Justin Kotalik

UW/CSE net Id: jkotalik

To motivate two of my difficulties, I will first start by talking about a scenario I encountered at an internship. During my internship at Cisco two summers ago, I was working on a platform called Cisco Spark. I made an applet that would read meeting information from Microsoft Outlook and display it conveniently on the website. I was writing some tests for this applet, which would take fake test users and process meetings in an E2E manner. There was a pool of ~1000 users that was for my use, so initially I wrote a single test that would just grab a random user and process their meetings. When I ran the test individually, everything seemed great, however, there were issues when I ran the test suite. The tests would always fail! So, I took a deeper dive into why it was failing. It took me at least a day to figure out what was going on, and the solution was frankly surprising. One big issue with the way Cisco's test suite ran is it used timers to wait for a response to be formed. What this means is responses may not have been processed in time. So initially, I just increased the wait time, however it seemed to keep failing. That's when I started looking at the user that were generated. The test users themselves had hundreds of meetings that needed to be processed because the users were being shared across thousands of tests. New users should be generated for each test, or at least a mock user should be used. The main difficulties highlighted here are **using timers in tests** and **large component sharing.**

Using timers in tests is always a dangerous idea, but is appropriate in some situations. People should notice whenever they are using a timer and ask themselves, "is this right?" The main issues that could arise from timers in tests is that each test will take at least as long as the timer value. Also, there is no guarantee that the action is completed once the timer expires, therefore causing flakey tests. Tests also must worry about correct synchronization between the test and the timer. But using timers sometimes is inevitable. For example, if a system needs to performant and doesn't meet some timing specification, timers should be used. But overall, they are hard to work around and can sometimes lead to odd solutions. This is not a "solvable problem," and rather a case by case issue, however there are some situations where one may be forced to use a timer. You could use an async/await model for awaiting a response, however if the black box doesn't send a termination signal (or something to await), this could not be possible. Also, there can be separation of the timer and tests by having the timer in a separate thread. This would allow for better performance; however, the response still may not be complete. And on top of this, there are times when implementing a solution around timers is not worth the company effort.

Next, a main difficulty I had was large component sharing. That is, using the same old large component of shared users across multiple tests. This was a good design decision for practicality, where each test user would only need to be created once, however there are scalability issues if multiple tests use the same resources. The large batch of test users though can have threading issues and assumes full control of these users. People should care about

using these large components for practicality and time saves, however without proper documentation and implementation, they could have fatal bugs causing the component to be worthless. I personally could have created new users for each execution of my tests, however it would eventually cause database and naming issues if too many users were created. Also, creating a new user is expensive, so the set-up time may not be worth a simple test. Hence, even in my case, **large component sharing is not an easy problem to solve**. The main reason there is no clear-cut solution is that there are too many assumptions to be made. Even if documentation is perfect, there could be a bug in the large component causing issues. My assumption that users have a reasonable number of meetings was not a valid assumption. Though sharing large components can be tough, it should be a goal of any company. In my situation, I should have tried to find a way to clear all meetings for a user and then add extra meetings afterwards. One cool way companies have been making large components more reliable is by open sourcing them. Then, companies can have users modify errors or bugs accordingly, improving the usability. Another solution is strong documentation and reliability; if a company is required to follow documentation, there is more likelihood the component can be used.

Finally, there is always a **trade-off between performance and readability** in all programs. People always want the fastest implementation possible. Though sometimes code can be made faster with small improvements, like making strings static or unrolling for loops, to the naked eye this code is not very readable and understandable. Some of these should be a job of the compiler, however there may be some improvements that cannot be accounted for. These problems are not easy to solve at all. What makes code readable? What makes performance improvements too crazy? Is 1% throughput improvement worth the hassle? All of these questions are incredibly application specific, however there is no clear answer at all. No has "solved this problem" because again it is so dependent on the application. There are some standards though for readability. Most programming languages have a style guide that they adhere too, allowing for most developers to understand code. There are also some general formatting rules. However, when it comes down to unrolling a for-loop, is it worth the readability issues? Most likely not, however some insane people still do it to this day.