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 they do not change 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 to change source control to be aware about items that do not change the functionality of the source code.

Acknowledgments

Any acknowledgments should go in here, between the title page and the table of contents. The acknowledgments do not form a proper chapter, and so don't get a number or appear in the table of contents.

Contents

Chapter 1

Introduction

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. *Version control systems* provide a way of mangaging the conflicting changes. One way they do this is by ensuring only one person can edit a file at a time. The problem with this is that one person stops other people 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 automaticaly merged are known as *merge conflicts*. The merge conflicts need to be manually checked and edited to make a merged file with the correct changes.

Internally the merge process need to determines 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 of the revisions against the common ancestor. This is often done as a linear comparision which work very well when there has been no change in the order.

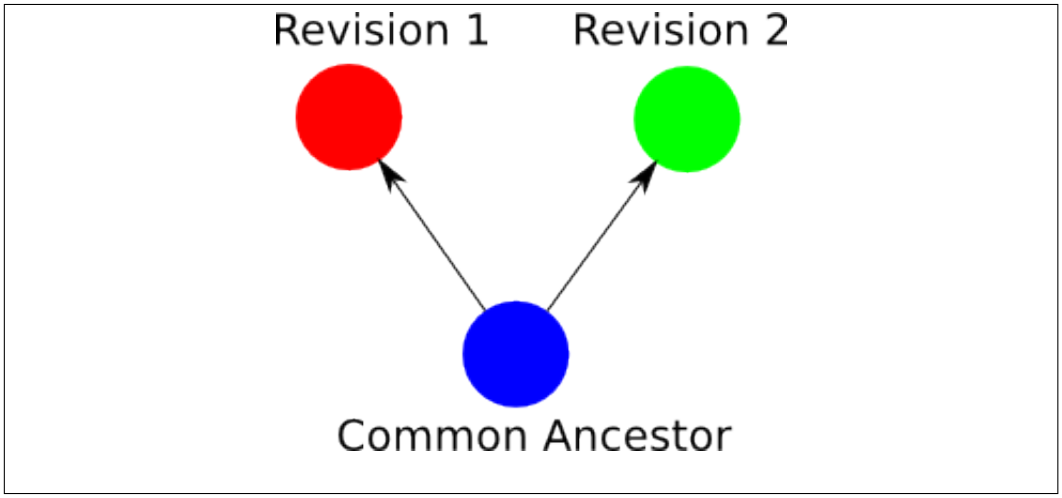

Figure 1.1: A file that has two different revisions

However, if there has been a change where a block of source code has been moved from one place to another a linear comparison determines that two changes have occured. This is equivalent to deleting a block of source code in the common ancestor and inserting that source code elsewhere. This becomes an issue because it is possible for a program to behave 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 could behave in the same manner as it did before the change took place.

Without any further analaysis 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 changing it unnecessarily could be

disconcerting.

If two different changes in the order of methods occur, one for each revision, it is possible to get a merge conflict in spite of the revisions being functionally equivalent.

This thesis explores a way of allowing a version source control system to detect when the source code has been reordered. It also introduces the concept of maintaining multiple seperate views of differently ordered but equivalent source code for the purpose of reducing the number of changes introduced in 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 difference between two files.

Chapter 2

Background

2.1 Version Control Systems

Version control systems are a way of managing different revisions (or versions). Most commonly, a version control system is used to maintain 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 expirimental feature, to associate additional documentation about a feature and to collaborate with mutiple developers on the same project.

Revist 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 emphtag. 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 expirimental features. It is also possible to maintain mutliple revisions of all the files in a project. This is useful if there is an experimental feature which you want to explore but want to maintain the original project. 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 a change or a set of changes to a document. The reason this is useful is that you can specify what the change was for. Instead of updating individual documents you could specify that changes over multiple documents were done for a single reason. For instance if you had a number of documents that had an address written in them and that address changes. 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 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.

Collaborate with multiple developers. The real power of a version control system is its ability to manage documents that are being changed by multiple people. In a multiple person systems it is possible to have individual revisions that contain each persons changes. The version control system then manages the way these changes are combined into a composite product. This 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 to 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 within the same environment there is a need to interact with each other. There is competition for access to the source code 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 there is a conflict about what the final can cause a conflict about how the final should be changed. 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. The advantage of this is that the file is always in a consistent state. The disadvantage of this 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 are changing distinctly different parts of the document or change the document in the same way this restriction is unnecessary.

This is how one of the original versioning 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[**?**]

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 conflict is known as a merge.

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 [**?**]. 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

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 3 way merging requires 3 different revisions are required, the changes you have made, the changes made by others, and the revision that is common to them both.

2.1.2 Architecture

Centralised version control In a centralised version repository all the changes are made to one location. This means that only one place needs to be checked in order to access the most up-to-date 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.

According to [**?**] centralised version control systems...

Distributed version control @

difference between git and mecurial reference

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.

2.2 Longest Common Subsequence

There are a number of issues in computer science that can be resolved by a longest common subsequence algorithm. It is very good at finding the differences between two different sets of ordered information.

2.2.1 Example

One method of discovering what has changed is to find the longest common subsequence (LCS). A simplified 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

There are a number of ways that the longest common subsequence is calculated. 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 [**?**].

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.

bran Cohen

because of the way a patience

Before the act

2.2.5 Histogram

A Histogram difference strategy is very similar to a patience algorithm Instead of looking at just the unique lines between any two subsets however it can examine lines that there are multiple copies of

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.

if

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 [**?**]. Similarly according to Kerievsky some of the motivations for refactoring include adding more code and understanding existing code [**?**]. 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

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

The work of Apel and Lesßnich has mostly been done in Java. 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 test if JDime is better than a text based merge we need to attempt to try something that would incorrectly cause a conflict in a text based merge. 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.

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 paper 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.

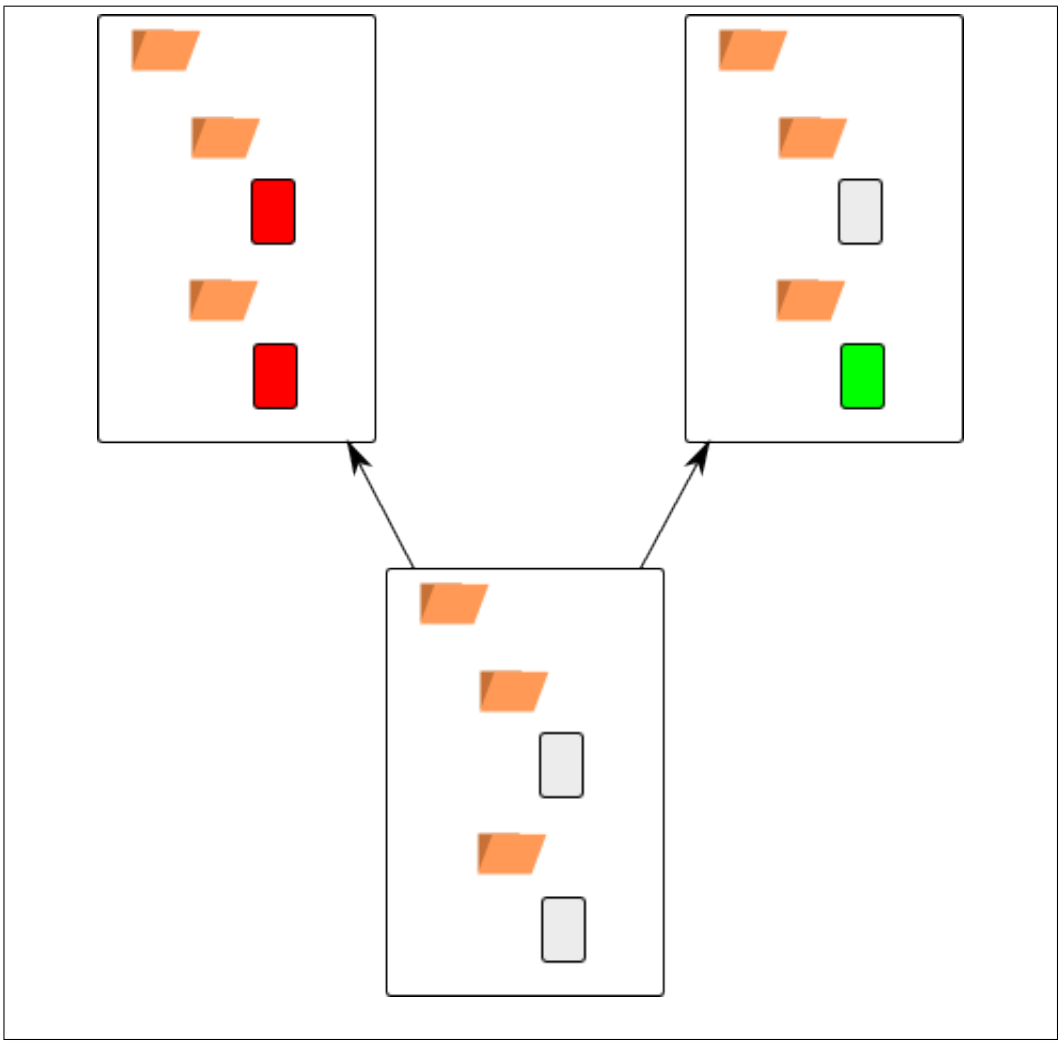

Figure 2.1: A project that has been split into two branches

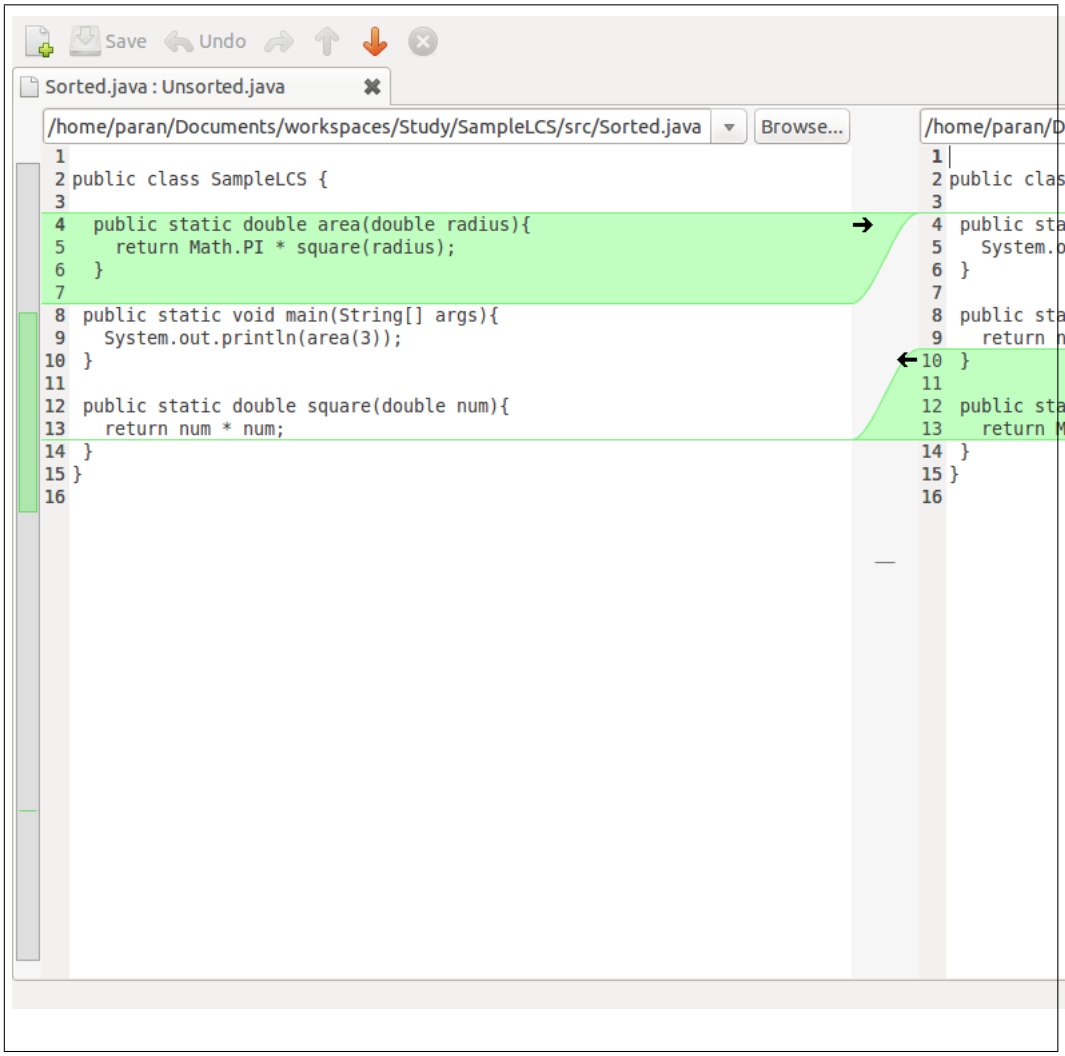

Figure 2.2: A graphical diff tool showing differences with two equivalent blocks of source code

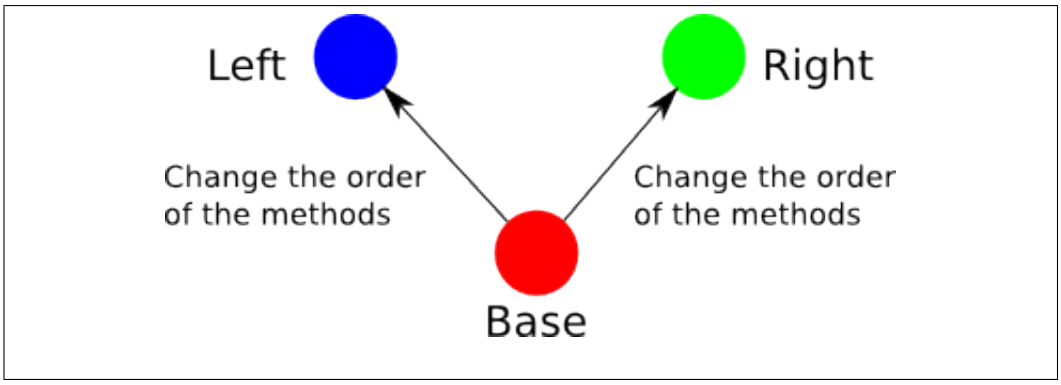

Figure 2.3: The setup for the test of JDime

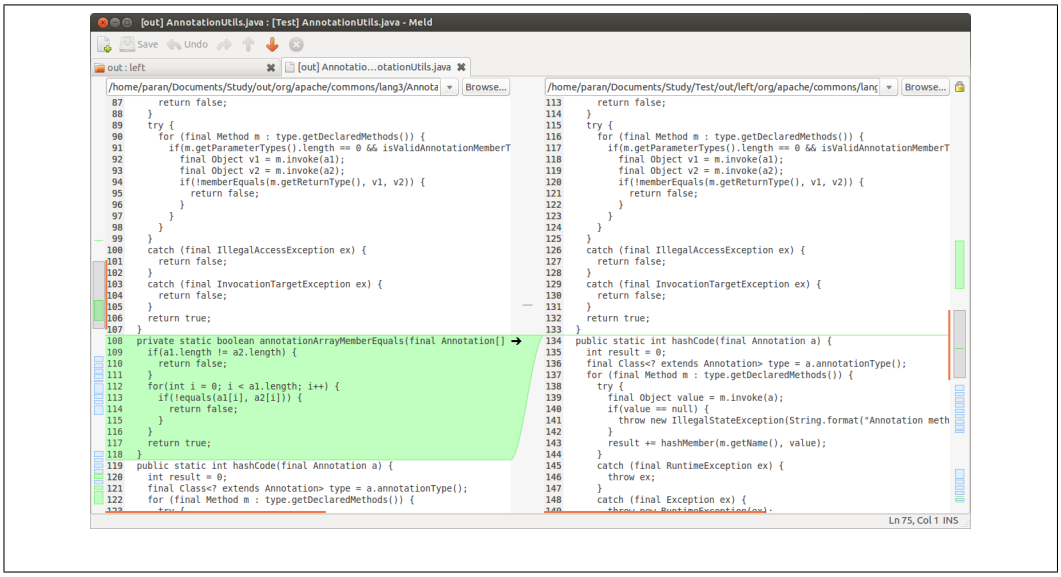

Figure 2.4: Screen-shot of Meld showing a different method order

Chapter 3

Refactoring Differences Tool

3.1 The aims for the tool

3.1.1 The problem

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 judgment call as Fowler claims that the main time to do refactoring is before making any changes [**?**]. 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 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

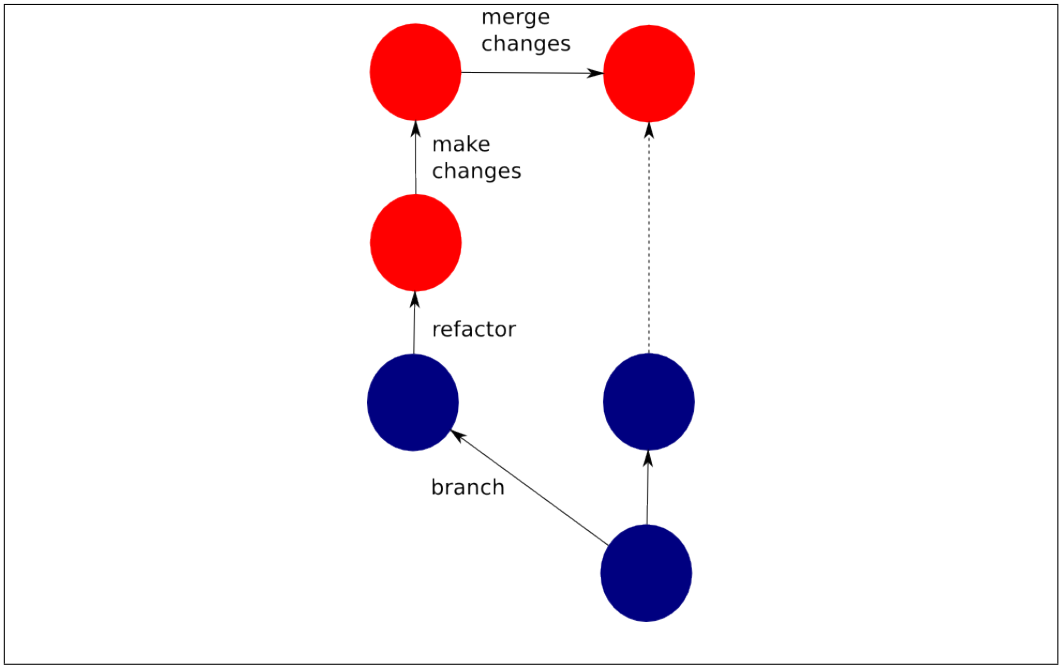

Figure 3.1: Merging changes with refactored code also merges any refactoring

refactoring that you have completed. This is because a refactoring often makes a large amount of global changes to the source code.

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 make a refactoring available to everyone. By checking in your refactoring code however, 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. This is a recipe for merge based bugs and time wasted doing unnecessary

merging.

When there is a large change on a separate branch it could be the case that there are multiple check-ins to keep other parts up to date and ensure that there is not too much divergence. 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 the repository each time it is checked in.

One of the ways this can be dealt with is by creating seperate branches for different projects however this has some issues with merging when working on two or more projects symaltaniously. In order to minimise the amount of divergence it is avisable 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.

One of these changes which is not catered for by current version control systems is the change 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.

3.1.2 Private 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 that you do not want

impact other developers. Maintaining your own independently ordered view of the source code could be valuable in these circumstances.

An example of this if you are working on a subproject in which you need to refactor. 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. [**?**] 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 is equivalent to the code after the partial refactoring.

3.1.3 How we would like to address the problem

We want to be able to maintain private views or separate branches that can have different but equivalent refactoring. When these branches are merged we want to make sure that any equivalent refactored code is not migrated to the view we are merging into. This is so that people working on the same project can freely refactor with minimal interference to others When two people refactor the code that they are able to hold their own individual refactoring with minimal change when they are merged. This also means that there will be less unnecessary merge conflicts when merging code.

We also want to be able to further classify more complex operations in a change set than insert, delete and modify. When JGit compares files with each other because they are structured it can determine if the file has been moved to a different location in the tree. At this level JGit can also detect copies and renames. However once JGit starts comparing source code it loses this structural information. By giving JGit some idea about the structure in the file it can determine these item at the finer granularity of sections of code rather than at a file basis.

We want to have views that can have different but equivalent refactor-

ing This is so that people working on the same project can freely refactor with minimal interference to others

3.2 What the tool does

The refactor categories tool enhances JGit by adding the concept of source code moving to JGits representation of changes. Currently JGit labels any of the differences it encounters as an insert, a delete, or a modification. 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. To a limited extent the refactor categories tool can also deal with copies and renaming.

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.

To a limited extent the refactor categories tool can also deal with copies and renaming.

To a limited extent the refactor categories tool can also deal with copies and 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.

3.3 How the tool works

The refactoring difference tool first works out which text has changed using the same method as JGit. This initial examination returns the differences based on a line by line basis rather than using a smaller granularity. This means that the set of changes found could still contain code that is

comparatively the same. The set of changes found in using the JGit histogram comparison are then evaluated. The reason for this is that some items of text could be in a differing order but still be a valid Java program

In order to resolve some limitations with JDime and the text only merge in GIT information about which line numbers are retained after the first text merge. In JDime these are ignored and the AST is relied upon to hold all the information. The change set has been taken from the original GIT 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 line numbers it is possible to figure out which AST items these changes affect. This is done loading the file into the JastAddJ parser to get an AST tree. The line numbers for each item in the tree are then comparing line numbers from the change.

There is a problem with using JastAddJ to do this. Some ASTs do not contain an equivalent position in the code. This could be because they are generated by the parser to reflect parts of the Java languge that are infered 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 a interface have a public type even if is not in the code.

To get aound this problem we have needed to record the previous valid position for a Java command as both the start and end of the AST that had no line numbers assigned to it. This means that the infered AST is in the right position but is not represented by a block of source code.

Rather than comparing everything with each other to determine matchs 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 relivant 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.

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 im using the position in the code to identify ASTs 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.

3.4 Design decisions

3.5 Limitations of the tool

the tool is based on changes not based on conflicts like JDime is. the drill down using line numbers

comments white-space

comparing ASTs the matcher and how using a score works finding the best match

3.6 Limitations of the tool

The refactor categories tool drills down only on the changes it is harder to investigate any change that has causes side effects in unchanged code. It is not often that a side effect will be purposely placed in the code as it reflects bad design decisions. This may however be an issue with bugs. This also means that the refactor categories tool will not be able to tell when some code is copied but the original remains unchanged. Instead it will assume that it is a completely new insertion of code.

Chapter 4

Expirimental Results

Chapter 5

Future Work

Chapter 6

Conclusions