CHRISTOPHER MACKIE
mackic

1. Large applications can often take a long time to build. This isn't necessarily a problem when pushing changes in some version control system, as continuous integration techniques often do a good job of handling this. However, long build times can be an issue when running a project's test suite locally, or when informally testing an application. I have experienced this issue while working on the visual development environment for an education scripting language. Often the simplest way to test anything was to build the app and mess around, then create an automated test based on what I did to manually test the program. Many projects make use of lazy building techniques to ensure that only the recently edited portions of an application are rebuilt if possible. These techniques can often shave off a lot of build time, and do help reduce the severity of the problem, but they often do not take enough advantage of various down times, since they are initiated by the programmer when it is time to test the program. For example, while working on this visual IDE, I would make many changes distributed across several files. It's entirely conceivable that some of those files could have been compiled while I worked on other things. If some building framework ran continuously, it could build recently edited modules while the programmer works on another one. For some languages, it may even be possible to compile portions of an individual file while the programmer works on another part of the file. When the programmer is ultimately ready to run the program the building framework would just have to verify that the whole project is built, and possibly build the most recently edited portion.

2. In large projects it is extremely useful, and often necessary, to reuse portions of the platform to extend the code base. For experienced contributors to the code base it can seem very natural to use the various parts of the project to do this. However, new developers may not know what all of the parts of the project are and how to use them to accomplish certain tasks. Even with very well documented projects, there can be a lot of exploration required to find the necessary modules within the code base, since documentation is often linked to the location of modules within the code base. For example, JavaDoc is organized in a hierarchical way, such that a reader must find the module they wish to learn about before they can see what it does. But what if the reader doesn't even know the module exists? For example, I did an internship at a company that had a very expansive platform. The modules in this platform were organized in such a way that related methods and classes were in very different locations within the code base, with some classes spanning over multiple directories. Luckily, someone had created documentation organized based on the semantics of the modules rather than their hierarchical organization, so I was able to find modules which I could use based on that document. Semantically organizing documentation can allow the programmer to discover the modules and

methods they need to accomplish a task more easily. This could potentially be done automatically by annotating modules based on possible uses to allow documentation for similar modules to be together compiled together in some document.

3. When writing tests, it is sometimes difficult to constrain a test entirely to the feature being tested. When writing unit tests for some application, it is possible that a test may be targetted at a specific feature, but may fail if an irrelevant part of the code base were changed. I once worked on an application with a test suite which consisted of screenshots and text logs of previous executions to determine correctness. Basically if any screenshots or logs were different than a baseline, or a baseline did not exist, the test failed pending human approval. I noticed that many logs were of unfiltered program state, which often included the whole state of some object. Often times, only a couple fields of the object were relevant, but whenever we changed the class at all, all tests using its logs would fail. Therefore, whenever a test failed, I changed it to filter out all parts of the log not relevant to that test. My manual fix to the problem was to check if a condition on the success of the test depended on a feature of the application not relevant to that test, and if it did, to filter it out of the test. This process could be automated by annotating the intent of a test and analyzing the constraints on the test's success to filter out irrelevant constraints which could cause the test to issue a false negative, without introducing false positives.