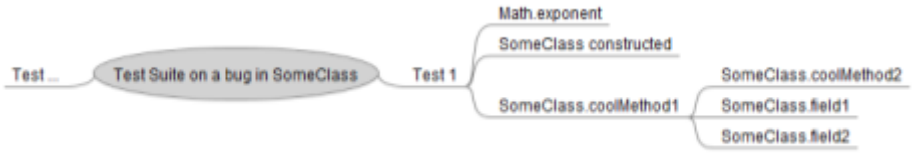
Abhinav Gottumukkala, Julius Christenson
anak4569, juliusc
Project 1 Pitch

Narrowing Commit Scope To Isolate Bug Fixes And Increase Team Effectiveness

Programmers often make large commits, solving numerous problems that they only noticed late into the night. Their commits are often vague about the details, too far-reaching in their scope and have messages like, “fixed three bugs, made variable names more clear, and added functionality that users requested.” But for other people looking at these types of commits, it might not be clear as to which changes in the code correspond to different parts of that message. These large commits can decrease team effectiveness as another team member might not know the intent of a change in the code without additional information. Being able to take a large commit and separate it into bug fixes and other tasks makes it easier for other programmers to understand each commit and easier for analysis tools to work. This would improve the readability of commits at a glance, allowing more effective programming when working with others. While users can use the patch function to split up changes in a file to multiple patches, which allows them to on their own split commits into “bug fixes” and “other”, this relies on the user to do it beforehand and would require all the code’s authors to do this.

To fix this, people have conceptualized many approaches to automatically minimize patches with respect to various import tasks, such as bug fixes. One traditional approach to minimizing a patch tried experimenting by removing lines added in the patch at random to see which changes were necessary for the task that is being minimized towards. However, this process is incredibly slow, as finding the smallest group of lines that still passes the task’s tests requires you run the corresponding test suite on many of the 2^{lines} subsets of the lines. For a tool that is designed to break apart large patches, which could have thousands of lines changed, this approach is untenable.

Although inspired by this method, our idea mitigates the performance issues using the user’s tests written to determine if the minimized patch fixed the desired task. Our idea starts by creating an overarching dependency tree and setting the root to be the task. From each test of our task, we build a subtree with the test itself as the root as a child of our overarching tree. With every call in that test to another piece of code in the user’s project, we add a child to the current node, continuing this process without repeating nodes in the entire tree and until the tree covers every sequence of calls in the entire test suite. In this way, we can take the user’s own test suite and try mapping the task’s ‘sphere of influence’ in the user’s code, limiting our minimization task to only what is needed. Finally, we would traverse through the tree, create a collection of all



An example dependency tree

the sections where the user added code and use the traditional approach mentioned above to see which of these additions are necessary. With a likely much smaller list of additions that is at most equal to the list of additions from before, the performance can only be improved.

This approach suffers from a few problems in implementation details, but our idea's largest challenge is that it assumes something about the user's coding and testing practices: modularity. The steps taken to improve the performance to the original approach only work if the tree does not inevitably branch into every change made in the commit. This means the user is forced to write tests for the task on as local a level as possible to help the tree reduce its internal scope of the bug. While making this sort of tool would require many more fine details and assumptions from potential users, the core idea is sound.