Jacob Ward: jward23; Evan Blajev: eblajev, Sarah Zhou: sarahsz, Julius Christenson: juliusc, Conner Knight: connerk

Commit-min: Minimize Bug Fix Commits

Problem & Motivation
Commits in a version control system should have a single purpose. However in the real world, commits often include extra code or refactorings beyond the purpose of the commit. For example, while a developer is working on finding a bug, they may also clean up surrounding code so it is easier to read and reason about. Once the bug has been fixed, the commit will include the code that fixed the bug as well as the extra refactoring. Ideally Commit-min would commit only the lines of code that fixed a bug.

When a developer needs to improve an old feature or revist a bug fix, they search the commit history, but with refactorings included in bug fix commits they often need to spend a large amount of time sifting through version control system diff statements to isolate the code that is significant. Commits consisting of only the necessary changes for a bug fix would improve a developer's workload since less time would need to be spent trying to understand old code, as it is more compartmentalized into individual, and manageable commits.

Current Approach
Git provides a couple methods for splitting up commits. The first is for unstaged code using "git add -patch" which interactively allows a developer to select "hunks" of code to stage for commit. The second is for code already committed using "git rebase -i" which allows the developer to walk through old commits and split them using the first method. These both work well if the developer already know how to split the commit, but the only approach to minimize bug fix commits currently is doing it by hand. The developer must go through and remove lines of code to see if they cause the desired test to fail. This approach is limited because it takes precious developer time, and if this could be done automatically, it would be beneficial. Because this is usually a pretty quick task, it may not seem obvious that this is a problem to be solved, but over time this can lead to hours wasted.

Our Approach
Commit-min will automate the bug minimization search process, freeing the developer from needed to comb over individual lines of code. Each bug that is fixed will target one or several unit tests, and our tool will attempt to minimize the number of lines that are needed to continue passing the tests. This test is one that was failing before the bug fix, and passing after the bug fix.

From a high level, our tool will be a script that runs from a Git pre-commit hook. The script will then run a three phase algorithm to minimize the bug fix. The first phase will determine which sections of the code are reached by the test. The set of lines under consideration will be narrowed down by executing a dynamic code profiler. Commit-min will only consider lines of code that were touched by the test.

The second phase looks for lines that are refactored and removes them. We will not try to determine all cases of logically equivalent code, but hope to tackle a few obvious cases. One case is easy formatting changes such as change in whitespace or change in brackets. Another case would be simple method refactoring. This would be if a block of code is pulled out into a method that is then called in replacement of the code. Figure 1 illustrates this type of refactoring. Finally, we could remove all comments. This part of the project is relatively open ended and many other optimizations could be added if time permits. After discussing, we found some tools that may help us in finding if a commit is a refactor. A research approach we took was to take a look at plagiarism detection tools. In the technical report, *A comparison of plagiarism detection tools*[1], we found a couple of open source tools that may be helpful such as

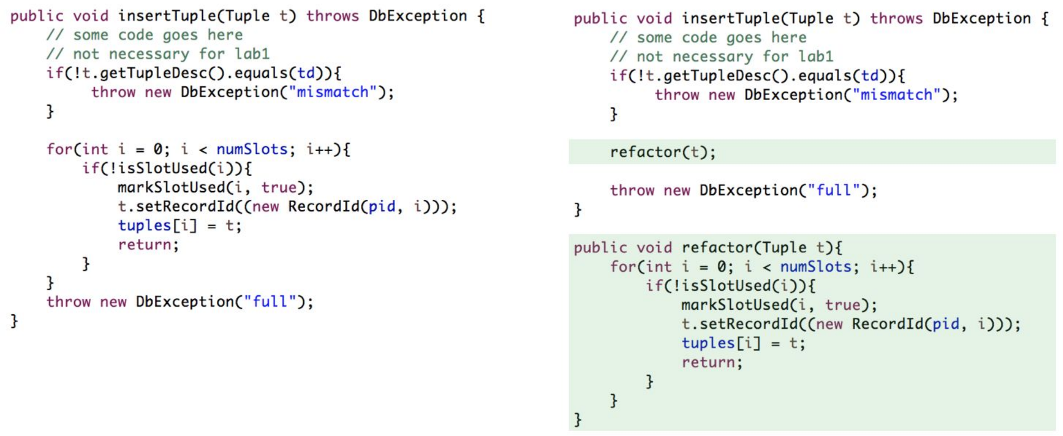
[1] Jurrian Hage, Peter Rademaker, Nike van Vugt, "A comparison of plagiarism detection tools", *Technical Report 2010.* Untrecht Univeristy, Department of Information and Computing Sciences, 2010, ISSN 0924-3275

SIM(supports C, Java, Modula-2, Lisp, Miranda) and Plaggie (supports Java), which detects similarities by source code tokenization.

The third phase will remove subsets of changed lines until it finds the largest subset of lines it can remove without causing the test to fail. An approach to determine the 'minimum change set' could be delta debugging, which is essentially takes a binary search approach to removing lines and seeing if they cause a failure. A tool that could be potentially useful for our delta debugging is AutoFlow[2], which uses a delta debugging algorithm to determine a minimal set of faulty changes, as well as automatically reduce a large portion of irrelevant changes in an early phase of development. Once this is done, the tool commit the minimized change set, add the removed lines back, and commits again. This dual commit is illustrated in Figure 2, which showcases how the the changes will be temporary duplicated so that they can be reapplied after the first commit.

Commit-min aims to be successful by combining three simpler minimization steps into one cohesive tool. Each phase of the algorithm on its own would provide the developer with useful information to determine which lines of code were involved in the bug fix. Phase 1 informs the developer where to begin the search for the bug fix, phase 2 eliminates changes that are basic refactorings, and step 3 experimentally determines which lines are unnecessary. Combining three relevant approaches into one systematic tool should increase the speed of the delta debugging since less lines are under consideration and improve the overall accuracy of the minimized commit.

Figure 1, Phase 2: Refactoring that would be caught and removed



```
public void insertTuple(Tuple t) throws DbException {
    // some code goes here
    // not necessary for lab1
    if(!t.getTupleDesc().equals(td)){
         throw new DbException("mismatch");
    }

    for(int i = 0; i < numSlots; i++){
        if(!isSlotUsed(i)){
            markSlotUsed(i, true);
            t.setRecordId((new RecordId(pid, i)));
            tuples[i] = t;
            return;
        }
    }
    throw new DbException("full");
}
```

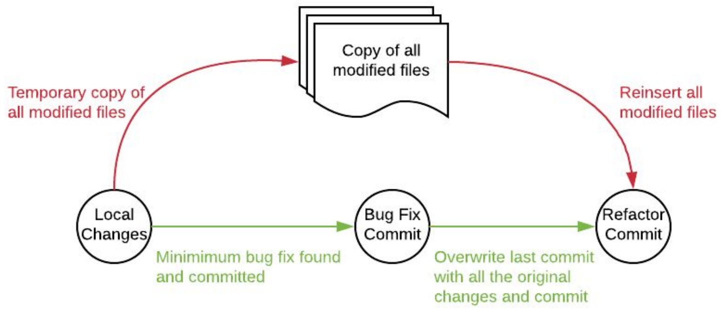
```
public void insertTuple(Tuple t) throws DbException {
    // some code goes here
    // not necessary for lab1
    if(!t.getTupleDesc().equals(td)){
         throw new DbException("mismatch");
    }

    refactor(t);

    throw new DbException("full");
}

public void refactor(Tuple t){
    for(int i = 0; i < numSlots; i++){
        if(!isSlotUsed(i)){
            markSlotUsed(i, true);
            t.setRecordId((new RecordId(pid, i)));
            tuples[i] = t;
            return;
        }
    }
}
```

Figure 2, Phase 2: Commit splitting



Copy of all modified files
Temporary copy of all modified files
Reinsert all modified files
Local Changes
Bug Fix Commit
Refactor Commit
Minimimum bug fix found and committed
Overwrite last commit with all the original changes and commit

[2] Sai Zhang, Zhongxian Gu, Yu Lin, Jianjun Zhao, "AutoFlow: An automatic debugging tool for AspectJ software", *Software Maintenance 2008. ICSM 2008. IEEE International Conference on*, pp. 470-471, 2008, ISSN 1063-6773.

Challenges and Payoffs
Since our approach consists of removing lines and re-running tests to determine which lines of code, if any, can be removed without affecting functionality we see a scalability problem arising with large commits. There are potentially many combinations of line removal that would still allow for the tests to pass. For example, removing line number 3, 4 & 5, vs 3 & 4, vs 4 & 5, vs 3 & 5. With this type of checking, large commits would take exponentially longer to fully check and resolve.

Designing a solution that runs efficiently on large commits will be the most difficult part of development. To reduce the problem space we could create an algorithm that does not try every single possible combination of line removal, such as delta debugging. It may rely on a heuristic to choose potential lines for removal. This approach may improve runtime but could introduce suboptimal minimizations because no heuristic is perfect and not every superfluous line will be removed. We will be able to manage our risk by removing any additional planned optimizations.

Successful implementation would result in high quality minimizations. Since running the tests with removed lines of code is a sure way to isolate the bug fix, if this process is time efficient, it will allow for narrowing down the scope of the bug fix further than code profiling and refactor detection could do alone.

Commit-min aims to automate the bug fix detection process in order to free up developer time before the commit, and after. If developers were previously spending time minimizing commits by hand, they will no longer need to do so. More dramatically, commit logs will be cleaner for future use. A developer will now spend next to no time in the future attempting to isolate code if they need to modify the bug fix, since it has automatically been split into its own commit message. This would result in hours of saved time, making a difference by freeing up developer time.

Development Schedule

Week	**Goals**
3	-Design data structures and algorithms for our modules and how they will interact with each other. -Setup the basic framework for our code such as necessary files, function headers, and comments.
4	-Build and test git diff parser -Implement test coverage tool to reduce lines to consider -Implement basic refactoring reverts (whitespace, renames, comments)
5	-Complete early prototype, will not have subset removal of lines yet -Create test suite for test coverage and refactoring tools
6	-Have others use our tool on their commits and get feedback -Evaluate the tool ourselves by running it on different scenarios -Implement basic subset line reverting
7	-Build more complex refactoring algorithms (method refactor, lines moved around) -Add heuristics to speed up subset line reverting and test
8	-Add functionality for user to control tool execution (run longer, choose whether to split commit) -Create high level how to use documentation
9	-Have others use our tool with new user controls and give feedback

	-Implement feasible feedback from users
10	-Evaluate tool over many examples to determine how effectively in minimizes commits -Build tool for deployment

Cost and Testing Experiments
What we indicated in our timeline is how much this is going to cost in terms of effort and time from our group. We will also need the use of other tools like git pre-commit hooks, as well as possibly some sort of UI for our project. The time that this will take is going to hopefully be around 8 weeks to finish the project, then polishing up and testing for around 2 weeks. So 10 weeks in total.

We plan to do multiple experiments for various code projects, analyzing their commits and testing across different sized commits. Some projects that we thought would be good to try are https://github.com/tabler/tabler.git, https://github.com/ReactiveX/RxJava.git, and https://github.com/google/guava.git. To do this, we will analyze commits from these projects and run our tool against them in hopes to test how accurate, precise, and fast our tool is. In addition, through analyzing different sized commits we can figure out which lines of code were possibly responsible for fixing the bug. This would specifically look at how many lines of code we missed that were a part of solving the bug. For example, if the bug fix was 4 lines, but we thought only 3 of those lines solved the bug that would be a 75% accuracy - hopefully through our approach this number would be 100% for all sizes of commits.

Additionally, we would test how specific we can get our prediction for the section of the code that solved the bug. For this, we would be testing the multiple of the number of lines it takes to solve the bug by the number of lines we predict the bug is solved in. Say, for example, the bug was fixed in 4 lines, but we include those 4 lines plus a 5th line, we would consider that 20% over - this is something that we will probably work the most on trying to get low, seeing as it is the crux of our entire project essentially - the ideal/complete success would be to get this to 0%, but realistically we will want to balance the time it takes to run the tool vs getting as precise as possible in our estimation of what lines caused the bug fix.

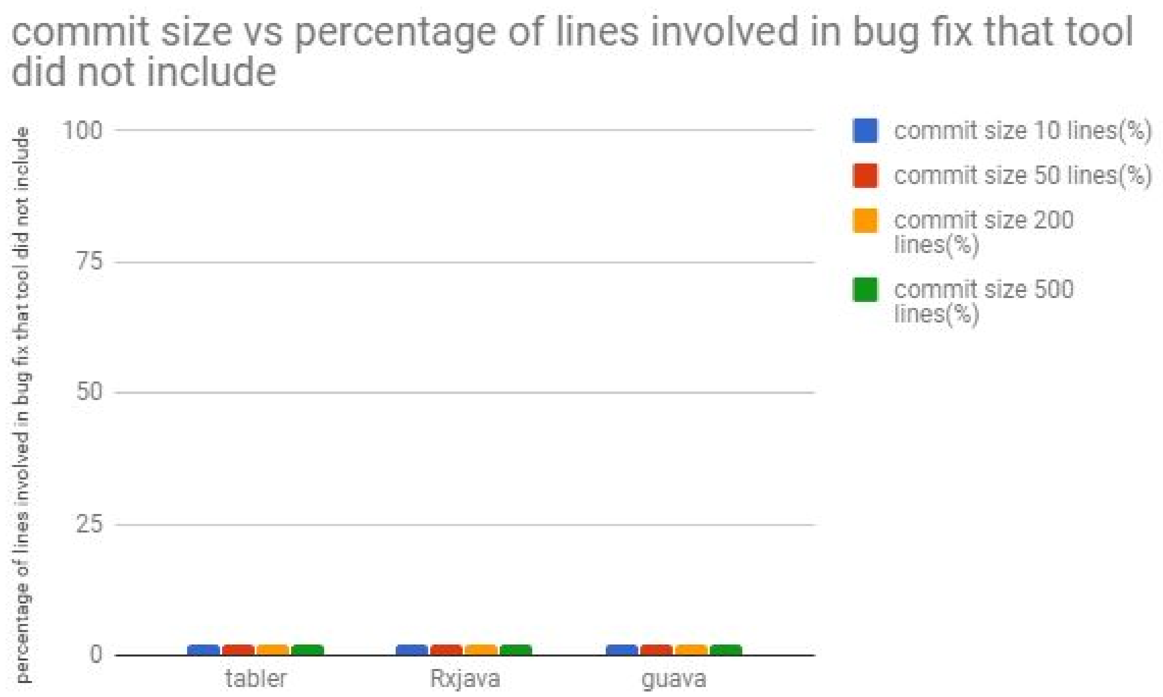
The last test would show how the amount of time in seconds it takes us to figure out which lines of code were responsible for fixing the bug vs the size of the commit(in lines of code). This amount of time will probably be, depending on our strategy of narrowing the code that fixed the bug, dependent on the tests that the code base has. So, we will want to keep the number of tests run constant while changing the number of lines of code in the commit. We would consider this a success if we see a linear speedup of our program as our commit size(in lines of code) increases, with a lower constant factor, so for a commit of around 50 lines we would shoot for around 5 seconds.

Percentage of lines involved in bug fix that tool did not include	commit size 10 lines (%)	commit size 50 lines (%)	commit size 200 lines (%)	commit size 500 lines (%)
tabler				
Rxjava				
guava				

Percentage of lines tool included that were not needed for bug fix	commit size 10 lines (%)	commit size 50 lines (%)	commit size 200 lines (%)	commit size 500 lines (%)
tabler				

Rxjava				
guava				

Time it takes to run tool (seconds)	commit size 10 lines (sec)	commit size 50 lines (sec)	commit size 200 lines (sec)	commit size 500 lines (sec)
tabler				
Rxjava				
guava				



Midterm and Final Checks
Our midterm check entails checking our early prototype, which we are aiming to finish by week 5. The prototype should be able to determine the lines of code that are accessed by the test and remove very simple refactored lines. This check would confirm that the tool itself successfully breaks up a commit into smaller ones. However, we do not expect our bug fixes to be completely minimized. The final check would involve running our tool with several examples as well as user testing to gain more information on how close our tool gets to minimum commit at a reasonable amount of time.