Università
della
Svizzera
italiana

Facoltà
di scienze
informatiche

Bachelor Project

EclipsEye

Spying on Eclipse

Yuval Sharon

supervised by
Prof. Dr. Michele Lanza
Romain Robbes

Abstract

Software evolution analysis permits developers and researchers to analyze software systems. It allows seeing the trends that the development of software has, with its present state and predict its future.
Today, researchers use the repositories of versioning systems to extract the history of a software, by downloading several versions of a system and then analyzing it. The problem with this approach is that versioning systems do not store all the important details of the development process of software, such as the changes that occur between two successive versions.
When developing software, programmers use an Integrated Development Environment (IDE). IDEs make the life of developers easier, by providing automated or semi-automated tools (e.g. refactoring actions). In order to gather accurate information about the development process of a software system, a change-based system can be built, which integrates within the IDE the developer is using. We can record relevant events generated in the IDE and process them incrementally. The events we are mainly interested in are class addition, modification and removal, method addition, modification and removal, refactoring actions, etc.
We implemented a plugin, called EclipsEye, for the Eclipse IDE, and more specifically for the Java programming language. By implementing it we were able to validate the applicability of the change-based approach and the plugin itself. For case studies we installed the plugin on our own Eclipse and we monitored the development of the visualization part. Furthermore we asked some of the second semester university students to install it as well, since they were doing a Java project using Eclipse.

Contents

Chapter 1

Introduction

Programmers and researchers are often interested in seeing the evolution of a software system. They want to understand how the software evolved, and predict how it will evolve in the future.
The history of a software system is usually extracted from the versioning systems developers use (such as CVS[1] or Subversion[2]). These allow them to share a project within a team and to see the changes that happened to it during its lifetime.
Versioning systems are widely used in the production of software systems. Their main advantage is that they are not programming language specific. They can be used with every programming language and even for other purposes (e.g. software documentations, any text document, etc).
For software evolution analysis developers and researchers need accurate data about the system. Since versioning systems are file-based, and not language-specific, they are not ideal to accomplish this type of analysis. The other problem is that changes are stored in the repository only when a developer commits his work. This means that we cannot control how often a commit will occur, since it depends on the programmer. With this mechanism a single commit could consist of many changes to a piece of software, but to us it would result as a single, merged change. This makes it hard and time-consuming to link entities across different versions, since an internal language specific model of each version has to be created. Furthermore it becomes difficult to distinguish and extract the originating events. The exact sequence of the changes and their time stamps are lost.
Today, most developers use an Integrated Development Environment (IDE). An IDE makes the development process easier, by providing automated or semi-automated tools (e.g. for refactoring actions). These IDEs contain all the information that is needed to perform evolution analysis, since each programmer's activity is performed within that IDE.
Instead of using a versioning system's repository as data source, an alternative is to provide a fine-grained monitoring of the programmer's activity extracted from the IDE he is working with. Getting information directly from the IDE allows us to record much more detailed and program-level events, such as: additions/removals/modification of classes/methods/fields, refactoring actions, navigation, etc.
From these events a change-based representation of the software system can be incrementally built.

[1] http://www.nongnu.org/cvs/
[2] http://subversion.tigris.org/

Romain Robbes[3] has developed a prototype of a change-based system as part of his ph.D. studies , called SpyWare[RL06]. It monitors a programmer's activity within the Squeak IDE[4], for the Smalltalk[5] programming language. In order to validate this new approach new systems have to be built.

1.1 Goals of the project

This project is aimed at porting SpyWare to the Eclipse IDE[6], for the Java[7] programming language. The porting is needed to validate the applicability of the change-based approach on different IDEs and different programming languages. Furthermore it gives more opportunities for case studies.
An important objective of the project is to understand the differences between the architecture of Squeak and Eclipse, and between the Smalltalk and Java programming languages. Due to these differences the implementation of the plugin for Eclipse is quite different from the one written for Squeak.
Eclipse is a file-based IDE, since each class in Java has its own file (except inner-classes and anonymous-classes). Squeak instead is a structured IDE which has a single file (called image) that contains the whole system. Java is a statically typed programming language, while Smalltalk is a dynamic language. Java has some concepts that are not present in Smalltalk (e.g. interfaces, enum types, explicit visibility, etc.). Furthermore the notification mechanism of Squeak is different from the one of Eclipse.

1.2 EclipsEye

We built a change-based system as a plugin for the Eclipse IDE, called EclipsEye. It monitors the developer's activity within this IDE, and it is specific for the Java programming language. EclipsEye implements specific listeners and participants in the Eclipse environment, to be notified of relevant events. These events are stored in external files to allow evolution analysis.
To validate the plugin we installed it first on our own Eclipse IDE. We have monitored our own plugin development and another Java project. Furthermore we asked some of the university students who were doing projects using Eclipse to install it. Some of them agreed to be monitored, and provided us with the recorded data about their projects.
With these preliminary case studies we were able to validate the change-based approach and the plugin itself.

[3] http://www.inf.unisi.ch/phd/robbes/
[4] http://www.squeak.org
[5] http://www.smalltalk.org
[6] http://www.eclipse.org
[7] http://java.sun.com

Structure of the Document

Chapter 2 describes the domain of the project and the problems of the current approaches to software evolution. Chapter 3 presents the solution to the versioning system's problems and the EclipsEye plugin. Chapter 4 outlines the preliminary validation. Chapter 5 concludes the document. Appendix A contains instructions on the installation of the plugin and its architecture.

Chapter 2

Software Evolution Analysis

Structure of the chapter

This chapter expands the main problems of software evolution analysis. Section 2.1 explains what is software maintenance, and why it is important in the development process of software. Section 2.2 introduces to software evolution. Section 2.3 gives an overview of the current approach to software evolution analysis: versioning systems. The problems of these approach are explained, with specific examples.

2.1 Software Maintenance

Our work is located in Software Engineering, which is a discipline that is concerned with all aspects of software production, development, and maintenance.

Software Evolution is a subfield of Software Engineering. Software Evolution is about the life cycle of software, starting from the initial phase of the development process. It includes software maintenance and reengineering[YW03, DDM+03].

In software development, after the initial phase, comes the maintenance phase, in which developers have to maintain and enhance the software. This includes mainly fixing bugs, adding new features and updating software to support new technologies and environments.
Fred Brooks states in his book The Mythical Man-Month[Bro75] that about 70-80% of the costs of a system comes from the maintenance phase, which means that this phase is very important in the software development process.

Software maintenance, in 1950 - 1960, was a very small part of the software life cycle[YW03, Ben93]. Later, as software grew bigger and was used longer, it became an important activity in the software development life cycle. Once the development of software is complete, and it is ready to be deployed, its maintenance phase starts. This means that software does not simply die after being developed.
Since the end of 1970 and the beginning of 1980, in some sectors, software maintenance was even taking more effort and costs than the initial development of the software.
Since then, the need for doing changes to software kept increasing and today it is still increasing. The main changes consist in fixing bugs, adding new features, updating

software to support new technologies and making optimizations. Often, changes done to maintain software, after being implemented, generate new problems that will need to be fixed too.

Maintenance has been divided into four categories by Lientz and Swanson in 1980[LS80]:

corrective: When developing software systems it is very likely that a bug or a fault will be present in the system. This leads to a behavior of the software that does not follow its specifications. In corrective maintenance known bugs and faults in the software are fixed.

perfective: During the process of developing software users (customers) often change their requirements. They require new features and modifications to the software system. Perfective maintenance takes care of modifying the system in such a way that the new requirements are satisfied. This category is large in software maintenance, since it includes addition of new features and it requires a lot of effort.

adaptive: The environments in which software operates is often subject to change. For example a modification in the operating system is done, or a computer hardware is changed, which may require also to modify the software to support the new environment.

preventive: In this category developers try to anticipate future changes to the software in order to reduce future maintenance. For example by applying refactoring principles to the current code will ease the future maintenance to that code. Another example is the *Y2K Bug*: companies and organizations had to check their systems before the year 2000 to prevent that software systems would operate incorrectly after that date.

Software is hard to maintain. When developers have to apply maintenance to software they have first to understand its structure, its design, and then modify it. Often, the system documentation is incomplete, or outdated, and this makes it even more difficult to maintain a system. About 50-60% of the time spent in software maintenance is used in reading and understanding code, before the actual modifications can be done.

2.2 Evolution Analysis

Software Evolution analysis is used by researchers and developers to understand software. This field started in the 70's with Lehman's work, while he was studying the OS/360 project. He formulated the laws of software evolution. They were derived from a direct observation and measurement of the evolution of a number of systems.
The eight laws are[LB85]:

Continuing Change A large successful software system must be continually changed and adapted, otherwise it will become progressively less useful. The process of continuing change stops when the cost of changing the system is higher than replacing it with a new one.

Increasing Complexity As a software system grows and is often changed, its complexity increases and it becomes more and more difficult to evolve, unless work is done to maintain or reduce the system's complexity.

Statistically regular growth The measures of a project and a system's attributes are cyclically self-regulating with statistically determinable trends and invariance.

Invariant work rate The work rate of a team evolving a large software system tends to be constant over the operational lifetime of that system.

Incremental growth limit Generally, the incremental growth and long term growth of a system is limited and it tends to decline.

Continuing Growth The functional capability of software systems must be continually increased and adapted to maintain user satisfaction over system lifetime.

Declining Quality Unless rigorously adapted to take into account for changes in the operational environment, the quality of a system will appear to be declining.

Feedback System Evolution processes are multi-level, multi-loop and multi-agent feedback systems.

An example of software evolution analysis can be done using the evolution chart visualization. This visualization shows the evolution of a property (e.g. number of methods) in the system. This mechanism is useful when we have to reason about the evolution of an entity in terms of one property[DGF04, GD06].
The evolution chart shown in Figure 2.1 shows the evolution of two classes with the property *number of methods*: class Foo, on the left, and of class Bar, on the right. Class Foo is growing: its number of methods increases with time. Class Bar, instead, remains constant: its number of methods does not increase quickly.

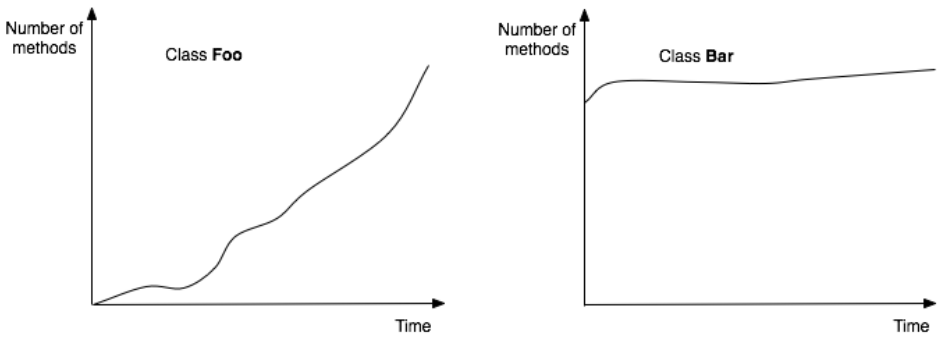


Figure 2.1: The evolution chart

2.3 Versioning Systems

2.3.1 Introduction

Version control (also known as revision control, source control, or source code management) is the management of multiple versions of a system. It is most commonly used in software development, as it allows to retrieve an old version of the system (or even only of a single part of it) to fix bugs, update features, or generate statistics about the differences between two or more versions.
With these systems it is very easy to setup a team-project environment. Developers can check-out a project from a versioning system repository and start working on it. After making some changes to the software system they can commit back to the repository their files. The versioning system will create a new version for each file modified. To allow a more efficient storage (less disk space usage) they usually use the delta compression, which stores only the differences between successive versions).
The most commonly used versioning systems are CVS[1] and Subversion[2].

2.3.2 Versioning Systems for Evolution analysis

Analyzing the evolution of a software system requires detailed data about it, or more precisely about its history.
Today, researchers and developers use versioning systems to acquire information about a software system, since they store all the versions generated through its development [RL05]. Although these versions are easily accessible they do not contain enough fine-grained information.

The main reasons for which versioning systems lose information are:

File-based vs domain-based The most used versioning system (e.g. CVS, Subversion) are file-based. Being file-based they are able to support any programming language (they are language-independent), or even any other kind of information which needs to be under revision control (such as text documents, images, etc.). This means that to understand a version of a software system written in a specific programming language extra computations are required.
All the entities of a software system (e.g. classes, interfaces, methods) lie within text files that have to be parsed manually to reconstruct the system's structure. The relationships of entities among many versions has to be constructed too, to discover what changed in the system between them. This last operation can be very hard if we consider refactoring actions that a developer may do; for example extracting some lines of code from a method and moving them to a new helper method would create no behavior difference in the system, but for a file-based version it would mean that a method has shortened, and a new method was added (see Section 2.3.3 for Examples).

Commits Versioning systems create a new version of a file only when the developer commits its work to the central repository. The action of committing is done at the programmer's will, and we cannot control it. This means that they may commit

[1] http://www.nongnu.org/cvs/
[2] http://subversion.tigris.org/

every hour, day, or even week. In fact developers don't commit after every single change they do on the software system. This creates two problems:

Merged changes A single commit can contain several modifications to the system, such as additions of classes, removal of others, and so on. All of these changes are merged, and are seen by the versioning system as a single, large, modification. This creates even more difficulties to discover all these changes.

Information loss Since changes are merged, they lose their time-stamps. By analyzing different versions of a system it's not possible to determine the exact sequence of those changes. The only time stamp available is the one of the commit action.

A real-world example is the *patch practice* that big open source projects adopt. Any developer is allowed to modify source code of an open source project, but usually he cannot commit it's work directly to the main repository. Instead he has to finish its feature development and then provide its code to the administrators of the project. They will analyze it and will commit it to the main repository by creating a patch. This approach creates even more difficulties for software evolution analysis through versioning systems, since with these practice changes can be very large.

2.3.3 Example of information loss

Using versioning systems to gather information about software systems to analyze their evolution is not ideal.

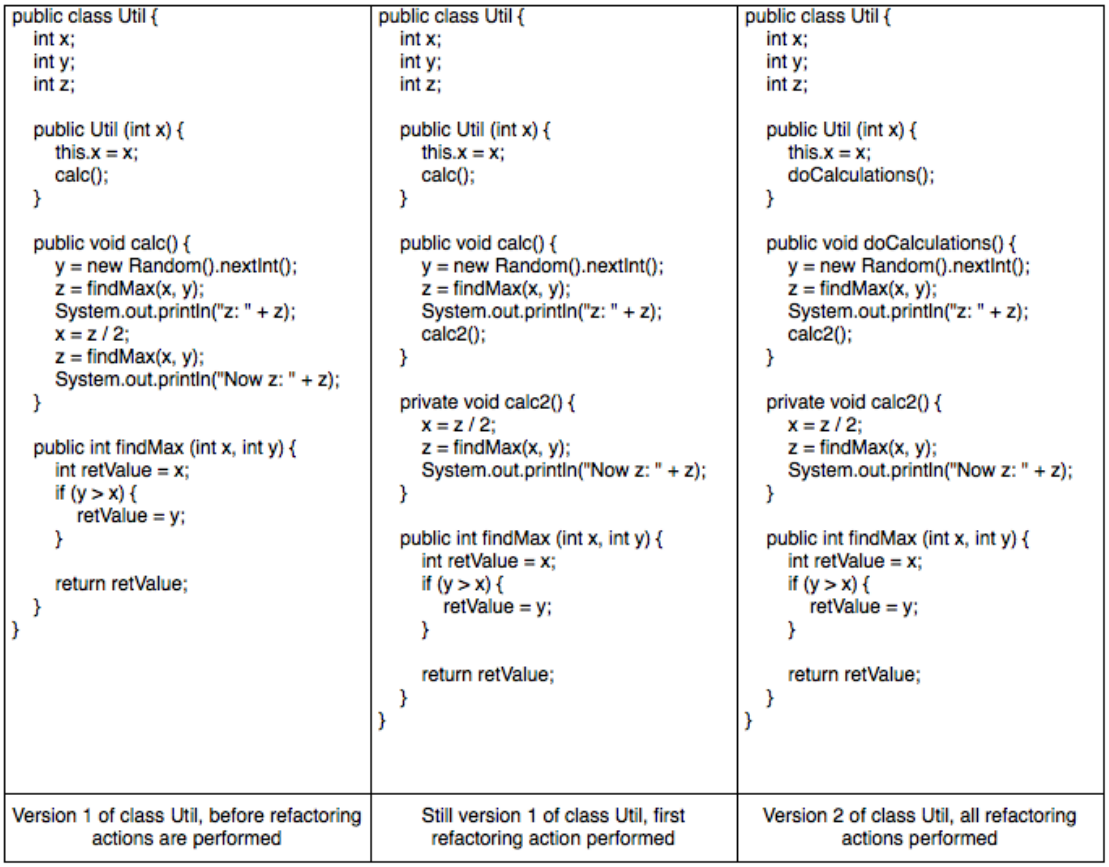
In this example consider a developer who performs the following refactoring actions:

- 1. Extract some lines of code from method *calc()* to a new method *calc2()*
- 2. Rename action:

  Rename method *calc()* to *doCalculations()*

  Fix all references to that method

After performing these small changes the developer commits its work to the repository. Using a versioning system we would have two versions to compare. Figure 2.2 shows these versions and the intermediate step. The first version of *class Util* is the one without any refactoring action performed. Version 2 is generated after all the refactoring actions are executed. The intermediate step is only present in the IDE of the developer, and it is instead lost by the versioning system.



```
public class Util {
   int x;
   int y;
   int z;

   public Util (int x) {
      this.x = x;
      calc();
   }

   public void calc() {
      y = new Random().nextInt();
      z = findMax(x, y);
      System.out.println("z: " + z);
      x = z / 2;
      z = findMax(x, y);
      System.out.println("Now z: " + z);
   }

   public int findMax (int x, int y) {
      int retValue = x;
      if (y > x) {
         retValue = y;
      }

      return retValue;
   }
}
```

Version 1 of class Util, before refactoring actions are performed

```
public class Util {
   int x;
   int y;
   int z;

   public Util (int x) {
      this.x = x;
      calc();
   }

   public void calc() {
      y = new Random().nextInt();
      z = findMax(x, y);
      System.out.println("z: " + z);
      calc2();
   }

   private void calc2() {
      x = z / 2;
      z = findMax(x, y);
      System.out.println("Now z: " + z);
   }

   public int findMax (int x, int y) {
      int retValue = x;
      if (y > x) {
         retValue = y;
      }

      return retValue;
   }
}
```

Still version 1 of class Util, first refactoring action performed

```
public class Util {
   int x;
   int y;
   int z;

   public Util (int x) {
      this.x = x;
      doCalculations();
   }

   public void doCalculations() {
      y = new Random().nextInt();
      z = findMax(x, y);
      System.out.println("z: " + z);
      calc2();
   }

   private void calc2() {
      x = z / 2;
      z = findMax(x, y);
      System.out.println("Now z: " + z);
   }

   public int findMax (int x, int y) {
      int retValue = x;
      if (y > x) {
         retValue = y;
      }

      return retValue;
   }
}
```

Version 2 of class Util, all refactoring actions performed

Figure 2.2: Versions example before and after the refactoring action is executed

By comparing only these two versions the physical changes that happened are:

- Method *calc()* was renamed to *doCalculations()* and some of its lines were moved to *calc2()*. These changes are hard to understand, since now the method *calc()* doesn't exist anymore, and the new method *doCalculations()* is shorter than it was before the extraction of code. This makes it even more difficult to find out whether *doCalculations()* is the same entity as *calc()* was or it is just a new method.
- The constructor *Util* changed since it called *calc()* which is now renamed to *doCalculations()*
- One method has been added: *calc2()*

Figure 2.3 shows how it becomes difficult to link entities between two successive versions. This example is a very simple one that consists of small changes to the system. In fact it would take a programmer about 1 or 2 minutes of work to do these changes. It is important to notice that already these small changes create information loss, causing evolution analysis not to be accurate enough.
Furthermore in real-life software development programmers do not commit their work so often to have always such little changes. This causes changes to be much larger and

detecting fine-grained differences between versions becomes even more hard.

In this specific example the method *doCalculations()* could be a new entity and thus method *calc()* was removed. Or method *calc()* was renamed to *doCalculations()* without creating or removing entities.

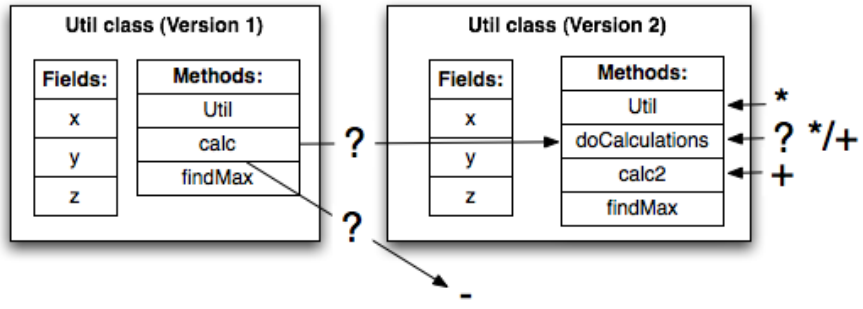


Figure 2.3: Difficulties in linking entities between two versions

Chapter 3

Solution

Structure of the chapter

Section 3.1 introduces the change-based approach with its advantages for software evolution analysis. Section 3.2 explains why we want to port the SpyWare plugin to the Eclipse IDE. Section 3.3 highlights the differences between the Squeak and Eclipse IDEs, and between the Smalltalk and Java programming languages. Section 3.4 shows the list of events that are recorded in EclipsEye. In Section 3.5 we describe how the plugin is implemented, and the problems we encountered during its development.

3.1 Change-based approach

To be able to monitor software evolution, without losing detailed information (such as time stamps, merged changes, etc.), a change-based approach can be used, based on two concepts:

1. An integration with the IDE to record as much data as possible
2. A model that is based on first-class change operations which helps in matching the incremental process of developing software.

Today most of the developers use IDEs for their programming work. These IDEs provide many useful tools to facilitate programming, such as automatic or semi-automatic refactoring actions, advanced debugging, easy browsing of the whole system, etc. These IDEs are usually extensible by plug-ins, to provide extra functionalities.

As said above, a programmer can easily browse the software system, perform refactoring actions, etc. This means that an IDE has a very good knowledge about the system being developed and the developer itself.
The idea is to integrate monitoring tools within the IDE and thus being able to record high-level events that would not be possible to get with versioning systems repositories.

Usually IDEs provide event notification systems, which are accessible through hooks. These hooks allow plugins to be notified of events that occur in the IDE. Tools can react to these events by creating first-class change entities.

First-class change entities are objects which contain the history of a system. The history is constructed incrementally.
Each change operation contains interesting information for evolution researchers, such as the time-stamp of the event, and who generated it.

Using the change-based approach only program-level differences between entities in the system are stored. At the opposite, by using traditional approaches, the history is modeled as a sequence of versions. When downloading several versions to analyze a system it may happen that they contain duplications among them (e.g. when a part of a system does not change in time).

The advantages of using a change-based system for software evolution analysis, rather than using repositories data, are:

Accuracy of events The first-class changes are generated as they happen, providing more accurate information than the one retrieved from a versioning system's repository. These events occur one at a time, giving more context to process them. Their time-stamps are more precise and they are not limited to the time-stamp of the commit. The exact sequence of the developer's actions are known.

Incremental processing By gathering events one at a time we are able to incrementally process them. With this mechanism it is much easier to maintain a representation of the model. The events generated by the IDE are high-level and their granularity contains classes, methods, variables. Instead, by using a versioning system's repository it would have been text files and lines. Parsing code is only needed at the method level, depending on the IDE.

3.2 Porting the approach to Eclipse

Romain Robbes has developed a prototype of a change-based system, called SpyWare. This system integrates with the Squeak IDE, and is developed specifically for the Smalltalk programming language.
The preliminary validation of this plugin has been done by monitoring SpyWare itself and by installing the plugin on the Squeak environment that second semester students used during their projects. SpyWare has limited case studies and it would need more to be successfully validated. Case studies are important to validate both the monitoring tool itself, and the change-based approach, which is still new in Software Evolution.
The main problem with Squeak and Smalltalk is that they are not widely used, and there are not as many users as there are for Eclipse and Java.
This project is aimed at porting the SpyWare plugin into the Eclipse IDE, and more specifically for the Java programming language. To accomplish this we had first to investigate the Eclipse event handling and notification mechanisms, and then experiment with the plugin development. We had also to understand the differences between the Eclipse and Squeak IDEs, and between the Java and Smalltalk programming languages.
Eclipse is a widely used IDE for software development and mainly for developing Java programs.
The plugin developed for Eclipse stores the monitored events in files, and the Squeak

version will still be used for creating the changes: for example to be able to browse the software system's code at any point in time.

3.3 Differences

The differences between Squeak and Eclipse, and between Smalltalk and Java led to different implementations of SpyWare and EclipsEye . Table 3.1 shows the main differences between the two IDEs. Squeak is a structured IDE which use a single, large, image file that contains all the environment of a software system. Eclipse, instead, uses several files to compose its environment: each Java class has its own file (except for inner-classes and anonymous-classes). This creates problems for example when monitoring refactoring actions, since these can affect more than a single file and we have to find which files have been modified and the program-level changes. In Squeak the changes happen only inside the single image file.
Squeak has an explicit *compile* operation for methods, and thus recording method modification in monitoring tools is quite straight forward; in Eclipse these events are harder to record (see Section 3.5). Eclipse has a continuous notification mechanism, where more than one event can occur at a time and they need a lightweight processing.

Squeak	**Eclipse**
structured editor	file-based editor
explicit *compile* operation for methods incremental compilation	continuous compilation
single notification	continuous notification

Table 3.1: The main differences between Squeak and Eclipse, relevant for change-based systems

The differences between Smalltalk and Java led mainly to having different types of events. Smalltalk has the concept of method protocols, which is not present in Java. The elements in Java that are not present in Smalltalk are:

Interfaces An interface is a type whose members are constants and abstract methods. This type has no implementation. Concrete classes can implement this interface by providing the implementation of the abstract methods.

Enum types An enum is a type that was introduced in Java 5.0. It contains mainly constants, but also methods, and fields.

Visibility In Java, classes, methods and fields can have different visibility: *public*, *private*, *protected*, *"default"*.

Abstract It is possible to define explicit abstract classes and methods. Abstract classes can contain both concrete methods and abstract methods. Abstract methods have no implementation.

Inner-classes Inner-classes are types declared inside an existing class.

Anonymous-classes Anonymous-classes are classes that do not have a name.

Imports A class has to declare import declarations when it needs to access external packages

Because of these differences we had to add java-specific events to our implementation of EclipsEye, and exclude some events that were not needed.

3.4 Events

3.4.1 High-level events

Here is the list of events, with their description, that EclipsEye catches in the Eclipse IDE:

Class Events:

A class entity contains a type, that can be a *class*, an *interface*, an *inner-class*, an *anonymous-class*, or an *enum type*.

Class addition a class was added to the system

Class removal a class was removed from the system

Class modification a class changed superclass, the implemented interfaces, or its modifiers

Method Events:

Method addition a method was added to a class

Method removal a method was removed from a class

Method modification the source code of a method was modified

Field Events:

Field addition an instance or a class variable was added to a class

Field removal an instance or a class variable was removed from a class

Field modification the type or the modifiers of an instance or class variable was modified

Package and Import Events:

Package addition a package was added to the system

Package removal a package was removed from the system

Import declaration addition an import declaration was added to a class

Import declaration removal an import declaration was removed from a class

3.4.2 Refactoring events

Eclipse provides many useful refactoring actions. *"Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs."*[Fow99]. It useful to monitor the refactoring actions because they show how the developer is working on a software system and how the code changes without modifying the system's behavior.
EclipsEye records the following refactorings:

General Refactorings:

Renaming java elements can be renamed. When a rename occurs all the references to that element are corrected (also references that are in other files). These elements can be renamed:

- Methods
- Method parameters
- Fields
- Local variables
- Classes
- Class parameters
- Enum constants
- Packages

Moving java elements can be moved. When a move occurs all the references to that element are corrected (also references that are in other files). These elements can be moved:

- Static methods
- Static fields
- Classes
- Packages

Changing method signature method's signature can be modified (e.g. parameter types)

Conversion Refactorings:

Converting anonymous class to nested converts an anonymous-class into a inner-class

Converting member type to top level converts an inner-class into a top level class (it creates a new compilation unit - java file)

Converting local variable to field converts a local variable into a field

Extract Refactorings:

Extract method extracting lines of code from a method to a new one

Extract a local variable creates a new variable assigned to the selected expression

Extract constant creates a static final field from the selected expression

Inline these elements can be deleted through *inline*:

Methods
Local variables
Static final fields

Hierarchy Refactorings:

Extract superclass extracts a common superclass from a set of selected sibling types.

Extract interface extracts a new interface from the selected class

Pushing down moves a set of methods and fields from a class to its subclasses:

Methods
Fields

Pushing up moves a field or a method to the superclass of its declaring type

Methods
Fields

Creation Refactorings:

Encapsulating fields replaces all references to a field with getters and setters

Introducing indirection creates a static indirection method delegating to the selected method

Introducing factory creates a new factory method (static), which will call a selected constructor and return the created object

Introducing parameter replaces an expression with a reference to a new method parameter

Generalization Refactorings:

Using supertypes replaces occurrences of a type with one of its supertypes (if possible)

Generalizing declared types change to a supertype of type references and declarations, it applies on:

Fields
Local variables
Parameters

Infer generic type arguments replaces type occurrences of generic types with parameterized types (where possibile)

3.5 The plugin

The solution is implemented as a plugin for the Eclipse IDE and specifically for the Java programming language. In Figure 3.1 the main window of the Eclipse IDE is shown, with the parts that are mostly used:

Package Explorer The package explorer shows the structure of Java classes and packages that are inside different projects.

Editor The text editor allows the developer to edit and write code.

Outline View The outline view shows in a list which methods and attributes are defined inside a Java class. This view is always in synch with the editor part.

Console The console is used to display the standard output inside Eclipse itself.

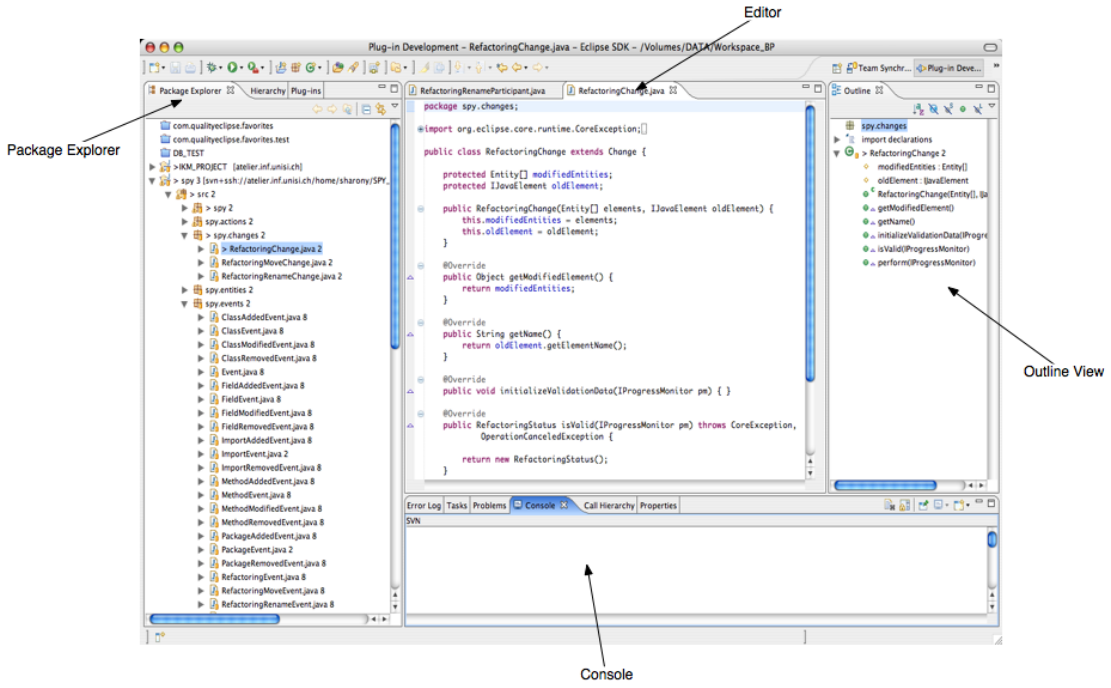


Figure 3.1: Eclipse view

In the Eclipse of Figure 3.1 EclipsEye is installed, and it is completely invisible to the developer. The only part that can be visible is the *EclipsEye View*, which is described in Chapter 4.
When developing the plugin we had to decide when to get the events. The alternatives are:

Save The developer can save the file he is currently working on when he decides to. The problem is that we can lose information by waiting for notifications at the point where the developer saves the file. Since he could have done many changes

before saving we are back to the previously discussed problems of snapshot-based versioning systems where we had to wait for the developer to commit its code.

Key stroke Being notified at key strokes level: each key pressed that causes modifications to java elements generate events. With this method we get continuous notifications about changes in the system, and we do not lose any event.

The strategy we adopted is to monitor changes at the key-stroke level. This allows us not to lose any event, and we do not get merged changes.

3.5.1 Listeners

In order to build EclipsEye we installed specific listeners in the Eclipse environment. These listeners send us notifications about changes that happen within the IDE.[CR04] The alternatives that Eclipse provides for notifications are the Resource change listener, or the JavaCore element listener.

A resource change listener can be notified of events indicating that resources have been added, modified, or removed during the course of an operation. Interested objects can subscribe to these events and react to them in such a way to keep themselves synchronized with Eclipse. The resources that are monitored are:

- Files
- Folders
- Projects
- Workspace root

The problem of using the resource change listener is that notifications are at the file-level, and therefore to find high-level changes (e.g. method addition) we would have had to parse each file on each notification. We used this mechanism only to get comfortable with the Eclipse plugin environment.

The JavaCore element listener provided by Eclipse notify of events that are specific to the Java programming language. With this approach the resources that are monitored are java elements:

- Classes
- Compilation Units
- Fields
- Import declarations
- Java Projects
- Methods

- Packages

We adopted this solution, since it allows us to be be notified of events at the program-level, such as:

- Class addition - modification - removal
- Method addition - removal (**not** modification)
- Field addition - modification - removal
- Package addition - removal
- Import declaration addition - removal

The events received are structured as java delta trees, and therefore we have to walk through them to identify the modifications that were performed. In Figure 3.2 an example of a delta tree is shown, where a field named *foo* was added and *bar* was removed from the system.

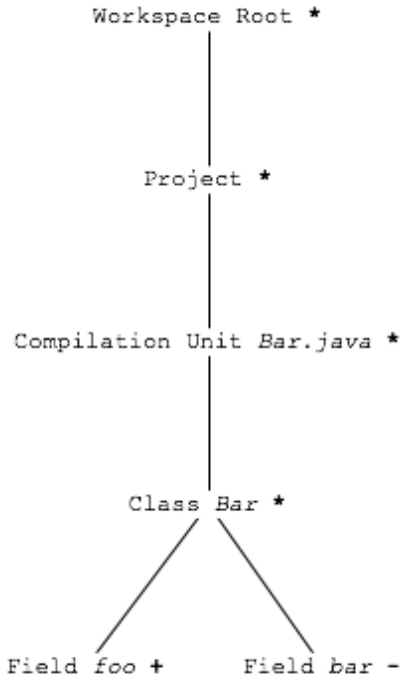


Figure 3.2: Example of a java delta tree

3.5.2 Method modifications

Changing source code of methods does not generate an explicit *method modified* event. Instead it is only notified that the class containing that method was changed. In Figure 3.3 we show an example of a method modification event as a java delta tree: on the left there is the actual tree that we get; on the right there is the tree that is not generated, but that was expected.

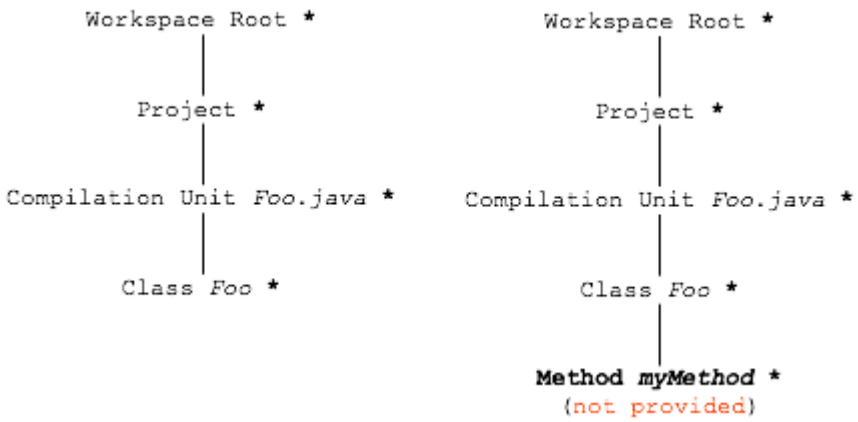


Figure 3.3: Example of a java delta tree for method modification

Since method modifications are not notified automatically we had to implement our own mechanism for recording this type of event. We experimented with several alternative implementations:

Cursor position when there is a modification in a class we can see where the cursor position is, and thus determine if it is inside a method. If it is the case that method was modified. The problem with this solution is that the cursor position is not reliable: one could quickly change its position before we actually get the notification and thus losing the method modification event, or getting a wrong method.

Outline view in the outline view of Eclipse the currently selected element (e.g. field, method) is known. If there is a modification in a class we can determine if it was done to a method and which one. The problem with this solution is that we rely on a GUI part of Eclipse: the outline view. This view can be closed by the developer and, when closed, it becomes unaccessible.

Modified class entities the solution we adopted consists in storing in memory the class entities that are currently being modified. When a modification in a class occur we compare the methods in the stored class entity with the current one. With this mechanism we can determine which methods were modified without relying on GUI parts or the cursor position.

3.5.3 Refactorings

An important part of the plugin is the recording of refactoring actions. In order to be notified of these actions we registered a *Refactoring Execution Listener* in the Eclipse environment. This listener is notified of refactoring executions, before they are executed, and after they have been executed. The problem with this listener is that it does not contain detailed enough information (e.g. it does not contain all the elements that were modified during the refactoring).
In Figure 3.4 an example of such a problem is shown: class *A* is declared in file *A.java* and it has an instance variable of type *B*. Class *B* is declared instead in another file *B.java*. Lets consider a rename refactoring in which we rename *B* to *Foo*. With this refactoring not only class *B* is modified, but also *A*, since it had a reference to *B*.
By using the Refactoring execution listener we are **not** notified of the change to class *A*.

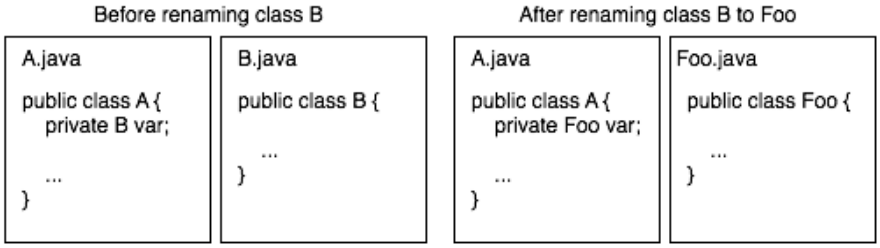


Figure 3.4: Example of a rename refactoring

Although external files modifications are not notified with this mechanism we are able to record the internal file modifications, where the refactoring occurs. To accomplish this we store two types of events:

Refactoring Start Event this event is recorded when a refactoring is about to start.

Refactoring Stop Event this event is recorded after the refactoring has been executed.

With this mechanism the events that are stored between a *Refactoring Start Event* and a *Refactoring Stop Event* are the ones caused by the refactoring action itself.

Plugins written for Eclipse can register specific *participants*. Participants are registered at plugin-level, and not as Java code. Eclipse provides the participant mechanism only for a small set of refactorings. We used the following:

- Rename
- Move
- Delete

These refactorings are the ones that are more likely to modify also other files. By using these participants we are able to find exactly which java elements are going to be modified in a refactoring action. And thus record these details.

3.5.4 Output

The events that are monitored by the plugin are stored in external files using the Java serialization. By using the serialization it is very simple to load the written files to analyze the recorded data.
Since the files can grow to be very large we added a *zip* feature: when the events stored reach 10 Mb of size they are automatically zipped so that space is saved. The defined size can be easily changed.
Once the events are stored they are also converted into the SpyWare output format, so that we can generate the changes and use the visualizations that SpyWare provides.

Chapter 4

Validation

Structure of the chapter

Section 4.1 explains the setup of the validation phase, the case studies, and shows the number of events we recorded. Section 4.2 shows the various visualizations we implemented. Section 4.3 highlights the results we obtained by analyzing the data we monitored.

4.1 Case studies

The change-based approach to software evolution still needs to be validated. By developing more and more systems monitored with this approach we will be able to understand its applicability on various IDEs and programming languages, and how it is effective.
EclipsEye has been tested on various, small-scale, projects:

EclipsEye View The visualization part of the plugin development has been monitored from the 3rd of June to the 26th of June 2007, recording 4161 events.

IR Project A pair project about building an information retrieval system (using the Lucene framework) has been monitored from the 26th of May to the 13th of June 2007, recording 3284 events.

Java Projects Some of the java projects of second semester students has been monitored from the 7th of June to the 12th of June 2007, recording 2132 events.

Table 4.1 shows the number of events that we recorded for the projects.

Event type	IR Project	Second semester's Project	EclipsEye View
Class Added	161	30	143
Class Modified	28	24	24
Class Removed	112	26	117
Method Added	371	223	424
Method Modified	1265	1123	2375
Method Removed	236	155	265
Field Added	265	180	165
Field Modified	67	52	25
Field Removed	197	105	93
Package Added	36	2	6
Package Removed	16	0	1
Import Added	344	223	339
Import Removed	123	55	149
Rename Refactoring	21	9	14
Move Refactoring	0	0	5
Other Refactoring	0	0	2
Working Sessions	36	29	47

Table 4.1: Events recorded for the projects

4.2 Visualizing data and metrics

The last part of the project consisted in developing a plugin that visualizes the information gathered by EclipsEye.
Figure 4.1 shows the Eclipse IDE with the EclipsEye view opened. In this view it is possible to choose between several visualizations and then navigate through the history of the system.

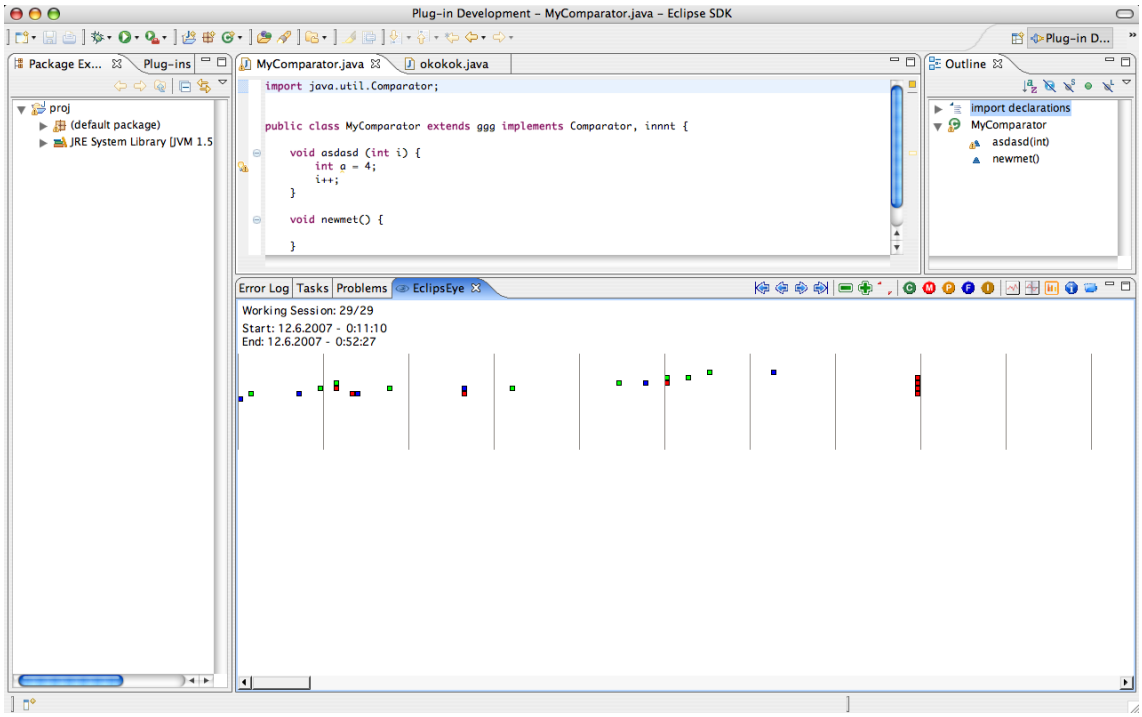


Figure 4.1: Eclipse showing the EclipsEye View

The visualizations we implemented are:

Histogram View The Histogram view shows all the events that were monitored, classified by each type of event (e.g. class addition, method modification, etc.). The height of each figure is fixed, while its width represents the number of events of a specific type. An example of a histogram view is shown in Figure 4.2.

Evolution Chart The evolution chart is a graph that shows the evolution of the system reasoning on a single property (e.g. number of classes). On the y axis is the property, while on the x axis is time.
The available properties that can be used for evolution charts are:

- Packages
- Classes
- Methods

- Fields
- Import declarations
- All of the above

An example of an Evolution chart is shown in Figure 4.3. In this example the property *number of methods* has been chosen. Therefore the events regarding methods are shown: method added (green), method removed (red), method modified (blue).

System Evolution Chart Another type of evolution chart is available, that is a global view of the whole history of the system. In this view all type of events are considered. An example is shown in Figure 4.4, in which the history of the last part of the plugin development is shown.

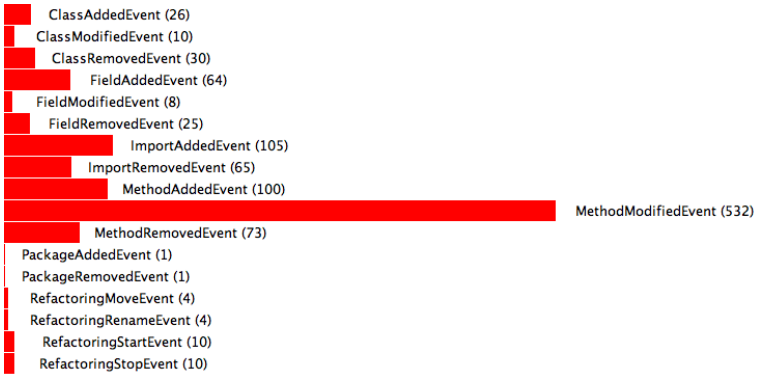


Figure 4.2: Example of a Histogram view

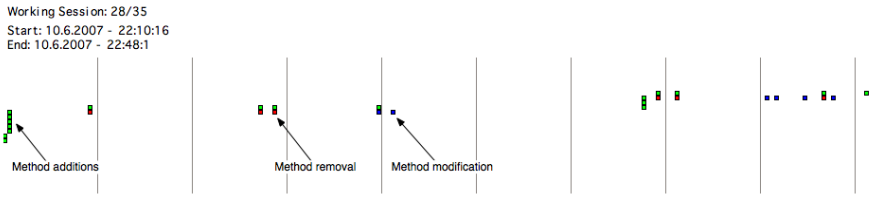


Figure 4.3: Example of an Evolution Chart

Since the total number of events can be very high, we used the concept of a *working session*. A working session is composed of successive events that are close to each other. We decided to split each working session when there is 30 minutes of inactivity. By using working sessions we have to visualize less events on a single screen. The working sessions are used to display the evolution chart since it shows the evolution of the system with respect to the time. If we had to display all the events on a single chart it would have become too large to be easily readable. In the visualization it is possible to navigate through the working sessions.

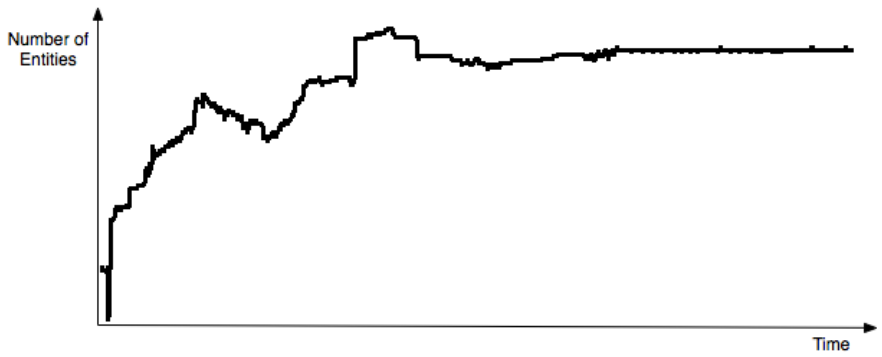


Figure 4.4: Example of a Global Evolution Chart

4.3 Results

4.3.1 IR Project

Figure 4.5 shows the IR Project evolution chart. The red circles are the particular developer sessions we analyzed.

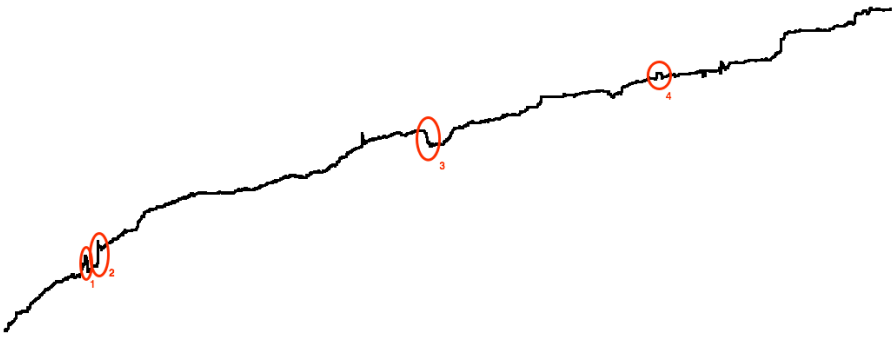


Figure 4.5: The evolution chart of the IR Project

Session 1 is shown in Figure 4.6. In this session we can see many removing events. These are removals of a package and its classes. The package which is removed is named *org.apache.lucene.demo*, and the classes removed are: *IndexFiles*, *SearchFiles*, *IndexHTML*, *DeleteFiles*, *HTMLDocument*, *Entities*, etc. These classes belong to an external package provided by Lucene, that were copied into the project. These removals are probably related to a small cleanup of the system, since the developer did not need these external classes anymore.

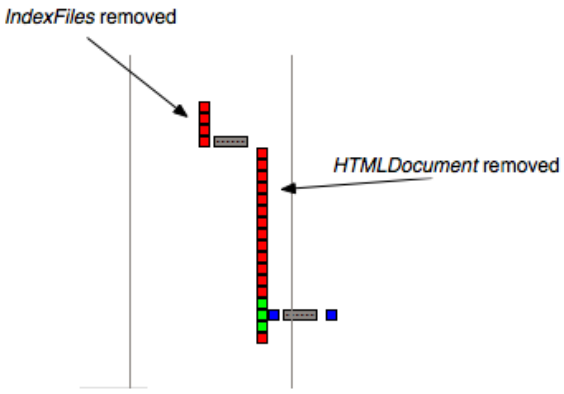


Figure 4.6: Session 1 of the IR Project

Session 2 is shown in Figure 4.7. In this session there are many addition events, and more precisely addition of import declarations. In Eclipse when code is copied and pasted the imports are automatically added. Therefore in this session probably some

code was copied and pasted, generating many import declaration additions. All the import declarations are added to a class named *TextSamplerDemo*, and they are mainly related to the Java Swing and AWT: *javax.swing.JSplitPane*, *javax.swing.JTextArea*, *java.awt.Cursor*, *java.awt.GridBagLayout*, etc.

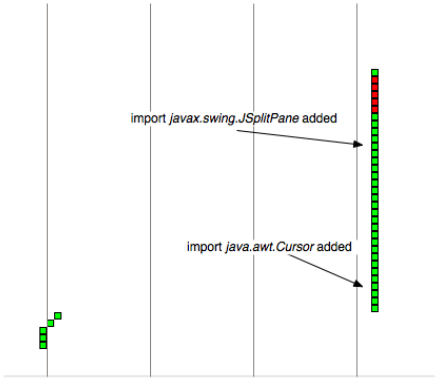


Figure 4.7: Session 2 of the IR Project

Session 3 is shown in Figure 4.8. This session is composed of several removals of entities. This is a bigger cleanup than the one of session 1, since in this case several packages, import declarations and classes are removed. The classes that are removed are named: *MainTest*, *ReaderTest*, *BrowserLauncher*. Several import declarations are removed from *HTMLTest*.

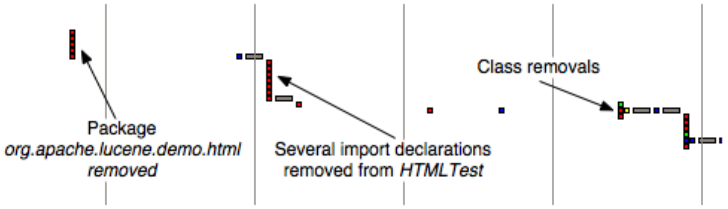


Figure 4.8: Session 3 of the IR Project

Session 4 is shown in Figure 4.9. In this session the developer adds some methods to the system, modify them and then removes them. It was probably a testing session. The programmer didn't really know how to accomplish what he had to do, and thus he tested some code. In fact by analyzing the events we see that a class named *Interface* is changed to implement the *WindowListener* interface, then methods related to this interface are added: *windowClosing*, *windowClosed*, *windowIconified*, *windowDeiconified*, *windowActivated*, *windowDeactivated*, *windowOpened*. At this point some of these methods are modified, and later removed.

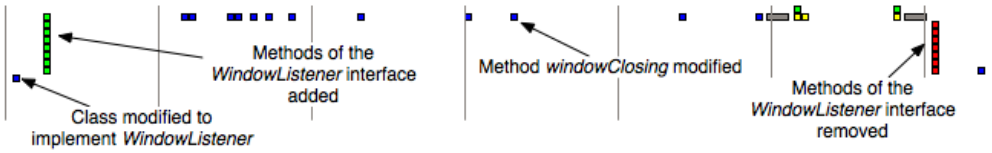


Figure 4.9: Session 4 of the IR Project

4.3.2 Second Semester's Project

In Figure 4.10 we present one of the second semester student's projects.

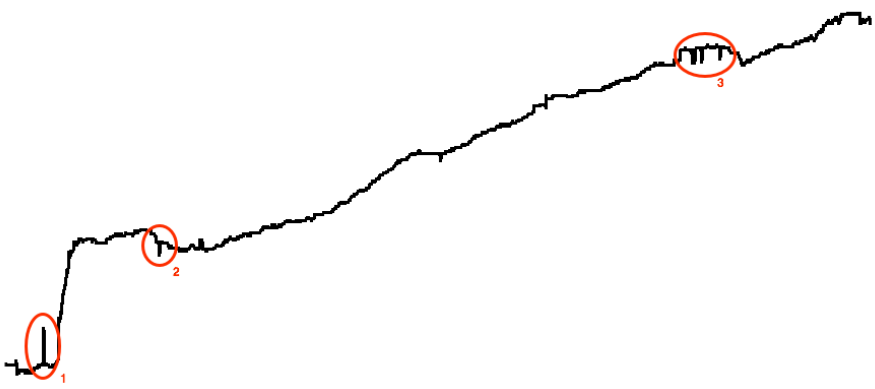


Figure 4.10: The evolution chart of a second semester student's project

Session 1 is shown in Figure 4.11. In this session a class called *StatsColumnModel* is created. This class implements the interface *TableColumnModel* and many methods are added to it. These methods are auto-generated by Eclipse. After a while the developer seems to have changed his ideas and removed all the methods, and the class *StatsColumnModel* too.

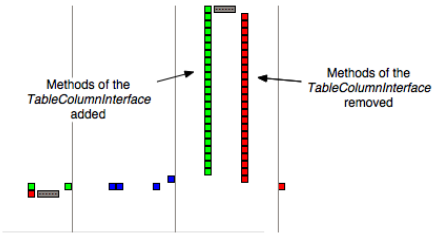


Figure 4.11: Session 1 of a student's project

Session 2 is shown in Figure 4.12. The developer removes some methods from a class named *PanelBets*, and immediately after he adds them again, probably by *undoing* his action. In this session the programmer seems to be quite unsure.

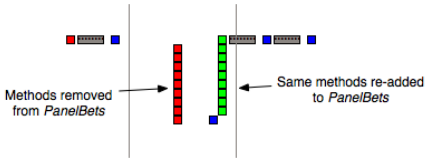


Figure 4.12: Session 2 of a student's project

Session 3 is shown in Figure 4.13. This session is similar to session 2. In this case methods are removed from class *PreferencePanel* and are re-added immediately after. These are methods of the interfaces *MouseListener* and *KeyListener*. In this session the developer seems to be very unsure, since he removes and adds the same methods of the same class four times in a row. After the fourth time he actually starts implementing the methods.

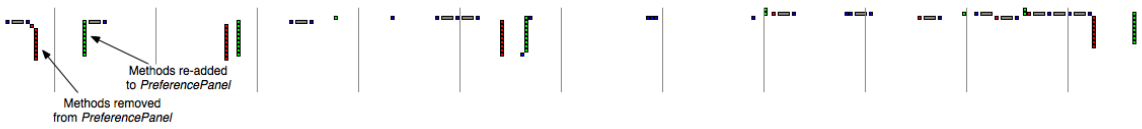


Figure 4.13: Session 3 of a student's project

4.3.3 EclipsEye View

In Figure 4.14 we show the evolution chart of the EclipsEye View development.

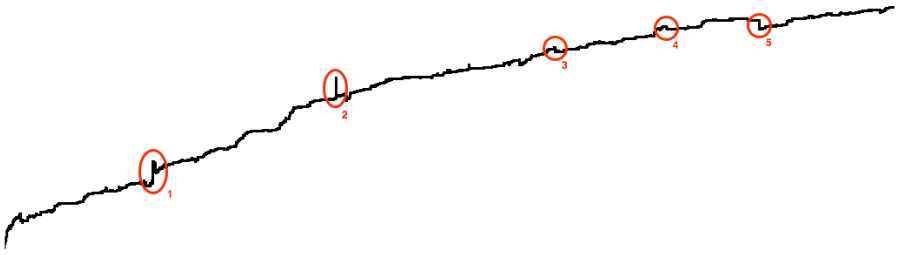


Figure 4.14: The evolution chart of the EclipsEye View development

Session 1 is shown in Figure 4.15. In this session many methods, fields, inner-classes and import declarations are added to a class named *AnimationView* at the same time. Later in the session some of these java elements are removed from another class, named *SpyView*. This indicates that the developer copied and pasted some code from one class to the other, and then removed what he didn't need from the original class.

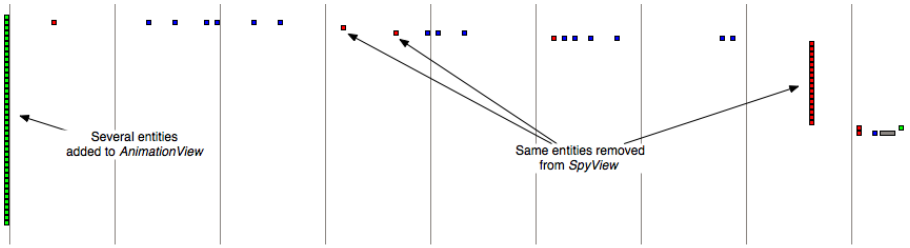


Figure 4.15: Session 1 of the EclipsEye View development

Session 2 is shown in Figure 4.16. In this session an anonymous-class is created with many methods, and immediately after all the methods are removed. By analyzing the events we can see that the anonymous-class created was an *IAction*, which is in a interface containing many abstract methods that have to be implemented, such as: *setHelpListener*, *setToolTipText*, *run*, *getId*, etc. After removing the methods the developer changes the anonymous-class to be an *Action*, which is an abstract class that allows him to implement only the methods he is interested in. In this case the method he implements is called *run*, which is called when the action is executed.
Sessions 3, 4 and 5 are cleanup sessions in which classes are removed from the system.

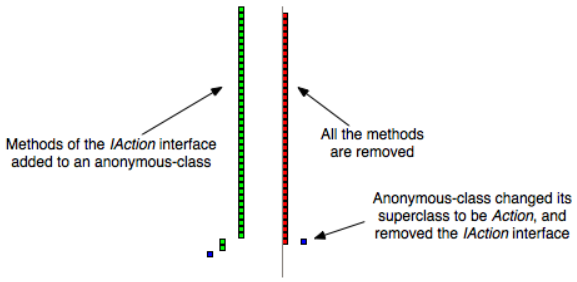


Figure 4.16: Session 2 of the EclipsEye View development

Chapter 5

Conclusions

Software evolution analysis requires accurate data about the system. Using versioning system's repositories to construct the history of software leads to information loss.
Change-based systems are integrated within the IDE the developer is using, and they process the events incrementally, as they happen, without losing information.
SpyWare is a prototype of a change-based system developed for the Squeak IDE, and for the Smalltalk programming language. Since the case studies for SpyWare and the change-based approach are limited we ported it to the Eclipse IDE, and for the Java programming language. This let us validate the applicability of this approach on a different IDE and language.
We built EclipsEye, that is a plugin for Eclipse which monitors the developer's activity within this IDE at the key-stroke level. Each change done to a Java element generates an high-level event, that is stored in an external file. The files generated are used for software evolution analysis.
During the development of the plugin we had to evaluate different possible solutions to the problems we encountered.

5.1 Contributions

We developed a change-based system that monitors the programmer's activity within the Eclipse IDE:

High-level changes Monitoring high-level changes in a software system. These changes contain detailed information about its development (see Section 3.4.1 for the list of events).

Refactoring actions Refactoring actions done inside Eclipse are monitored. The modified elements that such action affects are recorded (see Section 3.4.2 for the list of refactoring events).

Storing the events The recorded events are stored in external files that can be loaded into the Visualization plugin.

Visualizing data The monitored data can be visualized with views, such as the Histogram view, the Evolution Chart, or the System Evolution Chart.

5.2 Future Work

This project can be extended by adding new features, such as:

Monitor navigation It can be useful to monitor the navigation of the developer inside the Eclipse IDE. It can show the developer's intentions and exactly what he does.

Monitor copy and paste actions Copy and paste actions often occur in IDEs. By monitoring these actions we can see how the developer write its code.

Monitor executions of programs Java applications can be executed within Eclipse. It is possible to monitor these executions. We could see for example whether the developer executes some JUnit test suites, or simply how often he runs its programs.

Add views Adding more views to the visualization plugin can be useful. We could see the evolution of a system reasoning on other properties. A 3D view can be also implemented.

Appendix A

Plugin

A.1 How to use

In order to install the EclipsEye plugin copy the .jar file into the plugins directory of Eclipse. Then restart Eclipse.

EclipsEye depends on some plugins. These plugins are included in the default installation of Eclipse, and are:

- org.eclipse.jface.text
- org.eclipse.jdt.ui
- org.eclipse.jdt.core
- org.eclipse.core.resources
- org.eclipse.core.runtime
- org.eclipse.ui
- org.eclipse.ui.editors
- org.eclipse.ui.part
- org.eclipse.ui.texteditor
- org.eclipse.ui.views
- org.eclipse.ltk.ui.refactoring
- org.eclipse.ltk.core.refactoring

The visualization plugin of EclipsEye uses the following plugins:

- eclipseye
- org.eclipse.draw2d
- org.eclipse.core.runtime

- org.eclipse.ui

The draw2d plugin is not included within Eclipse, and has to be downloaded separately from *http://www.eclipse.org/gef/*.

A.2 Architecture

The architecture of the plugin is shown in the next Figures as UML Class Diagrams.

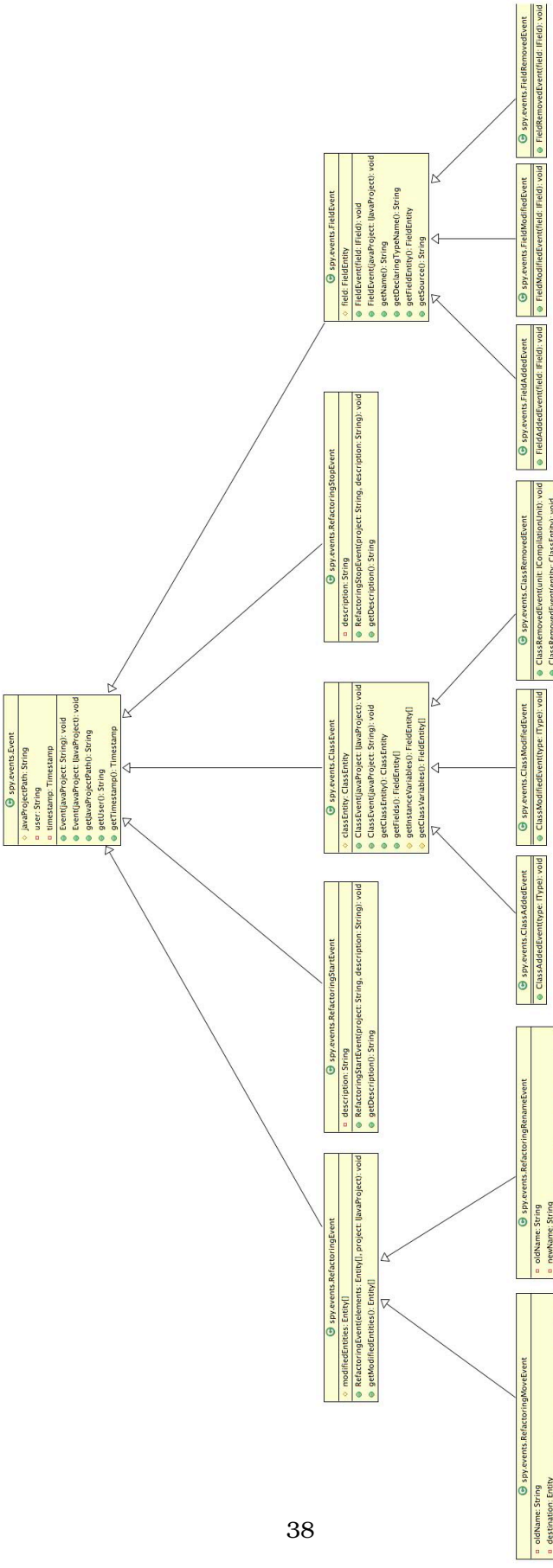


Figure A.1: Events (1/2)

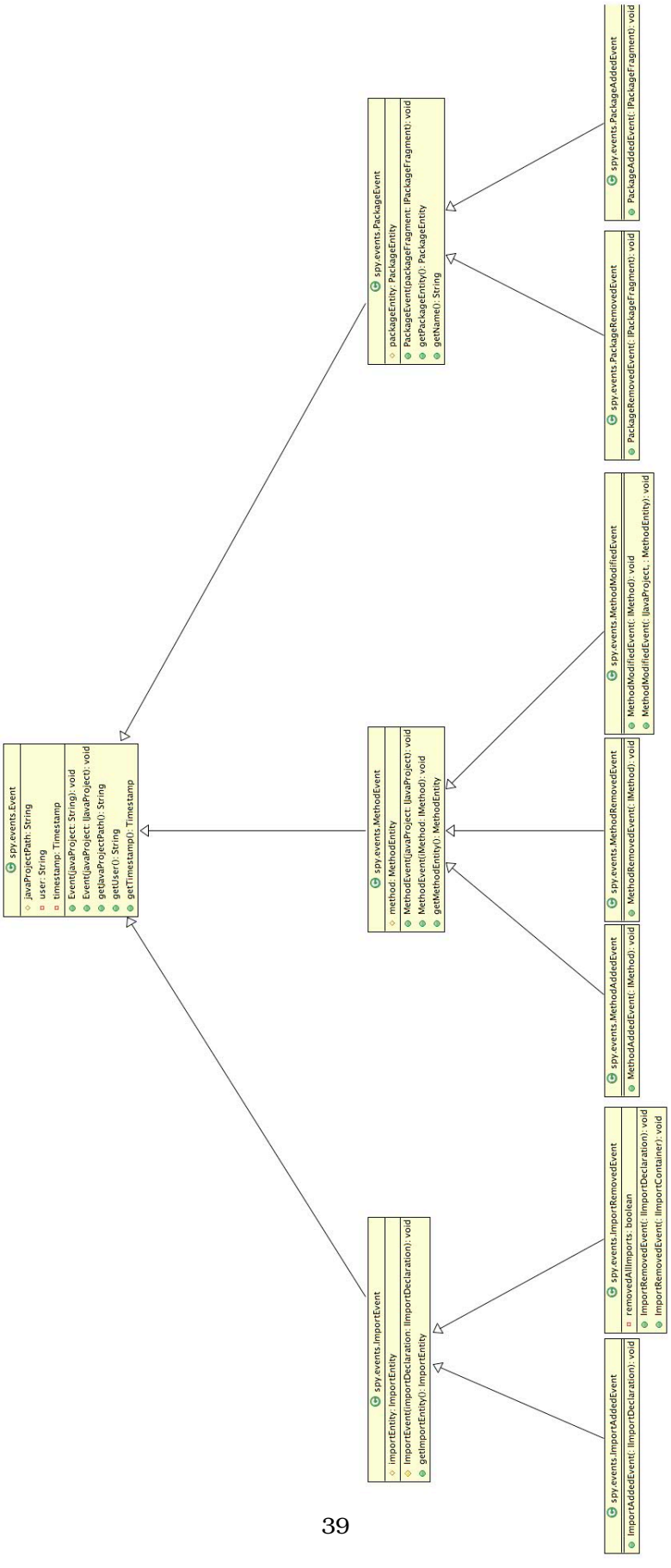


Figure A.2: Events (2/2)

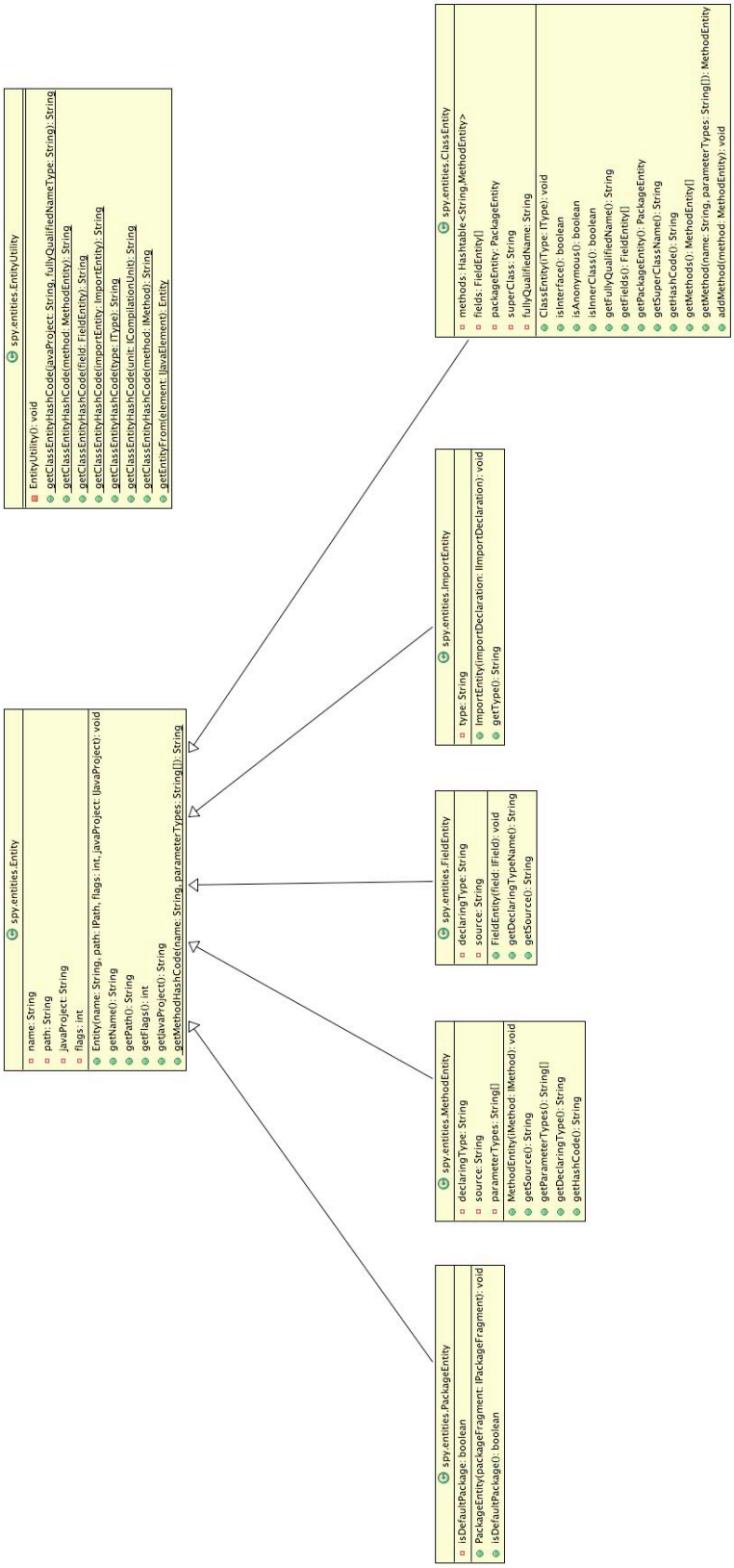


Figure A.3: Entities

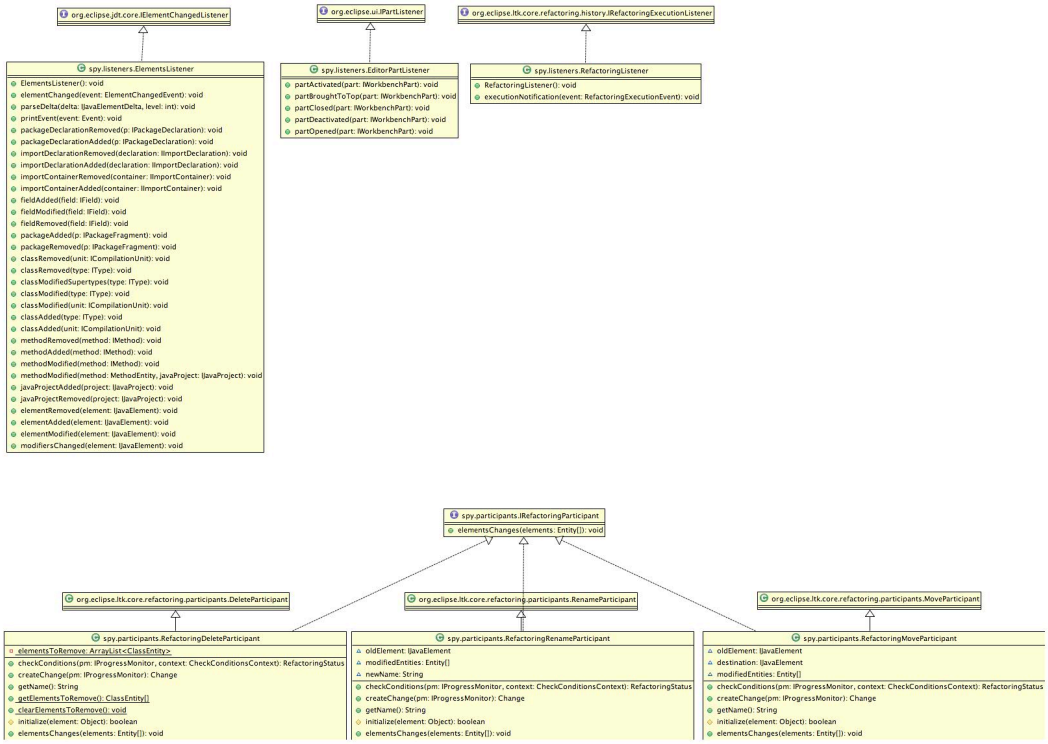


Figure A.4: Listeners and Participants

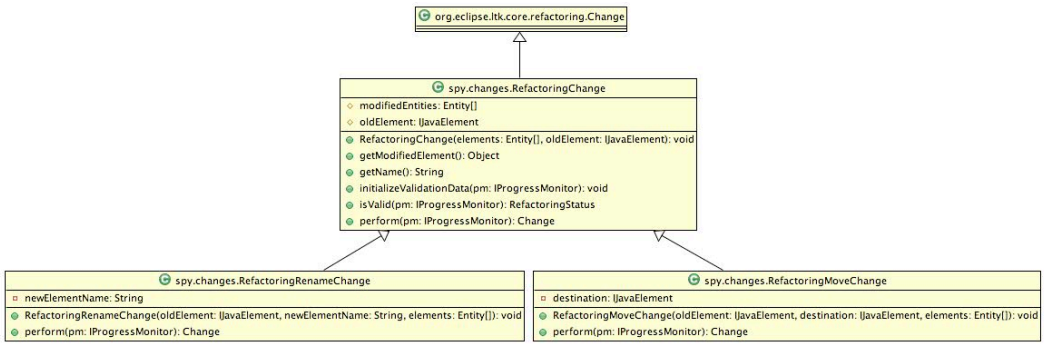


Figure A.5: Refactoring Changes

List of Figures

List of Tables

Bibliography

[Ben93] K. Bennett. An overview of maintenance and reverse engineering. pages 13–34, 1993.

[Bro75] Frederick P. Brooks. *The Mythical Man-Month*. Addison Wesley, Reading, Mass., 1975.

[CR04] Eric Clayberg and Dan Rubel. *Eclipse: Building Commercial-Quality Plug-Ins*. Pearson Higher Education, 2004.

[DDM+03] Serge Demeyer, Stéphane Ducasse, Kim Mens, Adrian Trifu, and Rajesh Vasa. Report of the ECOOP'03 workshop on object-oriented reengineering, 2003.

[DGF04] Stéphane Ducasse, Tudor Gîrba, and Jean-Marie Favre. Modeling software evolution by treating history as a first class entity. In *Proceedings Workshop on Software Evolution Through Transformation (SETra 2004)*, pages 75–86, Amsterdam, 2004. Elsevier.

[Fow99] Martin Fowler. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley, 1999.

[GD06] Tudor Gîrba and Stéphane Ducasse. Modeling history to analyze software evolution. *Journal of Software Maintenance: Research and Practice (JSME)*, 18:207–236, 2006.

[LB85] Manny Lehman and Les Belady. *Program Evolution: Processes of Software Change*. London Academic Press, London, 1985.

[LS80] Bennett Lientz and Burton Swanson. *Software Maintenance Management*. Addison Wesley, Boston, MA, 1980.

[RL05] Romain Robbes and Michele Lanza. Versioning systems for evolution research. In *Proceedings of IWPSE 2005 (8th International Workshop on Principles of Software Evolution)*, pages 155–164. IEEE Computer Society, 2005.

[RL06] Romain Robbes and Michele Lanza. Change-based software evolution. In *Proceedings of EVOL 2006 (1st International ERCIM Workshop on Challenges in Software Evolution)*, pages 159–164, 2006.

[YW03] Hongji Yang and Martin Ward. *Successful Evolution of Software Systems*. Artech House, Inc., Norwood, MA, USA, 2003.