Taylor Yoon (yoont4)
Yifei Song (yifeis)
CSE 403
Project Proposal

Motivation:

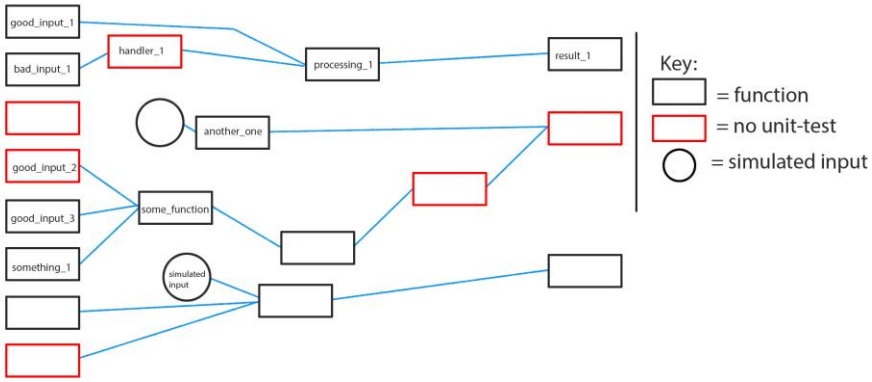
To my knowledge, there are many software packages available to generate tests, but none to comprehensively analyze a test suite. There are simple code/branch coverage metrics software like EclEmma, but I will discuss why these metrics are not useful enough.

Often times, just like the main product code base, the associated test suite can be huge and expansive. It can be take a long time for people being introduced to the test suite to understand what the tests cover, and how. Even for people just maintaining a large test suite, it can be difficult to remember every implementation detail, and what needs better coverage. This often leads to wasted time, going line by line through the tests, trying to figure out what is covered and what isn't. Finding redundant tests can also take a long time, as they may be physically spaced far apart in a file, and the content could be functionally equivalent, but appear different to the eye.

This is only time consuming, because there are no good implementations as of now, to visualize how comprehensive a test suite is in its coverage. Code coverage and branch coverage metrics alone are not useful enough to indicate whether a test suite is good or not, as Cem Kaner has taught. While they are useful to a degree, accurately pinpointing weaknesses in a test suite needs to display the context of a test within the entire program. This is not something that code/branch coverage metrics give.

Key Solution Element:

Visualization is one of the best ways to take data, and restructure it in a way where we can easily notice patterns and behaviors, that would be difficult by looking at the data itself. Visualization of a test suite's coverage also solves the issue of the tests in context, since we can see where in the program processing "pipeline" that a particular event or string of events is being tested. Rough example of function level coverage visualization in the following figure:



Function Level Coverage Map

While code coverage might say that the functions with unit tests are 99% covered, and that 60% of branches are covered, we can gain more insight through this on the connectivity of the tests at the functional level. If this was, say, all the functions in a pipeline for processing orders in a web store, we can easily see what has unit tests implemented and what doesn't. But, we can also see now that some functions without unit tests, are covered through integration tests (context!). Also, now it is easy to see that one case, where code and branch coverage might say are we're fine, is missing an end-to-end test, with the input being simulated input.

How This Solves the Issue:

What this does is let the developer make decisions for themselves, based on an easier to interpret representation of a test suite's comprehensiveness. Instead of jumping to every red flag that automated checkers raise, or trying to make assumptions based on shallow coverage metrics, the developer can see it in the context of everything else and say "this is okay, we are already aware of this weakness in the test suite," or say "this needs to be tested immediately, a critical case has not been tested."

We could further expand on the functionality of this visualization, by making it interactive, in that they can see what inputs are being passed into certain functions, particular end-to-end paths of a particular test, and some metrics on the range of inputs functions are given.

Challenges and Risks:

Building interactive visualizations can be very difficult, as UX and UI are key factors in the usefulness to the developer. Since it is a separate software package in itself, it also needs to have an external way of accessing all the source code and test files, without being irritating or confusing. Figuring out a logical way of creating the function map using an algorithm could prove to be difficult as well, as there is no easy way of denoting a function as a typical "starting point," another function as an "intermediate point," etc. without some kind of markup protocol in the source code.

There is also very little test suite analysis software of this kind available to reference either. Tools like EclEmma provide simple metrics and markups, but nothing of this type of visualization and analysis, meaning we have no references to bounce design decisions off of. The decisions would then have to be very carefully chosen, otherwise the entire project could fall out of the time frame, or prove to be ineffective in its intent.