Hadar Greinsmark
hadar@cs.washington.edu

Software Development Difficulties

Opt-out in higher-level languages

Three years ago, I had an internship at NASDAQ where I worked with porting older C++/C# applications to Java. The project was part of an effort to follow the decision of only using the Java ecosystem inside the company. This was made primarily to make the applications more uniform, so developers could be moved around more easily between teams without having to learn a new platform and language. The downside though, is that a higher-level language like Java isn't designed for lower level optimizations.

In some applications, they had very long arrays that stored different kinds of table information. In Java, the obvious approach would be to use classes and inheritance to represent table rows. Then we could store those rows as objects in an array. But when working with arrays of objects in Java, there's a relatively high linear space overhead. For every object, you store a 4- or 8-byte reference to it in the array. Each object then has to store a header containing type info and garbage collection status, that is fragmented on the heap. To ensure a better running performance, the solution was to store primitive types in arrays and using enum names and other strategies to make it easier to access values. As primitive types are stored in the array directly, you reduce overhead and fragmentation. Though, you circumvent Java's type checking and the automatic memory management making it more error prone.

So how could you go about this problem? It seems that Java isn't designed for these lower-level optimizations. Maybe it would have been better to use C++ in this case. In C++, the objects - rather than references - are stored in the array, and there's no headers appended. Then we could've just done an RPC call from Java to the performance critical component. But the management and HR didn't like multiple software platforms. Why not use C++ for all the applications in the company then? If you would want to use higher-level features, C++'s *opt-in* approach makes it easy to use "headers" on objects using virtual classes for inheritance types and pointer template containers for automatic memory management. Here, you can opt-in to a higher level. But there isn't much ways to *opt-out* from a higher-level language to a lower level to avoid the heavyweight stuff in a fashionable way, and still maintain the same programming environment. E.g. the JVM is giving you a simplified environment for simplicity, and it would be tricky to let the application step out of that environment. Though, it's starting to pop up solutions to this in newer generation of languages. Swift is a good example. It distinguishes between struct and class types in that they differentiate between passing by value or reference. You can also use unsafe pointers that circumvent the default automatic reference counting. If looking at virtual machines, WebAssembly is planned to allow different kinds of memory areas within one application; one for manual memory management, and another for garbage collected memory.

Deploying libraries

Another software development difficulty often experienced, is the waste of time spent on installing and setting up a new library or system rather than learning how its API is supposed to be used. The widely used ones usually have a lot of documentation and a wide community on the Internet, that can help you deploy it. But on less used libraries, there's often a lack of documentation and examples, so you end up reading the source code in order to understand how it's supposed to be used. The problem it that it can often be time-consuming and unnecessary to understand the internals. The maintainer of the code may not be very interested in spending time on documentation. What if we could have auto generated tests? One way could be to randomly generate calls to the exported/public API, that follows the logic rules and soundness of the language. If a function expects type X as argument, we try to generate a line of code above that returns type X. Though, this would probably give a lot of weird examples. Another way would be to learn example structures from experience by e.g. doing data mining on open-source repositories like GitHub and Stack Overflow. Turns out that there's already a startup called Kite trying to do something similar.

Long feedback loop when testing

One problem when developing and testing large applications, is the long feedback loop from doing changes in code, to going through all tests. Often when you do a change, you rerun all the tests ever written, just in case something broke. That can take quite a bit of time to run. Therefore, you often compile and run the tests at night using Jenkins or similar tools. Sometimes - before you leave office for the day - you commit your work of the day, but forget to commit a new file. When the test suite is starting for the night, it fails immediately as a referenced file is missing. When you're back in office the day after, you realize that none of the tests have actually been executed. You would then have to wait another day to get the results. A simple name analysis that could have checked the difference between new and old declarations and their usages, would have found the error very quickly. But as that analysis is often built into the compiler and reevaluated from the beginning with other compilation steps, the analysis is done as a batch job rather than instantly online.

There's some measures to shorten the feedback loop. Continuous integration is one example, where you rerun a lighter test suite for every commit. This wastes a lot of resources though, as probably 99% of the code is the same. And you still have to wait 10 minutes or so; enough time that the developer would switch to another task in between and lose focus. One idea would be to see commits as a transaction in a database management system, where you do consistency checks before approving a transaction. The repository software could then optimize for fast commits by cashing the AST, namespace/scoping and a dependency graph of tests, and update those online. The challenge is that it would require different repositories for different languages. Mixing languages would become tricky, and things like code generation (ex. a parser generator) would require a special setup.