HW #1

Glenn Hanawalt
grh95@cs.washington.edu

Testing GUI-Based Programs

GUI-based applications that rely on user interaction are hard to test. Simulating user interaction is challenging, and tests can become obsolete quickly. Additionally, there may be practically infinite combinations of user actions possible, making it hard to verify that a program cannot be broken. I know from personal experience that failing to write sufficient tests can lead to surprising regression bugs, so I want to suggest a method that combines multiple testing paradigms to provide better test coverage. Here are three possible ways to write tests:

- Using a testing framework that symbolically links a UI item to its abstract notion. For example, an annotation on the "align center" button in MS Word. Ideally this link should allow the programmer should be able to trigger that button in a test by referring to its abstract function, without requiring knowledge of its precise location or identity. The programmer can write complicated tests using this framework and be reasonably confident that they will not become obsolete. Tests written with this approach are fairly stable BUT they do not necessarily reflect end user experience.
- Writing tests that record and automate mouse, keyboard, or touch events in the app. These tests would do a better job of capturing user experience, than the previous strategy, but they will fail (causing false alarms) if the layout of the UI is changed.
- Writing tests like those above that ALSO record and compare screenshots or video. These tests would verify specific user-facing behavior (buttons highlt when they are hovered over, etc) but they are very brittle. Even small unrelated changes to the program's appearance will cause them to fail.

I'd suggest writing comprehensive tests using the first strategy, because these tests should be stable over the longest period of time. They also may afford for a higher degree of control and knowledge over program behavior.

The second and third kinds of tests are useful because they capture realistic user interactions, but they are probably tedious to record, difficult to verify, and easy to break. I would prioritize writing only very short and focused tests of these varieties. In addition, it would be great if the testing framework had a 'fast re-record' button that could auto-adjust clicks and/or expected pixels if a programmer can manually verify that the tests have been broken for superficial reasons.

It's also pretty challenging to simulate real (and possibly malicious) user behavior. To get a better idea about whether the program will can avoid failure (or at least fail graciously), high-stress tests and randomized user tests might be a good addition to a testing suite. The "check-rep" methods from CSE 331 would be useful and ought to be running in these tests, since the only real goal is to avoid crashing or achieving an invalid state.

Testing Programs with Numerical Outputs

While some programs or functions can be tested easily, it's been my experience that ones that involve large amounts of floating point operations and numerical analysis often cannot. Here, I am think specifically of scientific computing and machine learning. When testing these programs for correctness, there are several issues:

- Outputs may be hard to quantify.
- Improvements (or the opposite) to the program will cause the numerical output to change. It may not be immediately obvious if these changes are positive.
- Math may be inaccurate or inefficient at different scales.
- General case outputs may be extremely hard to verify due to scale.

I propose a testing methodology that compares outputs from many different inputs with known properties, and outputs from the same inputs on previous versions of the program.

For example, consider a machine learning application intended to identify images of trucks. The tests could contain a number of pictures, some of trucks and some of other objects. A simple policy would be to verify that all the truck pictures are above a certain threshold of confidence, and all the non-truck pictures are below a certain threshold. More sophisticated schemes could reward increases and penalize decreases, or track true/false positive rates. This performance could be tracked over time, showing how the program's behavior has changed.

As a second example, consider a computational biology program that uses statistical analysis to predict the locations of genes in various genomes. The tests, instead of passing or failing, could compare the program's results to known (hand-verified) genomes and assess the overlap. Setting these tests up would be time consuming, but they'd give a view of how hard-to-quantify changes to the program affect real-world performance.

Time Management

On some days, unproductive time can outweigh productive time. I don't refer to the difference between maintenance and development; rather, the difference between working and twiddling thumbs.

Things that have slowed me down as a developer:

- Waiting to ask someone for help.
- Waiting for code review or to be assigned new tasks.
- Switching between tasks (mental cost).
- Forced interruptions (meetings, etc).
- Being blocked ("I need them to finish their module before I can finish mine").
- Feeling stuck.

Unfortunately, practices that solve some of these problems can exacerbate others. More meetings and communication between the team may reduce the occurrences of blocking and waiting type hindrances, but they also contribute to interruptions and context switches.

I think this has had a huge effect on my productivity, but I often underestimate or ignore these issues because they rely more on 'soft skills' than engineering prowess. These issues are well known, but it still takes strategic work to optimize your time:

- Most organizations or software companies use some kind of team management software. It's easy to be inattentive to it because it's not nearly as interesting as work, but it's very important to update and read this religiously to keep tabs on what you and your teammates are doing. This helps you stay focused, anticipate problems, and ask for new work early.
- It's good to have several independent tasks available at a given time. Time spent being truly blocked or stuck is usually unproductive, so in these cases it's best to just switch tasks. Unfortunately, my natural instinct when faced with a problem I don't know how to solve is to think very hard for a long time. This might be necessary sometimes, but it mostly wastes my time. Always maintain a setup that allows you to switch between tasks with a minimum amount of effort.
- Other than critical meetings, keeping interruptions at a minimum is ideal! I am noticeably less productive over periods of time that include multiple classes or meetings (accounting for the time these take, of course). I suspect that one big meeting is better than multiple small ones. Additionally, organizations should strive to set up infrastructure that helps avoid unnecessary interpersonal communication (ie. "I'd better go ask Susan how X component works." This slows both you AND Susan down, and her time is probably more valuable than yours.) Documentation is critical here. Most organizations have a wiki, but it's only as good as its contributions, so you should contribute to it when you have something meaningful to say.