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 your changes before you apply them. The ability to have a separate view which although functionally equivalent to other views can present the code in a different form in these situations would be valuable. It enables the programmer to refactor or change the code with minimal impact on others. 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 source code has been moved within a file or comments has been added removed or edited. This gives us an indication 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 acknowelge the help of both my supervisors David Pearce and Lindsay Groves but also my wife Hye Eun Park who has been a great help by supporting me though this thesis.

Contents

Chapter 1

Introduction

According to Bertino [5] *Version control systems* provide a way of allowing multiple developer to collaborate. When multiple developers work on the same source code there is a risk that they have conflicting changes for the same portion of the source code. 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 [21] 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 for one version with the changes made on the other revision. 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 revision that are derived from a common ancestor. It is possible to determine what has been deleted, inserted or changed by comparing

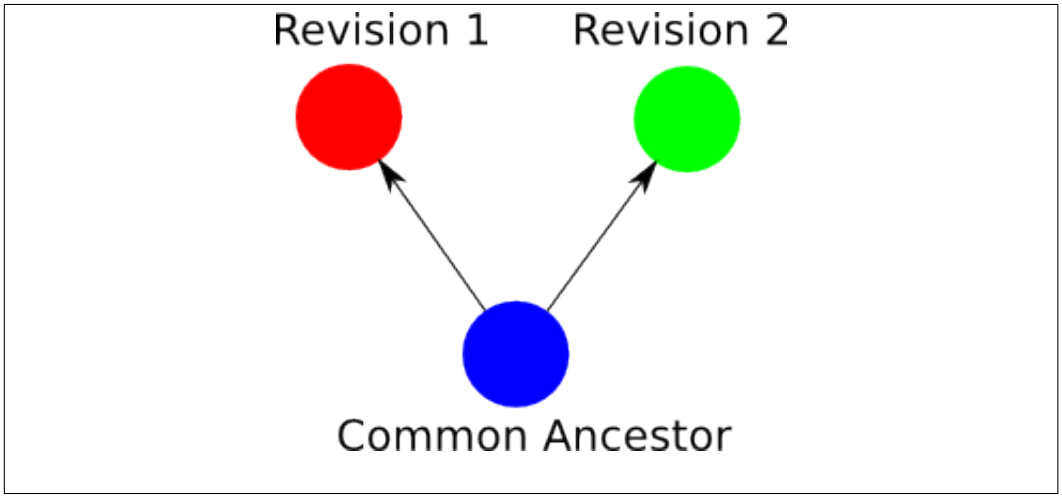

Figure 1.1: A file that has two different revisions

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 systems

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 consistant the swapping of methods could be disconcerting.

Another issue that non-functional changes raise is an increase in the number of *merge conflicts*. These could occur when there are two views both have changes in the order of methods. 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 methods alphabetically. For an ordinary text-based merge these changes to the structure of the source code require manual intervention. These issues highlight the need to develop smarter way to merge.

It is becoming more important to have accurate 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 can make source code available to many developers at a time. 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 separate views of differently ordered but equivalent source code for the purpose of reducing the number of changes introduced during a merge. It also explores a way of allowing a version source control system to detect when

there is a change in the source code but not a funtional change in a program. Examples of this is if items have been reordered, if comments have been inserted or there have been changes to the formating. We have used the Refactor Categories Tool we have created in an expiriment that iterates though all the revisions in the version source control for a handful of Java projects.

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 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 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. Once the change has been made is possible in most version control systems to write a message when the documents are checked in. In some version control

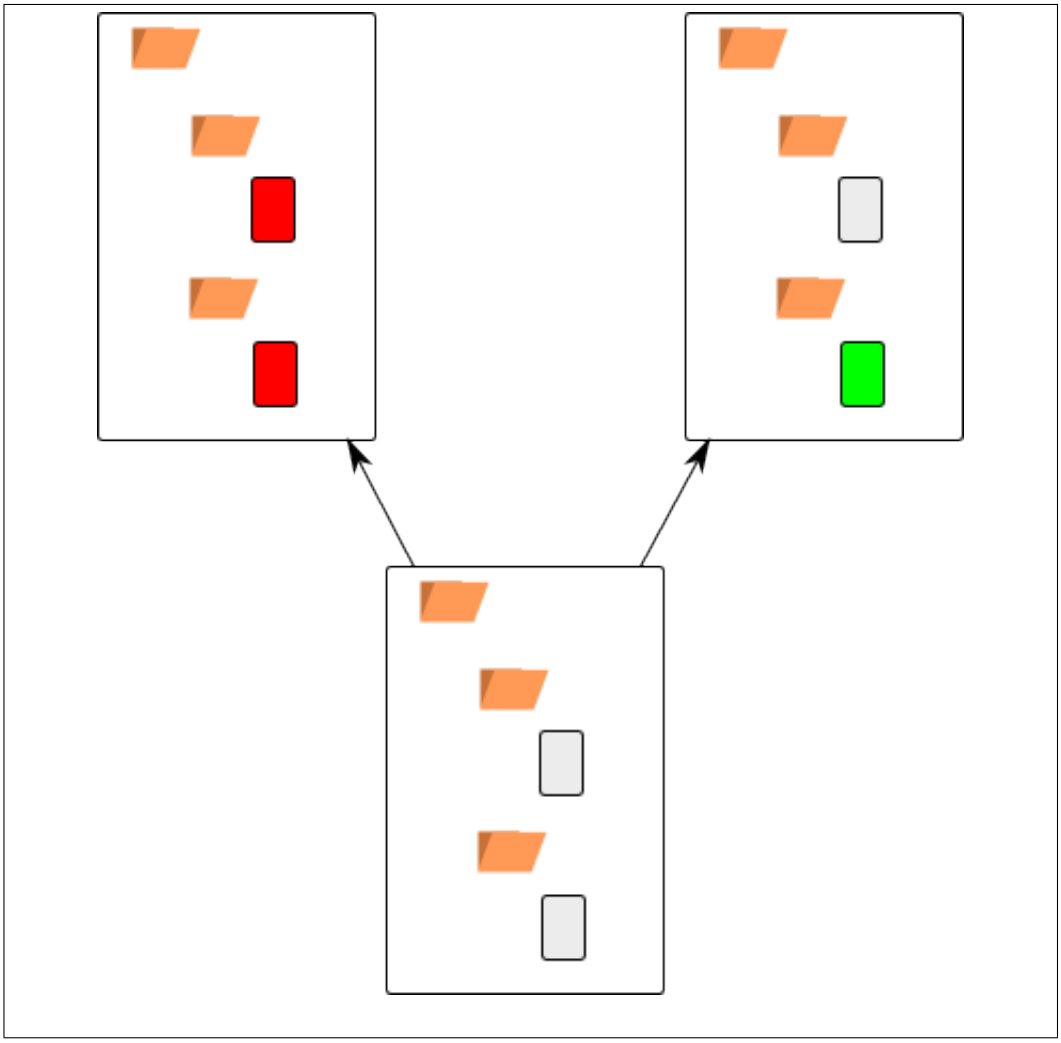

Figure 2.1: A project that has been split into two branches

systems this message is required to check-in any set of documents. The reason this is useful is 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 an issue tracking system

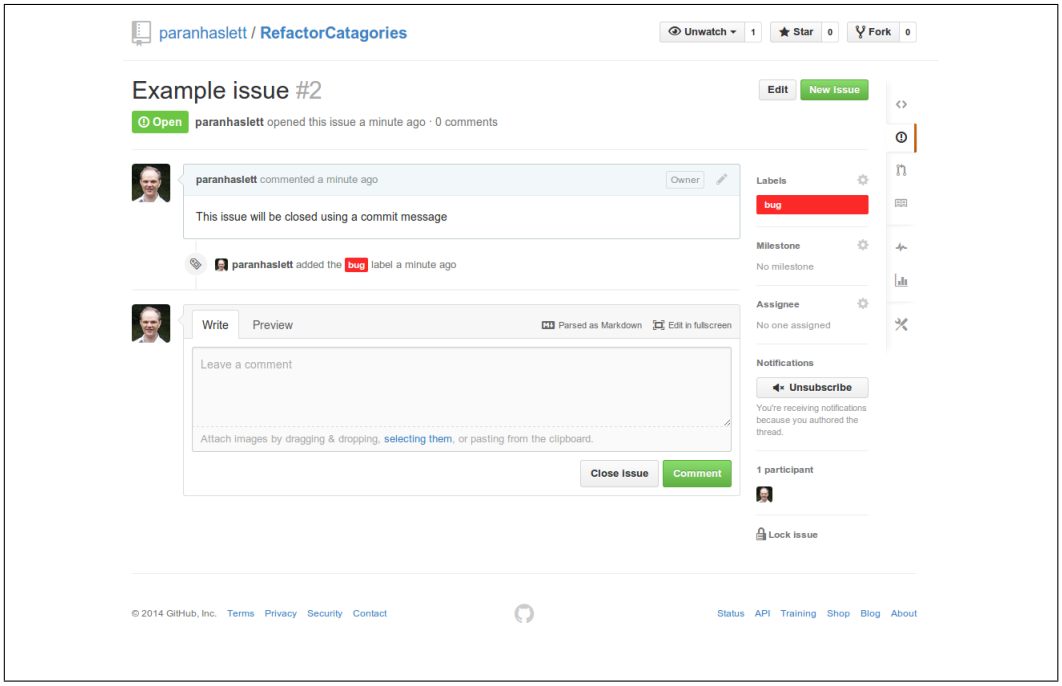

Figure 2.2: A issue that has been created in GitHub

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 issue tracking system. An example of this is the issue tracking ability that is built in to GitHub, an online version control system. In GitHub an issue can be created as shown in Figure 2.2. The issue tracker has also been linked with the messages that you need to write when you check in some files. If you include a hash sign followed by a number in the message you check in then GitHub updates the issue with that number as shown in Figure 2.3.

Collaborate with multiple developers. Version control systems make it possible to have individual revisions that contain each persons' changes. The version control system then manages the way these changes are merged into a composite product. Bertino [5] describes the ability

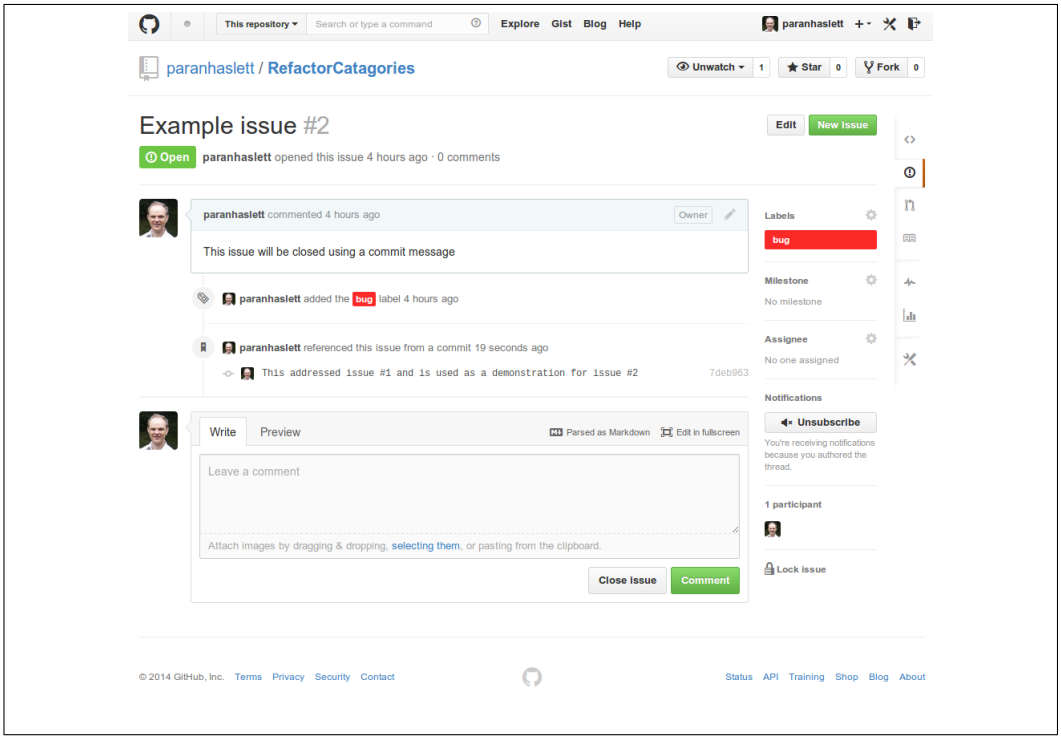

Figure 2.3: The issue has been updated from a commit message

to merge the work of multiple people as being a powerful collaborative tool. This is beacuse 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 exporing 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 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 it 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 has explained why he considers locking in a version control system to be a good idea in the design for RCS[21] 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. Condering 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. Version control still can have problems with merge conflicts.

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 can not 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 peoples changes and a change only occurs in our code then that change is retained. If we attempt to update our code by merging other peoples changes and a change only occur in the other persons code the the change is inserted into our code. Only if a different change is made in both your revision and others revision to the same area in the code from the original that a merge conflict can occur.

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 allowed multiple developers the ability to work on the same source code but often had a large overhead. Some centralised system it required 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

An advantage to having a complete copy of a project from the repository is that it is possible to make changes to the program remotely. You are also able to select which changes others have made you want to incorporate 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 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 server that is on the web may be a better solution. Loeliger [16] shows how it is possible to access and use a web based version control server to achieve this. A good example of a web based version control server 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. Figure 2.4 shows the amount of activity that there has been on the JGit project. With 74 contributers 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.

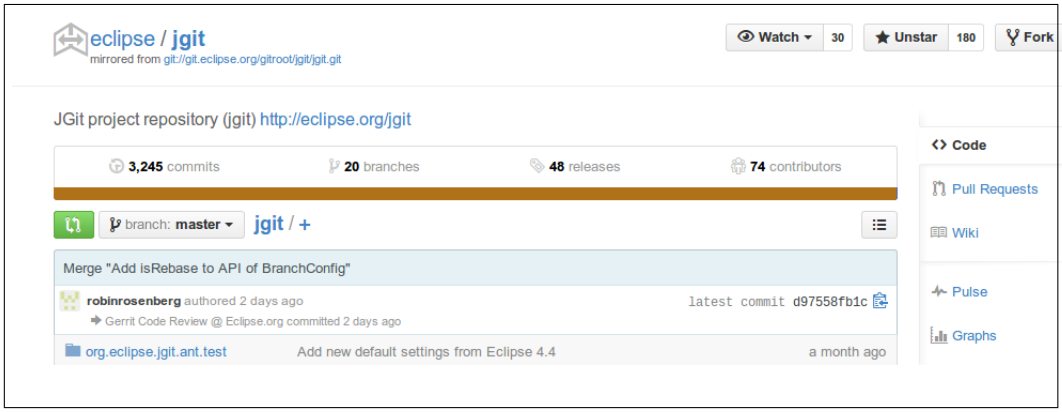

Figure 2.4: The overview page of the Jgit project in GitHub. This shows that there are 74 contributors to the source code and 3245 individual changes over 20 branches.

2.2 Longest Common Subsequence

A simple definitionof 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 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 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
```

```
3    public static double area(double radius){
4      return Math.PI * square(radius);
5    }
6
7   public static void main(String[] args){
8     System.out.println(area(3));
9   }
10
11  public static double square(double num){
12    return num * num;
13  }
14 }
```

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
3   public static void main(String[] args){
4     System.out.println(area(3));
5   }
6
7   public static double square(double num){
8     return num * num;
9   }
10
11  public static double area(double radius){
12    return Math.PI * square(radius);
13  }
14 }
```

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
3   public static double area(double radius){
4     return Math.PI * square(radius);
5   }
6
7 }
```

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
3  public static void main(String[] args){
4    System.out.println(area(3));
5  }
6
7 }
```

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.

2.2.3 Myers

The Myers algorithm was discovered by Eugene Myers who claimed that finding the minimal differences between any two documents was the equivalent to finding the shortest or longest path in a graph [19].

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 it finds the longest increasing subsquence is similar to a single player card game. When this is used with line numbers from the source code the longest common subsequence can be established.

When it is used to determine the longest common subsequence only the lines in the code that only appear once in both revisons and that match are considered. The lines that only appear once are then used as markers to determine the location for any of the other changes.

As Bram Cohen has pointed out in his blog there are instances where a traditional LCS algorithm can return results that are not as helpful to a developer as they could be. Instead of noticing that a method has been inserted it is possible for a LCS algorithm to return that the closing brace of one method followed by another method without the closing brace has changed instead. By initially ignoring items that might occur many times in the source code such as brackets and whitespace the similarities are instead centered around the markers that have been found.

2.2.5 Histogram

According to comments in the source code a histogram algorithm claims to be equivalent to a patience sort in some circumstances but provides a better result for when there are multiple copies of some items being merged.

2.2.6 The problem with LCS

There is still a problem with longest common subsequence. It does not notice changes of order in a document. For the example we have been looking at two methods that 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.

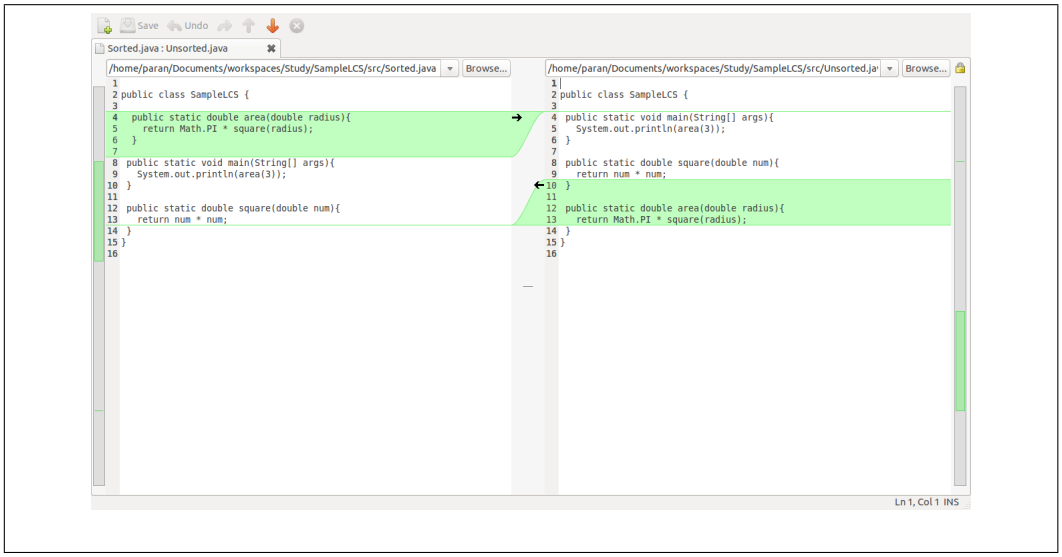

Figure 2.5: A graphical diff tool showing differences with two equivalent blocks of source code

2.2.7 How LCS is used in differencing tools

Differencing tools (often shortened to ”diff tools”) are a range of standalone programs that compare the contents of two files and show the similarities and differences. In many diff tools 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 option 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 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.

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 [12]. Similarly according to Kerievsky some of the motivations for refactoring include adding more code and understanding existing code [14]. 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. A simple example is illustrated as follows:

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

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

```
public TempConv() {
  Scanner keyboard = new Scanner(System.in);
  System.out.println("Enter the temperature in Celsius");
  int celsius = keyboard.nextInt();
  System.out.println("Degrees Fahrenheit is approx "
    + celsiusToFahrenheit(celsius));
  keyboard.close();
}

public int celsiusToFahrenheit(int celsius){
  return celsius * 2 + 30;
}
```

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:

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

```
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 }
```

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 alternate way of refactoring is as follows:

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 Lesßnich [2] [3] [15] to study how to get a balance between fast text-based merges and slow but more acurrate semantic merges. 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 perfomed a small expirmenmt to investigate JDime as a tool for automatically merging code which has been reordered. As JDime performs a type of automatic merge it requires 3 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

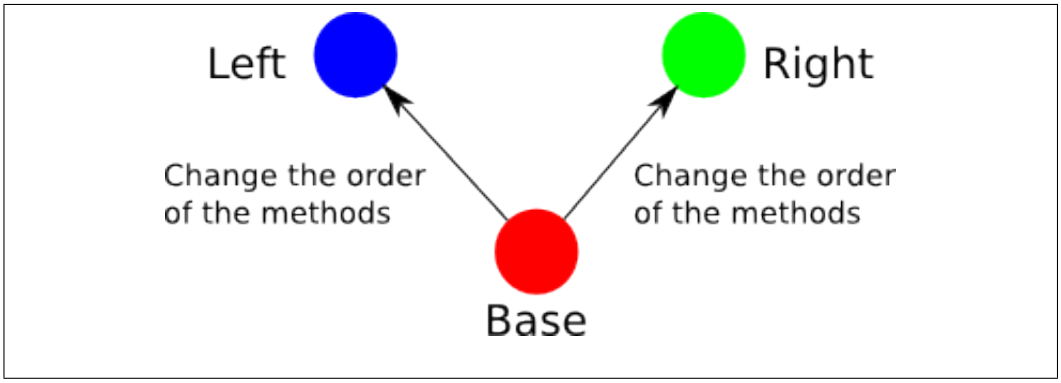

Figure 2.6: The set-up for the test of JDime

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 loading 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 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 left and the right directories. Figure 2.4.2 demonstrates how the test files were arranged.

Once the test was set up using the test handler the JDime 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 output did not match the order of any of the equivalent input files.

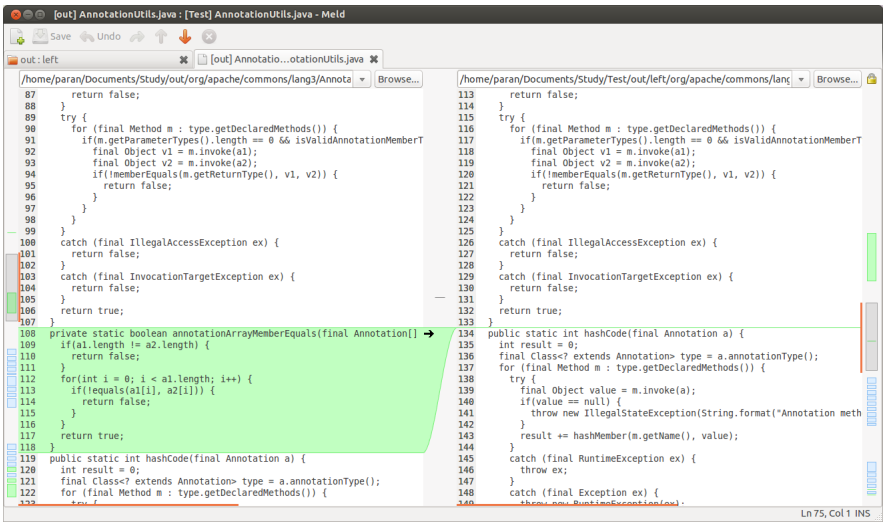

Figure 2.7: Screen-shot of Meld showing a different method order

2.4.3 Reasons why Jdime cannot currently be used to create separate 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 only time an AST based merge is performed in JDime is if there

are plain text conflicts. However there could be a hidden conflict that the plain text merge does not detect. For example if we have two different views and a method has both been relocated in both of them. Athoungh the method in the original version is in the middle of the code one revison relocates it near to the top of the file and the other to the bottom. If in addition to this a change is introduced inside the method in both revisons that would normally produce a conflict had the method not been moved.

A text based merge would not recognise this as a conflict however a AST based one would. According to the text based merge both revisions would agree that the method had been deleted from the middle of the file. One revison however would insist that a new method with the same name as the deleted one has been created near the top of the file. The other revision would insist that a new method with the same name had been created but near the bottom. When merged both the method at the top and method at the bootm would be inserted inot the merged code.

An merge using an AST would recognise the methods described as being a conflicting change. As the names and method signature has not changed for either method it would recognise the conflicting change that has been introduced.

As JDime does not use an AST comparison to check any items that have been deemed non-conflicting by the text based merge it would miss this conflict.

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 [10] 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 Leßnich when they developed JDime. Both make use of ASTs to record information about any refactoring that took place. Freese [13] also developed a tool that was very similar written in Object-Z. Despite this interest in refactoring aware merges however the idea of keeping separate branches as consistent as possible 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 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 contol 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 already in making diff tools and version control systems refactor aware. Some example of this are as follows:

- MolhadoRef [9] [8] attempts to incorporate refactoring in version control systems.
- Semantic Merge is a series of stand alone diff tools for different languages.

3.1 The problems

There are a couple of challanges when collaborating using versioning 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. It is possible that the code will need to be refactored before any of your changes can be added. This would be a fair judgment call as Fowler claims that the main time to do refactoring is before making any changes [12]. 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 conflicts that they experience could be because the changes you made directly compete with the changes they have made. Potentially more conflicts would occur between the changes they have made and the refactoring that you have completed. This is because a refactoring often makes a large amount of global changes to the source code.

As shown in figure 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 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 also include any formatting changes, or code restructuring used to create a programming enviroment to allow you to be more productive. During these occasions you may want to avoid changing other peoples code in such a dramatic fashion.

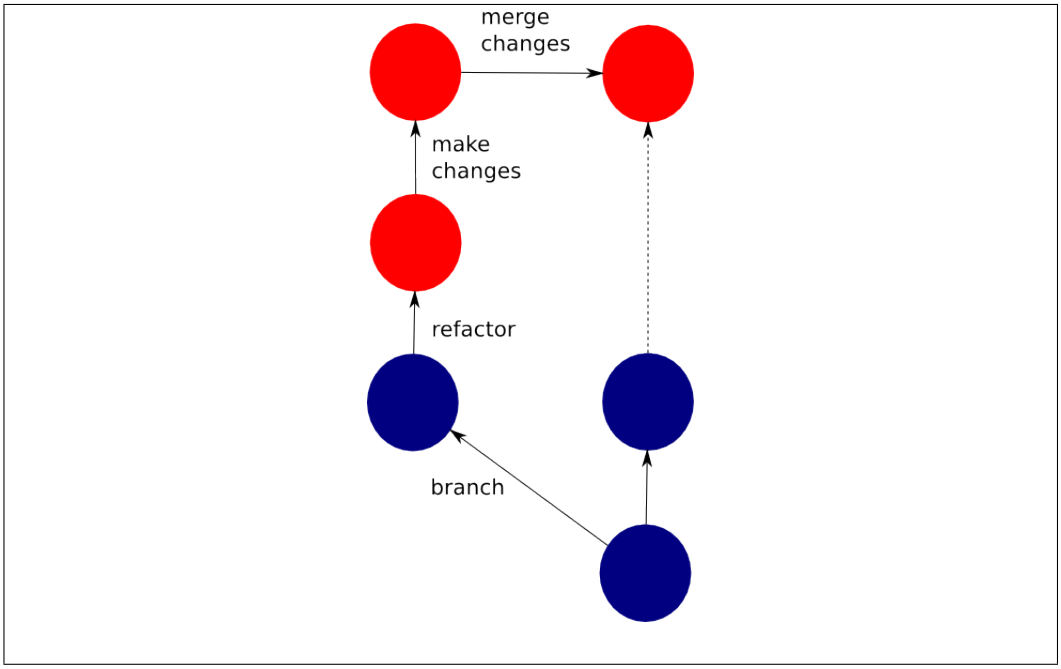

Figure 3.1: Merging changes with refactored code also merges any refactoring

In some conditions refactoring is only required to simplify the code to implement a small change as opposed to cleaning up the entire code base. This partial refactoring is likely if the code base is large. According to Melina et. al. [17] refactoring is a challenge when the code base is large. By definition refactoring does not change any functionality, but changes the source code. This means that the code previous to being partially refactored has the equivalent functionality to the code after the partial refactoring.

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 to submit your code periodically. This could be done to ensure that there is not too much divergence between the separate branch and other development projects. The desire to regularly merge the code makes the issue we have discussed even worse. Currently if you have a project where there are periodic check-ins for each development milestone there could be a large impact each time there is a commit. This is because any refactoring for the large project is imposed upon others each time it is merged.

Differences in how code is understood. According to Kerievsky a reason for refactoring code is to better understand it [14]. As proven 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 [14]. 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 sometime 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 visualizing 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 systems is the changes of order. 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 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 refactoring. 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 at 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 to work 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 purlely for formating 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 comment 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 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. The items will still need to be ordered.

The problems with using this approach to reconcile to different ordered sets is that 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.

Normalizing 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 tranformed 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.

The number of items that this strategy will resolve however are limited. It also seems more efficent 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 plugin for eclipse that maintains a record different refactorings in addition to the source code [10].

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 regular expressions It could be possible to use regular experssions to control a lot of the merge process. We could eliminate a number of comparisons that contain equivalent functionality in this manner.

Regular expressions however could be to combursome to use to represent a programing language.

Using a tool like JDime solely as a method of comparison. As mentioned earlier 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. Molding JDime into something that we could use was too complex and it was easier to explore using the JastAddJ compilier.

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 achive 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 whitespace as these segmants of the source code were not covered by an AST node.

3.4 Are comments important?

Although in this thesis we have focused mostly on changes to the behaviour of a program as opposed to astetic changes to the source code we recognise that sometimes changes to comments may be important. There are occiasions 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 revelant 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 occured. 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 prove that having private views would provide some useful results we created the Refactor Categories Tool. Version control systems such a JGit compare and merge files by differentiating between insert, delete and modify opperations. The Refactor Categories Tool enhances this by also including instances where a block of code has moved but the functionailty 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 causes a change in the behaviour of the program and some instances where it has been refactored. Although it does not detect more complex differences beyond this we intend to show that changes exist in real world software projects that have no impact on the projects' behaviour.

4.1 Overview

The test we use with the Refactor Categories Tool analyses all the historical changes ever made on a software project by extracting successive revisions from a Git repository using JGit. An ordinary text comparison is used as a starting point. The algorithm used for the text comparison is the histogram LCS one normally used by JGit with white space ignored. The text

comparison returns the minimal number of text changes in an `EditList` object. The `EditList` contains a list of changes to the plain text. The information about each change that the `EditList` object retains are:

- 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

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 revisions we are comparing into an AST using JastAddJ [**?**]. We then need to discover which AST node matches which change to the source code.

Each of the AST nodes contains information about the line of source the AST node starts and ends at. Unlike the `EditList` object which we have previously discussed, 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 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 a text based 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 only partly overlaps an AST node or an AST node only partly covers a text item. This issue is shown in Figure 4.2. In the situation where

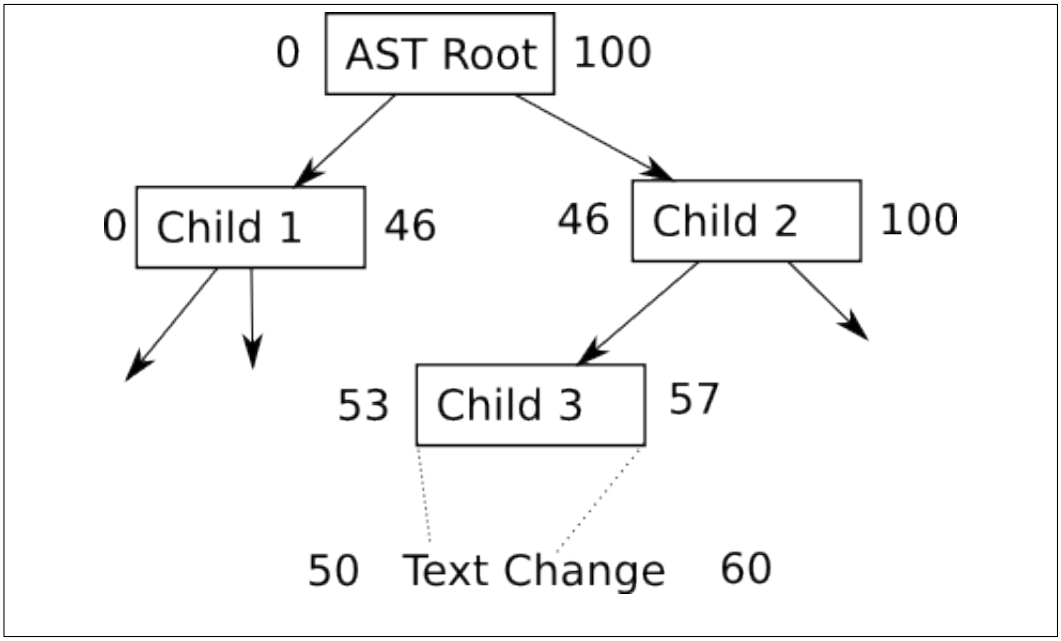

Figure 4.1: An AST showing each AST Node arranged in a tree and with Child 3 consisting of a text change. Each AST Node and text change has a start row recorded before the node and a end row recorded after the node

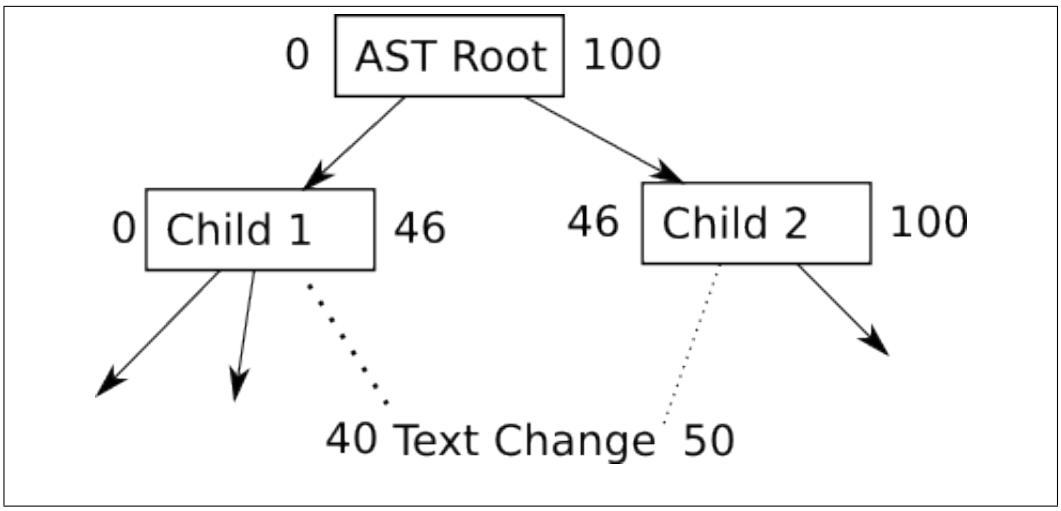

Figure 4.2: An AST showing a partial overlap between child 2 and a text change, and child 3 and a text change

only part of and AST Node contains the text change the children of that AST Node are examined to see if they are fully within the text change. If not it keeps recursively checking until either a child falls completely outside the change or is completely in the change.

In the case where portions of a text change which are outside of a AST node then that change could be a comment or white space. This can be seen in both Figure 4.1 and Figure 4.2 because in both of these portions of text are outside a text 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. Once the AST node for both revisions has been matched with the text change we can compare the AST nodes and their children to discover if they would behave in the same fashion. If they do behave in the same fashion it could indicate that the change is an ascetic one.

Another change we are interested in is if items have been moved. We would notice this if there is a both insert and a delete for similar AST node types within the same scope. If a method has been shifted within a class we would notice similar AST nodes, one being deleted at one location, the other being inserted at a different location. This is another example of a text change that does not change the behaviour of the program.

If we find examples of text based changes that do not change the programs' behaviour we have an indication that we can reduce the amount of changes necessary to update another view. This in turn could indicate that we would have less merge conflicts by reducing the overall number of changes.

4.2 What the tool does

By checking to see if an insert at one point is similar to a delete at another point it is possible to see if the code block has moved. In order to deter-

mine 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. For example if a method has been shifted within the same class the program will still act the same. If a method is moved from one class into an inner class there is no guarantee that they will be evaluated to determine if there is a suitable match. If a method is shifted from one class to an inner class however the programs' behaviour could have changed.

To determine if blocks of code have been moved we need to find a deleted block of code that is most similar to an inserted block of code. An example of this is if we have the original program shown in Figure 4.5 and the modified program shown in Figure **??**

A comparison of the source code using a graphical merge tool is shown in Figure **??**. Because two methods have moved there are two deletions from the original and two insertions into modified version. However the method that calculates the area for a circle has also changed slightly. In the Refactor Categories Tool all the Java are changes are represented by the AST for the original source code and the AST for the modified source code. We are not able to directly compare the AST nodes to see if they are equal because the `circleArea` method has been changed slightly. By recursively 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. As in the example there could be multiple inserts and we need to compare them against multiple deletes to obtain all the scores.

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 are similar we can then check their children. By perfoming 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

```
1  public class SampleMoveAndChange {
2
3    public static void main(String[] args) {
4      System.out.println(circleArea(3));
5    }
6
7    public static double triangleArea(double base, double height) {
8      return base / 2 * height;
9    }
10
11   public static double cubeSurface(double length) {
12     return 6 * squareArea(length);
13   }
14
15   public static double circleArea(double radius) {
16     return Math.PI * squareArea(radius);
17
18   }
19
20   public static double squareArea(double num) {
21     return num * num;
22   }
23
24 }
```

Figure 4.3: The original source code before methods have been moved

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

```
1  public class SampleMoveAndChange {
2
3    public static double triangleArea(double base, double height) {
4      return base / 2 * height;
5    }
6
7    public static void main(String[] args) {
8      System.out.println(circleArea(3));
9    }
10
11   public static double cubeSurface(double length) {
12     return 6 * squareArea(length);
13   }
14
15   public static double squareArea(double num) {
16     return num * num;
17   }
18
19   public static double circleArea(double radius) {
20     return squareArea(radius) * Math.PI;
21
22   }
23
24 }
```

Figure 4.4: The modified with methods that have both moved and have changed slightly

AST nodes we are scoring.

In the example in Figure 4.6 two AST node 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

```
\begin{center}
 \includegraphics[scale=1]{movediff}
 \end{center}
```

Figure 4.5: A comparison of similar code with methods that have moved and changed

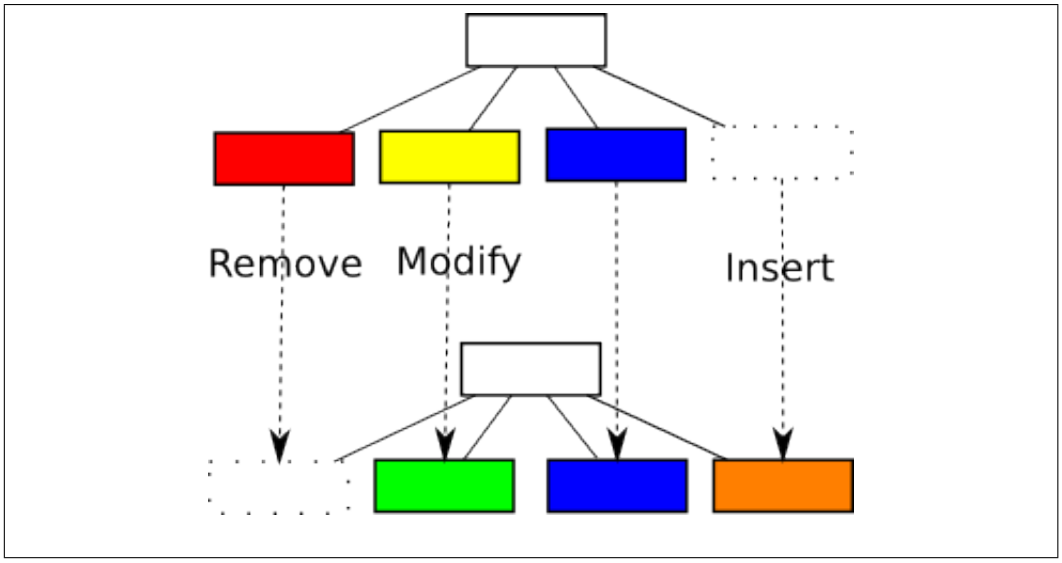

Figure 4.6: Two AST nodes being compared to each other by comparing the differences in their child nodes.

effectively been deleted and a green node has been inserted. This means has been a total of 4 changes to this level of the AST.

In addition to the the comparision of AST nodes to test for where java code has move we also need to determine when comments move. As it is possible for the comment to change we also need to test for similarity. This is done by comparing the characters in from the code block that has been deleted with characters from the code block that has been inserted. A diff 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 of a 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 chracters in both code blocks.

The 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. The insert and delete canidate 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 compariosns 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.

To a limited extent the Refactor Categories Tool can also deal with renaming. It does this by examing the node and seeing if its children have remained the same and its type remain the same but its name has changed. It can not analyse nodes where the name has not changed as these are not picked up by the initial text comparison.

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 whitespace.

4.3 Performance

This means that performance is an issue especially for 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 analyzing 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 analyzed 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 analyze the program using an AST if there are conflicts the categorization tool examines all files that have a difference in them. Although this takes longer and is more memory intensive there are some advantages to this. The main advantage is related to the concept of keeping branches as consistent as possible whenever there is a change. An example of this advantage is if a merge was done using an ordinary text comparison and there is a non functional change to only one revision. Examples of a non functional change could include reordering methods or inserting a comment. As the changes were only done to one of the revisions there is no conflict and JDime only does a text based merge. During this text based merge, in addition to any of the changes to functionality that we want, we 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 categorization tool needs to be efficient in how it tests changes. Assuming that changes occur in select areas within the file there are portions of the file that have not been changed. We have developed a method that spends a smaller amount time in within the portions that have already been identified as not containing changes. Like JDime we initially do a text based merge. The text based merge we use however uses the histogram merge in JGit. This allows us to use information from the text based merge when we analyze the AST tree. The information used is the ranges of line numbers and operations identified by the histogram

merge. The change set has been taken from the original JGit based diff contains the start and end of the change in both files and what type of change it is (insert delete or modify). By reusing these ranges of line numbers it is possible to figure out which AST items these changes affect. This is done by loading the file into the JastAddJ parser to get an AST tree. Line numbers for each item in the tree are then compared to line numbers from the change set. The line numbers are matched to the position information stored in each AST node using the following method.

The root of the AST is identified as the AST node we need to start at. We only begin any analysis, if the AST node resides completely within a block of text changes. If there are any separate blocks of changes that occur between the start position of the AST node and the end position of the AST node then we recursively examine the AST nodes children.

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 have 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.

4.5 Limitations of the tool

The Refactor Categories Tool focuses only on areas where there has been a text change in the source code. It is harder to investigate any change that has causes side effects in unchanged code. Fortunately it is not often that this type of side effect will be purposely placed in the code as it reflects bad design decisions. This may however be an issue with bugs, which are unintentionally occur in the code. This also means that the Refactor Categories Tool will not be able to tell when some code has been copied but the original remains unchanged. Instead it will assume that it is a completely new insertion of code.

If a change of a variable name is noticed the program still retains the same behaviour if the rename operation has worked on all instances where the old name was previously used. As we do not examine parts of the code that have not experienced a text based change we cannot guarantee that a rename operation is a valid one. We have an indication that it might be if the code still compiles without error. If a variable name is changed to that of an already existing variable however the behavior of the program could change. This is known as a *semantic conflict*. This issue already exists in a

text based merge and Fan [11], Shao [20] and others address this issue.

Chapter 5

Experimental Results

5.1 Purpose

The purpose of the test is to establish if there are

5.2 Methodology

As we are testing the complete revision history for several projects there is a high number of changes.

The test we use with the Refactor Categories Tool analyses all the historical changes ever made on a software project by extracting successive revisions from a Git repository using JGit.

By comparing only successive changes. There are a greater amounts of changes if we compare the head against a very old revision. By checking against successive we reduce the number of changes. This also could mean that we need to parse fewer files.

5.2.1 Setup

This

5.3 Results

5.3.1 overview

5.3.2 detailed examination of some results

The data sets following are complete git repositories which have been examined to discover the proportion of changes to behavior compared to ascetic changes that do not affect the behaviour of the program.

- Jasm - this a a Java bytecode assembler written for use with the Whiley programming language. https://github.com/Whiley/Jasm
- Jpp - this is a preproccessor for Java based on Bash shell script. https://github.com/maanc
- AST Java - a small parser wriiten to transform Java into an AST. https://github.com/klangner/ast-java
- Java Object Diff - allows two Java objects to be compared https://github.com/SQiShER/ja object-diff
- DiffJ - a diff tool which in addition to ignoring white space ignores changes of ordering in package names for a Java file https://github.com/jpace/diffj

There are a number of Java categories that the Refactor Categories Tool will recognise. Apart from the traditional insert, delete and modify it will aslo recogise if an Ast Node has been renamed but the content and type of the node remains the same. It will recognise valid moves within a scope. It will also recognise when the text based merge got it wrong by claiming that the code was not fuctionally equivalent when in fact it was.

Apart from this it will also diferentiate between Java code and comments. It can tell if comments have been inserted, modified, moved or deleted. It will also pick up any changes to that whitespace that have not already been figured out by the text based comparison.

Chapter 6

Future Work

Performance of Refactor Categories Tool could be further enhanced by only parsing nodes that are part of the change set or are ancestors of this. This however would require major changes to the parser or rewriting it. It would save memory in addition to speeding up the parsing of Java code into AST nodes.

Chapter 7

Conclusions

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.

[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, p. 190.

[4] ARSLAN, A. N. *Language and Automata Theory and Applications*, vol. 6031 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 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* (New York, New York, USA, Oct. 2012), ACM Press, p. 219.

[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] 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.

[9] 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* (New York, New York, USA, 2006), ACM Press, pp. 25–29.

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

[11] FAN, H., AND SUN, C. Supporting semantic conflict prevention in real-time collaborative programming environments. *ACM SIGAPP Applied Computing Review 12*, 2 (June 2012), 39–52.

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

[13] 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* (New York, New York, USA, May 2006), ACM Press, pp. 953–956.

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

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

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

[17] MILEA, N. A., JIANG, L., AND KHOO, S.-C. Scalable detection of missed cross-function refactorings. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014* (New York, New York, USA, July 2014), ACM Press, pp. 138–148.

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

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

[20] SHAO, D., KHURSHID, S., AND PERRY, D. E. SCA. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering on European software engineering conference and foundations of software engineering symposium - E* (New York, New York, USA, Aug. 2009), ACM Press, p. 291.

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