CSE 490E HW 1: Software development difficulties

Jasper Hugunin (uw/cs netid: jasperh)

January 5, 2017

1 Assertions

Most commonly used programming languages support testing and specification checking through assertions. This is usually a statement accepting a boolean parameter which is evaluated, and if false signals that something has gone wrong. Adding assertions to a program can help clarify how it works by making explicit (and automatically checking) the invariants of an algorithm. For example, adding assertions at the beginning and end of function definitions can serve to document preconditions and post-conditions. In the same manner, many tests are written in the form "do this; assert it worked".

However, there are many properties that we would like to check in this way but are impractical to. For example, we may want to check that the input list to a binary search algorithm is sorted, but checking this turns the procedure from $O(\log n)$ to $O(n)$. Or perhaps we have some value function on an ordered list, and some algorithm that quickly returns the highest value permutation of an input list; checking that it in fact has the highest value attainable would require $n!$ time. This same difficulty applies to testing as well; testing such a property usually involves precomputing the correct answer via some external method and hard-coding it into the test suite, opening up the possibility that your analysis and the program are both mistaken in the same way.

Even when adding assertions doesn't cause an asymptotic slowdown, calculating assertions still takes time, for a result that will hopefully be thrown away. So in production, it may seem wise to optimize the program by removing the assertions. If there are bugs, they won't be caught in this mode, but this allows for a trade-off between running fast and catching bugs. Adding additional annotations to assertions could also allow for turning off only some assertions, perhaps those which are especially slow or in the inner-most loop.

Turning off assertions can have consequences though. If computing the assertion condition has side effects (writing to memory, performing IO, etc), then we have potential differences between behavior between the version with assertions and without assertions. If you have a bug that is entangled with a side effect of your assertions (the side effect makes the bug disappear when assertions are enabled), then debugging becomes more difficult.

2 External dependencies in tests

Consider you are developing a web app and you want to test it. You could spin up a web browser for each test case, having the whole system processing each test. But that is slow and potentially expensive. Furthermore, such large scale tests are fundamentally more prone to unrelated failures than smaller, faster tests that test each component in isolation (network outage, disk failure, cosmic ray event)

So we can try to isolate small, self-contained components that can be tested in isolation. This lets our tests run faster, which encourages them to be run more often. However, separating the design into such components is not trivial. Writing so-called testable code, which makes testing individual components easier (by dependency inversion, small public interfaces, etc.) requires adjustments to the program. Refactoring takes time, has the potential to introduce new bugs (or remove old ones, to be fair).

And what does the final code look like in the end? We have added another requirement to the program, that it be easily testable. Writing code that is testable is not always the most easily understandable or the most performant design.

Furthermore, to test components in isolation requires some stand-in for the rest of the system. Creating such a mock/dummy can require reimplementing a restricted version of the component you are trying to mock out. As the complexity of that component increases, the value gained from mocking it out increases but so does the difficulty of creating a suitable stand-in. If the stand-in doesn't match the behavior of the real component, then the test is worse than useless, since you now have to debug your test code.

An important part of such a complex system is how it does handle inevitable failures, so testing that is also important. Accidental failures that happen during a full-scale integration test suffer from non-repeatability. The mock components can be enriched to allow simulating various failures, and thus testing the response, but this adds even more complexity to the mock components.

3 Package size

Consider a package in the sense of package managers, like pip for Python or npm for Node. It provides a set of features, and may depend on other packages. How to resolve when two packages depend on different versions of the same package is a tough problem, but not one I've been personally bit by, so I will ignore it here.

The question I pose is this: how finely divided should the package set be? That is, when should a package be split into multiple packages, or multiple packages combined into one?

Large packages have the advantage of providing a unified viewpoint on a variety of problems. Frameworks, for example, like Django for python web applications, tend to be on the large side. This also facilitates internal code

reuse and avoids issues that may arise with coordinating releases and versions of several small but heavily inter-related independent modules. From the client's perspective, it is easier to keep track of one large dependencies than n tiny dependencies.

Small packages are also very appealing however. They allow clients to select only the functionality they need, reducing transitive dependencies and therefore code size, compile time, and potentially runtime performance. There is also the benefit that smaller systems are more easily understood and maintained.