Maintaining private views in Java

by

Paran Haslett

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Master of Science
in Computer Science.

Victoria University of Wellington
2014

Abstract

When developers collaborate on a project there are times when the code diverges. When this happens the code may need to be refactored to best suit the changes before they are applied. In these situations it would be valuable to have a *private view*. This view would be functionally equivalent to other views and be able to present the code in a different form. It enables a developer to refactor or change the code to their tastes, with minimal impact on other developers. Changes in the order of methods and the addition of comments currently impact other developers even if there is no change in how the code works. The Refactor Categories Tool has been written to detect where Java source code has been moved within a file or comments have been added, removed or edited. This indicates that it would be useful for version control systems to differentiate between changes to a program that also change the behaviour and those that do not.

Acknowledgments

I would like to acknowledge the invaluable help of both my supervisors David Pearce and Lindsay Groves and also my wife Hye Eun Park who has been a great support during this thesis.

Contents

Chapter 1

Introduction

According to Bertino [5] *version control systems* provide a way of allowing multiple developers to collaborate. When multiple developers work on the same portion of source code there is a risk that they have conflicting changes. One way of managing these conflicting changes is by ensuring only one person can edit a file at a time. This locking mechanism was recommended by Tichy [23] for the RCS version control system. Of course, the problem here is that one person can stop others from being able to edit the file.

An alternative approach is to allow multiple changes to a file and to automatically resolve most of them in a process called a *merge*. The merge process compares the changes made in one version with the changes made on the other version. If the merge process determines that changes can coexist, it creates a merged file that contains all the changes. The changes that cannot be automatically merged are known as *merge conflicts*. The merge conflicts need to be manually checked and edited to form a merged file with the correct changes.

Internally the merge process needs to determine what changes have happened to both of the revisions being compared. In Figure 1.1 there are two revisions that are derived from a common ancestor. It is possible to determine what has been deleted, inserted or changed by comparing

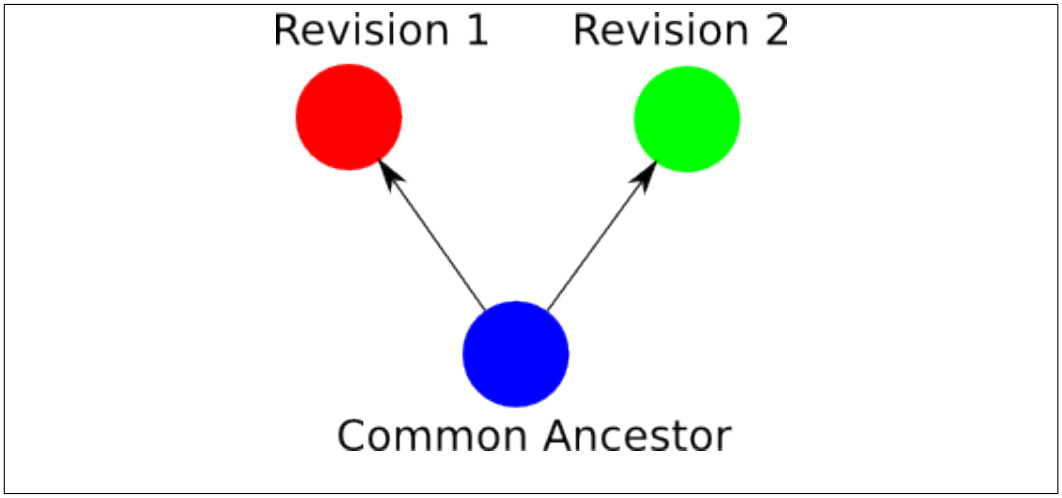

Figure 1.1: A file that has two different revisions

each of the revisions against the common ancestor. This is often done as a linear comparison of the source code for each revision. This works very well provided there has been no change in the order or structure of the file. However, if there has been a change where a block of source code has been moved from one place to another a linear comparison instead determines that two changes have occurred. This is equivalent to deleting a block of source code from the common ancestor and inserting that source code elsewhere. It is possible that this change is not important to other programmers as the program behaves in the same manner even when source code is in a different order. An example of this is if a Java programmer changes the order of methods within a program. The program will behave in the same way as changing the order of methods does not change any functionality, however the source code is now different. The swapping of the order of the method is still counted as being two different changes even though the program behaves in the same manner as it did before the change took place.

Without any further analysis this change is recorded in the merged file even if the reordering was a personal preference for the programmer. Although there has been no functional change the version control system

will treat the relocation of blocks of source code exactly like a change in functionality. Whenever a programmer attempts to update their code to incorporate any change in functionality, the change to the order of methods is also made to their code. If a programmer is already familiar with the old structure of code and expects the code to remain relatively consistent the swapping of methods could be disconcerting.

Another issue that non-functional changes raise is an increase in the number of *merge conflicts*. This could occur when two views have had a small amount of refactoring. Since in both views the behaviour of the program has not changed it is possible that the merge conflict occurs about something trivial. An example of this could be different formatting or ordering of methods. For an ordinary text merge these changes to the structure of the source code require manual intervention. These issues highlight the need to develop smarter ways to merge.

It is becoming more important to have smarter merges because of the scale of many software projects and the number of developers working on them. Large online repositories, like GitHub, contain many open source projects. It is possible for these projects to make source code available to many developers at a time. This means it is possible for two developers to work on the same project while having little personal contact. Care needs to be taken when their individual work is combined. Preferably most of the problems with merging their work should be automatically resolved. However, there still could be instances where either or both developers will have to figure out how the code should interact. Having better automatic merges reduces the risk that time will be spent manually figuring out how different changes should be combined.

This thesis introduces the concept of maintaining multiple private views which can be ordered differently but have functionally equivalent source code. The purpose of these views is to reduce the number of changes introduced during a merge. It also explores a way of allowing a version control system to detect when there is a change in the source code but not

a functional change in a program. Examples of this are if items have been reordered, if comments have been inserted or there have been changes to the formatting. We developed the Refactor Categories Tool for the purpose of identifying these changes. We report on the design and implementation of this tool. The Refactor Categories Tool is then used in an experiment evaluating a range of real-world benchmarks.

1.1 Overview

This thesis is organised in the following manner: In Chapter 2 we go over the background of version control, the longest common subsequence problem and refactoring as well as look at JDime, an existing tool for merging refactored views. In Chapter 3 we discuss our concept of private views and what they could mean. In Chapter 4 we examine the Refactor Categories Tool, a precursor to developing private views. This will allow us to evaluate if additional merge operations are possible. In Chapter 5 we evaluate the results produced from the Refactor Categories Tool and determine what this could mean for our concept of private views. In Chapter 6 we will conclude and discuss future work.

Chapter 2

Background

As this thesis is about maintaining private views within a version control system what follows is some background concerning version control systems and how they determine if a change has occurred in source code. We will also cover refactoring before looking more closely at JDime and other tools that attempt to reduce merge conflicts caused by refactoring or reformatting.

2.1 Version control systems

Version control systems are a way of managing different revisions (or versions). Version control systems can be used to keep revisions of files that are in any format. Most commonly they are used for maintaining source code written for a plain text programming language (e.g. Java, C, etc.). There are a number of reasons why we might want to use a version control system. It can be used to refer to previous revisions, to maintain a revision that has an experimental feature, to associate additional documentation about a feature or to collaborate with multiple developers who are on the same programming project:

Revisit revisions using tagging. A version control system can be used by

a single person to manage different revisions of their program. A previous revision can always be revisited at a later date and changed. If there is something significant about a particular revision it can be labelled with a *tag*. A tag assigns a name to all the files in the revision you are interested in so that you can more easily revisit the code at a certain point. This is helpful if a software package has a number of released versions. If you need to go back and revisit a particular release it becomes a lot easier if you have tagged the code at that point with the release name or other identification.

Use branching for experimental features. It is also possible to maintain multiple revisions of all the files for a software project. This is useful if there is an experimental feature which you want to explore but want to maintain the original as a separate project. As shown in Figure 2.1 a version control system can keep these multiple interests separate by putting them on different *branches*. It is still possible to easily switch between the different branches depending on which project you want to make changes to. A good use of this feature is if you have a software project that you have written on behalf of two different companies but each of them would like their own unique customisations on top of the base product. By making two copies of the base product and having a record of when it was divided the branches can later be recombined to include some or all of the features that have been introduced.

Attach documentation to a feature. Another useful feature of version control is the ability to record meta information beside changes to a set of files. The reason this is useful is that you can specify what the change was for. It is possible to associate a change that has occurred over multiple files as being for the same reason. Most version control systems allow a message to be written when documents are checked in. In some version control systems this message is required. The rea-

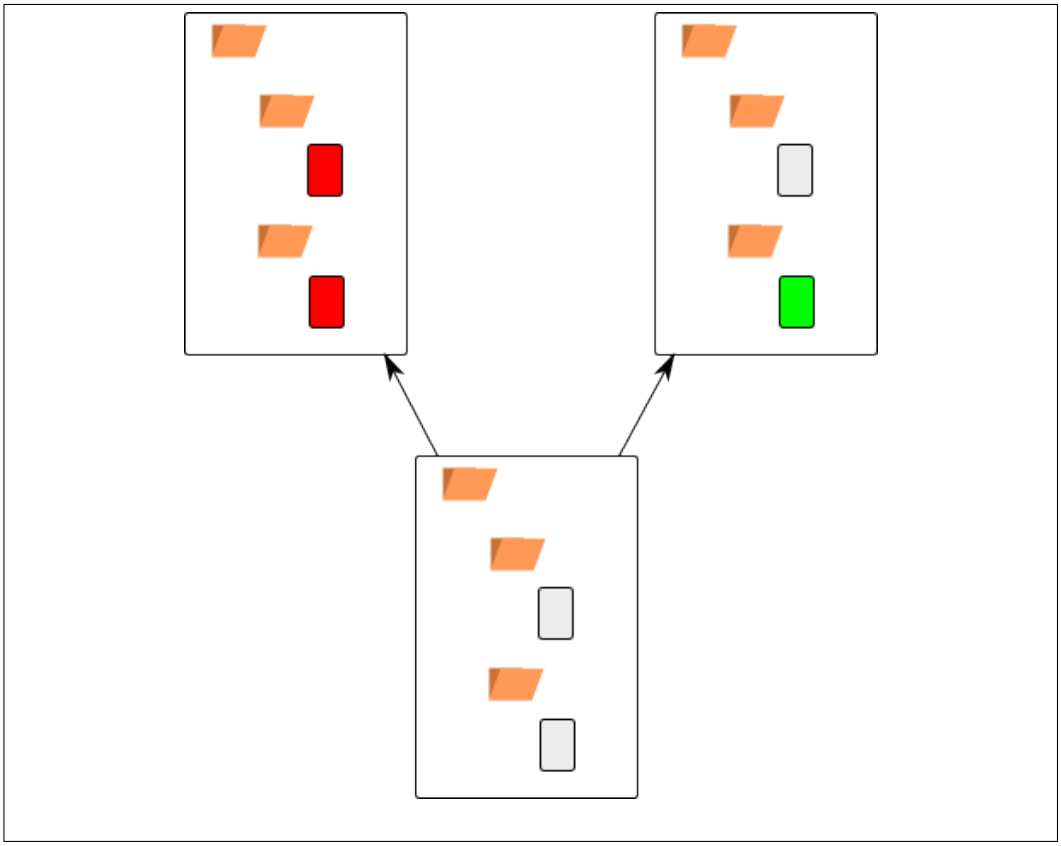

Figure 2.1: A project that has been split into two branches. Both the red files have been changed for the left branch whilst only the green one has changed in the right branch.

son this is useful is for when queries are made about what a certain change to a document was for. Since there is a message beside all the documents about the reason for a particular change it becomes easier to figure out the reason for the individual change we are interested in.

When used on source code in tandem with a *bug tracking system* the message can contain the identification number for the bug being fixed or feature being added. This means that anybody who is examining the revision to see the reasoning for the change has access to a lot more information via the bug tracking system. An example

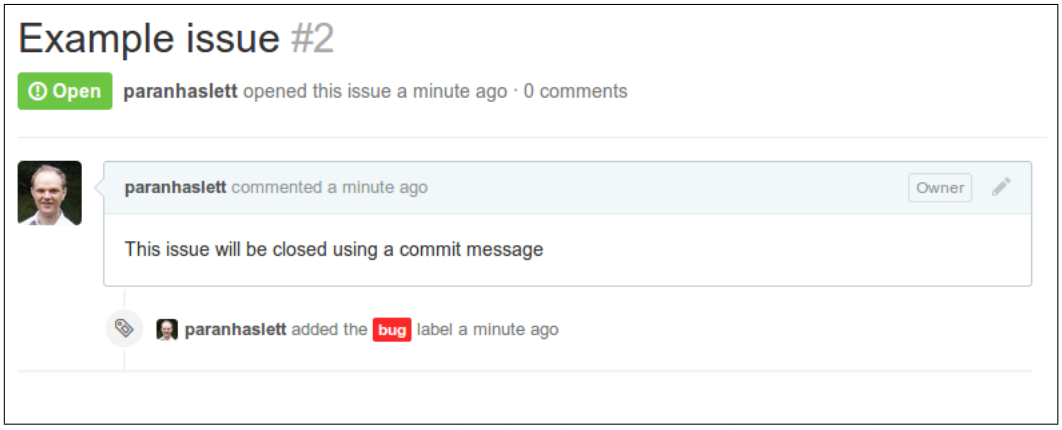

Figure 2.2: A issue that has been created in GitHub

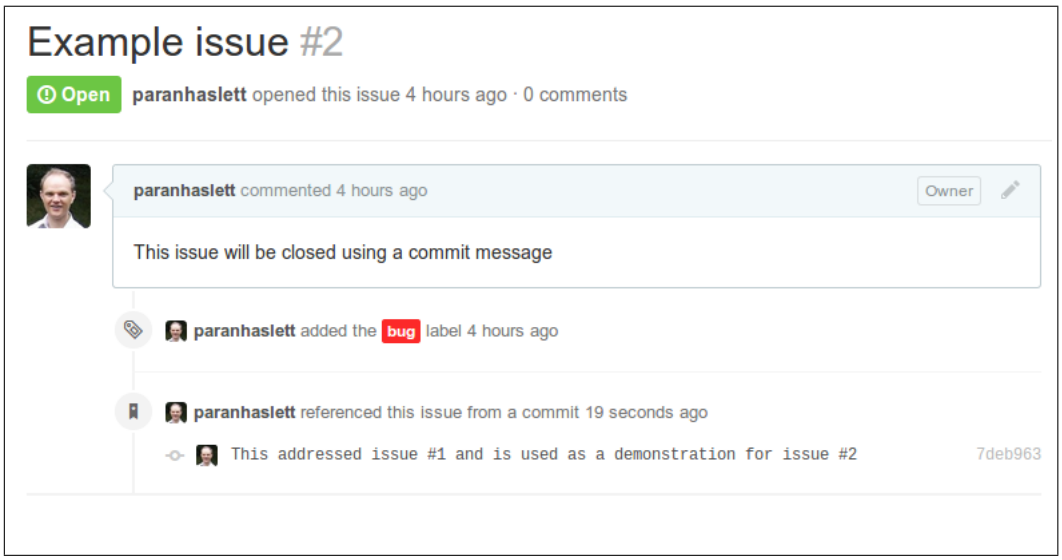

Figure 2.3: The issue has been updated using a commit message

of this is the *issue tracker* that is built in to GitHub, an online version control system. In GitHub an issue (e.g. a bug report or feature request) can be created as shown in Figure 2.2. The issue tracker is linked with the messages that you need to write when you check in files. If you include a hash sign followed by an issues' identification number in the message you check in then GitHub updates that issue, as shown in Figure 2.3.

Collaborate with multiple developers. Version control systems make it possible to have individual revisions that contain each person's changes. The version control system then manages the way these changes are merged into a composite product. Bertino [5] describes the ability to merge the work of multiple people as being a powerful collaborative tool. This is because a version control system allows multiple people with different ideas to collaborate on the same document. In some circumstances it allows them to work on the document at the same time. This feature seems similar to the concept of having a private view that we are exploring in this thesis. The difference is that a version control system has no awareness about the format of the documents. This means it is harder for it to evaluate the difference between features of the document that will be helpful to collaboration and features that express the individual tastes of each of the collaborators.

2.1.1 Dealing with conflicts

When people work on the same software project there is a need to interact with each other. If they require the same source code then there is competition to access that file for each of them to successfully do their work. There is the risk that they will attempt to change the same block of source code at the same time. If different changes are made to the same block of source code then they have a *merge conflict* and need to figure out how to combine the changes manually. There are a few ways of dealing with these conflicts.

Locking

One approach is to require that a file is only able to be used by one person at a time and that anyone else has to wait. This method avoids the need to resolve any conflicting changes. The advantage of locking a file like this is

that we can be certain about the contents of the file at any given time. This is how one of the original version control systems, RCS ensured that the document stayed consistent. Tichy [23] has explained why he considers locking in a version control system to be a good idea in the design for RCS. The disadvantage is if one person retains the document for extended periods of time, it cannot be changed by anybody else. Furthermore the resulting document may be barely recognisable as the original if extensive work is done on it. However, if the two parties change distinctly different parts of the file, or both independently make exactly the same changes this restriction is unnecessary.

Smaller structured units

Another way to reduce conflicts is to split the programming code into smaller units. The advantage of this is that if you are using locking you minimise the risk that two people need access to the same unit of source code. Consider two people working on the same project made up of a number of files. Instead of one person locking a file at a time that person could be allowed to only lock the functions they are changing. As those functions are smaller than the file there is likely to be less changes and they are likely to unlock them sooner.

Merging documents

Finally we could allow both parties to change the document and try to figure out what the problems are afterwards. This resolution of anything that remains a problem is known as resolving a *merge conflict*. This occurs whenever the computer cannot automatically process a merge. The merge then needs to be sorted out manually. Merge conflicts are more likely to occur if there is a dramatic change such as refactoring.

If not regularly merged it is possible for the source code to diverge greatly and it becomes harder and harder to reconcile. According to Bertino

it is possible to keep a smaller more easily deployed repository by excluding files that can be generated. [5]. Although Bertino refers to unnecessary files this premise may also be applicable for the smaller blocks of code we are interested in. This suggests that maintaining a record about what is relevant and what is irrelevant may have some benefit (e.g. the non-functional changes).

Manual Merging. If you have two files you want to merge but no common starting point for them both you will need to manually merge any differences. The computer has no record about what the original file looked like so it cannot determine which changes were intended. Features that they have in common will not need to be affected. However, a decision needs to be made about any differences by the developers responsible for those changes.

Automatic Merging. An automatic merge is possible if three revisions are available: the original revision, the revision containing changes we have made and the revision containing changes made by other developers. By comparing the differences between our revision and the original one it is possible to determine which changes we have made. If we compare the changes made by other developers to the original, the changes that they have made can be determined. If we attempt to update our code by merging other people's changes and a change only occurs in our code then that change is retained. If we attempt to update our code by merging other people's changes and a change only occurs in the other persons code the the change is inserted into our code. A merge conflict can however still occur when a change is made in the same area of code in both revisions. The merge conflicts need to be dealt with manually.

2.1.2 Types of version controls systems

Centralised version control. In a centralised version control system all the changes are made to one location. This is called a centralised repository. Having a centralised repository means that only one place needs to be checked in order to access the most up-to-date and agreed upon source code. The need to be connected to a central system allows multiple developers the ability to work on the same source code but often has a large overhead. Some centralised systems require a specialist to be involved just to look after the server and ensure that merges were done correctly. However according to Chacon [7] this single point of management has some advantages as it is possible to manage what developers had access to. According to Chacon [7] and Bertino [5] the main flaw with centralised version control systems is that they have a single point of failure. If anything goes wrong with the server you could lose all your work.

Distributed version control. According to Chacon [7] and Bertino [5] this is like having a complete copy of a repository present on every computer that has access to that project

One advantage of having a complete copy of a project from the repository are that it eliminates the single point of failure that centralised version control systems have. It also makes it possible to make changes to the program remotely without being connected to a central server. At a later time you are able to merge you changes with other people's work. You are also able to select changes others have made and incorporate them into your personal copy.

Online version control systems. Whilst is is possible for a measure of collaboration just by using a version control system on its own, it requires that you have some method of obtaining the separate branches on one machine before they can be merged. One way of doing this

within a company is to set up a server for the version control system. This might be suitable for projects that are closed source and have a select group of people who work on the source code. For larger projects that have programmers in different parts of the world a publicly accessible version control system that is on the web may be a better solution. Loeliger [17] shows how it is possible to access and use a web based version control system to achieve this. A good example of a web based version control system is GitHub. GitHub provides a way for many developers in different parts of the world to change source code for an open source project. It is possible that the developers for a particular project have not even met in real life or even know about each other.

As an example we could look at the JGit project in GitHub. JGit is a pure Java implementation of the Git version control system. Figure 2.4 shows part of the GitHub page for JGit highlighting the amount of activity that there has been on the JGit project. With 74 contributors supplying 3245 changes means that the potential to have conflicting changes must be high. Having 20 separate branches may indicate that merges often need to happen. This one example could indicate the need for good merge tools that can reconcile conflicts even if the developers have little contact with each other.

2.2 Longest Common Subsequence

The longest common subsequence problem is relevant to this thesis as it concerns comparing code to determine similarities and differences. The similarities and differences can then be used to automatically merge code in a version control system.

A simple definition of the longest common subsequence problem is attempting to find the maximum number of common items in two strings when the strings are examined from left to right. A subsequence does not

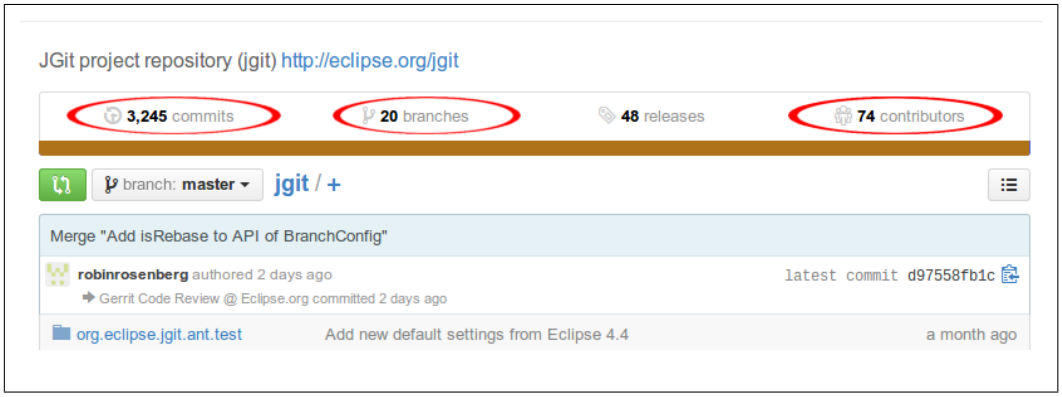

Figure 2.4: The overview page of the Jgit project in GitHub. This shows that there are 74 contributors to the source code and 3245 individual changes over 20 branches.

need to be in a single contiguous block however. Algorithms that solve the longest common subsequence can determine the differences between two lists by working out what is the same.

2.2.1 Example

An example of finding the longest common subsequence is as follows. Imagine we have two similar sets of Java source code that we want to compare with each other. We would like to know what is the same and what is different. A longest common subsequence for the source would contain a list of all the lines that are the same and in the same order.

The first listing is as follows:

```
1  public class SampleLCS {
2    public static double area(double radius){
3      return Math.PI * square(radius);
4    }
5
6    public static void main(String[] args){
7      System.out.println(area(3));
8    }
9
10   public static double square(double num){
11     return num * num;
12   }
13 }
```

In the second listing the order of a number of methods has changed but the way the code works has not been changed.

```
1  public class SampleLCS {
2    public static void main(String[] args){
3      System.out.println(area(3));
4    }
5
6    public static double square(double num){
7      return num * num;
8    }
9
10   public static double area(double radius){
11     return Math.PI * square(radius);
12   }
13 }
```

A listing containing only the common lines in the same order between both listings follows. Since this is one of the longest listings possible it is known as the longest common subsequence.

```
1 public class SampleLCS {
2   public static double area(double radius){
3     return Math.PI * square(radius);
4   }
5
6 }
```

It is possible to have more than one longest common subsequence if there are multiple listings of common lines that have the same number of lines in common and have the maximum number of lines that match. For instance the following listing is also a longest common subsequence of the above example.

```
1 public class SampleLCS {
2   public static void main(String[] args){
3     System.out.println(area(3));
4   }
5
6 }
```

As there are possibly multiple longest common subsequences identifying the longest common subsequence that is going to be most useful becomes difficult.

2.2.2 Methods of calculating LCS

According to Arslan [4] there are many algorithms that solve longest common subsequence problem. As is it possible for there to be multiple correct solutions to a LCS problem a reason for having a different algorithm may be to find the LCS that make the most intuitive sense. The algorithms used

in JGit (an open source implementation of Git in Java) for example are the the Myers, Patience and Histogram algorithms. JGit predominately uses the Histogram algorithm with a fall-back of using the Myers algorithm if it gets too computationally expensive. There is also the option of using the Patience algorithm however this will produce similar results to the Histogram algorithm which has been derived from it.

2.2.3 Myers

The Myers algorithm was discovered by Eugene Myers [19] who claimed that finding the minimal differences between any two documents was the equivalent to finding the shortest path in an *edit graph*. If we have the two sequences that we wish to compare we can use them to create an edit graph. For example to discover that the longest common subsequence for the two strings ”ABBCABAB” and ”BACCABBAC” we would create the edit graph shown in Figure 2.5

Note that the diagonals are on the graph at every point where the letters coincide. If we draw a path from the top corner to the bottom corner by following either the edges of the boxes or the diagonals we can determine a subsequence of both of the strings. Each time the path crosses a diagonal is an instance where both strings have a common letter. If the path uses the fewest number of box edges possible then it contains a lowest common subsequence. Figure 2.6 shows a path for a longest common subsequence. There may be more than one longest common subsequence. For our example another longest common subsequence of ”BCABB” is also equally valid.

2.2.4 Patience

The Patience algorithm instead of figuring out the longest common subsequence directly uses the longest increasing subsequence. The example used by Aldous [1] to explain how the Patience algorithm finds the longest

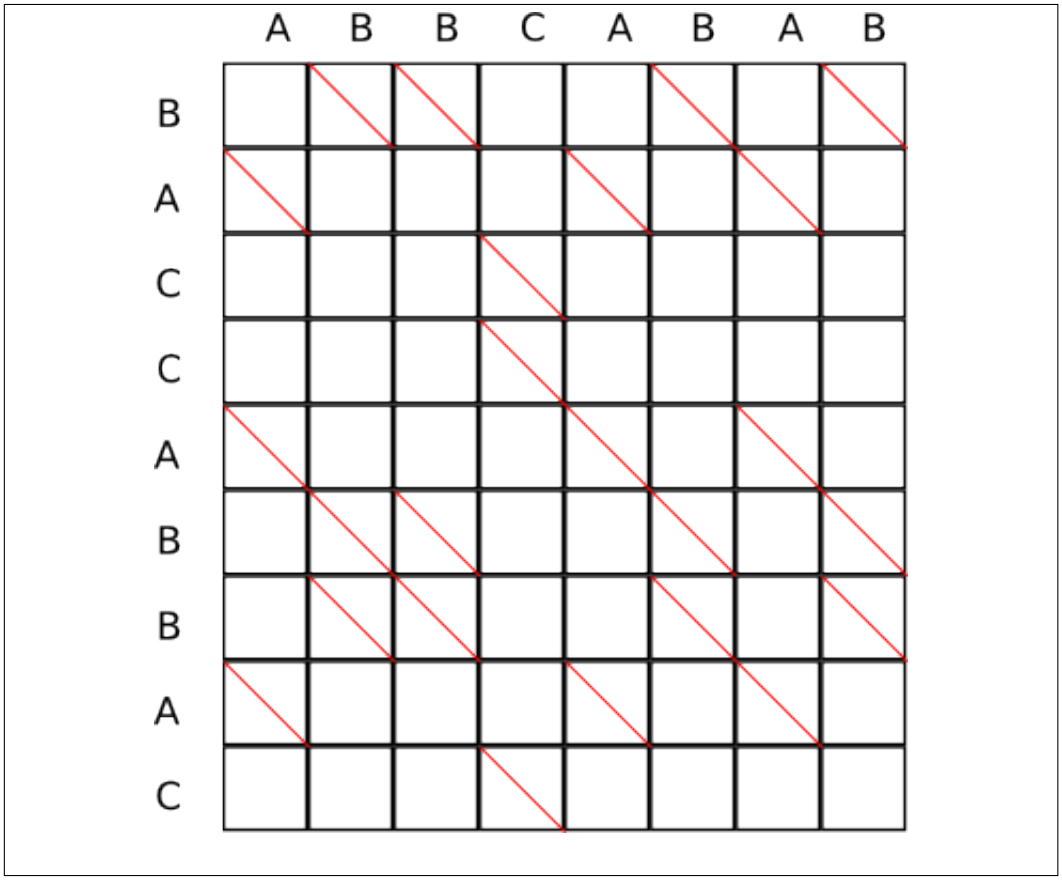

Figure 2.5: An edit graph for two strings, "ABBCABAB" and "BACCABBAC"

increasing subsequence is similar to a single player card game. The aim of the game is to create the minimal number of piles of cards in a row. A higher card may not be placed on a pile with a lower one in it. Cards need to be placed on the leftmost valid pile. A new pile needs to be created at the end of the row for any cards that cannot be placed on any existing pile. This game discovers the longest increasing subsequence of the cards when they were shuffled before the game is played. The cards in the pile to the immediate left of each card when it is played are all possible elements that could come before it in a longest increasing subsequence. By taking notes of the card on top of the pile to the immediate left whenever

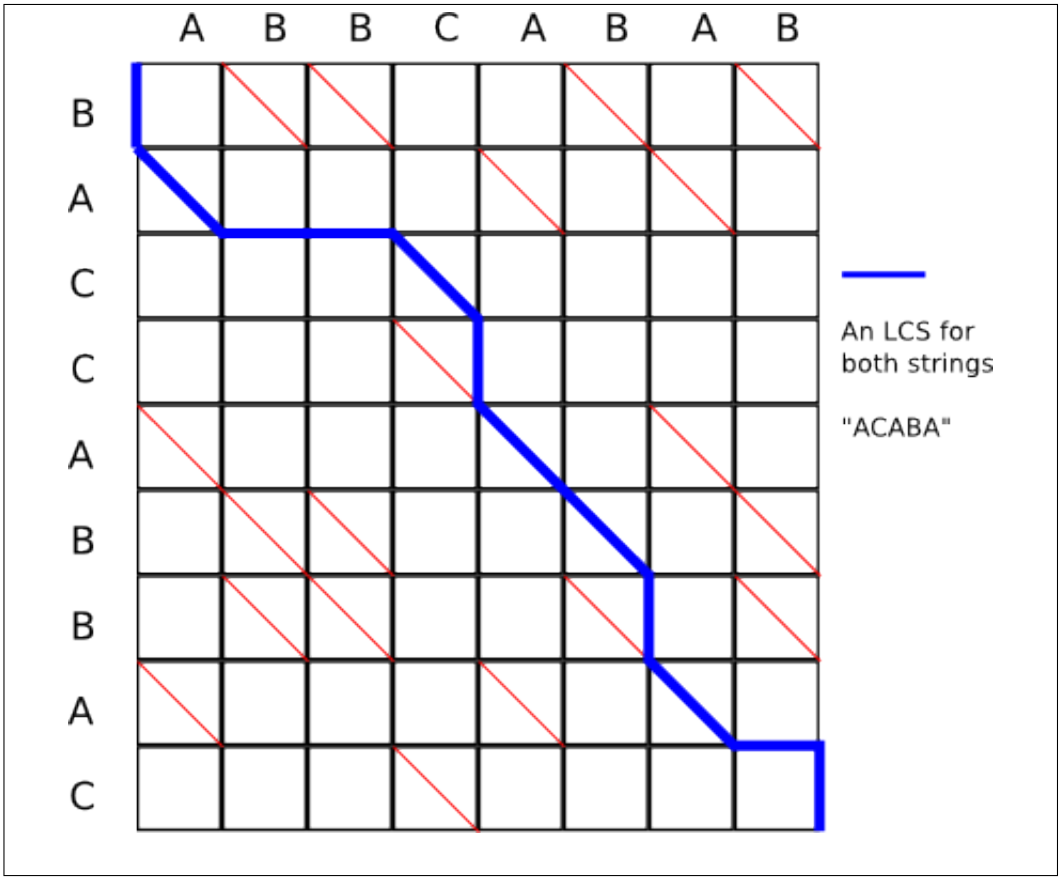

Figure 2.6: An edit graph for two strings, "ABBCABAB" and "BACCABBAC", showing the path for the longest common subsequence of "ACABA"

a card is played a longest subsequence can be calculated.

When the Patience algorithm is used on source code line numbers are compared (rather than cards). The Patience algorithm is not applied to every line in the source code, but is only applied to *markers*. A marker is when there are lines of code that match in both revisions and appear only once. The longest common subsequence is computed just for those markers. The portions of code between those markers in the longest common subsequence are then computed recursively.

As Bram Cohan [9] has pointed out in his blog there are instances

where a traditional LCS algorithm can return results that although correct are not as helpful as they could be. This is especially true when there are multiple possible longest common subsequences. The patience algorithm initially ignores lines that appear multiple times and focuses first on solving the longest common subsequence problem for the markers, which only appear once. By doing this it has a clearer picture about where to place the lines that occur multiple times.

2.2.5 Histogram

The patience algorithm works well when there are matches that only occur once in both revisions, but has difficulty in determining what to do if lines appear more than once. It is possible to fall back on a Myers algorithm for the segments where multiple matches occur, however it is possible for Myers to produce unhelpful results. The Histogram algorithm attempts to overcome this by also detecting lines that occur in both revisions a small number of times. The algorithm is solely used in JGit at the moment and is a derivative of Bram Cohens patience algorithm. More details about this algorithm can be found in the Javadoc for HistogramDiff in the JGit source code [22].

2.2.6 How LCS is used in differencing tools

Differencing tools (often shortened to ”diff tools”) are programs that compare the contents of two files and show the similarities and differences. In many diff tools a hash code is assigned to each line of the files to speed up the differencing process. This means that the differencing tool can work much faster as it does not need to compare each character in the line but can compare hash codes instead. However the granularity of what is compared is more coarse as it shows complete line differences rather than word or character differences. In the source code for many programming languages the white space is not relevant so many diff tools have the op-

tion of ignoring the white space and only comparing the code. This has an impact on the hash codes for each line as the hash code needs to be generated just from the text without including white spaces.

Additionally with some diff tools it is possible to use regular expressions to ignore program features such as comments when doing a diff. The reason this is important is if it is possible to exclude changes that have no affect on behaviour from a diff then it is possible to also exclude them from a merge. If we exclude them from a merge we could have fewer conflicts. An example of something that is already excluded in JGit is white-space.

2.2.7 The problem with LCS

From the perspective of this thesis there is still a problem with longest common subsequence. It does not notice changes of order in a document. For the example in section 2.2.1 two methods have swapped positions. The program still behaves in the same manner when it is run. It is unnecessary to make any changes to this code in order to get them to behave the same way. Diff tools that solely use the longest common subsequence do not take different ordered items into account even if they can be considered equivalent.

2.3 Refactoring

A common concern with coding is the need to periodically refactor the code. When we use the word refactored here we refer to the definition of refactoring presented by Murphy-Hill [18] who claims that refactoring simply changes the structure of the code but not the behaviour. Refactoring simply reorganises the source code so that it is easier to read and add changes. According to Fowler et al. the main time for refactoring is when new functionality is added [13]. Similarly according to Kerievsky some of the motivations for refactoring include adding more code and understand-

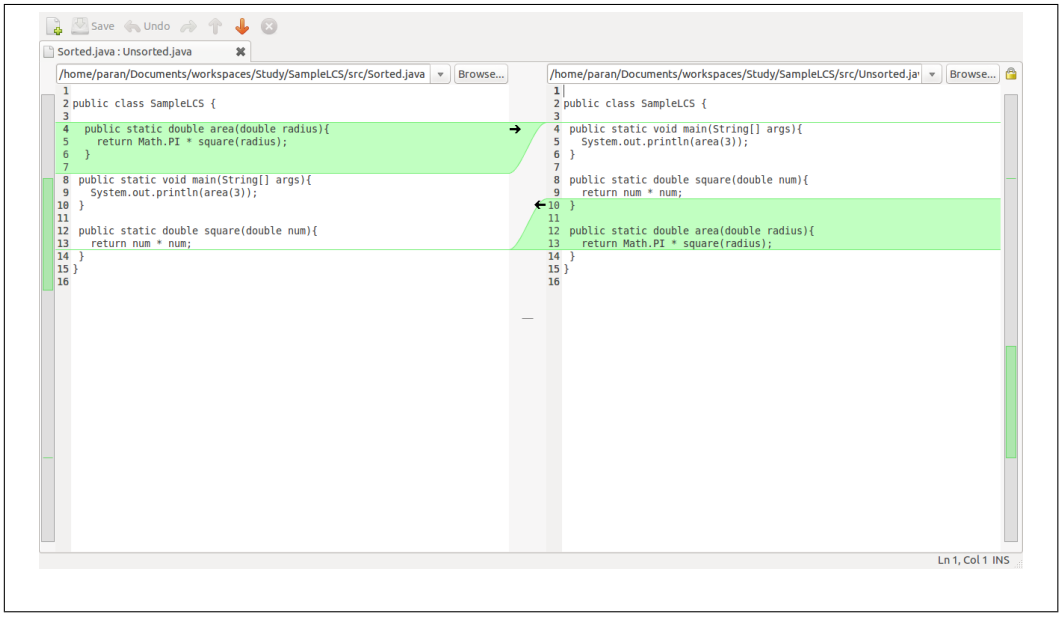

Figure 2.7: A graphical diff tool showing differences with two equivalent blocks of source code

ing existing code [15]. As adding more functionality is one of the motivations for refactoring let us consider what happens in a multi-developer environment. Two developers could have different views on what is considered an appropriate refactoring. This is especially true if they need to add different functionality from each other. We will now demonstrate this with the following two code examples:

```
1 public TempConv() {
2   Scanner keyboard = new Scanner(System.in);
3   System.out.println("Enter the temperature in Celsius");
4   int celsius = keyboard.nextInt();
5   System.out.println("Degrees Fahrenheit is approx "
6     + (celsius * 2 + 30) );
7   keyboard.close();
8 }
```

Refactoring this code depends on what functionality you need to add.

One developer may recognize that conversion from Celsius may be used several times throughout the code and so extract the calculations as a separate method as follows:

```
1  public TempConv() {
2    Scanner keyboard = new Scanner(System.in);
3    System.out.println("Enter␣the␣temperature␣in␣Celsius");
4    int celsius = keyboard.nextInt();
5    System.out.println("Degrees␣Fahrenheit␣is␣approx␣"
6      + celsiusToFahrenheit(celsius));
7    keyboard.close();
8  }
9
10 public int celsiusToFahrenheit(int celsius){
11   return celsius * 2 + 30;
12 }
```

This change, in spite of producing the same output as the first, provides a number of advantages. Firstly if other programs need to convert from Celsius to Fahrenheit the new method can easily be reused. Secondly since the calculation is a crude estimation it becomes a lot clearer where the code needs to be changed to improve the formula. The ability to add a method that clearly indicates that the calculation is from Celsius to Fahrenheit helps with the readability of the code. There are also disadvantages to doing this refactoring. If we do not care about conversion between Celsius and Fahrenheit the refactoring simply adds to the amount of code we need to examine before understanding what the code does. An alternative way of refactoring is as follows:

```
1 public TempConv(){
2   Scanner keyboard = new Scanner(System.in);
3   System.out.println("Enter␣the␣temperature␣in␣Celsius");
4   int celsius = keyboard.nextInt();
5   int celsiusToFahrenheit = celsius *2 + 30;
6   System.out.println("Degrees␣Fahrenheit␣is␣approx␣"
7     + celsiusToFahrenheit);
8   keyboard.close();
9 }
```

While this again expresses the same functionality as the code above it has not created a new method to do so. This has some of the same advantages. It separates and identifies the formula to convert between Celsius and Fahrenheit. It also uses less code to express this separation than forming a new method. It does not expose the conversion formula outside this method to be used by other calculations however.

As the value of a particular refactoring appears to depend on what is trying to be achieved it is very hard to claim that one refactoring is better than another. Rather, it depends on the wider context of the intention for the refactoring, in this case the level of access required for the approximation to convert Celsius to Fahrenheit.

Although this was a simple example it is easy to imagine a case where a much larger refactoring process is undertaken. In such circumstances a merge becomes difficult.

2.4 JDime

JDime is a tool written by Apel and Leßenich [2] [3] [16] to study how to get a balance between fast text-based merges and slow but more accurate *semantic merge*. A semantic merge is done by parsing the class files into AST and evaluating the AST. JDime is designed to merge two sets of code

even if both of them have undergone refactoring. In order to increase performance, only if there is a conflict in a text based merge does any of the more expensive semantic merge take place.

2.4.1 How JDime works

Before doing any calculations, JDime runs a regular text merge over the source code. If the regular text merge has conflicts then JDime parses the file into an abstract syntax tree (AST). JDime uses the AST to determine if sections of the source code need to be in a particular order or could be in any order. What then happens depends on if order is required in the section of code JDime is examining.

2.4.2 Investigating JDime

We performed a small experiment to investigate JDime as a tool for automatically merging code which has been reordered. As JDime performs a type of automatic merge it requires three different revisions. JDime requires a revision that has changes that we want included. This is commonly called the right revision however I will call this the merger revision as the changes in it are meant to be merged. JDime also requires a revision that we want to merge into. This is commonly called the left revision, however I will refer to this as being the mergee. Finally JDime requires an original revision that both the merger and the mergee are based on. This is commonly called the base revision.

In order to show how JDime performs extra refactoring based merging we need to attempt to try something that would incorrectly cause a conflict in a text based merge. The reason that this is necessary is that if there are no conflicts in a text based merge the refactoring aware portion of JDime will not be run. This saves the overhead of parsing the program into an AST in the event that the initial text merge has no conflicts. One way to get a lot of text conflicts between two pieces of code that are equivalent when

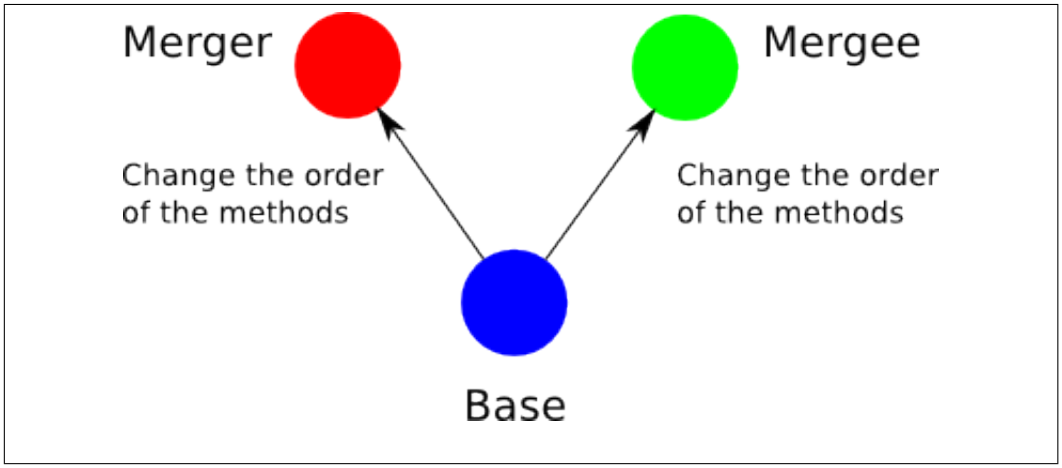

Figure 2.8: The set-up for the test of JDime

they run is to change the order of the methods. Although the methods are in different order the programs are still ”functionally equivalent”. In order to examine how JDime works and test its suitability a test handler was written. The test handler creates all of the directories and files for JDime to process. The methods inside the files are reordered differently for both the merger and the mergee. Figure 2.4.2 demonstrates how the test files were arranged.

Once the test was set up using the test handler JDime was run to process the directories. What we expected to happen was that JDime would reorder the methods to match the order in the mergee. As shown in Figure 2.4.2 when we compared the methods using a graphical merge tool however we found that the order of the methods in the files did not match.

Further investigation revealed that the order of the methods in the final merged code example did not match the order of any of the equivalent input files. In other words JDime normalises its output so that methods are in a particular order, but this does not necessarily reflect the order of any of the original files.

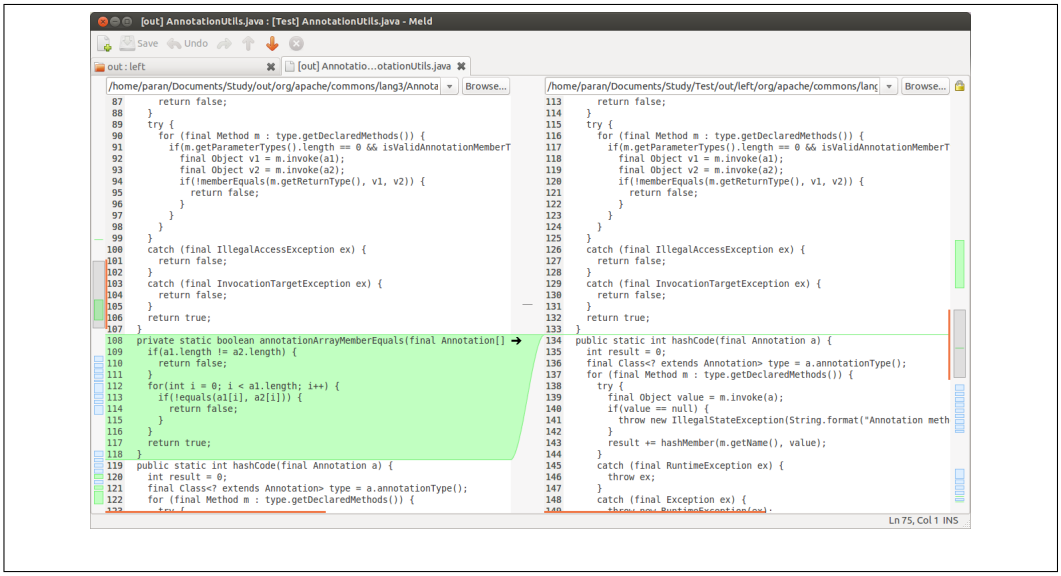

Figure 2.9: Screen-shot of Meld showing a different method order

2.4.3 Reasons why JDime cannot currently be used to create private views

The aim of this thesis is to be able to maintain two views of Java that, although having a different format, function in the same manner. Although JDime seems like it would be able to help achieve those aims there are a few reasons why it cannot be used without changes.

The first issue is that as explained above that the merged code could be in a totally different order to the original file and both of the revisions.

The second issue is that when JDime parses the code into an AST it strips out any comments or white-space placed in the code. Although the comments do not have any functional impact on how the program runs they do have an impact on how the source code is understood. To limit the impact a merge makes on one view comments need to be evaluated as well. In some ways retaining comments or even white-space in the code aids in determining if a section of the code has been copied verbatim from one place to another.

The final concern is that after JDime does the initial comparison of text

and finds conflicts it discards those results. It parses the entire file into an AST and begins analysing it again without knowing which parts differ.

2.5 Other refactoring aware versioning tools

JDime is not the only tool that can be used to address refactor aware version control. Ekman and Asklund [12] introduced a plug-in for eclipse that recorded information about refactoring in to version control so that it was easier to recognise where refactoring took place. They did this in a very similar manner as Apel and Leßenich when they developed JDime. Both make use of an AST to record information about any refactoring that took place. Freese [14] also developed a tool that was very similar written in Object-Z. Despite this interest in refactoring aware merges however the idea of maintaining private views has not been discussed.

Chapter 3

Private views

In a project with multiple developers situations may arise where you need to make a change to the structure of the source code. This becomes a problem if you also want to limit the impact on other developers. Maintaining your own private view of the source code could be valuable in these circumstances. One way of doing this is by making version control systems aware of refactoring. Interference with the structure of the source code in each view could then be kept to a minimum, with only what is necessary merged between views. There is already a significant amount of interest in making diff tools and version control systems refactor aware. Some example of this are as follows:

- MolhadoRef [11] [10] attempts to incorporate refactoring in version control systems.

- Semantic Merge [8] is a series of stand alone diff tools for different languages.

- JDime as already discussed in Section 2.4.

3.1 The problems to address

There are a couple of challenges when collaborating using version control systems. We believe that having private views will help address these issues.

Repeated structure changes for other software developers. Imagine a situation where you are working jointly on a project with other people. Since you want to collaborate on different aspects of the same source code you have set up the project in a merge based version control system. You have checked out your own copy of the code so that you can work on the source code without interfering with any of the changes others are making. You decide that it would be helpful to make a small refactoring. This lightweight refactoring will help you to better understand the source code and could involve some reformatting. You complete your changes and check your code back into the version control system. While you are doing this other people have been working on the code. If you manage to check in your code before anyone else you will not need to merge any of your changes. Anybody who checks in after you however, could have a merge conflict. A few of the merge conflicts that they experience could be because the changes you made directly compete with the changes they have made. Potentially merge conflicts could also occur between the changes they have made and the lightweight refactoring that you have completed.

As shown in Figure 3.1 the difficulty lies in the fact that not only the functionality that you have added is checked in but also the changes brought about by lightweight refactoring. These refactored changes have not changed how the program functions but have simplified and tidied the code to make the addition of your changes easier. These could include any formatting changes, or code restructuring used to create a programming environment to allow you to be more

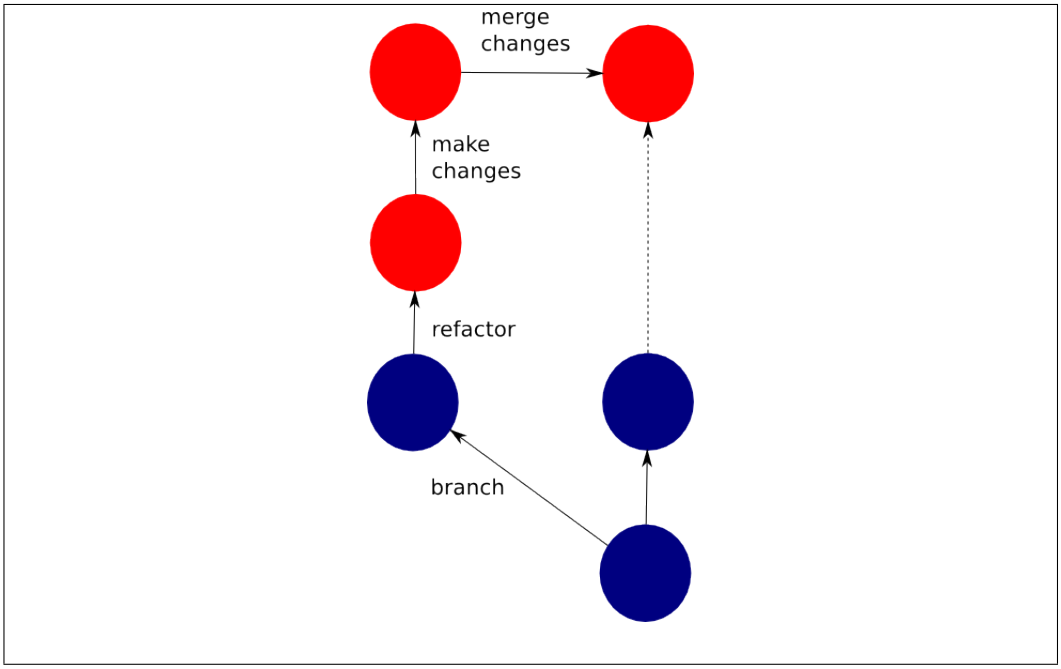

Figure 3.1: Merging changes with refactored code also merges any refactoring

productive. During these occasions you may want to avoid changing other peoples code in such a dramatic fashion.

By checking in your refactoring code you are forcing others to comply with your vision about how the code should be structured. This occurs even though you could have no awareness about what changes to the code others have made or intend to make. Everyone who attempts to check in their code after you will need to merge into a restructured code source that they are unfamiliar with. The potential for merge based bugs and time wasted doing unnecessary merging increases.

Difficulty if there are multiple check-ins. When there is a large change in a separate branch with many development milestones it is desirable to have the ability update your code periodically. By regularly pulling updates from a main branch in the version control system

this is possible. This could be done to ensure that there is not too much divergence between the separate branch and other development projects. If there is some lightweight refactoring in your branch however every time you pull the changed updates your code could change format as it applies the changes that others have completed. It is possible that each time you pull the changes you would need to reapply any lightweight refactoring that you have done. In the worst case other peoples lightweight refactoring conflicts with your lightweight refactoring despite both of them being non-functional changes. As this happens periodically a lot of time could be spent adjusting the code back into to the format and style that you wanted.

Differences in how code is understood. According to Kerievsky a reason for refactoring code is to better understand it [15]. As shown by [6] the very act of going through the source code and reprocessing it in a clearer form can help with the understanding of it. This would suggest that developers tend to leave the code in a difficult to understand state or that different developers understand things differently. Kerievsky also relates a tale about how the lack of knowledge of patterns makes a particular refactoring look a lot more complex [15]. The different perspectives meant that the programmer he refers to as John has a differing opinion that the refactored code was not an improvement. This shows that it is not just different functionality that influences the need to refactor but sometimes the knowledge and experience of the developers themselves. It is often the case that two developers could have different views about what is an appropriate refactoring. This could be because each person brings different skills, notices different issues and has a preferred way of visualising a problem and solution.

Version control systems not being aware of changes in the order. One of the changes which is not catered for by current version control sys-

tems is changes in the order of methods. The first person to check-in their code will have no issue as the version control system assumes that all the changes are simply a new revision. When the second person attempts to reconcile their view there is the possibility of having unnecessary conflicts. A lot of these conflicts will be with refactored or reformatted code which although works the same has a different structure.

3.2 Benefits of private views

We want to be able to maintain private views that can have different but equivalent refactorings. Different structures of code that function the same way have a number of features that could be of interest.

Reduced interference with other software developers. One benefit is that it is possible to keep the structure of code that each programmer works on as consistent as possible. This means that when the developer examines the code again that it remains familiar and in a similar state to how it was when they last examined the code. The location of methods and variables are more likely to remain in the place the software developer left them even if a merge occurs. By maintaining two private views it allows software developers working on the same programming project to freely refactor or add notes with minimal interference from others. It also means that the software developer will not interfere with others. If changes that are purely for formatting are not included when the code is merged then there will be less changes. The reduced changes would also mean that there will be less merge conflicts when merging code.

The number of changes is reduced when merging. The number of changes when merging is reduced if you omit any changes that don't also

have any change in behaviour. This in turn means that there is less chance for there to be a merge conflict.

The ability to have comments tied to a specific view. It would be possible to have comments that are not considered when doing a merge. This would be a benefit if there are comments that are specific to a view and that are not necessary to share. This would allow a programmer to keep a lot more personal notes about a change. As some comments will remain only in a private view there is less chance that it will be hard to read the code due to the surplus amount of comments.

3.3 Implementing private views

There are a number of ways we have considered about how to provide private views that have the same functionality.

Comparing differences using a non-ordered comparison algorithm. Instead of using the Longest Common Sub-sequence (LCS) based approach we could instead use a non-ordered comparison. The easiest way to consider this concept is that the LCS algorithm compares two lists whereas a non-ordered algorithm is a bit more like comparing sets of items. In this illustration each item would be a chunk of code. Node that the items within these items may still need to be ordered.

The problems with using this approach to reconcile two different sets is that the comparison would not know the difference between what needs to strictly remain in order and what is allowed to be in a different order. If the version control system has an understanding of a particular computer language it is much easier to determine what items can be moved without changing functionality and which ones need to stay in the same order.

Normalising the source code before placing it in the version control system. Before placing the item into source control it could be automatically transformed into an agreed upon format. That is before doing a merge both the code currently in the view and the code that is in source control would need to be transformed into normal form. Here normal form could mean that all the methods are arranged alphabetically. Once we have the normal form of both items being merged it will be easier to compare versions to see if they are equivalent. If both versions are equivalent there is no need to merge them.

It is complicated to do this well and it requires careful thought about what features would be normalised and which will not. It seems more efficient to compare two revisions directly with each other rather than to go to the effort of transforming them before comparing them.

Storing additional information in the version control system. By storing additional information within the version control system different views could be managed and recreated. This concept is very similar to Ekmans plug-in for eclipse that maintains a record of different refactorings in addition to the source code [12].

The problem with this is that it would have to store information about every private view. In a distributed version control system especially an online one the number of distinct views could be large and change often.

Using a tool like JDime solely as a method of comparison. As mentioned earlier (see Section 2.4.3)there are a number of reasons why JDime cannot currently be used as a method of keeping two private views. If changed however it could still be useful. One idea we had was to attach it to Git and solely use it to detect equivalent pieces of Java source code.

The difficulty we experienced with this idea was that JDime had features that we did not want. Moulding JDime into something that

we could use was too complex and it was easier to explore using the JastAddJ compiler.

We selected using a non-ordered comparison algorithm as we had a method of figuring out what could be reordered and what had to stay in the same order. To achieve this we used a parser to discover the AST. As each AST node had a start and an end position we could relate each AST Node back to its position in the source code. We could also determine which parts of the source code were comments and white space as these segments of the source code were not covered by an AST node. We will discuss this further in the next chapter.

3.4 Are comments important?

Although in this thesis we have focused mostly on changes to the behaviour of a program as opposed to aesthetic changes to the source code we recognise that sometimes changes to comments may be important. There are occasions were it is practical to require that an important comment or a change to a comment are merged. There may also be instances where is it better not to merge a comment, as it is specific to this view. We propose that inserted or deleted comments should be treated as if they are specific to the view and that modifications to existing comments need to be copied into other views. We believe the best idea however is to allow comments to be marked with an annotation to specify if they are only relevant to the view they are currently in or need to be included in any merge.

Existing comments and even white space can also provide useful information about any changes of order in the code. If a programmer has cut and pasted a block of code the white space and comments are also moved and provide hints to what has occurred. Even if some of the code has been modified there could be enough clues left behind to suggest that the most likely event is that the code has moved and then adjusted.

For these reasons comments are investigated as part of the Refactor Categories Tool.

Chapter 4

Refactor Categories Tool

In order to show that having private views would be useful we created the Refactor Categories Tool. Normally Version Control Systems such a JGit compare and merge files by differentiating between insert, delete and modify operations. The Refactor Categories Tool enhances this by also identifying instances where a block of code has moved but the functionality remains unchanged. It also can differentiate between changes to comments, white space and Java code. This means it can differentiate between instances where a change to a source file does not cause a change in the behaviour of the program and some instances where it has been refactored.

4.1 Overview

When the Refactor Categories Tool is run an ordinary text comparison is used as a starting point. The text comparison within JGit is used with white space ignored. The text comparison returns the minimal number of text changes in an `EditList` object. The `EditList` object contains a list of changes to the plain text. Information about each change that the `EditList` object retains is:

- the starting line of the change for both revisions being compared

- The ending line of the change for both revisions
- The type of change that is being made

As this is a text comparison the types of changes that are detected between two files are limited to inserts, deletes, and modifications. In order to expand this list we need more information. We obtain information about the meaning of Java files by parsing both of the revisions we are comparing. We use JastAddJ [20] to parse the information into an AST. Once we have the AST we need to discover which AST nodes match which changes to the text in the source code.

Both the `EditList` object and AST nodes contain information about the line of source the AST node starts and ends at. Unlike the `EditList` object an AST node can also hold the column that the AST node starts and ends at. Using this positional information we can match the text changes in the `EditList` object to a set of AST nodes.

There are likely to be AST nodes that are not included in any of the change sets and can be safely ignored. In order to find those AST nodes we are interested in we need to traverse the AST. The root node spans the entire file. By examining the children of the root however, we can determine which children contain all or part of the matching text change.

An example of this can be found in Figure 4.1. In this Figure it is safe to ignore any children of the AST node marked 'Child 1' because 'Child 1' does not contain the text change. The AST Node marked 'Child 3' however is interesting as its location places it within a text change.

Another interesting question is how the tool deals with situations where a text change partly overlaps an AST node or an AST node only partly covers a text item. In Figure 4.2 two different AST nodes each contain part of a text change. By examining any possible children of 'Child 1' and 'Child 2' we could discover 4 types of AST Nodes:

AST nodes that are not included in the text change. Any AST node that cannot be associated with a text change can be safely ignored. It

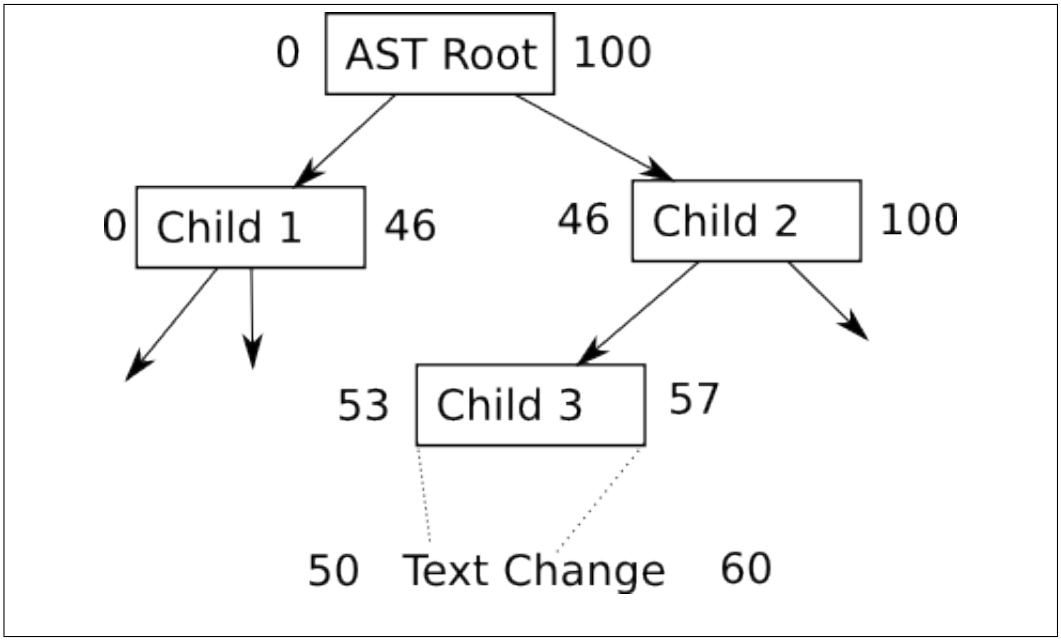

Figure 4.1: An AST showing each AST Node arranged in a tree and with Child 3 consisting of a text change. Each AST Node and text change has a start row recorded before the node and an end row recorded after the node.

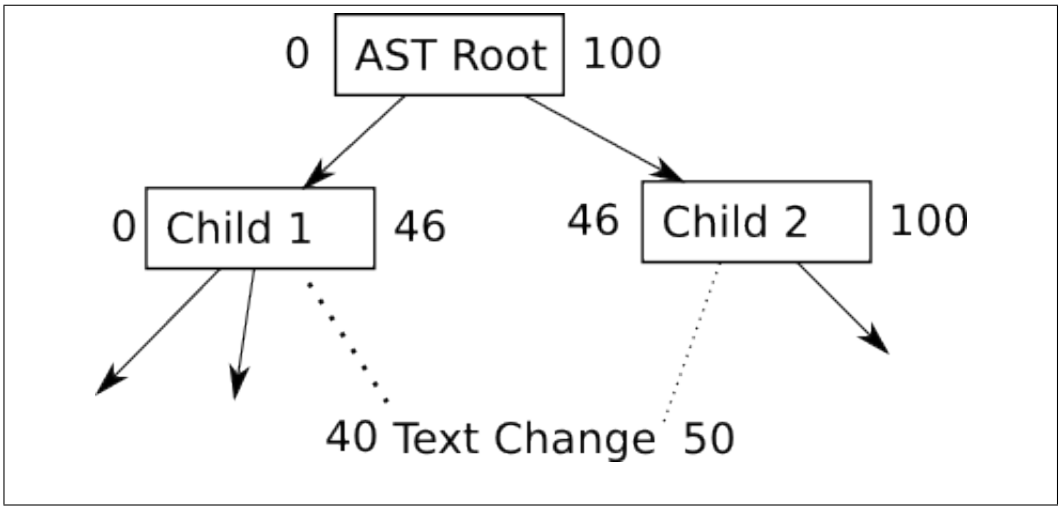

Figure 4.2: An AST showing a partial overlap between child 1 and a text change, and child 2 and a text change.

means that there are no changes to the text and therefore no changes to that AST Node.

AST nodes that are fully encapsulated by the text change. These are the AST nodes that we wanted to discover as they are entirely within the range of a text change. It means that this AST node has changed and warrants further investigation. There may also be text changes that are outside the AST node. If they are not part of any AST node they would be classified as comments or white-space.

AST nodes that partly include the text change but have child AST nodes. If an AST node has children and it is not fully contained by the text change we recursively check the children.

AST nodes that partly include the text change but have no child AST Nodes. If an AST node is a leaf node in an AST and it contains a text change then it is highly likely that the AST node has changed. As there has been changes to this AST node then it also warrants further investigation. Any part of the text change that is not part of the AST node and not part of any other AST node would be classified as comments or white-space.

In both Figure 4.1 and Figure 4.2 there are portions of text that are outside an AST node. These are of interest to us because these are instances where the text may have been changed but it has no impact on the behaviour of the program. The most likely instances of this is when a comment has changed or if white space has been introduced in the middle of source code.

4.2 What the tool does

4.2.1 Detects moves

If an AST Node has been inserted at one point and a similar one deleted it is possible to see if a code block has moved. The following shows an example of original source code before any methods have been moved:

```
1 public class SampleMoveAndChange {
2   double rect(double w, double h) {return w * h;}
3   double tri(double b, double h) {return b / 2 * h;}
4   double cube(double len) { return 6 * sqr(len);}
5   double circ(double rad) { return Math.PI * sqr(rad);}
6   double sqr(double num) {return num * num;}
7 }
```

Once we move the `tri` method to before the `rect` method and `circ` method to after the `sqr` method the code now looks like this:

```
1 public class SampleMoveAndChange {
2   double tri(double b, double h) {return b / 2 * h;}
3   double rect(double w, double h) {return w * h;}
4   double cube(double len) {return 6 * sqr(len);}
5   double sqr(double num) {return num * num;}
6   double circ(double rad) {return sqr(rad) * Math.PI;}
7 }
```

A comparison of the source code using Meld, a graphical merge tool is shown in Figure 4.3. Both `rect` and `circ` have been deleted from the original version and have been inserted into the modified version. However the `circ` method has also changed slightly. When this code is compared and parsed into an AST we will initially have four differences due to both the deletion and insertion of the two methods. We are not able to directly compare the deleted AST nodes with the inserted AST Node to see if they are equal because the `circ` method has been changed slightly. Instead of

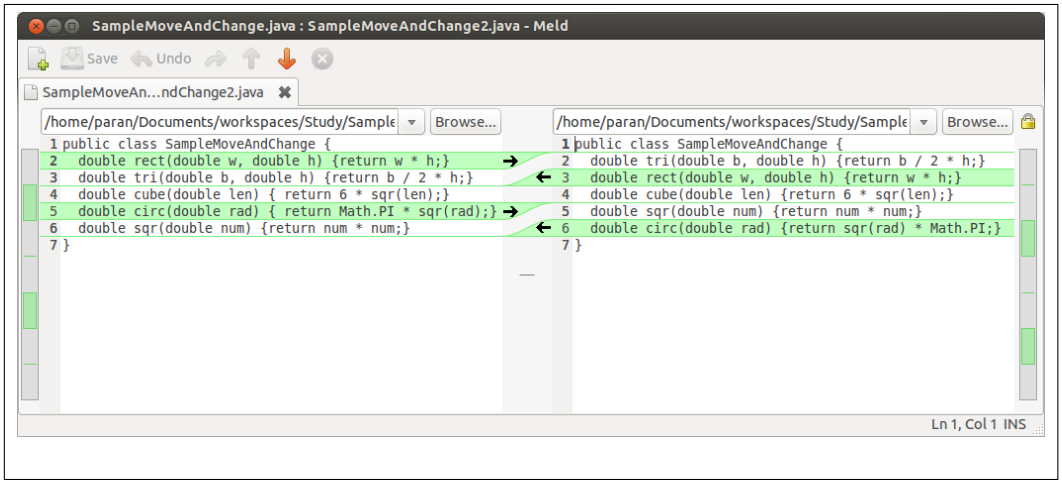

Figure 4.3: A comparison of similar code with methods that have moved and changed

checking for equality we need to check for similarity. By comparing all the AST nodes for a delete candidate against all the AST nodes for a insert candidate we are able to calculate a score that shows how similar they are.

There could be more than one insert and we need to compare them against all the deletes to obtain the scores for each combination. In the above example for instance we need to compare the deletion of `circ` with both the insertion of `rect` and the modified `circ` to determine which one is the better fit.

If a method is moved from one class into an inner class the programs' behaviour could have changed. When this occurs there is no guarantee that a match between that insert and delete is a valid one. If a method has been shifted within the same class however, the program will still act the same. Therefore in order to determine if the move is one that has no impact the matches are only counted if a match can be found within the same container. Here, a container is an enclosing scope within the source code, such as a class. Restricting any valid moves to a container ensures we get moves that don't change the behaviour of the program.

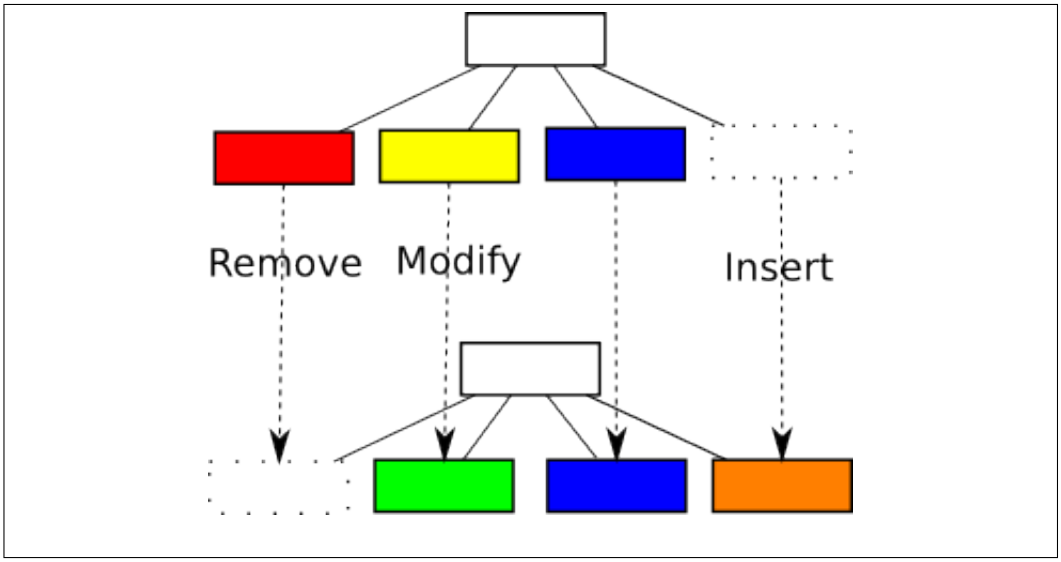

Figure 4.4: Two AST nodes being compared to each other by comparing the differences in their child nodes.

To calculate the scores for a single change we start at the root AST node for both the insert and delete candidates. If the root nodes have the same type and fields then we can then check their children. By performing a diff between the lists of children for both the insert and delete candidates we obtain the minimum number of differences. If an extra child AST node has been inserted or a child AST node is deleted then 1 is added to the score. If a child AST node has been modified it is the equivalent of both a child AST node being inserted and one being deleted so 2 is added to the score. If there has been no change to a child AST node we need to recursively assign a score using its children. To get the total percentage of children that changed we divide the score against the total number of children for both the delete and insert AST nodes we are scoring.

In the example in Figure 4.4 two AST nodes are being compared with each other. The red AST node has been deleted and an orange AST node have been inserted. Because the yellow node has been modified it has effectively been deleted and a green node has been inserted. This means there have been a total of four changes to this level of the AST.

Scores are stored in a two dimensional array where the rows are insert candidates and the columns are delete candidates. We use the greedy algorithm to select the lowest score in the array. Insert and delete candidates that are selected are recorded as a move and should not be compared with anything else. The comparisons that are no longer possible are in the same column or row in the array. The comparisons that are no longer possible are removed and the next lowest score for a match is evaluated. To ensure that anything cannot be matched with anything once all good matches are eliminated a limit is set ensuring that any further matches are ignored. The two items that could have otherwise been matched will remain a solitary insert and a delete.

4.2.2 Detects renaming

The Refactor Categories Tool can also detects some renaming. It does this by comparing the names for each each node. If the name has changed but the node type and all the children nodes are still the same then it places the change into the 'rename' category. When the Refactor Categories Tool does not check every AST node it can not determine if the rename operation is going to have impact on other code. This means that if all copies of a methods or variable have not been uniformly renamed it cannot find the ones that remain unchanged.

4.2.3 Detects equivalent code

In some situations when the code has been modified the ASTs still remain the same. This means that although there have been changes the behaviour of the code still has not changed.

4.2.4 Detects changes to comments

In addition to doing a comparison of AST Nodes the Refactor Categories tool also detects text changes that are located between AST nodes. These changes could be comments or white space and are checked to see if they have been inserted, deleted, modified or moved. The Refactor Categories Tool first does a cursory examination of the text differences between two files. It then examines both the differences in executable Java code and the differences between comments and white-space.

This detection of comments and white-space also extends to move operations As it is possible for the comment to change we also need to test for similarity rather than equality. This is done by comparing the characters in the code block that has been deleted with characters from the code block that has been inserted. A comparison is done between the two code blocks using a diff to obtain the smallest number of changes. A score of 1 is added for each deleted or inserted character. If a character has been modified a score of 2 is added. To obtain the percentage of items changed the score is divided by the total number of characters in both code blocks.

4.3 Performance decisions

Performance is an issue especially for when examining large software projects that have many changes. This means we have not only had to look at using more memory for the Refactor Categories Tool but also had to make some changes to the code to free up more memory where required.

By setting object references back to null when they are no longer used.
Once a difference has been detected we attempt to remove any of the information we are not going to use in the future. This should help the garbage collector free up memory. We do this by setting the AST nodes recorded against a difference to null once the difference has been discovered.

By not analysing AST node that did not have a text based change. By ignoring AST nodes and any of their child nodes that do not have a text based change comparing all the AST nodes can be avoided. This will also help to speed up analysis as these nodes do not need to be analysed to see if they match.

matching within only the required scope. If we were not testing for moves in the same scope we would need to test every deleted AST node against every inserted AST node for the entire file. By stipulating that we can only be sure if it is a legal move if it is in the same scope we not only eliminate a lot of relocations of source code that are illegal but also reduce the number of items that need to be compared. In addition to this the AST node along with its type are recorded in a hash map. This ensures that only AST nodes that are a similar type are compared against each other.

4.4 Design decisions

Java was chosen as both the language to write the tool in and to be the language that the tool would recognise. This was done as it was a language we understood and to make use of JDime if it was going to be part of the tool. We wanted to use Git due to its distributed nature. The reasons for choosing a distributed version control system was that we wanted each private view being able to function on its own as a fully fledged project without being dependant on a server. As we were already using Java rather than using the main Git distribution which is written in C we needed to use JGit instead.

We could not use JDime because of the issues discussed earlier. Instead we used JastAddJ — the same Java implementation that JDime uses so that we could implement some of the differences we required including being comment aware and only examining each of the copies of source code.

There are a number of design differences between JDime and the Refactor Categories Tool. Instead of doing a text comparison first and only proceeding to analyse the program using an AST if there are conflicts the Refactor Categories Tool examines all entries that have a difference in them. Although this takes longer and is more memory intensive there is an advantage to this. If a merge was done using an ordinary text comparison and there is a non functional change to only one revision there is no conflict and JDime only does a text based merge. During this text based merge we could get the non functional changes that change the source code without changing the programs' behaviour. By examining all changes irrespective of if the text has conflicts means that the Refactor Categories Tool can determine if it is a change that would not affect the behaviour of a program.

As there is a cost overhead with testing all the changes rather than just the conflicting ones the Refactor Categories Tool needs to be efficient in how it tests changes. Assuming that changes occur in select areas in the file there are portions of the file that have not been changed. As mentioned in Section 4.1 we use positional information to match AST Nodes to text changes. By ignoring portions of the AST that do not have any text changes we avoid spending extra time where we know changes have not occurred.

In some instances there is no position information stored in the JastAddJ AST nodes. This could be because they are generated by the parser to reflect parts of the Java language that are inferred rather than directly mentioned in the code. An example of this would be the use of super in the constructor. Even if it is not written in the code for every constructor has a super. Likewise all methods mentioned in an interface have a public type even if is not in the code.

To get around this problem we needed to discover the end position of the previous AST Node to determine the position the inferred AST node should occupy. This means that the inferred AST node is in the right position but is not represented by a block of text in the source code.

Comment and white-space are also examined separately as they also could give some indication of where code has been moved from or to. Before being checked to find matches unnecessary white-space is identified and recorded. Any text that remains is examined to determine if its is a comment.

Because of the way we are using the position in the code to identify AST nodes there are circumstances when parts of the Java programming language are identified as being surplus text. These have already been identified and represented as an AST Node. By identifying comments we can eliminate any of the items falsely recorded as comments.

Rather than comparing everything with each other to determine matches it is more efficient to match just the items that are under the same AST structure. This means that it is more likely that we get a match that is going to be relevant and valid. An example of this is matching methods. If the methods are under the same container (a class) they may be legally swapped without causing issues. If the method has been moved to an inner class from an outer one however it becomes more complicated and we cannot guarantee that the code is equivalent.

Chapter 5

Experimental results

5.1 Purpose

The purpose of this experiment is to measure how many insertions, deletion and modifications are performed on the Java AST verses comments across a range of realistic benchmarks. Similarly, how many insert and delete pairs can be classified as moves.

5.2 Methodology

We will now outline the experimental methodology. Our goal is make it possible for someone to reproduce our experiment.

The experiment was run on a Lenovo laptop with 3.5GB Ram and 113.2 GB disk space assigned to the 64 bit Ubuntu 13.10 (Saucy Salamander) operating system. The tests were written in Java and run within the Eclipse IDE

This text is further evaluated to determine if it is a change to comments or a change to white space. An example of this is if the text comparison detects a single change that inserts two new methods and a comment. The Refactor Categories Tool will recognise this as three distinct changes. This

Benchmark	Description	Commits	LOC
Jasm	Java bytecode assembler written for use with the Whiley programming language. github.com/Whiley/Jasm	74	29139
Jpp	Pre-processor for Java based on Bash shell script. github.com/maandree/jpp	40	254
AST Java	Small parser written to transform Java into an AST. github.com/klangner/ast-java	24	10174
Java Object Diff	Allows two Java objects to be compared at runtime. github.com/SQiShER/java-object-diff	291	10023
DiffJ	Diff tool which in addition to ignoring white space ignores changes of ordering in package names for a Java file. github.com/jpace/diffj	490	13712
IRE	Java library for regular expression matching. github.com/jkff/ire	41	2714
Syntax	Compiler compiler used for teaching. github.com/jaimegarza/syntax	89	9376
Auto Refactor	Eclipse plug-in that automatically refactors code github.com/JnRouvignac/AutoRefactor	212	13400

Table 5.1: A range of benchmarks that will be included in the test

means that it is possible for the Refactor Categories Tool to notice more changes than a text based comparison.

We had some garbage collection memory issues with some benchmarks that we were testing. To resolve these issues the Refactor Categories Tool was run individually once for each benchmark to further ensure that there were no memory leaks between runs. We also increased the memory by changing the parameters for the run configuration. The following parameters were added to the run configuration for the test.

```
-Xms512M
-Xmx4096M
```

5.3 Results

5.3.1 Overview

After running the Refactor Categories Tool over our benchmark suite we obtained the results shown in Table 5.2

The number of text changes that were recognised during ordinary text comparison done using JGit for each benchmark are shown in Table 5.3. This table allows us to show the difference between an ordinary text merge (reported by JGit) and the results produced by the Refactor Categories Tool shown in Table 5.2. Note that the values in this table are lower as the Refactor Categories tool detects multiple changes which a text based comparison aggregates.

5.3.2 Discussion

Figure 5.1, which has been generated from the data in Table 5.3, shows the average for all the benchmarks of the types of operations found during an ordinary text comparison. Figure 5.2, which has been generated from the

Benchmark	Java						WS	Comments		
	Ins	Del	Mod	Mov	Ren	Eqv		Ins	Del	Mod
Jasm	264	76	385	7	0	0	26	7	6	95
Jpp	43	9	50	0	0	0	6	1	2	11
AST Java	87	28	72	1	0	11	5	2	0	22
Java Object Diff	2645	1961	8862	5	9	187	270	52	25	881
DiffJ	3106	3164	5192	58	82	18	276	36	39	291
IRE	263	213	640	1	2	30	49	10	3	79
Syntax	1142	544	2216	6	7	12	204	14	81	451
Auto refactor	1702	1621	2809	7	16	25	295	30	16	568

Table 5.2: Results over our benchmark suite showing the number of inserts (Ins), deletes (Del), modifications (Mod), moves (Mov), renames (Ren) and equivalences (Eqv) for Java AST nodes, comments and white-space (WS) changes

Benchmark	Ordinary Text Diff (reported by JGit)		
	Ins	Del	Mod
Jasm	91	44	337
Jpp	23	3	41
AST Java	44	15	85
Java Object Diff	1186	837	8219
DiffJ	746	991	3544
IRE	112	49	465
Syntax	691	214	1863
Auto Refactor	639	309	1621

Table 5.3: The result of doing an ordinary text comparison showing the number of inserts (Ins), deletes (Del) and modifications (Mod)

data in Table 5.2, shows the average for all the benchmarks of the types of operations found during a Java AST node comparison.

When we compare the figures the number of modifications is different. The percentage is much higher in the text comparison that JGit normally uses and lower for the Refactor Categories tool. The reason for this is because a change done to a block of text recorded in the `EditList` object is made up of a number of smaller changes. For instance a single line detected by the JGit text comparison may contain a Java AST node change and a change to a comment. The Refactor Categories Tool records each of these changes separately. A change detected by JGit could be a number of lines, so a single block of text could contain multiple Java AST nodes. If a block of text is a delete all the Java AST nodes or comments in that range will also be deletes. If a block of text is an insert all the Java AST node or comments in that range will also be inserts. It is slightly different for blocks of text that have been modified. It is possible that some of the changes could be Java AST nodes or comments that have been inserted. They could also be deletes rather than modifications. Some of the changes that an ordinary text diff recognises as modifications could actually be made up of individual inserts and deletes. Note that the number of changes reported by the Refactor Categories Tool will increase exactly for this reason. Furthermore, when we examine these changes more closely using the Refactor Categories Tool the comparative percentage of insert and delete increases while the percentage of modifications tends to decrease.

Instead of having insert, delete and modifications Figure 5.2 also has move and rename. Predominately the changes were still identified as modifications, inserts and deletes rather than the move and rename we are interested in. What this could mean is although some non-functional changes have been introduced it does not occur very often. However, it is also possible that a more advanced identification algorithm could improve the results.

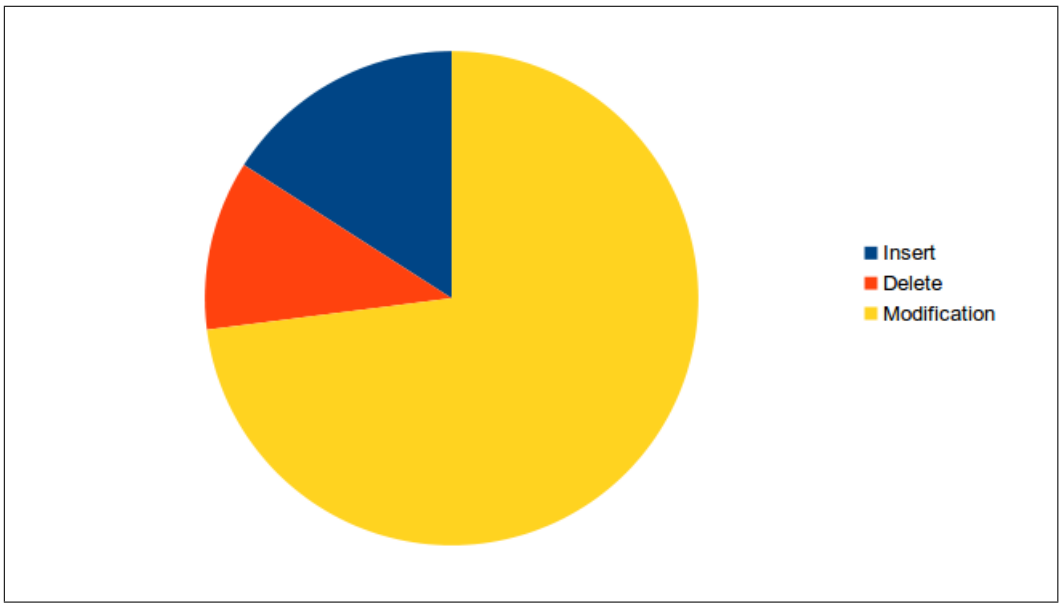

Figure 5.1: Type of operations for an ordinary text comparison on average for all the benchmarks

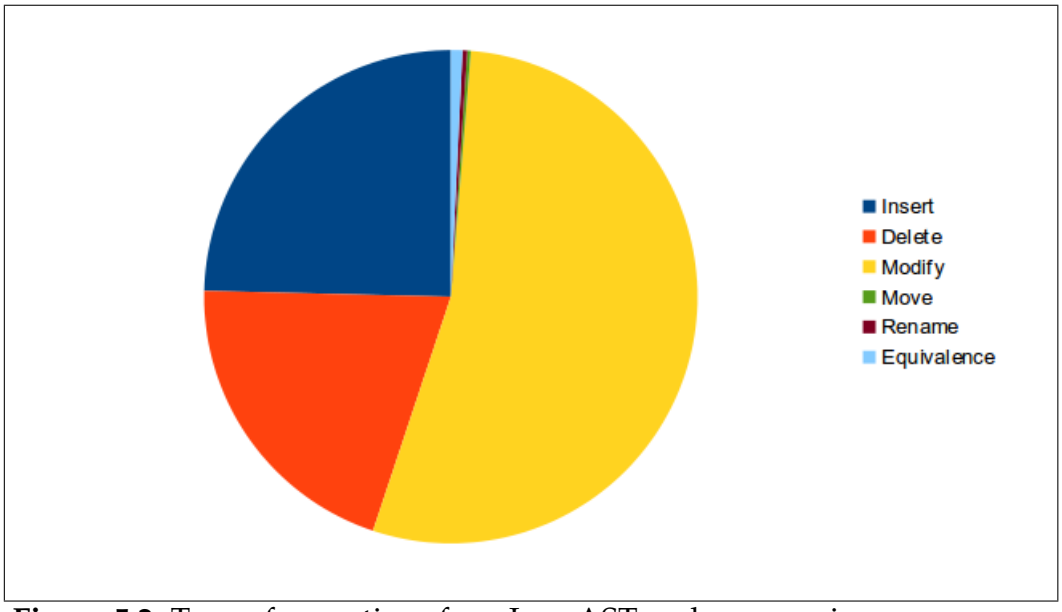

Figure 5.2: Type of operations for a Java AST node comparison on average for all the benchmarks

Another aspect we were interested in was the difference between Java based differences and non-Java changes such as comments or white-space. Figure 5.3 shows the how many Java changes were detected as compared to comments and white-space. This shows that the predominate change in the source code was a change in functionality rather than a change to the documentation. It implies that in programming projects there tends to be many more lines of program than lines of comments. What this means for this thesis is that currently there are more changes to functionality rather than to coding style and documentation. As there are still significant amount of changes to comments it is possible that being able to modify comments that will not be merged could be useful.

Figure 5.4 more closely examines the type of changes that have occurred to comments in the source code. By far the main change to comments is modification. Unlike Java AST Nodes blocks of code containing comments are not further divided. This means that the block of code marked as a modification to a comment may in fact contain a number of deletes or inserts. Another possible cause for a large number of comment changes to be modification is if a programming project is well established. It is possible, in an existing project, that changes are made to the code and the existing comments are modified to reflect that change. The changes to the code could include insert or deletes in addition to modifications to the source code. In this situation when comparing the amount of inserts, deletes and modifications between the source code changes and the comment changes the source code will have comparatively more inserts and deletes whilst the comments will have comparatively more modifications.

A significant number of comments have been inserted and deleted. Examining these items a little closer revealed that some included the words ”TODO” or ”FIXME”. Although the addition of these comments are likely to be associated with a related code change they may only be relevant to

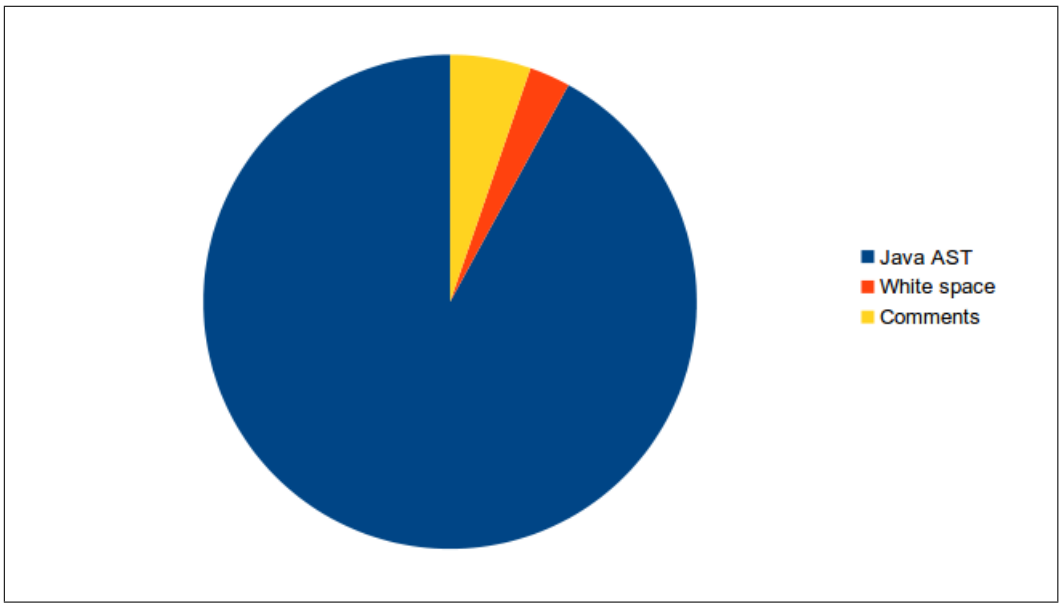

Figure 5.3: The type of changes that most often occurred over all the benchmarks

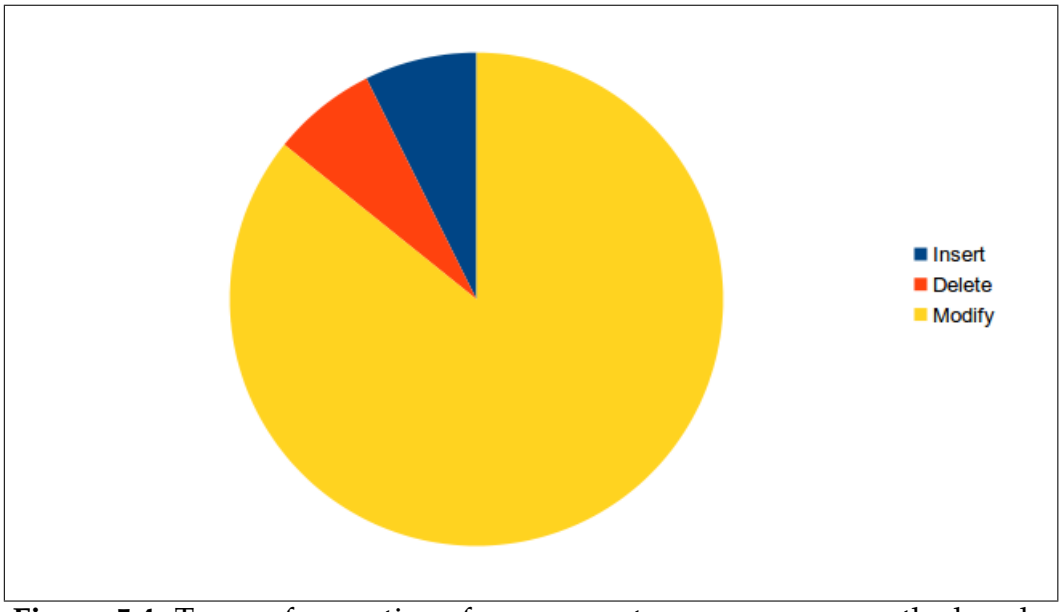

Figure 5.4: Types of operations for comments on average over the benchmarks

people working on that section of the code. The same is true for blocks of commented out code. In this situation having a private view could be useful.

Chapter 6

Conclusions and future work

In this thesis we presented the concept of maintaining private views in Java. A private view presented here is an environment that allows a developer to import changes they want while avoiding hidden unwanted changes. This would also allow programmers to implement lightweight refactoring to their tastes, while minimising the impact on others. In evaluating what these private view will look like we used version control systems as a starting point. There are some features of version control systems that already temporarily limit unwanted changes (e.g. branches). However, during a merge any unwanted refactoring is imported. To this end we created the Refactor Categories Tool as a precursor to creating private views. This tool analyses the difference between two revisions such as encountered during a commit and identifies some examples of lightweight refactoring. The way that the Refactor Categories Tool analyses these differences is by first parsing the source for both commits into a Java Abstract Syntax Tree (AST). Once the AST is populated we then identify which parts of the AST match the differences we want to examine. We then use the AST to identify additional features that have been changed. The features we have focused on are ones that do not change any functionality such as methods being moved or comments being changed. The results show that some of these lightweight refactorings are encountered

in practice. As the Refactor Categories Tool is a prototype it did not, unfortunately, identify as many as we hoped. We believe that it is possible to detect many more non-functional changes using more advanced identification algorithms.

6.1 Future work

In order to further the research into private views it would be useful to evaluate how the Refactor Categories Tool could be enhanced to detect more non functional changes. In addition to this some other tools could be adapted to create and evaluate the usefulness of private views.

6.1.1 Changes to the Refactor Categories Tool

There are a number of ways that the Refactor Categories Tool could be changed to discover more moves, and renames:

- At the moment the Refactor Categories Tool only examines moves that occur within a class, however, there could be non-functional changes that occur inside a method. An example would be if a local variable declaration was moved. Sometimes this move would have no effect on the code and others it could cause the code to no longer compile.
- At the moment the Refactor Categories Tool only compares matches within a limited scope (i.e. a class). Allowing the Refactor Categories Tool to check other parts of the code, such as inner classes or even other files may also produce some interesting results. Although we cannot guarantee that the moves discovered are valid ones this could give us more information about the source code we are examining.
- At the moment the Refactor Categories Tool only examines files that have been identified by JGit as being modified or renamed. In some

instances JGit could have incorrectly determined that a file as been deleted and reinserted rather than being renamed or moved. This could happen easily since during a move or rename Java changes the package reference and class name within the file. This is especially true if the class has both been renamed and modified.

- Revising the scoring system for matching up inserts and deletes may produce some better results. At the moment modifications are counted as two changes using the scoring system to match inserts and deletes. Experimenting by reducing this value could improve the number of matches.

In addition to moves and renames, other lightweight refactoring may be of interest. One of these are changes to access modifiers. An example would be if a methods access changes from being private to being public. Each of the method calls would then need to be rechecked to ensure that the change does not affect functionality. Due to the possibilities of overloaded methods in Java this would be complicated.

An additional lightweight refactoring that could be considered is code that has been duplicated. This could be done in a similar manner as how the Refactor Categories Tool check for code that has moved. If we also check for code that has been modified slightly we may be able to determine that a copy and paste has been used to generate new code. However, at the moment the Refactor Categories Tool only considers code that has been changed. If we want to analyse where code has been copied we would need to check the entire source for copies as opposed to just the items that have changed.

Comments could be associated with the AST Node they relate to. With this change would be possible to tell if changing a comment should be reflected in other views when there is a source code change. This change is difficult as it is hard to tell which block of code the comment refers to. One way this could be done would be to associate single-line comments

at the end of the line with the AST Node that appears directly before them and other comments with the AST node that appears directly after them. This however is only a rough approximation so it may be helpful to also be able to specify exceptions to these rules by using annotations that tie the comment to a block of code. Annotations could also be used to specify how important the comment is. If the comment is marked as unimportant it would indicate that it still should not be considered a change even if it differs between revisions.

The Refactor Categories Tool could be re-purposed to allow it to be used as a merge tool rather than a comparison tool that we are currently using it for. This would bring us a step closer to being able to realise the vision of having better separated private views.

Performance of Refactor Categories Tool could be further enhanced by only parsing nodes that contain the text change. This however would require major changes to the parser or rewriting it. There would also be the complexity of figuring out how to only partially parse a source code. The benefits of rewriting the parser would save memory in addition to speeding up the parsing of Java code into AST nodes.

6.1.2 Other lines of enquiry

There are other tools that could be modified to determine when a refactoring has taken place.

JDime has already been investigates as part of this thesis. Although JDime cannot recognise changes to comments or white-space it could be re-purposed. If it could be converted into a comparison tool rather than a merge tool then code that has been refactored differently could be compared without the result being normalised.

According to Pace [21] *DiffJ* is able to find the functional differences between two revisions of Java source code. When computing the difference DiffJ ignores a range of lightweight refactorings such as moved methods,

moved imports and the code being reformatted. As it ignores comments and white-space however, it will not be able to determine if there have been comment based changes that may be important.

Bibliography

[1] ALDOUS, D., AND DIACONIS, P. Longest increasing subsequences: from patience sorting to the Baik-Deift-Johansson theorem. *Bulletin of the American Mathematical Society 36*, 04 (July 1999), 413–433.

[2] APEL, S., LESS ENICH, O., AND LENGAUER, C. Structured merge with auto-tuning: balancing precision and performance. *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012* (2012), 120 – 129.

[3] APEL, S., LIEBIG, J., BRANDL, B., LENGAUER, C., AND KÄSTNER, C. Semistructured merge. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11* (New York, New York, USA, Sept. 2011), ACM Press, pp. 190 – 200.

[4] ARSLAN, A. N. *Language and Automata Theory and Applications*, vol. 6031 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, May 2010.

[5] BERTINO, N. Modern version control. In *Proceedings of the ACM SIGUCCS 40th annual conference on Special interest group on university and college computing services - SIGUCCS '12* (Oct. 2012), ACM Press, pp. 219–222.

[6] BOIS, B. D., DEMEYER, S., AND VERELST, J. Does the "Refactor to Understand" reverse engineering pattern improve program com-

prehension? *Ninth European Conference on Software Maintenance and Reengineering* (2005).

[7] CHACON, S., CORNELL, G., GENNICK, J., LOWMAN, M., MOODIE, M., PEPPER, J., POHLMANN, F., RENOW-CLARKE, B., SHAKESHAFT, D., WADE, M., AND WELSH, T. Pro Git. *Control* (2009), 1–210.

[8] CODICE SOFTWARE. Semantic Merge. `http://www.semanticmerge.com/`, 2013. Accessed: August 2014.

[9] COHAN, B. Patience Diff Advantages. `http://bramcohen.livejournal.com/73318.html`, 2010. Accessed: August 2014.

[10] DIG, D., MANZOOR, K., JOHNSON, R. E. R. R. E., AND NGUYEN, T. N. Effective software merging in the presence of object-oriented refactorings. *IEEE Transactions on Software Engineering 34*, 3 (May 2008), 321–335.

[11] DIG, D., MANZOOR, K., NGUYEN, T. N., AND JOHNSON, R. E. MolhadoRef: A Refactoring-aware Infrastructure for OO Programs. In *Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange - eclipse '06* (2006), ACM Press, pp. 25–29.

[12] EKMAN, T., AND ASKLUND, U. Refactoring-aware versioning in Eclipse. *Electronic Notes in Theoretical Computer Science 107* (Dec. 2004), 57–69.

[13] FOWLER, M., BECK, K., BRANT, J., OPDYKE, W., AND ROBERTS, D. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[14] FREESE, T. Refactoring-Aware Version Control Towards Refactoring Support in API Evolution and Team Development. In *Proceeding of the 28th international conference on Software engineering - ICSE '06* (May 2006), ACM Press, pp. 953–956.

[15] KERIEVSKY, J. *Refactoring to Patterns*. Addison-Wesley Professional, 2004.

[16] LESS ENICH, O. Master Thesis Adjustable Syntactic Merge of Java Programs.

[17] LOELIGER, J. Collaborating With Git. *Linux Magazine 46* (2006), 32–35.

[18] MURPHY-HILL, E., AND BLACK, A. Breaking the barriers to successful refactoring. *2008 ACM/IEEE 30th International Conference on Software Engineering* (2008), 421–430.

[19] MYERS, E. W. AnO(ND) difference algorithm and its variations. *Algorithmica 1*, 1-4 (Nov. 1986), 251–266.

[20] ÖQVIST, J., AND HEDIN, G. Extending the JastAdd extensible Java compiler to Java 7. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages, and Tools - PPPJ '13* (Sept. 2013), ACM Press, pp. 147–152.

[21] PACE, J. JDiff. `https://github.com/jpace/diffj`. Accessed: August 2014.

[22] THE ECLIPSE FOUNDATION. JGit. `http://download.eclipse.org/jgit/docs/jgit-2.0.0.201206130900-r/apidocs/org/eclipse/jgit/diff/HistogramDiff.html`, 2014. Accessed: August 2014.

[23] TICHY, W. F. Design, implementation, and evaluation of a Revision Control System. 58–67.