we do not know what we need to change, it can take a lot of time to change the system correctly. Opacity is closely related to all other code smells:

- *Opacity leads to rigidity.* In an opaque system, no one clearly knows where changes must be applied. This can lead to several changes in parts of the system that do not necessarily need change.
- *Opacity leads to fragility.* An example is a method that spans over several pages. If we change a part of the multi-functional method it affects all the responsibilities of the method. The same applies on a class that has several responsibilities. Problems in the class affect all responsibilities. Although the responsibilities do not have a conceptual relationship to the changed part, they will break because something does not work correctly in that class.
- *Opacity leads to immobility.* The example of the tapeworm applies here too. A so called tapeworm is a method which is extremely long and therefore hard to move to another place. An extremely long method that handles many things is hard to understand.
- *Viscosity leads to opacity.* In an opaque system, changes are seldom made correctly. If no one actually understands what he is going to change, new problems arise.
- *Needles complexity leads to opacity.* Complex abstract structures that are not necessary needed make a system hard to understand.
- *Needles repetitions lead to opacity.* Needles repetitions that are scattered everywhere in the system lead to an opaque system, since the one that *copy & pasted* the source code and altered it slightly might have copied it from someone else that already *copy & pasted* it from somewhere else.

Chapter 4

Approach

The goal of this thesis is to detect violations of design principles and code smells by analyzing the *changes* made for fixing *problems*.

To keep track of changes made to the system, we need a versioning system. Versioning systems store all changes made to a file. Every file receives its own revision number. Each time the file is changed, the revision number is increased. Every change is stored in the system. With the data from the versioning system we have the possibility to locate parts of the system that have many changes. Those parts may be risky parts where problems can emerge.

To detect violations of design principles and code smells we also need problem reports. A problem report is written, if a part of the system malfunctions. After a problem is reported there will be a change to the system to fix the problem. This change is the change we are interested in. With the problem report alone, we can only guess which part of the classes need to change. Therefore, with the combination of the history data, where the information of changes is stored, and the appropriate problem reports, which induced the changes, we expect a valuable source of data to detect violations of design principles and code smells

Figure 4.1 shows the approach on how we intend to detect problematic areas in the source code of a project. Most of todays software projects have some kind of versioning system. A versioning system has many benefits. The source code is located on a server and developers working on the project have always the possibility to get the newest version. If a machine crashes or the source code on a machine is lost for any reasons, there is always the possibility to restore the desired version from the versioning system. In addition, *changes* to certain parts of a system can be located. Source files that are updated on the versioning system receive a new version. The basis for our detection of risky areas is therefore the data under version control (1).

The version history data alone already serves many purposes. We can detect classes that are changed much more than other classes. Those classes are maybe difficult to change, but we can not know that for certain. Source files receive new versions for many reason. Already a line-break or space character inserted into a source file leads to a new version in the versioning system. Indeed, Fluri et al. [FGP05] showed in their approach on the analysis of change couplings, that many changes are no structural changes.

Critical parts of a software system, where principles may be breached and the code started to rot, may be found after a problem emerges. The problem causes the system to break and needs fixing. At this point we want to know which parts of the system are affected by the changes made to fix the bug. In mid-sized projects or larger projects, there can be many problems. To be able to keep track of problems that need fixing or problems that are already fixed, many software developers rely on a bug tracking system (2).

By combining the data from the versioning system with the data from the bug tracking system (3), we receive a valuable source of information to detect code smells and violations of design

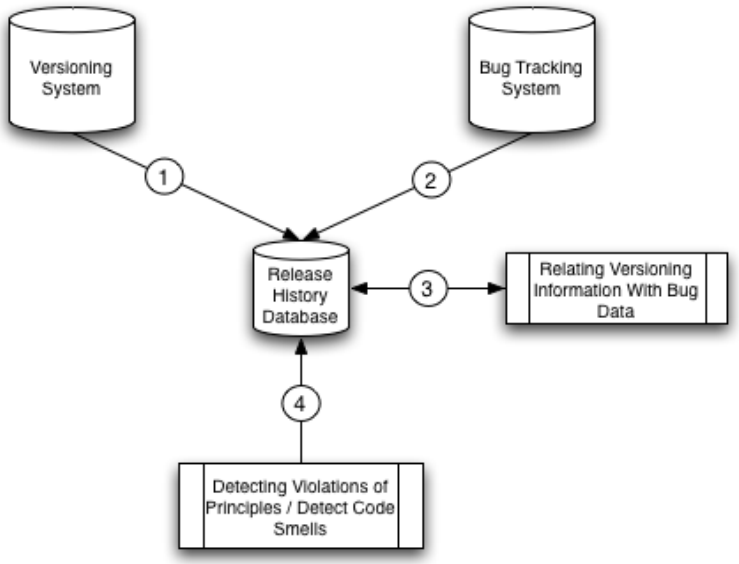

Figure 4.1: Basic idea to detect violations of principles and code smells

principles in a project (4). The goal is to provide the ability to analyze a software system directly in the IDE (integrated development environment) a developer works with by extending it with a plugin. With the ability to visualize problems where violations of principles or code smells are detected, we expect possibilities to reduce the risks involved in a software project.

In chapter 3 we discussed a range of *object oriented design principles* and *code smells*. In this chapter we already sorted out some of the principles that are not going to be implemented for the plugin. Those are the Reuse/Release Equivalency Principle, the Common Closure Principle, the Common Reuse Principle and the Acyclic Dependency Principle. Those are all principles that deal with packages. For the first prototype we are less interested in principles on package level than on principles that deal with classes, methods or interfaces. For the analyzation of the principles and code smells, we created a pattern that will help us to consider which of the principles or code smells are going to be implemented in a first prototype. Those that are not implemented yet can be implemented in a further version. The analyzation is made with the following scheme:

Level This is the level of detail a principle or code smell covers. Possible levels are package, method, interface or class.

Implementation This is not a detailed implementation, only implementation ideas on how a principle or code smell could be detected.

Problems Problems that make it difficult to implement a principle or code smell.

Example An example that explains the principle or a code smell.

Importance This is the importance for the plugin. We use data from a bug tracking system and data from a version archive. Therefore we want to implement code smells or principles that use both data. The importance can be low, medium or high. High means it uses both data and should be implemented. Medium uses only one side of the data, bug data or version history data. Low uses neither of the data.

4.1 Detecting Violations in Object Oriented Design Principles

The Single Responsibility Principle

A class should have only one responsibility.

Level Class

Implementation If the Single Responsibility Principle is breached, an update or a fix on a problem affects multiple classes and multiple methods. Although this is an indication that the principle might be breached, it is not for certain.

Problems Knowing what a class should be responsible for is a hard task even for humans. Automatically finding classes with multiple responsibilities is therefore difficult. If no problems are detected and that part of the code is only seldom changed, the problematic classes are hard to detect.

Example Telephone call - Instead of one class, that manages the call, the responsibilities need to be broken off into a class that handles the connection and one that handles the data.

Importance Medium

The Open-Closed Principle

Modules should be open for extension, but closed for modification.

Level Class / Method

Implementation Searching for fine-grained changes of classes in interfaces or abstract methods reveals classes that breach the Open-Closed Principle. Changed interfaces or changed abstract methods breach the Open-Closed Principle. All the classes that extend the abstract class need to modify the changed method. In general, classes or interfaces that are being extended or implemented should not be changed anymore.

Problems If developers stick too close to the principle, there might be too much build for the future. This could lead to Needless Complexity.

Example A class that prints to a specified medium: First only the specified medium is supported. When the customers need more media to print to, the class needs to be changed once to prevent future changes.

Importance Medium

The Liskov Substitution Principle

Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it. Every subclass must be able to substitute its base class.

Level Method

Implementation Methods in the derivative that override the methods of the base class can breach the Liskov Substitution Principle.

Problems Methods that override a method of the base class, but do not change any fields of the base class are substitutable. Detection of overridden methods which only change fields in the subclass are not easy to detect.

Example We know that, a square is a rectangle. Therefore we extend the `Rectangle` class and override the method `setWidth(double width)`. We want the `Square` to be always a `Square`, that is why we need to change the method. In the overridden method the height is set too when the width is set and vice versa. Now we can not destroy the `Square` object, but we lose the ability of the subclass to substitute the base class. This can be illustrated with the following example: We use `Square` instead of `Rectangle`. Then we set its width first to 15. The `setWidth` method sets the width and height automatically to 15. Now we set the height to 30. The `setHeight` method sets the width and height to 30. The length of the square is now 30. If we had first used the `setHeight` method and then the `setWidth` method, we would get 15 as the result. If the `Square` would be able to substitute the base class, the result would be always the same.

```
public class Rectangle{
   private double fWidth;
   private double fHeight;
   public void setWidth(double width){
      fWidth = width;
   }
   public void setHeight(double height){
      fHeight = height;
   }
}

public class Square extends Rectangle{
   public void setWidth(double sideLength){
      super.setWidth(sideLength);
      super.setHeight(sideLength);
   }
}
```

Listing 4.1: A `Rectangle` class that has a width and height and methods to set the width and height. The second class is a `Square` class extending the `Rectangle` class.

Importance Low

The Dependency Inversion Principle

High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.

Level Class / Interface

Implementation Several heuristics can be applied to detect if this principle is breached or not:

- No variable should hold a pointer or reference to a concrete class.

- No class should derive from a concrete class.
- No method should override an implemented method of any of its base classes (see Liskov Substitution Principle).

Problems Sticking too close to that principle may lead to Needless Complexity.

Example In chapter 3 we made an example of the copy class using an `AbstractReader` and `AbstractWriter` (see Figure 3.1).

Importance Low

The Interface Segregation Principle

Clients should not be forced to depend on methods that they do not use.

Level Class / Method

Implementation Clients are forced to depend on methods if they extend an abstract base class with abstract methods or if they implement interfaces.

1. *"fat" base classes:*
   If the base class contains abstract methods, the subclass that does not use them must have at least an empty method body. In the case of a return statement other than void, the method body contains a return null statement.
2. *"fat" interfaces:*
   Fattened interfaces can be detected by checking the method bodies in the source code. Usually interface-methods, that are not used, have empty method bodies.

Problems In some cases, it is necessary that an interface contains more methods than the implementing class. Examples are event listeners, for example the `MouseListener` in Java [Jav]. It contains the following methods:

mouseClicked(MouseEvent e) Invoked when the mouse button has been clicked (pressed and released) on a component.

mouseEntered(MouseEvent e) Invoked when the mouse enters a component.

mouseExited(MouseEvent e) Invoked when the mouse exits a component.

mousePressed(MouseEvent e) Invoked when a mouse button has been pressed on a component.

mouseReleased(MouseEvent e) Invoked when a mouse button has been released on a component.

An implementing class may actually only use the mousePressed event and all the other events may not be interesting. Although the methods are not used, they must be implemented. A solution to this problem would be an abstract adapter class. In the case of the adapter class, the method bodies are empty and subclasses of the adapter class can implement only what they actually need.

Example A class extending an abstract base class with abstract methods.

```
public String aString() {
   return null;
}
public void something() {
}
```

Listing 4.2: Even if the abstract methods in the base class are not used, the extending class must implement all the abstract methods

Importance Medium

4.2 Detecting Code Smells

Rigidity

Changes affect many parts of the system.

Level Class / Method

Implementation To detect problematic changes of the system, we can use the information we get from linking the data of the bug tracking system and the version history. With a previously defined threshold value which defines how many classes are allowed to be affected by a bug, we can detect those classes that are risky to change.

Possible implementation:

1. for each bug, count number of affected files
2. for each affected file, check if there is a major change (a change to a method, constructor etc.)

The check on the major change is necessary, because it is possible that a file is committed with other files that had only a minor change, such as adding an import declaration. Such changes do not make the system rigid. Changes on methods and constructors however can be seen as major changes. Even if the developer only needed to change a single line, he had to read at least the method again to know what he is going to change.

Problems Files can be committed together even if they have nothing to do with the bug fix. An example of a false major fix could be a space character that was accidentally inserted in a method. Return characters are often inserted to structure the code. The class is then saved and committed with the other files which gives a false result.

Importance High

Fragility

System breaks in parts with no conceptual relationship to the changed parts.

Level Class / Method

Problems An automated check on bindings which have no conceptual relationship is difficult. The goal to solve the fragility problem is to break up the parts with no conceptual relationship.

Example A class uses the method of another class which does by chance exactly what is needed. If the functionality of that other class changes, the functionality may not fit anymore for the class (Figure 4.2).

Importance Medium

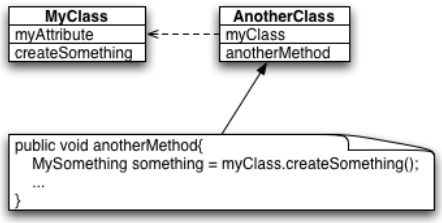

Figure 4.2: The class `AnotherClass` uses the `createSomething` function of `MyClass` just for convenience. By chance it just does what is needed in the `AnotherClass`. `AnotherClass` now depends on `MyClass`, although those classes would have normally nothing to do with each other. If the `createSomething` method of `MyClass` is now changed due to functional enhancements, it is possible, that the `createSomething` method does not fit anymore for the `AnotherClass` and the system will break.

Immobility

Difficult to disentangle parts from the system and move them to other places.

Level Package / Class / Method

Problems To detect immobile parts we must already know that we can not disentangle the specific part from the system. Automatic detection is therefore difficult.

Examples God classes which handle everything in a single method are good examples of immobile parts of a system. It is only possible to disentangle specific parts of the god classes if we re-factor it and make smaller parts out of it. Using only a small part of the god class is usually not possible.

Importance Low

Viscosity

Hard to do things right.

Level Class / Method

Implementation We can define a threshold value for how many changes a specific file should maximally have until a problem is resolved. Also, we can detect if a problem report appears directly after the problem was fixed.

Possible implementation:

1. For every bug count all the affected revisions of the source files, if it is a major change. Affected revisions are revisions that were created because of the changes that were necessary to fix a problem.

2. Revisions to a certain bug are found, if we check the commit messages for an indication of a bug number. Usually if there was a bug number mentioned, it is mentioned in every commit message until the problem is resolved.

Problems It is possible that files are committed together that had no relationship to the problem.

Examples The god class can be used here as well to illustrate viscosity. A god class usually needs a lot of changes until it works without errors. Since the god class is very big, it has a lot of places where changes must be applied. For every change that is committed, a new version of the class is made. Classes with lots of changes are not necessary god classes, but they are difficult to change correctly.

Importance High

Needless Complexity

Needless structures that have no direct benefit.

Level Class / Method

Problems With our data from the bug tracking system and the version history it is not possible to detect needless complexity. Since the application is working fine in most of the cases, no bugs are reported and therefore we can not detect needless complexity automatically.

Examples A system that is fully designed to handle future changes, although those changes may never be applied. Assuming we want to build a car. Needless complexity would be an arrangement on the car that facilitates adding wings, because there might be clients that want the car to fly someday.

Importance Low

Needless Repetition

Repeating code fragments.

Level Method

Implementation We can make a similarity search for every source file. Similar methods can be detected by checking the files that have a high percentage of equal code.

Example Assuming we have three developers working on an application: Sabine, Michael and Julia. Julia made a decent method that does almost exactly what Sabine need. Sabine copies the code that Julia already wrote. She does not want to invent the wheel again. She slightly modifies the code for her needs. Now Michael searching the project for code that already does what he needs stumbles upon Sabine's code. It does nearly exactly what Michael was looking for. So Michael copies the code and modifies it for his own needs. After some time a bug is detected in Julia's code. The change affects all the code clones.

Importance Low

Opacity

Hard to understand.

Level Package / Class / Interface / Method

Implementation An opaque system is hard to understand. Therefore there may be many bugs for a single class.

Possible implementation:

1. For every bug, get all affected source files.
2. For every affected source file count all bugs. If the number of bugs is higher than a predefined threshold, opacity is detected.

Importance High

4.3 Summary

In this chapter we presented the basic ideas of our approach. We want to detect violations of design principles and code smells by analyzing the changes made for fixing problems. We defined how important detecting a specific violation of a principle is and gave thought on how to implement the detection. The same was done for the code smells. Three of the analyzed code smells have importance level high and their detection will be implemented in chapter 5.

Chapter 5

Implementation

For the implementation of the plugin, a set of requirements must be met. In addition we need to evaluate the existing Components (Evolizer Base plugin and Bugzilla parser). Based on the requirements, we chose a system setup that will be used for the implementation. Components that are not reusable must be rewritten. We use the name *Class Evolution* for the plugin.

5.1 Requirements

In the case of a plugin there are two different kinds of requirements. On the one side, the requirements of the plugin itself needs to be specified. On the other side there is the surrounding application which is extended by the plugin and other applications the plugin communicates with. Therefore we classify the requirements into system requirements and plugin requirements.

5.1.1 System Requirements

Under system requirements we understand everything needed for the plugin to work with. This includes the surrounding IDE (Integrated Development Environment). The following system is desirable for our plugin:

Integrated Development Environment The development platform the *Class Evolution* plugin is extending supports Java. We want to evaluate Java projects and the plugin is written in Java. The environment the plugin is running on has an extensible plugin architecture. We do not want to change the source code of the IDE. The IDE is also well known to reach a large community and it is open source, since we want to work with standards. And finally it needs to be free of any charges.

Database The database supports SQL (Structured Query Language). We use Hibernate [Hib] as the object-relational mapping tool. The database is supported by the *Evolizer Base* plugin. It is fast and free of any charges. Setting up the database is easy and well documented.

Bug Tracking System The bug tracking system is easy accessible and it is open source and free of any charges. The system is well known and many companies use it.

Versioning System The versioning system is supported by the Evolizer Base plugin. It is free of any charges and keeps reliably track of the changes.

Parser There are two parsers needed in the system. One for the bug data and the other for the data under version control. Both parsers need to be fast and reliable. No additional hardware is required to run the parsers and the existing hardware must not be impaired by running the parsers.

5.1.2 Plugin Requirements

Other than the system requirements, the plugin requirements are based from the user's point of view. Instead of which system the plugin is running with, attributes such as how fast the plugin is are investigated here.

Availability The plugin must always be available for a developer. The parsing process which produces the data needed for the plugin must be separated from the actual analyzation of a project. Therefore, after the parsing process is finished, the plugin works even if there is no connection to the internet.

Speed The application of heuristics on bugs is a matter of seconds. No developer wants to wait a long time for a process to finish.

Usability A single click on a bug reveals all important information. In a well-structured view a developer can quickly analyze critical bugs and browse to the affected source code. A list of different heuristics can be applied on the bugs. Critical bugs are highlighted if a code smell is detected or if a design principle is breached.

5.2 Analyzation of existing Packages

There already exists a parser for CVS (Concurrent Versions System) and a parser for Bugzilla. Before reusing them we need to examine them closely in order to decide if they can be reused or not. For the analyzation we used several possible candidate projects for a later evaluation with our plugin. The chosen projects are all open source and Java-based. Landfill is the only exception. We used the bug entries of the Landfill project because it is probably the most difficult source to parse bug files.

- **argoUML** is an open source UML (Universal Modeling Language) modeling tool which is based on Java. It is able to create standard UML diagrams and provides the ability generate UML diagrams out of existing source code files [Arg] .

- **Landfill** is the *playground* for clients that want to "try out" Bugzilla. Since there may be many bug reports from many different users, it is perfectly suited to evaluate the existing `BugParser` on the XML data files from Landfill [Lan].

- **org.eclipse.team.core** is a sub project of Eclipse. These sub projects can all be seen as mid-sized projects. Eclipse is a very stable environment which almost never breaks. Therefore a sub project of Eclipse is an ideal candidate for a future evaluation with our plugin. We choose the org.eclipse.team.core [Eclb] project, because we use a parser that parses version history information. The org.eclipse.team.core project provides Eclipse with the necessary functions to create CVS connections.

The argoUML project and the org.eclipse.team.core are installed on a PowerBook G4 1 GHz with 1.25 GB Ram. We added additional lines of code to measure the beginning and the end of the parsing process for both, the CVS parser and the Bugzilla parser. The time difference indicates the amount of time used for the parsing process. In addition to the time measures, we examine if the packages are reusable or not. Reusable packages are desirable, because we do not have to care about the source code.

5.2.1 Evolizer Base

The Evolizer Base system is already a complete plugin. With a wizard function, we can enter the data required to store the parsed CVS information into the database. There are two supported databases, PostgreSQL [Pos] is selected as default and MySQL can be set in addition. The parsing process itself is done in a few minutes.

- *argoUML* time difference: 7m33s
- *org.eclipse.team.core* time difference: 1m53s

For all used Java projects under CVS control, there was never an error and all the data was successfully stored in the database. With the option if the database model should be recreated or not, the whole plugin can be used standalone and does not need to be integrated into our new plugin.
Reusability: Completely reusable

5.2.2 Bugzilla parser

The Bugzilla parser is a Java application that downloads XML bug files from a Bugzilla repository and stores them on the local file system. The number of XML files that are being downloaded must be modified in the source code. It is only possible to download a range of XML data files, for example file number 1 to file number 3412. In a second step, the downloaded files are being parsed. Again, to parse all the downloaded files, the source code must be modified for the downloaded range. The time measures were done in two steps. First, the time to download the bug was measured and afterwards the time to parse the downloaded files. Since the time to download the bugs can vary greatly because of the server workload, we do not put much weight on that time. However we put much weight on the time to parse the bug files.

Downloading the XML files

- *argoUML* We downloaded 3'451 XML files. The time difference was 1h40m.
- *Landfill* After having downloaded 8511 XML files, we received a *java.lang.OutOfMemoryError*. The time difference to download the 8511 bugs is 2h45m.
- *org.eclipse.team.core* Not applicable, because the bug data for the whole Eclipse project is stored in one place. To download a specific bug for the org.eclipse.team.core project, we need to know its identification number. Eclipse has a total of 138'521 bug entries (as of February 05, 2006). Downloading them all to make certain all bug files are included would take a great amount of time. Therefore we had to skip this project.

Parsing the downloaded XML files

- *argoUML* time difference: 55 minutes. The interesting thing here is that no database entries were made. Looking at the information printed in the console log revealed, that every XML

file was searched through and the root node was printed. Since there was no node with name *bug_id* to be found in the XML files, no entries were stored in the database.
Even after replacing all the *bug*-strings with *issue*-strings and fixing errors with *NullPointerExceptions* the `BugParser` could not succeed in parsing the bug files. The parser always receives an *OutOfMemoryError* (even after the memory for Eclipse was increased to 1GB). The error occurred after 50 minutes. Until then 1'890 files were parsed, but unfortunately nothing was stored into the database.

- *Landfill* The parser started to do well with the XML files of the Landfill Bugzilla database, but after two or three files were parsed, we received a *UTF-8* error. A UTF-8 error occurs when the parser receives a malformed character. This is a problem only parsers with the DOM API experience. In a DOM parser, text between nodes are elements, as well as the tags. The parser tries to create a text element out of the characters between tags. If there is a malformed character, we receive a UTF-8 error. The parser stopped after the error and there were no database entries.

- *org.eclipse.team.core* Not applicable because the XML files are not available.

Reusability: Only partially reusable. The meta model [Mar06] can, the parser can not be reused.

5.3 System Setup

Based on the requirements specifications we set and the analyzation of the existing packages, the following system setup is used:

Eclipse The Eclipse IDE [Ecla] is well known and widely used. It has a sophisticated plugin architecture and there already exist tools to build plugins. The Eclipse IDE also matches all the other requirements. It is open source, supports Java and it is free of any charges.

MySQL MySQL [MyS] is a database which is very popular among developers that use a database for their web-applications. Since there is a great number of users of MySQL it is also well supported by all common operation systems. The database can be installed by an installer, and further steps are well documented. Phpmyadmin [Php] is used to facilitate browsing data.

Bugzilla There are many bug tracking systems, such as the bug tracking system of SourceForge [Sou]. If we used this bug tracking system we could only evaluate SourceForge projects. Although there are plenty of promising Java projects that would be interesting to evaluate, such as the Azureus [Azu] project, we prefer projects that are not bound to one specific pool of projects such as the ones from SourceForge. With Bugzilla we have a well-known bug tracking system that is used for 523 companies. A list of projects using Bugzilla is found in [Bug]. Eclipse is also using Bugzilla.

CVS The two most popular versioning systems are CVS (Concurrent Versioning System) and Subversion. Since Subversion is rather new compared to CVS and not yet as well established, we prefer CVS to Subversion. In addition, only CVS logs are parsed with the Evolizer Base plugin. Subversion is not yet supported.

Evolizer Base The Evolizer Base plugin satisfies all the needs to parse the CVS logs and map the data to the MySQL database. It can be fully reused and does not need to be modified.

Bugparser Due to the unsatisfying results in the analyzation of the parser, we will not include it into the plugin. A new stable bugparser must be written. For this purpose we will use the existing meta model from the old parser and implement a SAX parser instead of a DOM parser, because the structure of the XML file does not need to be changed.

5.4 Plugin Work-Flow

The Class Evolution plugin has three major steps. First, the project that needs to be evaluated has to be checked out from a CVS repository. The Evolizer Base plugin requests the CVS logs (1 in Fig. 5.1) from the CVS server. The received logs are parsed and the data is mapped to the MySQL database (2). Part of this data are the *Revision* entries we are mainly interested in.

Second, the Class Evolution plugin searches the commit messages of the *Revisions* for any occurrences of bug numbers (3). The data of the bugs is then downloaded from the Bugzilla server (4). The Class Evolution plugin parses the data of the bug file and requests the source files of the files affected by this bug (5).

Every file has different revisions. The revision of a file before the bug was detected and the revision of the file after the bug was fixed are compared to each other and the structural changes are stored in the MySQL database (6).

The final step is the analyzation of a project. The Bugs view offers a drop down menu with different heuristics. The selected heuristic is applied to every bug in the view. If the heuristic matches a certain bug, that bug is highlighted in red. By clicking on a bug, the developer gets a detailed view of the bug and its problems in the Eclipse perspective (Fig. 5.2).

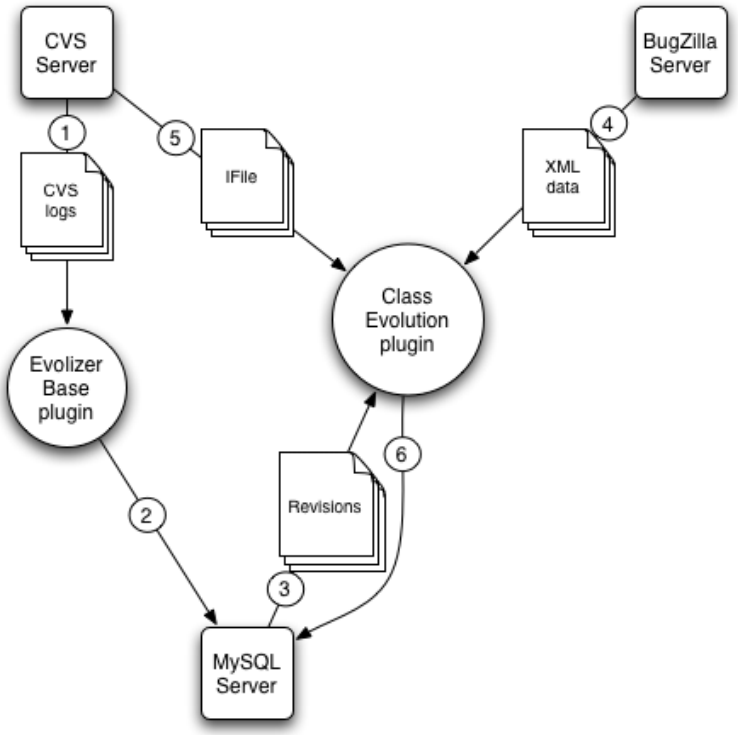

Figure 5.1: Data flow of the system

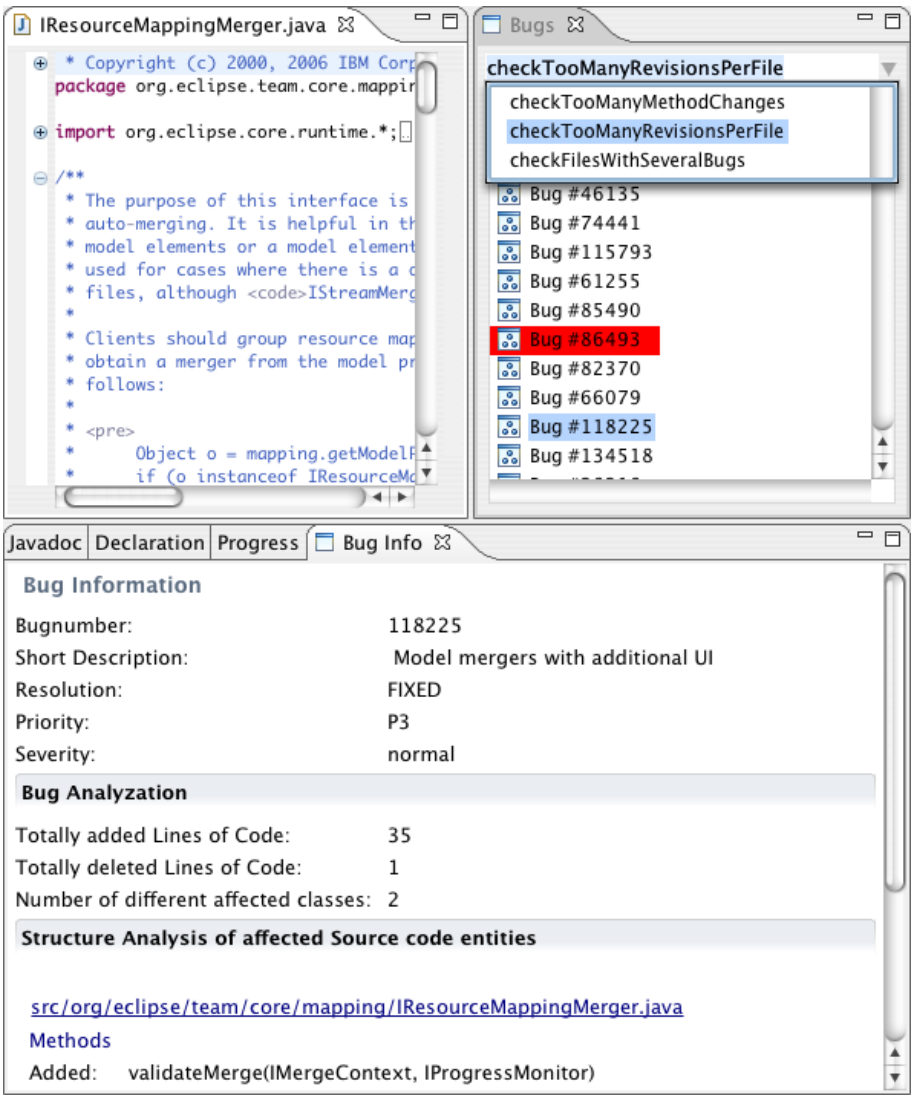

Figure 5.2: Eclipse Perspective of the Class Evolution Plugin. The Bugs view lists all the bugs that are mentioned somewhere in a commit message of a revision. In the Bugs view, the project can be analyzed by selecting heuristics from the drop down menu. In the above example, every bug that affects files with more than two revisions is marked. The Bug Info view displays all the details about the selected bug, including bug analyzation and structure analysis of the affected source files.

5.4.1 CVS Data Retrieval

The parsing process is straightforward. First, the CVS logs are retrieved from the CVS server. The connection to the server is set up using the CVS folder information.
The retrieved CVS log is then parsed line by line by applying regular expressions. The parsed information is then stored to the MySQL database using Hibernate. Details can be found in [WJ06].

5.4.2 Establishing the link between CVS data and bug data

Bugs in Bugzilla have a unique identification number. An application is able to retrieve a bug using its ID. Since we are only interested in bugs that can be related to the version history of changed files, we need the IDs of the bugs.

For this purpose we retrieve the revisions from the MySQL database. The goal is to find a Bugzilla bug number mentioned in the commit message of a revision. With these bug numbers and the location of the Bugzilla repository all the bugs useful for an analyzation can be downloaded. Bugs are only useful if they can be related to the version history.

Since commit messages are informal, bug numbers are noted in many different ways, if they are noted at all. Examples are "Fixed bug #8348", "Solved issue 5549" or "Fixed bug number 227". To catch these bug numbers, the following regular expression is applied to the commit messages:

```
(bug$|$issue)+\textbackslash D*(\textbackslash d+)
```
[1]

In natural language this regular expression can be interpreted as follows: "Find 'bug' or 'issue', followed by any non-digit characters, followed by at least one digit character.

The expressions in parenthesis form groups. In the regular expression above there are three groups. Group zero is the whole expression that has been found in the commit message. Group one is 'bug' or 'issue' and group two is the bug number we are looking for.

Bug numbers are stored in a `HashMap` that contains the bug number as the key and a list of revisions as value.

For every bug found in a commit message, the bug and the corresponding revision is stored. If the bug number already exists in the `HashMap`, the revision is added to the existing bug. With the list of revisions, the bug has the information about every change it was the cause of.

5.4.3 Bugzilla Data Retrieval

With the `HashMap` containing the bug numbers, all bugs related to the CVS repository can be downloaded from the Bugzilla repository.

Bugzilla offers many different formats (Table 5.1). For our bugparser we need a well structured and well formed document to reliably parse the received data. Therefore XML is the format we need from Bugzilla. Usually URLs where Bugzilla data is downloaded from have the following form:

http://*pathToRepository*/*cgiFunction*.cgi?[*functionParameter*]

pathToRepository is the path to the Bugzilla repository, for example *bugs.eclipse.org/bugs*
cgiFunction is the name of the cgi function, for example *show_bug*
functionParameter is a parameter the cgi function uses, for example *id=bugNumber* or *ctype=type*, where bugNumber is a unique integer value and type is one of the types listed in table 5.1.

---

[1]

- **\D** a non-digit character
- **\d** a digit character
- **|** logical 'or' operator
- **()** parenthesis form groups
- **+** at least one time
- ***** zero or more times

Table 5.1: List of available Bugzilla output formats

ctype	MIMEType
html	text/html
rdf	text/application/rdf+xml
xml	text/xml
js	application/x-javascript
csv	text/plain
png	image/png
ics	text/calendar

Since the path to the repository for a specific project can differ from project to project, the URL can be entered into the text-field of the `BugzillaDownloadWizard`. Figure 5.3 shows the wizard with an empty text-field. Into the URL field everything including the ID is entered, but excluding the bug number, because we already know every bug number we want to download. For every bug in the `HashMap`, a unique URL object is created. The stream from the URL object is then passed on to the `BugParser`.

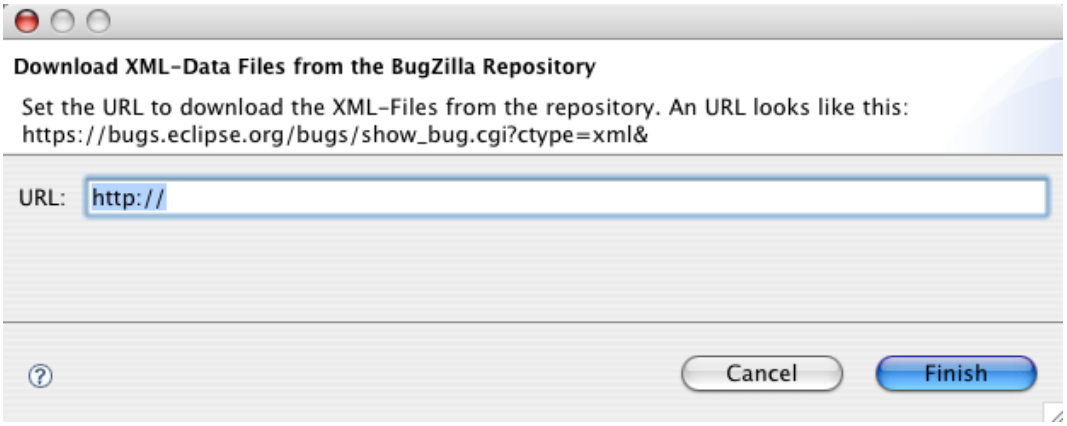

Figure 5.3: BugzillaDownloadWizard

BugParser

There are mainly three possible ways to retrieve data from an XML file.

- Writing a class which searches for node names and then tries to retrieve the data between the tags
- Using a tree-like API (Application Programming Interface), for example DOM (Document Object Model)
- Using an Event-Based API, for example the Simple API for XML (SAX)

Between the two kind of APIs are several differences

- Tree-like APIs map the XML document structure to an internal tree structure. The major advantage of this API is the memory in which the tree is stored. Since the tree resides in

memory, it is possible to restructure the whole tree, remove branches or copy branches to a different location in the tree. This can also be a major disadvantage, since building the tree and keeping it in memory costs both time and memory space.

- Event-Based APIs report parsing events such as the start of an element or the end of an element directly to the application through callbacks. There is no memory of a tree available and it is only possible to handle data in callback-methods. The major advantage of this API is that only very little memory is needed and it is much more robust to changes. Also the data is retrieved in a single sweep through the document whereas in a DOM tree, the needed information is always searched in the tree.

For our purposes with the Bugzilla data, restructuring the nodes of the document is not necessary and therefore we prefer SAX over DOM. The existing Bugzilla parser is a DOM parser. A comparison of the two parsers would be unfair, since they were designed with completely different goals.

While the old DOM parser downloads every available bug from the Bugzilla repository and then parses the downloaded files, the SAX parser we use here only downloads a small amount of bugs from the Bugzilla repository, namely the bugs that were identified with the information from the CVS data.

The `BugParser` consists of an `XMLReader` and a content handler, the `BugHandler`. The `BugHandler` is the heart of the `BugParser`. It handles the contents of the XML data. Names of the nodes we need information of are stored as global attributes. To retrieve data from the nodes, the `BugHandler` has several callback methods:

startDocument *Receive notification of the beginning of the document.*
In our implementation, nothing is done here.

endDocument *Receive notification of the end of the document.*
At the end of the document parsing, all revisions that have been added temporarily earlier are now added permanently to the bug object. At this point, structural changes between revisions are also added to the bug. Structural changes are changes between the revision that existed before the bug and the last revision that lead to fixing the bug.

startElement *Receive notification of the start of an element.*
At the start of an element, we store its name to use it later on if the element contains character data.

endElement *Receive notification of the end of an element.*
Nothing is done here.

characters *Receive notification of character data inside an element.*
At this point we do not know in which node we were. Therefore we stored the name of the element previously in the startElement event. Here we can get the character data if we wish information of the current element.

Information of an element is only taken in the case the name of the element is one we want information of. Table 5.2 shows all the element names the `BugHandler` takes data from. The element names are stored in global variables in the `BugHandler` class.

If the name of a node which the parser just passed through is not available in the global attributes, nothing happens. Like this the parser is highly adaptable to changes. In the case of the Bugzilla repositories there are mainly two different kind of XML documents. The documents created from Issuezilla, another fork of the Bugzilla project, have ”issue” elements instead of ”bug” elements. For our parser to adapt to this change we just have to tell it to handle those nodes

equally. Also, there are some nodes which do not exist in the documents created from Issuezilla that exist in the documents created form Bugzilla and vice versa. While the DOM parser stumbles into problems here, the SAX parser does nothing if a specified node name does not exist in the document.

The new `BugParser` parses all the necessary bug files in a reasonable amount of time. The parsing process we measured includes the following steps:

- downloading Bugzilla data
- parsing Bugzilla data
- retrieving structure change information

Note that the retrieving of the structure change information takes the biggest amount of time. For every affected source files we check out two revisions, the one before the bug appeared and the latest revision that fixed the bug. The same projects that were used for the Evolizer Base plugin were used for the time measures of the parser.

- *ArgoUML* time difference: 5m01s. 101 bugs were parsed.
- *org.eclipse.team.core* time difference: 12min41s. 131 bugs were parsed.

Based on the fact that the download-process alone took hours with the old parser, this is a massive improvement.

Retrieving Structure Change Information

The StructureChange class uses the possibilities of file comparison that eclipse uses. It extracts the change information of two files. The files used for the comparison are being checked out from the CVS repository.

In a first step, we use a `StructureCreatorDescriptor` to create a `StructureCreator`. Based on the file extension we receive the appropriate `StructureCreatorDescriptor`. The `StructureCreator` is used to create the structures used for the `Differencer` class to calculate the structure difference of the two checked out files. The differences are stored in `DiffNodes`. With the name, the kind and id of the node, we receive the following information:

- *name* The name of a node is the name of the method, field or whatever change has been made.
- *kind* The kind of `DiffNodes` specify whether it was an addition, a deletion or a change.
- *id* With the id we can extract a `DocumentRangeNode`. With this node, we can get the information about which kind of change occurred. Table 5.3 displays all the possible change types.

5.4.4 Detection of Violations

In the last step a developer can evaluate the project. From the list of heuristics in chapter 4, we implemented the ones with importance *High*. The analyzation can be made in several different modes. The developer can select each bug manually and explore the detail information in the Bug Info view or he can choose to select a heuristic.

After a heuristic is selected, all the bugs that contain classes with code smells or breached principles are highlighted. Highlighted bugs contain in addition the problem of this bug.

Table 5.2: This table shows the element names of the Bugzilla / Issuezilla XML data and the corresponding attribute in the MySQL database

Element name	**Attribute in the MySQL database**
bug_id / issue_id	Bugs.BUG_ID
creation_ts	Bugs.BUG_OPENED
short_desc	Bugs.summary
classification	Bugs.description
product	Products.prodName
component	Components.compName, Bugs.COMPONENT_ID
version	Components.compVersion
rep_platform	Computer_Sys.plattform
op_sys	Computer_Sys.Operating_system
bug_status / issue_status	Bugs.bugStatus
resolution	Bugs.bugResolution
bug_file_loc / issue_file_loc	Bugs.url
status_whiteboard	Bugs.status_whiteboard
keywords	Bugs.keyword
priority	Bugs.priority
bug_severity	Bugs.severity
target_milestone	Milestones.Milestn_description
reporter / assigned_to	Persons.personEmail
long_desc	
who	Comments.commentWho
bug_when / issue_when	Comments.commentDate
thetext	Comments.commentText
attachid	Attachments.ATTACH_ID
date	Attachments.attCreated
desc	Attachments.addDesc
ctype	Attachments.attType
data	Attachments.attSize
dependson	Dependencies.dependsOn
rep_platform	Computer_Sys.plattform

Table 5.3: List of change types

Change Type
Compilation Unit
Package
Import Container
Import
Interface
Class
Enum
Annotation
Field
Init
Constructor
Method

Implemented Heuristics

The only heuristics that matched the importance level high are code smells. Code smells are easier to detect than principles. Code smells are clearly defined, whereas violations of principles can only vaguely be guessed. Taking the Single Responsibility Principle as an example, we discovered, that a violation of the principle can affect multiple classes. But this is only one side of this principle, namely if there is a responsibility which is covered by many classes. The other side is one class that has many responsibilities. There is only one class affected in this case. The code smell that is generated by this principle in the first case is rigidity. Detecting rigidity is much easier. It is clearly defined that changes affect many classes. By defining a threshold value, any number of changed classes which is bigger than the threshold value counts as rigid parts of the system.

Rigidity Rigid parts of a system are parts with several changes to fix a problem. Listing 5.1 shows the implementation to detect rigidity.

```
public static void checkTooManyMethodChanges(Bug bug){
    int numberOfClasses = bug.getClassesCount();
    int numberOfClassesWithMethodChanges = 0;
    if (numberOfClasses > NUMBER_OF_CLASSES_THRESHOLD){
        for (StructureChange structureChange: bug.getStructureChanges()){
            if (structureChange.hasMethodChanges()){
                numberOfClassesWithMethodChanges++;
            }
        }
        if (numberOfClassesWithMethodChanges > NUMBER_OF_CLASSES_WITH_METHOD_CHANGES){
            bug.setHeuristicFlag(true);
            return;
        }
    }
    bug.setHeuristicFlag(false);
}
\end{normalsize}
```

Listing 5.1: The checkTooManyMethodChanges method counts method changes. A method change is either an added method, a deleted method or a changed method.

Viscosity We implemented the method checkTooManyRevisionsPerFile to detect Viscosity. Viscosity is defined as the problem that it is difficult to do the right thing. Therefore there can be many revisions until a problem is resolved. Listing 5.2 shows how Viscosity is detected.

```
public static void checkTooManyRevisionsPerFile(Bug bug) {
    HashMap<String, ArrayList<Revision>> map = new HashMap<String, ArrayList<Revision>>();
    for (Revision revision: bug.getRevisions()){
        //File is already in the HashMap, add revision to existing ArrayList
        if (map.containsKey(revision.getFile().getFullPath())){
            ArrayList<Revision> list = map.get(revision.getFile().getFullPath());
            list.add(revision);
        }
        //File is not yet inserted into the ArrayList. Insert new HashMap entry with the
        //revision and filepath
        else {
            ArrayList<Revision> list = new ArrayList<Revision>();
            list.add(revision);
            map.put(revision.getFile().getFullPath(), list);
```

```
            }
        }
        //HashMap is created. Check all the HashMap entries against the threshold value
        for (String file: map.keySet()){
            if (map.get(file).size() > NUMBER_OF_REVISIONS_PER_FILE){
                bug.setHeuristicFlag(true);
                return;
            }
        }
        bug.setHeuristicFlag(false);
    }
```

Listing 5.2: The checkTooManyRevisionsPerFile method uses a HashMap where all the revisions to a file are added. The files full path is the key and the list with revisions is the value of the HashMap. Flagged bugs will be colored.

Opacity To implement the detection of opacity, we created a HashMap where the key is a filepath and the values are bugs. Those bugs affect this file. In other words we count the number of bugs a file has and compare it to a threshold value (Listing 5.3).

```
    public static void createHashMap(Object[] bugs){
        hashMap = new HashMap<String, ArrayList<Bug>>();
        for (int i = 0; i < bugs.length; i++){
            Bug bug = (Bug) bugs[i];
            hashMap = new HashMap<String, ArrayList<Bug>>();
            for (Revision revision: bug.getRevisions()){
                if (hashMap.containsKey(revision.getFile().getFullPath())){
                    ArrayList<Bug> list = hashMap.get(revision.getFile().getFullPath());
                    list.add(bug);
                }
                else {
                    ArrayList<Bug> list = new ArrayList<Bug>();
                    list.add(bug);
                    hashMap.put(revision.getFile().getFullPath(), list);
                }
            }
        }
    }

    public static void checkFilesWithSeveralBugs(Bug bug){
        for (String file: hashMap.keySet()){
            if (hashMap.get(file).size() > NUMBER_OF_BUGS_PER_FILE){
                bug.setHeuristicFlag(true);
                return;
            }
        }
        bug.setHeuristicFlag(false);
    }
```

Listing 5.3: The createHashMap method fills a HashMap with the filepaths as key and the value as an ArrayList of Bugs. The checkFilesWithSeveralBugs method checks for every Bug if the threshold is exceeded by the number of Bugs

5.5 Summary

In this chapter we demonstrated the data flow of the Class Evolution plugin. We set up the requirements for the plugin and implemented a new bug parser that handles bug data much more efficient than the old parser. By downloading only the bug numbers we got from the CVS commit messages, we reduced the amount of bug files that need parsing. Finally, we explained the implementation of the three code smells opacity, viscosity and rigidity. In chapter 6 we will use the implemented plugin to analyze the org.eclipse.team.core sub project.

Chapter 6

Evaluation

This chapter covers the analyzation of the org.eclipse.team.core project. For the evaluation we used the implemented heuristics described in chapter 5 and the additional informations that is displayed for every bug, such as added lines of code, affected classes or number of revisions for a file.

6.1 Evaluation of the org.eclipse.team.core project

The argoUML project was not suited for an analyzation. All the bugs that were linked with the CVS information only affected non-java files. The intention of the plugin is to examine bad parts of the software in Java source code and not in additional files. Therefore we applied the Class Evolution plugin on the org.eclipse.team.core project. This project revealed many affected Java source files and was therefore suited for an analyzation.

6.1.1 Facts

The Class Evolution plugin searched in *3'043 Revisions* for bug numbers. Out of the *2'237* bugs found for the org.eclipse.team project in the Eclipse bugzilla list for version 3.x, *119 bugs* could be linked with the version history information. To fix the 119 bugs, 4'131 lines of code were added and 2'255 lines of code were deleted. For the affected classes we calculated the average for every bug. Every bug affects an average of 2.44 classes. We calculated the average also for the number of revisions a bug caused. The bugs caused a total of 363 revisions, which is an average of 3.05 revisions. Table 6.1 displays all the collected informations.

Table 6.1: Facts of the evaluation

Bugs	2'237
Total Revisions	3'043
Linked Bugs	119
Added LoC	4'131
Deleted LoC	2'255
Affected Classes in Average	2.44
Average of Revisions	3.05

6.1.2 Used heuristics

We used the implemented heuristics for detecting code smells. For every heuristic, we analyzed the highlighted bugs and noted special characteristics.

Rigidity

Out of the 119 bugs, we detected 6 bugs, which exceeded the threshold we set for the maximum of affected classes. If a bug needs to change more than five classes and five method changes, the bug is highlighted. Looking at those bugs in detail revealed the following informations:

Bug #61255 Reduce deprecation warnings for core.runtime.compatibility

- *Classes:* 11 different classes, 10 with method changes
- *packages:* 3 different packages
  - src/org/eclipse/team/internal/core/
  - src/org/eclipse/team/internal/core/subscribers/
  - src/org/eclipse/team/core
- *Conclusion:* The Common Closure Principle is violated. The changes to the classes might origin from a violation of the Single Responsibility Principle.

Bug #47193 ’

- *Classes:* 7 different classes, 7 with method changes
- *packages:* 2 different packages
  - src/org/eclipse/team/core/variants/
  - src/org/eclipse/team/internal/core/subscribers/
- *Conclusion:* The Common Closure Principle is violated. In four of seven changes, the method `run(IResource, IWorkspaceRunnable, IProgressMonitor)` is added, including the `DescendantResourceVariantByteStore`. Since the same method has been added in all of those classes, the assumption is that the Open-Closed Principle is violated because of an interface change.

Bug #126485 Changes to ISynchronizationScopeManager API

- *Classes:* 9 different classes, 9 with method changes
- *packages:* 5 different packages
  - src/org/eclipse/team/core/mapping/
  - src/org/eclipse/team/internal/core/mapping/
  - src/org/eclipse/team/internal/core/subscribers/
  - src/org/eclipse/team/core/mapping/provider/
  - src/org/eclipse/team/core/subscribers/
- *Conclusion:* The Common Closure Principle is violated. In five of the nine classes are the methods `refresh(ResourceMapping[]),getContext()` and `getProjects()` added. This indicates again a violation of the Open-Closed Principle.

Bug #120365 Model Participant still needs SyncInfoSet

- *Classes:* 21 different classes, 11 with method changes
- *packages:* 5 different packages
  - src/org/eclipse/team/core/mapping/
  - src/org/eclipse/team/internal/core/mapping/
  - src/org/eclipse/team/internal/core/subscribers/
  - src/org/eclipse/team/core/mapping/provider/
  - src/org/eclipse/team/core/subscribers/
- *Conclusion:* The Common Closure Principle is violated. There is not any change pattern to see in the changed classes. The SubscriberEventHandler class has many changes. Therefore the Single Responsibility Principle could be violated.

Bug #122849 Support for change sets

- *Classes:* 14 different classes, 10 with method changes
- *packages:* 5 different packages
  - src/org/eclipse/team/core/diff/
  - src/org/eclipse/team/internal/core/subscribers/
  - src/org/eclipse/team/internal/core/mapping/
  - src/org/eclipse/team/core/mapping/provider/
  - src/org/eclipse/team/core/subscribers/
- *Conclusion:* The Common Closure Principle is violated. Another pattern ca not be detected in the classes. All have some changes.

Bug #112927 Update CVS workspace job takes too long

- *Classes:* 8 different classes, 7 with method changes
- *packages:* 4 different packages
  - src/org/eclipse/team/internal/core/subscribers/
  - src/org/eclipse/team/internal/core/mapping/
  - src/org/eclipse/team/core/mapping/provider/
  - src/org/eclipse/team/core/subscribers/
- *Conclusion:* The Common Closure Principle is violated. Another pattern ca not be detected in the classes. All have some changes.

Viscosity

Only 2 Bugs out of the 119 bugs were detected that affected files which have been committed more than three times.

Bug #86493 Working set decorator does not change when contained resources change

- *Classes:* 7 different classes, 4 with method changes
- *packages:* 4 different packages

  - src/org/eclipse/team/core/
  - src/org/eclipse/team/internal/core/
  - src/org/eclipse/team/internal/core/subscribers/
  - src/org/eclipse/team/core/subscribers/
- *Conclusion:* The Common Closure Principle is violated. Other violations are not visible. The change for this bug is also on the edge of rigidity.

Bug #122849 Support for change sets This bug was already detected in the rigidity section. On top of the conclusions we made there, this part is also viscose.

Opacity

Out of the 119 bugs there was no bug that affects files which are already affected by other bugs. In our CVS and Bugzilla data, there is no file with different changes due to different bugs.

6.1.3 Conclusion

The source code of the org.eclipse.team.org project has a high quality. In average, there were 35 lines of codes added to fix a bug, which is rather small. There are only few critical parts in the system, such as the files that are affected by bug number 122849. This bug was highlighted for both rigidity and viscosity. The amount of 477 lines of codes to fix the bug is rather high compared to the average of 35. A surprising result is the check for opacity. There is no java source file that is affected by more than one bug, which is a very good result.

Although we suspected the quality of the project to be good, we could find a part of the system which is problematic. A refactoring, if the change was no refactoring, to ease changing the system in this part is necessary.

Chapter 7

Conclusions

In this thesis we implemented a plugin into the Eclipse IDE that analyzes projects by using the parsed information from the CVS log and the Bugzilla XML files. We implemented a new bug-parser, based on the meta-model of the old parser. By searching the commit-messages of the revisions for any occurrences of bug numbers, the link between Bugzilla data and CVS data could be established. In addition to the CVS and bug data we calculated the structural changes between revisions of a file. For the analyzation of a project, we collected and examined object oriented principles that aid the developers to create good quality software. On the other side we also examined code smells. Three of the code smells which were best suited for this thesis were implemented. The goal of this thesis was to analyze a middle-sized Java project by detecting violations of principles or by detecting rotting places in the source code. The org.eclipse.team.core project was successfully analyzed and although the quality of the project is high, we detected several code smells and violations of principles.

The Class Evolution plugin forms the base of a powerful tool to analyze small and middle-sized Java projects. The stored data from the version history and the data from the bug tracking system is a valuable resource to detect code smells and violations of object oriented principles. The data for an analyzation of a project is gathered in a few minutes and a developer can analyze the project directly in the Eclipse IDE.

7.1 Future Prospects

The plugin offers room for extensibility. By now the plugin parses only bug data from the Bugzilla system. The *Class Evolution* plugin can be extended to parse data from other bug tracking systems, such as the bug tracking system of SourceForge. The *Evolizer Base* plugin can be extended to parse data from a SVN (Subversion) log. The extension would open many more projects for an analyzation. Another important extension would be additional implemented principles to detect more weaknesses or to detect parts where several principles are breached. Right now, there are only a few automatic code smell detections implemented. The structure change information can be further refined. A change of an interface, such as changing the signature of a method is not a change, but an addition and deletion. This can lead to false analyzations. Browsing to the risky parts of the source code can be modified as well. At the moment, there is only the possibility to browse to the affected java source files. This possibility can be extended in browsing directly to the problematic part of the source file and highlighting it.

References

[Arg] ArgoUML. http://argouml.tigris.org/.

[Azu] Azureus - bittorrent client. http://sourceforge.net/projects/azureus/.

[Bug] Bugzilla installation list. http://www.bugzilla.org/installation-list/.

[Ecla] Eclipse. http://www.eclipse.org/.

[Eclb] Eclipse team - platform team integration framework. http://dev.eclipse.org/viewcvs/index.cgi/%7Echeckout%7E/platform-vcm-home/main.html.

[FGP05] Beat Fluri, Harald Gall, and Martin Pinzger. Fine-grained analysis of change couplings. In *Fifth IEEE International Workshop on Source Code Analysis and Manipulation*, September 2005.

[FPG03] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a Release History Database from Version Control and Bug Tracking Systems. In *Proceedings of the 19th International Conference on Software Maintenance (ICSM)*, pages 23–32, Amsterdam, The Netherlands, September 2003. IEEE, IEEE Computer Society.

[GHJV97] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, 1997.

[Hib] Hibernate - Relational Persistence for Java and .NET. http://www.hibernate.org/.

[Jav] Java™ 2 Platform, Standard Edition, v 1.4.2 API Specification. http://java.sun.com/j2se/1.4.2/docs/api/index.html/.

[Lan] Landfill: The Bugzilla Test Server. http://landfill.bugzilla.org/.

[Lis88] Barbara Liskov. Data Abstraction and Hierarchy. In *SIGPLAN Notices*, pages 17–34, New York, NY, May 1988. ACM Press.

[Mar02] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, first edition, October 2002.

[Mar06] Dane Marjanovic. Developing a Meta Model for Release History Systems. Master's thesis, University of Zurich, 2006.

[Mey97] Bertrand Meyer. *Object Oriented Software Construction*. Prentice Hall, 1997.

[Moz] Mozilla. http://www.bugzilla.org.

[MyS] Mysql. http://www.mysql.com.

[Php] The phpmyadmin project - effective mysql management. http://www.phpmyadmin.net/home_page/index.php/.

[Pos] Postgresql. http://www.postgresql.org/.

[Sou] Sourceforge - optimizing distributed development. http://sourceforge.net/.

[SZZ05] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. Hatari: Raising Risk Awareness. In *ESEC-FSE*, September 2005.

[WJ06] Michael Wuersch and Andreas Jetter. Evolizer Base, October 2006.