Thesis Proposal

Paran D Haslett (300274241)

School of Engineering and Computer Science
Room C168C
Cotton Building
Victoria University of Wellington
Gate 6, Kelburn Parade
Wellington,
Email : Paran.Haslett@ecs.vuw.ac.nz

Abstract

When collaborating on a project there are times when the code diverges. This could be due to refactoring or code being reused in another project. It could even be due to throwaway 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. This is especially true if there is a piece of functionality that you wish to work on that differs from what everyone else is working on. 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 situation would be valuable. It enables the programmer to refactor or change the code with minimal impact on others. If the relationships between views are maintained it also could better recognise changes that need to be communicated to other views or branches.

Keywords: Personal annotations, Collaboration, Multiple views, Semantic Merge

1 Introduction

There are some issues that arise in Software Development when code diverges from a point. This divergence could be caused for a number of reasons including adding new functionality, refactoring, or code being reused. When source code diverges it could be helpful to retain relationships between the code that has diverged. In keeping these relationships it may be useful to identify source code that has the same intention and source code that is only relevant within a particular branch or view.

1.1 Refactoring

A common concern with coding is the need to periodically refactor the code. According to Fowler et al. the main time for refactoring is when new functionality is added (Fowler et al. 1999). Similarly according to Kerievsky some of the motivations for refactoring include adding more code and understanding existing code (Kerievsky 2004). 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 view 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 with 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.

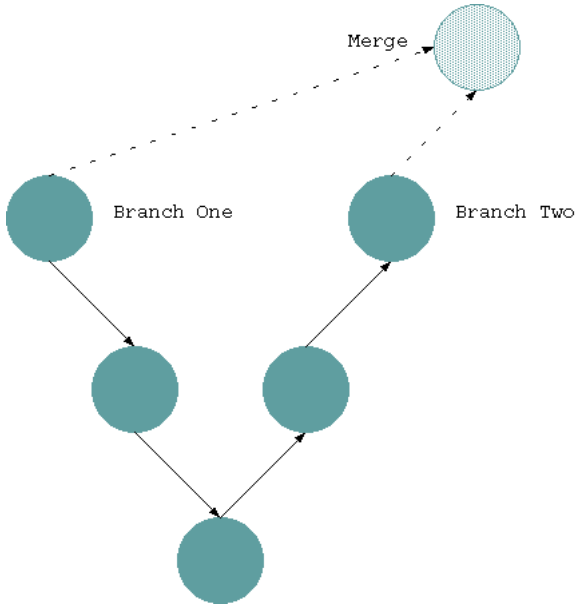
Although this was a simple example it is easy to imagine a case where a much larger refactoring process is undertaken. In such circumstances a merge becomes difficult.

1.2 Differences in how code is understood

Another reason that Kerievsky claims for refactoring code is to better understand it (Kerievsky 2004). 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 either developers tend to leave 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 (Kerievsky 2004). The different perspectives meant that the programmer he refers to as John having 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.

1.3 Working on large projects with multiple developers

Perry claimed that there are not adequate tools to use with projects with multiple developers (Perry et al. 2001). Whilst a lot of differences between two different versions of source code can be already merged by source control, refactoring often involves larger structural changes. According to Freese these larger changes can lead to merge problems if a traditional text difference approach is used (Freese 2007). Digg et al. have identified that merging after refactoring causes problems with some SCM systems and that tools that are more aware of semantic changes are necessary (Dig et al. 2008). This has lead to the development of smarter merge technology being used to compare code. The simplest of these being hunting for regular expressions so that white-space is not a factor during merging. This is already a feature on many merge tools. Some more complicated examples of this are presented by Leßenich and Apel who both identify in addition to text based merging both structured and semi-structured approaches to merging (Leß enich 2012) (Apel et al. 2011). Although Leßenich briefly talks about using GIT with a semi structured merge they have named “JDime” there are some interesting features of distributed version control system that have not been explored.



In the above diagram we have a situation where two branches have diverged in GIT and we would like to merge only the changes that would have a *functional difference*. We do not want to merge differences in branch one if they have been refactored but still perform the same function as branch two. There has been some success in using JDime to achieve this. JDime can do a refactoring based merge on the head of branch one with the head of branch two. As a refactoring merge is done from the head of each branch the code may have diverged a great distance. There may be benefits in instead doing a lot of micro refactoring merges, one for each check-in. As code has not diverged as far for a single check-in, the conflicts could be less severe and easier to automatically resolve.

Let us also consider if both branches in a merge have been refactored or dramatically changed. During a merge it would be advisable to favour one set of changes over another and then work through the conflicts. While this might be appropriate there could be times when further changes need to be made on the branch that contains the refactoring that was not so favoured. This would be true if there were periodic updates of functionality. When the next merge occurs for the diverging branches we could find ourselves having similar merge conflicts.

2 Related Work

The ability to have two differently refactored views of a functionally equivalent piece of source code is influenced by a number of software practices and tools. Some of these allow different views to interact by requiring them to adhere to certain standards and some are more flexible. The challenge is to allow flexibility while communicating what a change to one view should mean to the other.

2.1 Formal architecture and design

An approach that has been tried in an effort to manage different points of view is by ensuring that the source code has a strictly agreed upon architecture. The architecture could change dramatically when refactoring in order to add further functionality. According to Tang et al. a reason you might want to change an architecture is that is often hard to determine an appropriate design before something is implemented (Tang et al. 2009). A dramatic change to the architecture could mean a greater number of people who are affected by that change. The people who are affected may have to revise their understanding of the new structure even if it still performs the same function as before. It also requires the intervention of an architect and assumes that there is if not one right way to represent code that should be chosen and agreed upon. Taylor et al. claims that there will always be the need for design (Taylor et al. 2007). What this view overlooks however is that a different perspective or different design may be better suited to a different task. Even Taylor et al. spends quite some time discussing different perspectives and paradigms but does not develop this further into an understanding that there could be equivalently well designed artefacts for the same purpose that differ (Taylor et al. 2007).

2.2 Big Ball of Mud

According to Foote throwaway code is a major cause for the most often coded Big Ball of Mud pattern (Foote 1997). Some of this could be made worse by source control merges not being able to identify the throwaway code. In practice these are normally marked with a comment to manually indicate that this code is irrelivant or temporary and needs revision. The source control system is not aware about these comments and does not identify the throwaway code. Quick and dirty fixes can make their way from branches into the main stream without being checked to see if they are wanted.

2.3 Code Equivalence

The ability to detect code that has the same functionality is especially relevant to this thesis. Park et al. have suggested one way that this could be achieved however this may be too course grained (Park et al. 2013). Park suggests using some very basic rules to eliminate items in the compared source code that have very little impact on how the program functions. An example would be changing all string arguments ("string") into empty strings ("") before comparing the code. This requires ignoring items that could be relevant later. Using Parks method still may be a good initial step to determine if code blocks underwent large structural changes. The semi-semantic merge suggested by Leßnich and Apel may be a better approach (Leß enich 2012) (Apel et al. 2011).

2.4 Design patterns

Design patterns are another way which has been used to identify the best code for a particular issue. Due to the dynamic nature of source code however it may be useful to consider different design patterns for differing tasks. Designing according to patterns without making a clear judgement call about if the pattern is necessary could lead to a condition Kerievsky calls "Patterns Happy" (Kerievsky 2004). This refers to code as being over-engineered. This means there could still be some dispute about the correct pattern to use in a certain situation. Therefore it is possible for code to diverge even if patterns are used.

2.5 MViews

MViews as described by Grundy is a way to focus on a small subsection of code in a more graphical way (Grundy & Hosking 1993). This is of interest to us because it is similar to having two independent views of the code. Refactoring could change a number of widely separated files. This makes it harder to see if such a tool could be used to show two views that have entirely the same functionality have been refactored differently.

2.6 Reuse

Another reason you may want to diverge from a point is when the code is being reused. Tracz claims that often it needs to be refactored to fit in with the new code (Tracz 1995). Once it has been refactored any further changes in the original code need to be manually propagated. Normally this is how you would expect a typical reuse scenario if the code can be used without the refactoring. If we consider one form of reuse being the use of library package however sometimes when the library package updates the programmer may want to use the new library. Having the ability to make updating a smaller block of code smoother in spite of it being refactored may be desirable.

2.7 Version Control

A way to maintain divergent code that is currently being used is version control. According to Bertino it is possible to keep a smaller more easily deployed repository it is necessary to evaluate what is necessary and what is unnecessary (Bertino 2012). 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 merge conflict issues. These issues have a far greater chance of occurring if there is a dramatic change such as refactoring. By using a semi-semantic merge as proposed by Apel these issues can be reduced (Apel et al. 2011). The idea of a refactoring aware version control is not new as Freese also proposed a prototype eclipse plug-in for making the version control aware about refactoring code (Freese 2006). What this paper will investigate is if by maintaining information about equivalency relationships and by examining the changes between subsequent check-ins if merge conflicts are further reduced.

3 Proposed Approach

The ability to represent similar functionality in two different ways could allow developers to refactor for different purposes. Maintaining relationships between the refactored views may reduce the impact of merging two divergent branches.

Evidence of this could be found by a reduced number of conflicts. What this could mean for developers in a multi-developer environment is that the branch of code that they work on could be in a more consistent state. This would hopefully reduce the time spent trying to comprehend changes that another developer has made.

Being able to provide manual instructions and hints about how the source control should treat set blocks of code needs investigation. If a record is kept of conflicting items that are marked as being functionally equivalent they do not need to be included in merge. The simplest of these would be comments that have been marked as being functionally irrelevant. If there is a conflict with the comment in one branch with a line of code in another branch, their could be an indication about which should be preferred. Another reason to mark something as functionally irrelevant might be if there is throwaway code that you do not want propagated over your whole version control system.

One way of identify functionally equivalent code would be to use annotations in the source file as indicators. A unique key would possibly be required if the block of code was functionally equivalent to differing code in other branches. A different annotation may need to be used to indicate that items such as code for debugging purposes, throwaway code or comments specific to a view are not merged. The benefit of this is that it can eliminate unnecessary merging when either the code has been marked as having same intention or if the code is irrelevant in other branches.

A way to keep track of irrelevant code would be to create a third branch which only pulls down changes and is never used to commit changes. Although this would be suitable for the situation where there was throwaway code, comments or debugging it would be harder to maintain without additional tooling. It would also be harder to manually identify equivalent code segments that do not need to be merged.

There is already some refactoring aware merge tools that have been researched. An example would be the "JDime" system proposed by Apel (Apel et al. 2011). Most of these however, are general purpose and have been created as difference or merge tools that can be plugged into various version control systems. What this could mean is that using the progressive changes in a distributed system has not been explored.

The proposed approach is to look at the code associated with Git and JDime and figure out how we can relate each step in git to a merge. Using JDime and Git as a starting point see anything can be gained by using the additional steps rather than just merging the top items in a single merge . This should not be as computationally expensive as it sounds if the semi-structured approach set forward by Leßnich and Apel should mean that this is only necessary for conflicting items (Leß enich 2012) (Apel et al. 2011). By merging some of the refactoring aware functionaility of JDime with GIT we plan to further reduce the amount of merge conflicts.

This paper will test view that being able to automatically discern and manually mark items as being related between two will reduce the amount of conflicts that occur. Initially the automatic marking all comments as being irrelevant will be tested. The results will be compared against the "JDime" system without any additional features. As things progress this could give us a clearer indication about what things could be manually marked as being equivalent.

One of the things that will be investigated is if the changing of variable or method names is possible. This is going to be a challenge as if the method is or variable is not private or local the use of it could be spread over a number of files. For this reason initially only private methods, private variables and local variables will be considered. Investigation needs to be done to see if some of these changes are already covered by "JDime".

This project will also test if there are any advantages in comparing on a check-in by check-in basis rather than simply comparing the most recent check-in for two branches against each other. Again this will be done by examining if the total amount of conflicts can be reduced. Some of the conflicts will also be manually examined to see if they are simpler or more simply resolved using this process. This could identify if the maintenance of the version control system could be simplified.

References

Apel, S., Liebig, J., Brandl, B., Lengauer, C. & Kästner, C. (2011), Semistructured merge, *in* 'Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11', ACM Press, New York, New York, USA, p. 190.
URL: *http://dl.acm.org/citation.cfm?id=2025113.2025141*

Bertino, N. (2012), Modern version control, *in* 'Proceedings of the ACM SIGUCCS 40th annual conference on Special interest group on university and college computing services - SIGUCCS '12', ACM Press, New York, New York, USA, p. 219.
URL: *http://dl.acm.org.helicon.vuw.ac.nz/citation.cfm?id=2382456.2382510*

Dig, D., Manzoor, K., Johnson, R. E. & Nguyen, T. (2008), 'Effective software merging in the presence of object-oriented refactorings', *IEEE Transactions on Software Engineering* **34**(3), 321–335.
URL: *http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4509441*

Foote, B. (1997), 'Big Ball of Mud'.

Fowler, M., Beck, K., Brant, J., Opdyke, W. & Roberts, D. (1999), *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Professional.
URL: *http://www.amazon.com/Refactoring-Improving-Design-Existing-Code/dp/0201485672*

Freese, T. (2006), 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', ACM Press, New York, New York, USA, pp. 953–956.
URL: *http://dl.acm.org.helicon.vuw.ac.nz/citation.cfm?id=1134285.1134461*

Freese, T. (2007), 'Operation-based merging of development histories', *Refactoring Tools (WRT'07)* p. 47.
URL: *http://www.eecs.tu-berlin.de/fileadmin/f4/TechReports/2007/2007-08.pdf#page=55*

Grundy, J. C. & Hosking, J. G. (1993), 'Constructing multi-view editing environments using MViews', *Visual Languages, 1993., ...*
URL: *http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=269600*

Kerievsky, J. (2004), *Refactoring to Patterns*, Addison-Wesley Professional.
URL: *http://www.amazon.com/Refactoring-Patterns-Joshua-Kerievsky/dp/0321213351*

Leß enich, O. (2012), 'Master Thesis Adjustable Syntactic Merge of Java Programs', (February).

Park, S., Ko, S., Choi, J. J., Han, H. & Cho, S.-J. (2013), Detecting Source Code Similarity Using Code Abstraction Categories and Subject Descriptors, *in* 'Proceedings of the 7th International Conference on Ubiquitous Information Management and Communication - ICUIMC '13', ACM Press, New York, New York, USA, pp. 1–9.
URL: *http://dl.acm.org/citation.cfm?id=2448556.2448630*

Perry, D. E., Siy, H. P. & Votta, L. G. (2001), 'Parallel changes in large-scale software development: an observational case study', *ACM Transactions on Software Engineering and Methodology* **10**(3), 308–337.
URL: *http://dl.acm.org/citation.cfm?id=383876.383878*

Tang, A., Han, J., Vasa, R. & Architecture, S. (2009), 'Software Architecture Design Reasoning: A Case for Improved Methodology Support', *IEEE Software* **26**, 43–49.

Taylor, R. N., Hoek, A. V. D. & van der Hoek, A. (2007), Software Design and Architecture The once and future focus of software engineering, *in* 'Future of Software Engineering (FOSE '07)', IEEE, pp. 226–243.
URL: *http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4221623*

Tracz, W. (1995), *Confessions of a Used Program Salesman*, Addison Wesley Longman.
URL: *http://www.amazon.com/Confessions-Used-Program-Salesman-Tracz/dp/0201633698*