Difficulties in Understanding New Codebases

Qiao Zhang (qiao@cs.washington.edu)

As software developers, we have all faced the task of understanding someone else's code. Modern day developers inevitably stand on the shoulders of giants. We lookup open source libraries from GitHub in order to use them for our projects or extend the libraries to contribute back to the community. As developers in a large company, we often face the daunting task of understanding tens or hundreds of in-house software libraries and tools before we can write our own code that is deeply embedded in a large codebase. While it is feasible to read through a small codebase of a few files, it is often time-consuming and challenging to understand how different pieces of code work in a large codebase totaling tens or even hundreds of files.

For the ease of exposition, let me call those who try to understand a new codebase Readers, and those who have designed and implemented the codebase Programmers.

Underlying Causes of the Difficulties

1. Readers often only see/focus on shipped code.
    a. Programmers design with abstractions, and conveying those abstractions can help readers build a useful mental model for an effective understanding of the code. But readers often only see/focus on the final product -- the code itself. While well-designed and -written code can convey a good sense of the abstractions used, code is at too low of an abstraction level for building a useful mental model. Readers can benefit from a more top-down view.
    b. Almost all code start simple, no matter how complex they get eventually. In fact, start simple and test often is a mantra for good software engineering practices. Programmers write function stubs and simple straight line code to test the simplest use case before modularizing and generalizing the code. Following the development process can often unravel the complex logic in the final version of the code, and highlight the design hierarchy of different pieces of code.
    c. Shipped code is written to cover all use cases in production, so it has to be generic and handle all cases, e.g. it is not uncommon to see various if branches that cover all corner cases that "obfuscate" an otherwise simple code. Readers can benefit from seeing simple cases before understanding the more general ones.
2. Given the same abstractions, programmers implement them differently, e.g. naming convention, different belief about modularity, programming abilities, knowledge of algorithms, so it is hard to read and instantly understand others' code

    a. Programmers can be inconsistent in their implementation, e.g. variable names, programming idioms, inline documentation style/completeness
    b. Programmers can be lazy, e.g. without autocomplete feature of an IDE, programmers may use short uninformative names
3. Code is often designed to be easily parsed by compiler rather than to be easily understood by humans.
    a. Code needs to be stored in a flat file, so function definitions are spread out all over the place rather than follow the hierarchical abstraction layers that programmers think about during design and implementation.
    b. Code is organized by files that may or may not mirror the design abstractions.

Potential Solutions, Existing Deficiencies and What We Can Improve through Tools?

1. Present readers with more than shipped code
    a. A top-down architectural documentation, sometimes as simple as a README file summarizing what the purpose of each subdirectory, can facilitate readers' understanding. However, programmers are often not rewarded for writing documentations, intellectually or monetary-wise. Programmers at a large company are paid for the code they write, not the documentations. Moreover, architectural documentations often help readers from outside the team more than within the team, so the reward of writing them is even less visible and certain. We can dramatically lessen the burden of writing architectural documentation by automatically synthesizing them from all kinds of program artifacts, e.g. API doc, API type info, execution trace, test cases, usage by other programs, development history etc. The architecture documentation can be a hybrid of natural text, code snippets, or even diagrams.
    b. A simplified version of the codebase would be useful. Readers can glean the simplified version from version history and design doc. The design doc is potentially non-existent, while development history is often in a difficult-to-consume format (e.g. git history). Readers need a more digestible "executive summary" of some form. We can synthesize a skeletal (hybrid) code by filtering out if branches, inlining functions, removing boiler code, replacing code with text (from API doc or even test case names!), most of those steps can be guided by the development history (e.g. how functions were modulated can reveal how functions relate to each other, how if branches are incrementally added can tell us which cases are more common/apparent or thought to be more important).
    c. Example usage from sample user programs, tutorials and test cases is useful, but they are more available for resource-rich or more popular libraries/languages/codebases. We can synthesize example usage by collecting

usage by other programs and learn the pattern statistically. For example, for each typed argument in an API doc, collect all instantiated variable names in real programs, and then synthesize semantic examples that are embedded in real world usage.

2. Make programmers' code more consistent with general convention or reader's' own code
    a. Code review enforces a coding convention within an organization, but it is manual/costly/non-exhaustive/non-uniform/subjective. We can enforce a coding convention post-hoc! Use NLP/PL techniques, we can codify a naming convention/programming idioms, and then post-process the code to make it more organized and structured. For example, expand variable names to be verbose and informative using NLP when useful.
    b. Readers may run parts of the code, modify them to familiarize with the code. Interactive programming languages often make this easier. We can take hints from how readers interact with new code, transform code snippets on the fly (as simple as changing variable names that are more in line with readers' style and understanding), highlight different parts of the code while hiding others as the interactions reveal perplexity and resolutions in the reader's mind.
3. Modern IDEs can better present code than just flatly, and we already see various helpful meta-summarizes of code in IDEs, e.g. function hierarchy tree view, call stack, instant API doc lookup. These tools often help some readers more than others. We can use machine learning to learn what aids understanding and what fails, and then customize the tools for readers.