Maintaining differently refactored 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. This could be due to refactoring, or the code being reused in another project. It could even be due to throw away code or code used for debugging. This could at times also involve how the structure of the program is presented or the variable and method names that are being used. In these cases you may need to refactor the code 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 of methods and the addition of comments currently impact other developers even if there is no change in how the code works. A 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 changes to a program that do not change any behaviour.

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 supported me though this thesis.

Contents

1 Introduction **1**

2 Background **5**

2.1 Version Control Systems 5

2.1.1 Dealing with conflicts 8

2.1.2 Architecture 10

2.2 Longest Common Subsequence 12

2.2.1 Example 12

2.2.2 Methods of calculating LCS 14

2.2.3 Myers 14

2.2.4 Patience 15

2.2.5 Histogram 15

2.2.6 The problem with LCS 15

2.2.7 How LCS is used in differencing tools 16

2.3 Refactoring 16

2.4 JDime 18

2.4.1 What Jdime can be used for 19

2.4.2 How Jdime works 19

2.4.3 Testing Jdimes suitability 19

2.4.4 Resons why Jdime cannot currently be used to create seperate views 21

2.5 Other refactoring aware versioning tools 22

3 Individual refactored views **29**
3.1 The problem 29
3.2 Benefits of individual refactored views 33
3.3 Ways individual refactored views could be achieved 34

4 Refactor categories tool **37**
4.1 What the tool does 37
4.2 Overview 38
4.3 Performance 41
4.4 Design decisions 42
4.5 Limitations of the tool 44

5 Expirimental Results **45**

6 Future Work **47**

7 Conclusions **49**

Chapter 1

Introduction

According to Bertino [3] *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 [11] for the RCS version control system. The problem with this is that one person can stop anyone else 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 each

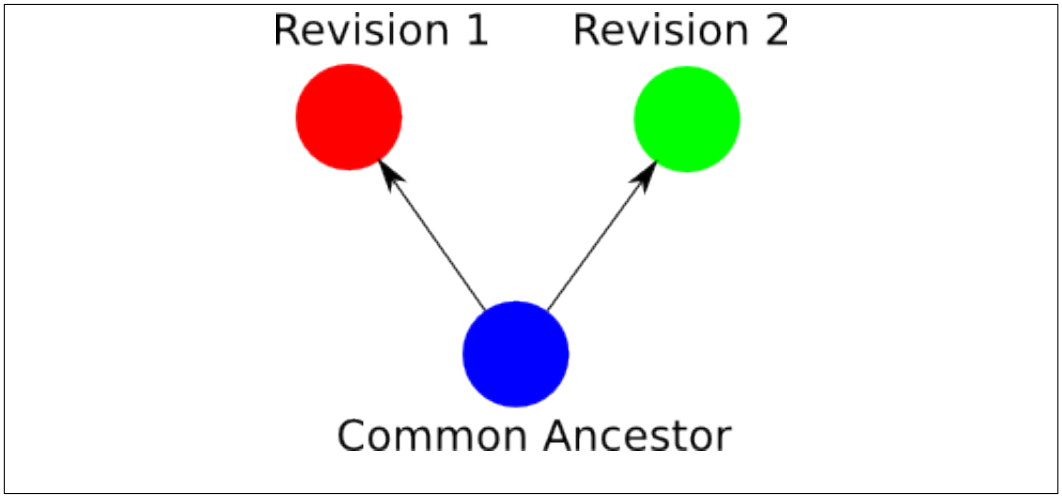

Figure 1.1: A file that has two different revisions

of the revisions against the common ancestor. This is often done as a linear comparison. 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. This change becomes pointless when a 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 unnecessary change is recorded in the merged file. 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, swapping methods unnecessarily could be disconcerting.

If there are two separate revisions that both have changes in the order of methods then it is possible to get a merge conflict. In both of the revisions the behaviour of the program has not changed. Conflicting but pointless changes to the structure of the source code require manual intervention. This issue has highlighted the need to develop smarter ways 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 explores a way of allowing a version source control system to detect when there is a change in the source code but not in the behaviour of a program. Examples of this is if items have been reordered, if comments have been inserted or there have been changes to the formating. It also 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.

In order to do this the source code needs to be divided into understandable sections. When each of these sections for each revising are compared

it is possible to determine if a section has been moved. This enhances what can be detected when examining the differences between two files.

Chapter 2

Background

As this thesis is about maintaining differently refactored branches within a version control system refactoring 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 reafctoring.

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 con-

trol systems this message is required to check-in any set of documents. The reason this is useful is at a later date if 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. If following the example above we were to examine a document and wonder why the address changed we could examine the check in with that change and see the message that the person changing it wrote.

When used on source code in tandem with an issue 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 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 [3] describes the ability to merge the work of multiple people as being a powerful collaborative tool. A version control system allows multiple people to work on the same document. In some circumstances it allows them to work on the document at the same time. A version control system allows people with different ideas to collaborate on a single document.

In order to manage the different version or revisions version control systems work better on plain text documents. One of the main uses for a version control system is to maintain source code written in a plain text programming language, such as Java or C.

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 the 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 conflict. The advantage of locking a file like this is that the file is always in a consistent state. 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[11] The disadvantage is if one person retains the document for extended periods of time, it cannot be changed by anybody else. If a lot of work has been done on the document before it is unlocked for others to edit it. Furthermore the resulting document may be barely recognisable as the original if extensive work is done on it. 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 similar. If we go back to the illustration of two people working on the same set of documents. If instead of one person having sole possession of a document at a time that person only has possession of the page or pages they are changing. As those pages are smaller than the whole document they are likely to retain them for shorter periods.

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 a *merge conflict*. This occurs whenever the computer cannot automatically process a merge. The merge then needs to be sorted out manually.

If not regularly merged 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 evaluating what is necessary and what is unnecessary [3]. 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. These issues have a far greater chance of occurring if there is a dramatic change such as refactoring.

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 devlopers responsible for those changes.

Automatic Merging

If there is a three way merge it is possible for the computer to calculate the merge independently provided that there are no merge conflicts. To do three way merging requires three different revisions are required, 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 revison 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 revison and others revison to the same area in the code from the original that a merge conflict can occur.

2.1.2 Architecture

Centralised version control In a centralised version system all the changes are made to one location. This is refered to a a centralised repository. Having a cenralised 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 solved a lot of issues but often had a large overhead. In 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 [4] this single point of mangement has some adavantages as it is possible to manage what developers had access to. According to ChaconA [4] and Bertino [3] 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 [4] and Bertino [3] 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 contol systems Whilst is is possible for a measure of collaboration just by using git on it own it requires that you have some method of obtaining the seperate branches on one machine before they can be merged. One way of doing this within a company is to set up a git server. 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 has programmers in different parts of the world a publicly accessible git server that is on the web may be a better solution. A good example of this it GitHub. GitHub provide 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. It shows that there are 74 contributers to the source code and 3245 indididual changes over 20 branches. With such a large development team the potential to have conflicting changes must be high. Having many seperate 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

According to Balcan [**?**]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 subsquenece doe not need to be in a single contiguos block however. Algorithms that solve the longest common subsequnce 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:

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:

```
public class SampleLCS {

  public static double area(double radius){
    return Math.PI * square(radius);
  }

 public static void main(String[] args){
   System.out.println(area(3));
 }

 public static double square(double num){
```

```
    return num * num;
  }
}
```

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

```
public class SampleLCS {

 public static void main(String[] args){
   System.out.println(area(3));
 }

 public static double square(double num){
   return num * num;
 }

 public static double area(double radius){
   return Math.PI * square(radius);
 }
}
```

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

```
public class SampleLCS {

  public static double area(double radius){
    return Math.PI * square(radius);
  }

}
```

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.

```
public class SampleLCS {

 public static void main(String[] args){
   System.out.println(area(3));
 }

}
```

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 [2] 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 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 [10]. This is achived

2.2.4 Patience

The patience algorithm instead of figuring out the longest common subsequence directly uses the longest increasing subsequence. When this is used with line numbers from the source code the longest common subsequence can be established. By examining only unique lines present in both copies of source code it is comparing it ignores items that are repeated multiple times. as they produce longest common subsequence could give a sub-optimal result. As Bram Cohen has pointed out in his blog there are instances where a LCS algorthm can return results that are not as helpful to a developer as they could be. Instaed 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.

2.2.5 Histogram

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

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 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.

2.2.7 How LCS is used in differencing tools

Often to speed up the differencing process each line is assigned a hash code depending on its contents. 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. 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 rather than the white spaces in the code.

2.3 Refactoring

A common concern with coding is the need to periodically refactor the code. Refactoring does not involve changing any of features the source code or change how the compiled program functions. 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 [6]. Similarly according to Kerievsky some of the motivations for refactoring include adding more code and understanding existing code [8]. 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:

```
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();
}
```

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:

```
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;
}
```

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 however. If we do not care about conversion between Celsius and Fahrenheit the refactoring simply adds to the amount of code we need to wade through before understanding what the code does. An alternate way of refactoring is as follows:

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

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. It depends entirely 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

Part of the inspiration for this tool come from JDime. If there is a conflict in a text based merge JDime provides a way for some refactored content to be merged.

2.4.1 What Jdime can be used for

JDime can be used to compare two different versions of Java source codes which have been refactored and to produce a copy common to them both.

2.4.2 How Jdime works

JDime instead of testing against a source code repository test against files in the system under the base, left and right directories. While this may be useful in quickly being able to show what JDime is able to achieve it requires that the inputs need to be previously extracted from a repository into the file system.

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.

2.4.3 Testing Jdimes suitability

Jdime has been written mostly in Java by Apel and Lesßnich [1] . There are a few exceptions including the linear programming libraries that need to be created. As we are attempting to combine some of their work with Git it was decided to use JGit rather than the C implementation of Git. As the Java implementation may run a bit slower then in order to get a good timing test running we need to run redo the tests of JDime using JGit instead.

Also the tests that Leßnich did on JDime were from files rather than from a repository. It is necessary to set the files back up in the original repository structure to get a adequate baseline.

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.

At the moment it cannot access a version control system so each of the revisions need to be set up as directories. Each directory needs a full copy of the source code for that revision This means that the necessary Java source code to be used by JDime in base, left and right directories.

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 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 f̈unctionally 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.

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. 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.

It is about this point in analysis that you begin to second guess yourself. For this reason the output of JDime was compared also with the merger

and the base. The order of the methods in the output did not match the order of any of the equivalent input files.

If it detects an unordered section it does not preserve the order of the output.

2.4.4 Resons why Jdime cannot currently be used to create seperate 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 this tool sounds 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 resulting code could be in a totally different order to any of the versions combined to create it.

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 is if there are conflicts. There could be advantage to determining if items that haven't conflicted in the text based merge but have moved from one position in the code to another. If a method has both been moved and changed in both branches it could have a conflict. This conflict would appear to the text-merge as a deletion agreed upon by both branches followed by two insertions at different points. It would not be picked up by the text-merge as containing a conflict even if one was potentially present. Since the conflict is not detected by a text-merge it is not set aside for testing by examining the AST

tree. In this case comparing AST trees could detect that there was a conflict. Although JDime is an improvement over a text-based merge this is a potential conflict that niether detect. There is a performance reason for this design decision. JDime can take a long time to determine if two files contain equivalent source code.

The final concern is that after JDime does the initial comparision 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 [5] introduced a plugin for eclipse that recorded information about refactoring in to version control so that it was eaiser 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 [7] 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 branchs as consistant as possible has not been discussed.

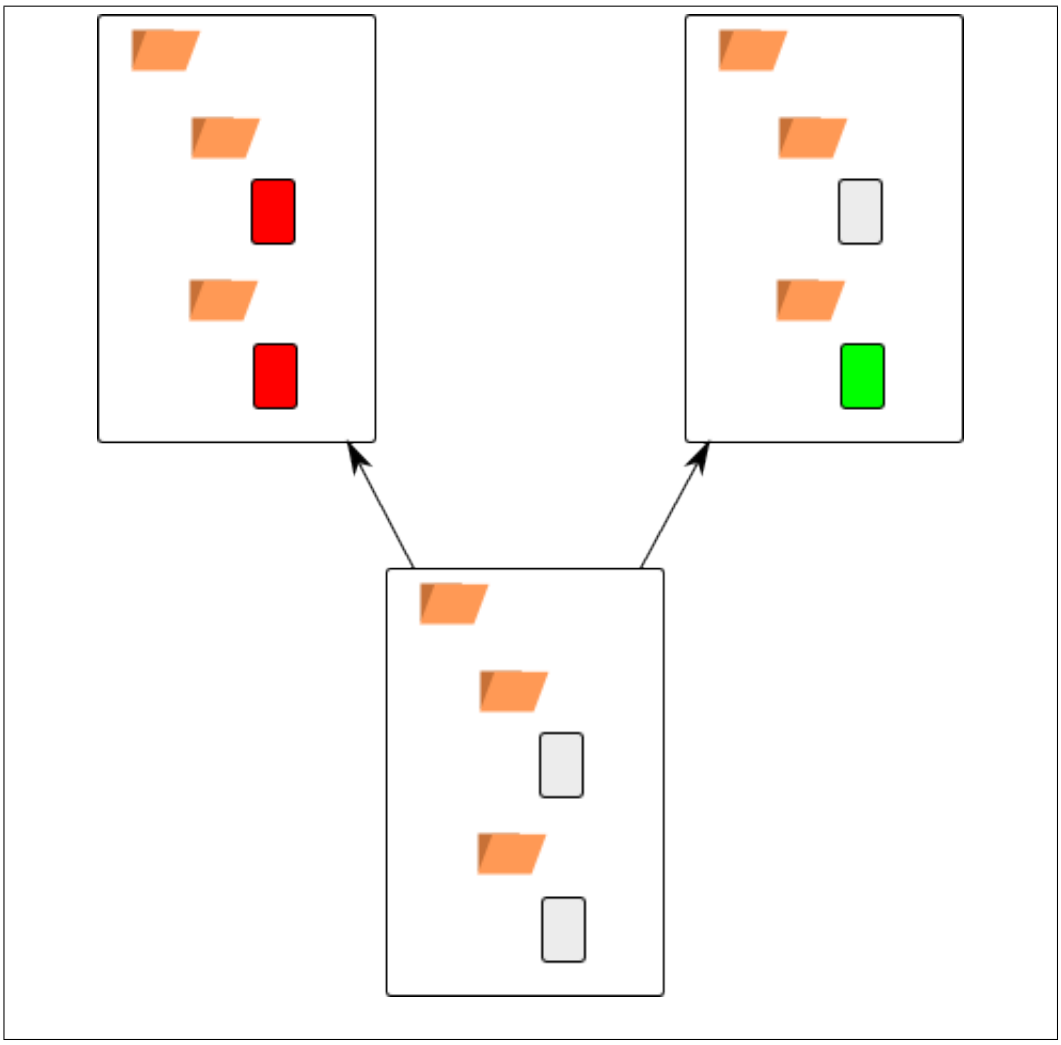

Figure 2.1: A project that has been split into two branches

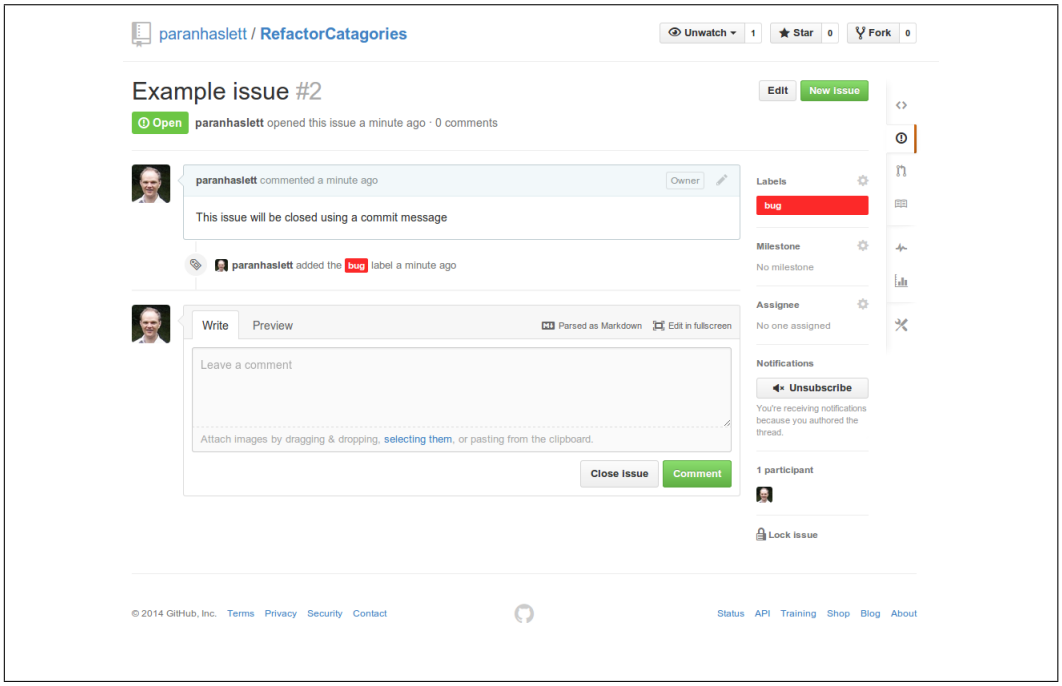

Figure 2.2: A issue that has been created in GitHub

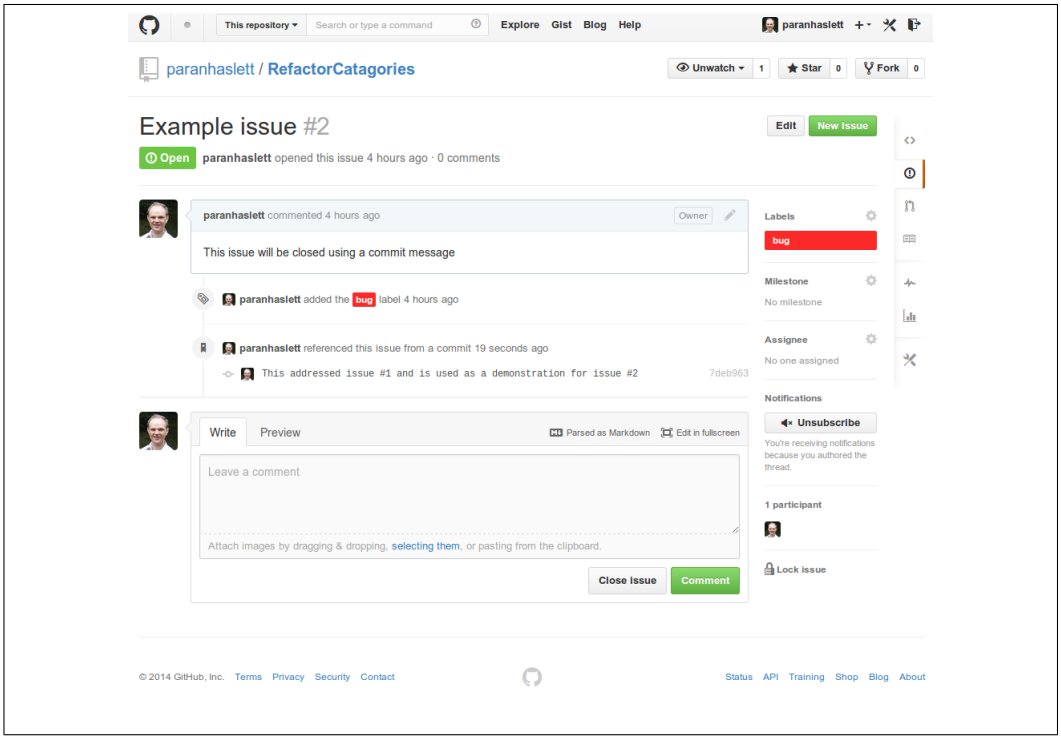

Figure 2.3: The issue has been updated from a commit message

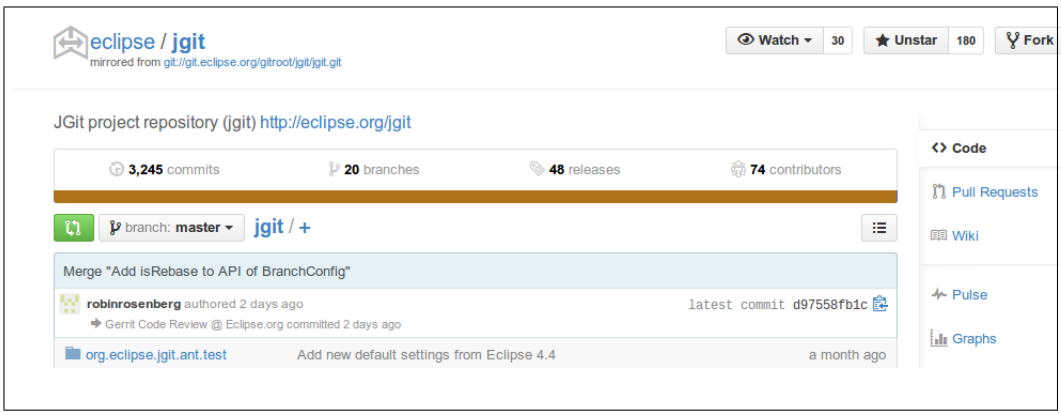

Figure 2.4: The number of contributers to the JGit project

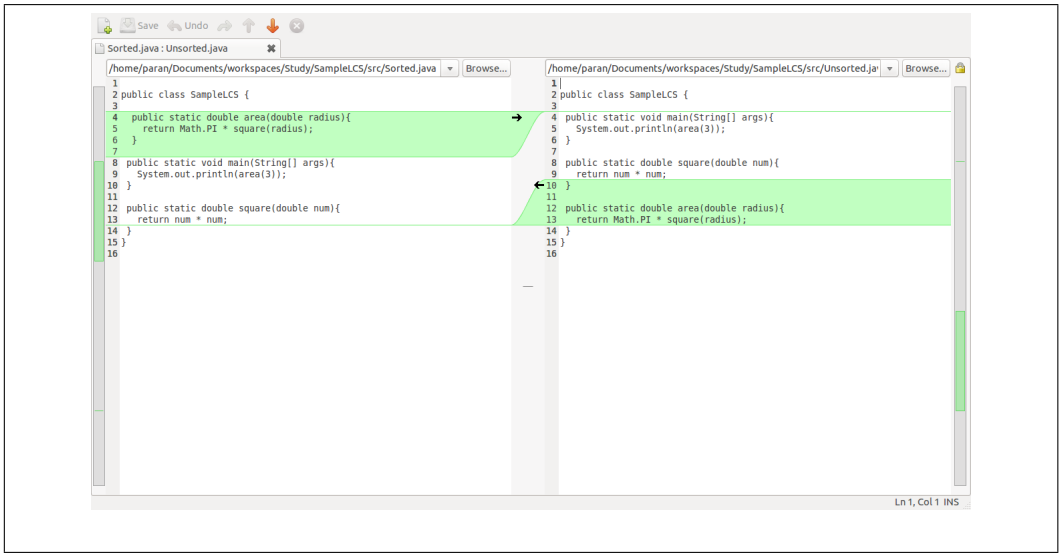

Figure 2.5: A graphical diff tool showing differences with two equivalent blocks of source code

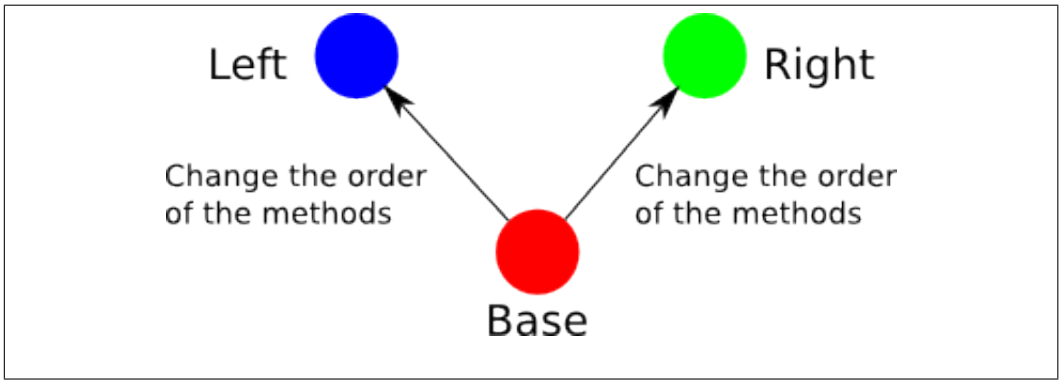

Figure 2.6: The setup for the test of JDime

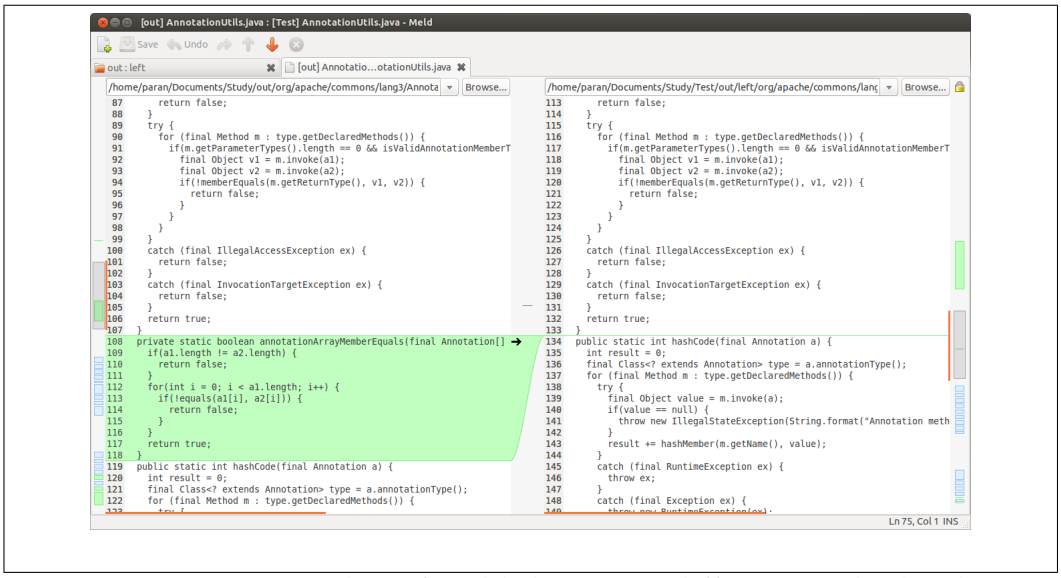

Figure 2.7: Screen-shot of Meld showing a different method order

Chapter 3

Individual refactored 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 independently refactored view of the source code could be valuable in these circumstances.

3.1 The problem

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 notice that you they are going to have to refactor the code before you add any of your changes. This would be a fair judgement call as Fowler claims that the main time to do refactoring is before making any changes [6]. You complete your changes and check in your code back into the version control system. While you are doing this other people have been working

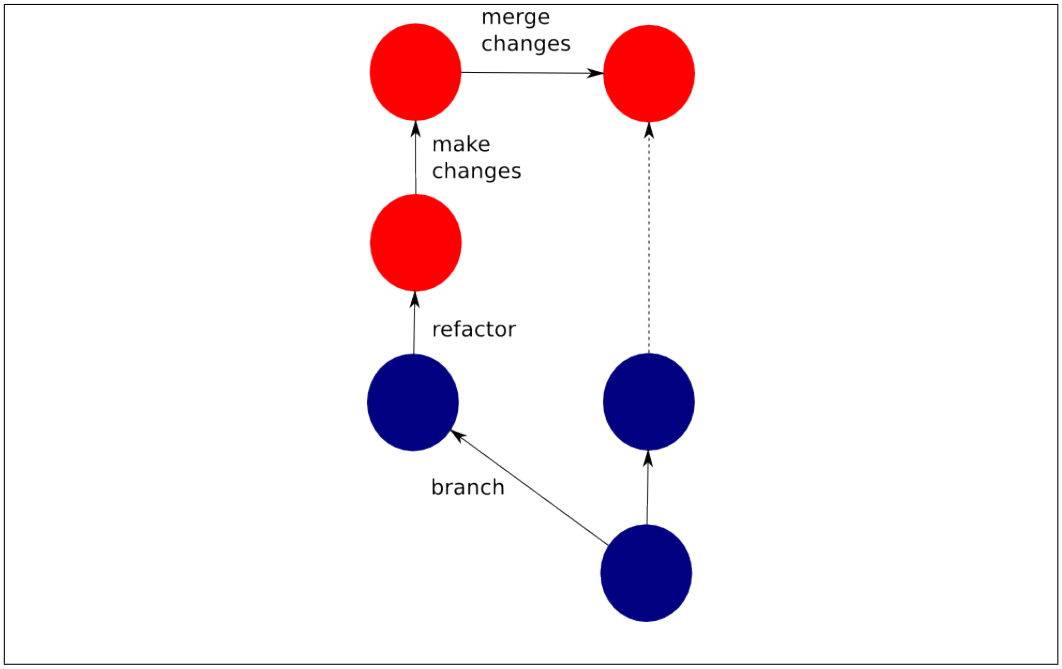

Figure 3.1: Merging changes with refactored code also merges any refactoring

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. Some conflicts that they experience could be because the changes you made directly compete with the changes you 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 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. There are some occasions where you may want to avoid changing other peoples code

in such a dramatic fashion.

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. [9] refactoring is a challenge when the code base is large. By definition refactoring does not 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 on 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 will be a large impact each time there is a commit. This is because the refactoring for the large refactored project is imposed upon others each time it is merged.

One of the ways this can be dealt with is by creating separate branches for different projects however this has some issues with merging when working on two or more projects simultaneously. In order to

minimise the amount of divergence it is advisable to merge each of the branches with the trunk often. If there has been global refactoring changes introduced by one project being merged the merge will be worse for any of the remaining branches when they are checked in. Instead of having an issue merging at the check-in level we now have an issue at the merging of branches. As these changes will occur less often than checking in the code, the code will possibly have more divergence. This divergence is likely to cause more rather than less merge issues at the expense of having to merge less often.

Differences in how code is understood. According to Kerievsky a reason for refactoring code is to better understand it [8]. 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 making a particular refactoring look a lot more complex [8]. 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 these 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 individual refactored views

We want to be able to maintain private views or separate branches that can have different but equivalent refactoring. Different structures of code that function the same way 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 software developer works on consistent. 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 differently refactored view it allows software developers to work on the same programming project to freely refactor with minimal interference to others. If two software developers refactor the code that they are able to hold their own individual refactoring with reduced changes when they are merged. The reduced changes would also mean that there will be less unnecessary 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 comment 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 branch and that are not necessary to share. This would allow a programmer

to keep a lot more personal notes. The risk of having so many comments from others that it is hard to see the code is reduced. By being able to identify that a block of text comment is a new addition

3.3 Ways individual refactored views could be achieved

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

Comparing differences using a non-ordered comparison algorithm. Instead of using the Longest Common Subsequece based approach we could instead use a non-ordered comparison. The easist 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. The items within the set will still be ordered.

```
while There are elements on both sides that match do
    Create a histogram of remaining elements;
    Select one of the elements that occurs in both sides the least but
    not zero times;
    while There is an element below the selected one that matches on both
    sides do
        Remove element from list of elements to compare, as it
        matches;
    end
    while There is an element above the selected one that matches on both
    sides do
        Remove element from list of elements to compare, as it
        matches;
    end
    Remove selected element from list of elements to compare, as it
    matches;
end
```

Algorithm 1: A non-ordered comparison algorithm

The problems with using this approach to reconcile to different ordered sets is that comparison whould 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 transformed into an agree upon format. Before allowing a merge on a block of code each the views could transform the block of code using the same method and compare it with the equivelent block of code that in the version control system. If the transformed code is the same

then there is no need to merge.

Storing additional infomation 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 [5].

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

Using a tool like JDime solely as a method of comparision. As mentioned earlier there are a number of reasons why JDime cannot currently be used as a method of keeping two different refactored 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.

Chapter 4

Refactor categories tool

In order to prove that having individual refactored views would provide some useful results we created a refator categories tool. The refactor categories tool examines the differences between files and arranges them into a greater variety of categories than used by version control systems or merge and diff tools. In addition to being able to detect instances of delete, insert and modify the refactor categories tool can also detect instances where the code has been relocated and to some extent renamed. 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 does not. 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 how the project behaviour.

4.1 What the tool does

By checking to see if an insert at one point matches a delete at another point it is possible to detect to see if the code block has moved. 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 or method. 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 guaramtee 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 a limited extent the refactor categories tool can also deal with renaming.

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.

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 in white-space.

4.2 Overview

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 EditList object. 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 can contain information about the line of source the AST node starts and ends at. Unlike the EditList object which we have previously discussed AST node can also hold the column that the AST node starts and ends at. Using this positional information we can match the text based changes to a set of AST nodes.

There is likely to be AST nodes that are not included in any of the change sets and can be safely ignored. In order to find just AST nodes we are interested in we need to traverse the AST. When we start at the root of the AST the start position for the root node is the first line and the first column. The end position for the root node is the last line and last column. 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.

In some instances items within the text change will not have a corresponding 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. 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 astetic one.

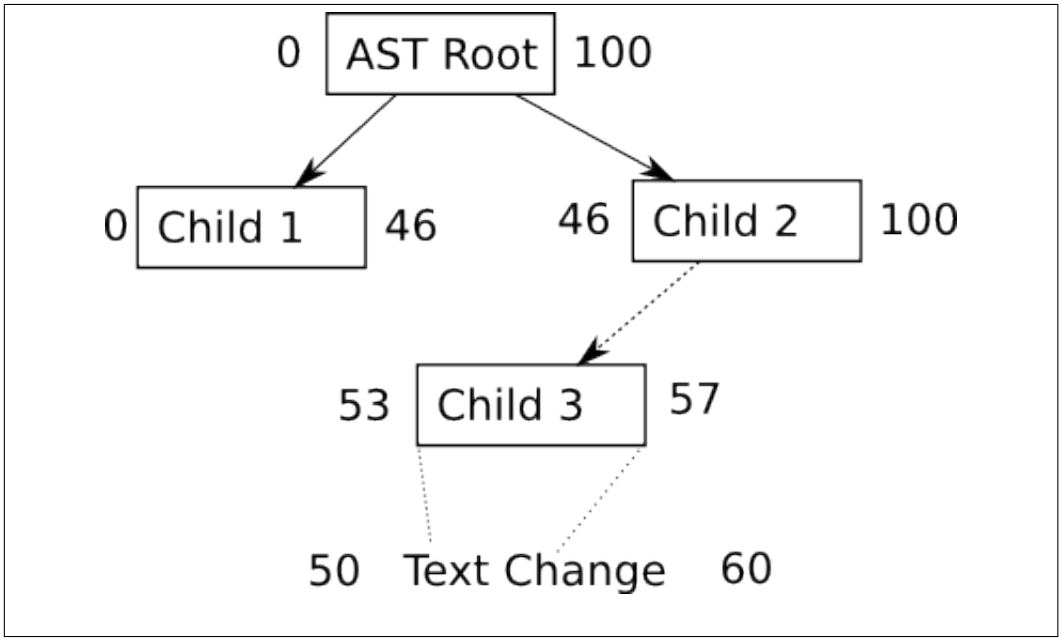

Figure 4.1: Finding an AST node we are interested in

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 based change that does not change the behaviour of the program.

If we can keep track of renaming

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 branch. This in turn could indicate that we would have less merge conflicts by reducing the overall number of changes.

4.3 Performance

As we are testing the complete revision history for several projects there is a high number of changes. This means that perfomance is an issue especally 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 collecter 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 analying 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 anaylisis as these nodes do not need to be analysed to see if they match.

By comparing only succsseive changes. There are a greater amount 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 less files.

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 hashMap. This ensures that only AST nodes that are a similar type are compared against each other

4.4 Design decisions

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 categorisation 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 categorisation 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 analyse 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 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 placed 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.

Chapter 5

Expirimental Results

Chapter 6

Future Work

Chapter 7

Conclusions

Bibliography

[1] 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.

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

[3] 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.

[4] 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.

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

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

[7] 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.

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

[9] 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.

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

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