Jacob Ward: jward23; Evan Blajev: eblajev

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 we would like the commit to only include 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 VCS 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 you to select "hunks" of code to stage for commit. The second is for code already committed using "git rebase -i" which allows you to walk through old commits and split them using the first method. These both work well if you already know how you want to split the commit, but the only approach to minimize bug fix commits currently is doing it by hand. You must go through and remove lines of code to see if they cause the desired test to fail.

Our Approach

From a high level, our tool will be a script that runs from a Git pre-commit hook. The script will then run two different algorithms over the changes. The first looks for lines that are refactored and removes them. The second will remove subsets of changed lines until it finds the largest subset of lines it can remove without causing the test to fail. This test is one that was failing before the bug fix, and passing after the bug fix. Once this is done, we commit the minimized change set, add the removed lines back, and commit again.

Determining refactored code could be its own project or even research project. 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. 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.

Phase 1: Refactoring that would be caught and removed

```java
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");
}
```

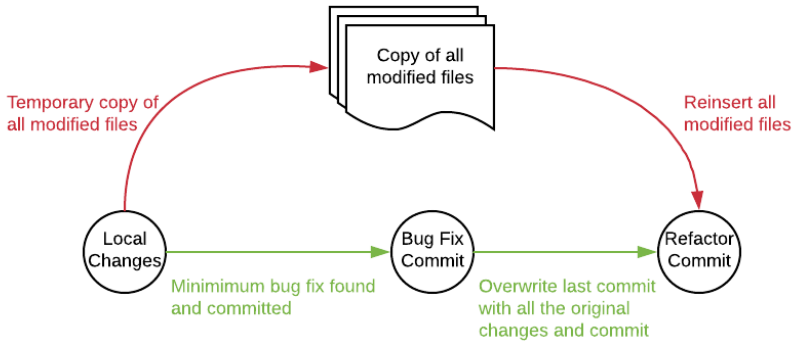
```java
public void insertTuple(Tuple t) throws DbException {
    // some code goes here
    // not necessary for lab1
    if(!t.getTupleDesc().equals(td)){
         throw new DbException("mismatch");
    }

    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;
        }
    }
}
```

Phase 2: Commit splitting



Challenges

Since our approach consists of removing lines and rerunning 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. 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.