Alex Vrhel
CSE/UW ID: avrhel
CSE 490E1 Assignment 1

1) In languages that use the same syntax for initializing and referencing variables, such as Python, I have encountered bugs that are difficult to track down due to typing errors. These occur when you attempt to modify a variable in some way, but make a typo in the variable name, leading instead to the initialization of a new, similarly-named variable that receives the modification instead. For example, rather than setting `inCount = 0`, one accidentally creates `imCount,` initializing it to 0 and leaving `inCount` unmodified. This is a fairly specific problem, but it has led me to great lengths of debugging in a few cases, because it can be so hard to detect visually when pouring over the code and it creates unexpected behavior.

This could perhaps be solved by IDEs highlighting variables that are being initialized, so developers can quickly verify that an initialization is actually what they were intending in that statement. Another solution could be for an IDE to highlight or underline the initialization of a variable that is similar to an already existing variable in that scope, perhaps using a particular edit distance between the variable names as a threshold of whether to activate. I expect that the reason that this problem has not been solved in such a way is the analysis that would be required to detect such issues would be relatively expensive considering the rarity of these sneaky typos. Also, improving typing accuracy, being aware of the issues that such typos can cause, and proofreading code more carefully are existing, although less comprehensive solutions that may be easier to implement.

2) Developing a comprehensive unit test suite can be difficult due to the inability to find boundary values for the different inputs. Boundary values are inputs that lie near the boundaries between partitions in inputs, and are useful because they are the cases that will test meaningfully different inputs. The difficulty comes with determining how to partition the input space to correctly catch all the boundary values that should be tested. For example, if a simple bit of code is expected to behave differently for positive and negative input values, then partitioning the inputs around 0 (choosing a positive and negative value with low absolute values, and 0 as the boundary values to test) will provide useful insight when testing, since we would be more likely to find bugs at the boundaries of the different input partitions. For more complicated code, choosing boundary values can become significantly more difficult.

Much of the difficulty around choosing boundary values is that the process is hard to automate, since it requires some understanding of what the different ranges of inputs mean for your program. To pick boundaries, one must semantically divide the input space on criteria that is relevant to how they think their code operates. Determining this semantic divide would be difficult for most tools that perform code analysis, but perhaps they could be somewhat approximated. For example, other testing metrics such as branch coverage could be used as a heuristic to ensure that at least some large portion of the different input spaces are being considered by a test suite.

3) Understanding code in large code bases can be difficult in most IDEs because of nested function calls and wrapper functions, causing a function call to span multiple (sometimes even 5-10 files) files of source code. In my experience, this can be extremely hard to follow and unpack, leading to a bad experience when attempting to get familiar with the code base. This is not always a poor design choice, since it can result from good modularity and use of wrapper functions/classes, but it can lead to architectures that are hard to unravel, and there are not many solutions for IDEs in my experience that allow one to track through multiple files of code easily. Difficulties in tracking parameters and returns through these long chains can lead to bugs when a new developer first attempts to work with the existing code base.

Solving this problem would involve developing a tool or view for an IDE that allows one to better visualize nested function calls. I have used a system in VIM that creates tags at function declarations that can be jumped to, but this felt incomplete, since it filled the view with the new function declaration and made one lose track of what they were looking at. A more complete solution would provide a way for someone to add snippets from multiple files as separate panes in a window, perhaps with added color coding or syntax highlighting that assists in following the flow of parameters or return values. This tool would be useful for people who are working in large code bases, where they need to track chains of code or keep a flow of function calls in mind. There are existing, more manual solutions, such as opening up each file and tracking through to the right place, or jotting down notes about the flow of data, but integrating this further into a UI of an IDE as a dedicated tool would significantly improve the ease of digging into large code bases.