Screaming Architecture

30 September 2011

Imagine that you are looking at the blueprints of a building. This document, prepared by an architect, tells you the plans for the building. What do these plans tell you?

If the plans you are looking at are for a single family residence, then you'll likely see a front entrance, a foyer leading to a living room and perhaps a dining room. There'll likely be a kitchen a short distance away, close to the dining room. Perhaps a dinette area next to the kitchen, and probably a family room close to that. As you looked at those plans, there'd be no question that you were looking at a *house*. The architecture would *scream*: **house**.

Or if you were looking at the architecture of a library, you'd likely see a grand entrance, an area for check-in-out clerks, reading areas, small conference rooms, and gallery after gallery capable of holding bookshelves for all the books in the library. That architecture would *scream*: **Library**.

So what does the architecture of your application scream? When you look at the top level directory structure, and the source files in the highest level package; do they scream: **Health Care System**, or **Accounting System**, or **Inventory Management System**? Or do they scream: **Rails**, or **Spring/Hibernate**, or **ASP**?

The Theme of an Architecture

Go back and read Ivar Jacobson's seminal work on software architecture: Object Oriented Software Engineering. Notice the subtitle of the book: *A use case driven approach.* In this book Ivar makes the point that software architectures are structures that support the use cases of the system. Just as the plans for a house or a library scream about the use cases of those buildings, so should the architecture of a software application scream about the use cases of the application.

Architectures are not (or should not) be about frameworks. Architectures should not be *supplied* by frameworks. Frameworks are tools to be used, not architectures to be conformed to. If your architecture is based on frameworks, then it *cannot* be based on your use cases.

The Purpose of an Architecture

The reason that good architectures are centered around use-cases is so that architects can safely describe the structures that support those use-cases without committing to frameworks, tools, and environment. Again, consider the plans for a house. The first concern of the architect is to make sure that the house is usable, it is not to ensure that the house is made of bricks. Indeed, the architect takes pains to ensure that the homeowner can decide about bricks, stone, or cedar *later*, after the plans ensure that the use cases are met.

A good software architecture allows decisions about frameworks, databases, web-servers, and other environmental issues and tools, to be deferred and delayed. A good architecture makes it unnecessary to decide on Rails, or Spring, or Hibernate, or Tomcat or MySql, until *much* later in the project. A good architecture makes it easy to change your mind about those decisions too. A good architecture emphasizes the use-cases and decouples them from peripheral concerns.

But what about the Web?

Is the web an architecture? Does the fact that your system is delivered on the web dictate the architecture of your system? *Of course not!* The Web is a delivery mechanism, and your application architecture should treat it as such. The fact that your application is delivered over the web is a *detail* and should not dominate your system structure. Indeed, the fact that your application is delivered over the web is something you should *defer*. Your system architecture should be as ignorant as possible about how it is to be delivered. You should be able to deliver it as a console app, or a web app, or a thick client app, or even a web service app, without undue complication or change to the fundamental architecture.

Frameworks are tools, not ways of life.

Frameworks can be very powerful and very useful. Framework authors often *believe* in their frameworks. The examples they write for how to use their frameworks are told from the point of view of a *true believer*. Other authors who write about the framework also tend to be disciples of the true belief. They show you *the way* to use the framework. Often it is an all-encompassing, all-pervading, let-the-framework-do-everything position. This is not the position you want to take.

Look at each framework with a jaded eye. View it skeptically. Yes, it might help, but at what cost. How should I use it, and how should I protect myself from it. How can I preserve the use-case emphasis of my architecture? How can I prevent the framework from taking over that architecture.

Testable Architectures.

If you system architecture is all about the use cases, and if you have kept your frameworks at arms-length. Then you should be able to unit-test all those use cases without any of the frameworks in place. You shouldn't need the web server running in order to run your tests. You shouldn't need the database connected in order to run your tests. Your business objects should be plain old objects that have no dependencies on frameworks or databases or other complications. Your use case objects should coordinate your business objects. And all of them together should be testable in-situ, without any of the complications of frameworks.

Conclusion

Your architectures should tell readers about the system, not about the frameworks you used in your system. If you are building a health-care system, then when new programmers look at the source repository, their first impression should be: "Oh, this is a heath-care system". Those new programmers should be able to learn all the

use cases of the system, and still not know how the system is delivered. They may come to you and say: “We see some things that look sorta like models, but where are the views and controllers”, and you should say: “Oh, those are details that needn’t concern you at the moment, we’ll show them to you later.”

For more on this topic, see Episode VII - Architecture, Use-cases, and High Level Design, at cleancoders.com.