The Engineering Triangle as it Applies to Web Development

Sheldon Linker
sheldon.linker@faculty.umgc.edu

Abstract
The "Engineering Triangle" tells use that everything can be good and fast, fast and cheap, or cheap and good, but not all three. In modern web programming, good and fast programming can both lead to a lowest-cost project.

Introduction

The Engineering Triangle is simply stated: "Good, Fast, Cheap: Pick two".

Definitions

First we have to define our terms. For web programming, what do Good, Fast, and Cheap mean? Each of these terms will take some consideration. Good means both that the application meets its requirements, but also that the application is relatively bug-free. Fast means that the application itself runs quickly, with a minimum lag time, and using a minimum of resources. Cheap means that the overall cost of the product, including creation, maintenance, and running the product (including the infrastructure cost) for some number of years is minimized.

Good

Except in exceptional circumstances, Good has to be the top priority. It turns out that the choice of language enters into the equation for the quality of the software produced. The more rigorous the compile-time checking, the more likely the software is to be correct, with a minimum of undiscovered bugs. Use a strongly typed language. For instance, in a weakly typed language such as JavaScript (including Node.JS), "`x = y;`" may cause a problem if X receives an unexpected data type. This type of problem can be somewhat eliminated by using add-ons like TypeScript. But, using a strongly typed language like Java will cite such an error at compile time. When using a language with inheritance (again, such as Java), the most specific declaration possible should be used. For instance, declaring a variable as `Object` works, but being more specific helps, so (in the case of this example), but `Map` works better, and `HashMap` works even better, and `HashMap<String,MyType>` works best. Always fix all warning messages. Normally, fixing warning messages will mean either fixing a problem, marking that specific line as fine (often using some sort of marker, such as "`(int)`" in C, or "`/*FALLTHROUGH*/`" in Lint), or by disabling the given warning for a specific line, if neither of the other methods will do. In all languages, set the warning level to the highest level, and treat all warnings as errors. If specific warnings are only turned on with certain flags, settings, or pragmata, use them. When using a pure-compiled language, such as C++, pay attention to portability warnings, such as data length or byte ordering. Having an application that's fully portable by recompiling means that the application can be scaled by running on a faster type of computer, such as moving from an x86 computer to z/Series computer. When using a database language, the more rigorous the data definition, the better the application will be, and the more rigorous the compiler, the better off the project is. Infrastructure as code, such as inversion of control and the use of text files as control mechanisms also leads to a greater possibility of unchecked code, and so should be avoided.

Good = Cheap

Following the rules above will likely lead to code that takes longer to write, but will likely decrease the cost of quality assurance, as well as making the likelihood of bugs found after release, and will thus likely reduce the overall cost of the project. As mentioned above, these techniques make vertical scaling fast and easy, and thus can decrease the overall cost that way too.

Cheap

Except in real-time work, or the case of medium-length-running requests (requests the run anywhere between 2 seconds and a few minutes), the benefits of a system that runs cheap outweigh the benefits of a system that runs fast. After all, most such systems are a part of a business, and the business wants the best bottom line. Note that following the rules of Good will also bring your project closer to Cheap.

Fast, and **Fast = Cheap**

A program or system should also run fast. (See Low Latency Web Programming.) It turns out that in many cases, a fast-running program is also a cheap-running program. If a program runs fast enough that it doesn't need to scale up (vertically, such as switching to a larger computer, or horizontally, such as starting new instances or longer-running lambdas), then it runs cheaper.