Software Development Difficulties

Calvin Loncaric

January 11, 2016

Problem 1: Avoiding unnecessary recomputation. Recently I found myself implementing a tool to process each one of about five hundred files. Even when run in parallel on `recycle.cs`, the entire process still took fifteen minutes. The tool was (intentionally) incomplete, and each time I run it I would get a histogram of missing features that caused failures on some files. I would implement those features, and then run the tool again to discover new problems, and so on in a loop until the tool ran on every file.

Most of the changes I made to the tool did not affect the output on most of the files, but the makefile I had in place was not smart enough to realize this fact. It would have been very difficult for me to write additional code to identify (1) which functionality I had changed in the tool and (2) which features in each file this would affect. However, I think this would have been possible for a computer to do; both the tool and each file were structured around the possible attributes a file might have. Avoiding the recomputation each time the tool ran could have saved many minutes of waiting for the entire process to complete.

Problem 2: Reading method implementations. As a graduate student this has been less important, but when I was working in industry I spent much more time reading code than I did writing it. In older code, it was frequently the case that the important classes and methods had many cross-cutting concerns: they would do some work, do some logging, do some cleanup, and so on. The additional work done by logging and so on obscures the main logic of the program.

This problem might be addressed by somehow labeling each program statement by its category, or at least by whether it has an impact on data or control flow. Being able to skip irrelevant lines might make it much easier to read the important parts of a method. This information might be determined using some sort of analysis that identifies statements that do not affect the output of the method, or by some sort of machine-learned model that predicts which cross-cutting concern a given statement relates to.

Problem 3: Reading class declarations. Along the same lines as problem 2, I have found that most classes in large projects fall into a few different categories, and quickly identifying which category a class belongs to can make it much easier to read the implementation. Some categories are: data (typically immutable objects that represent something outside the program, like Dog or

Webpage), data structures (mutable objects that organize data in some fashion), and workers (classes with relatively little state that implement a few functions for computing values). Different projects may have their own project-specific categories as well, like classes related specifically to logging or to interface with a database. This categorization is a fuzzy notion, but it might be possible to learn some features that distinguish one category from another and train a model to identify the category in most cases.