

An Architectural Perspective on Software Design

George Fairbanks
Rhino Research

November 2009

1 What is architecture design?

For technophiles like me, it was amazing. A friend was playing music at his party via iTunes – nothing unusual there – but he was controlling it from his sofa using his phone. Lots of folks play music at parties from their laptop, but he had an application that showed him the songs in his library and let him queue up songs to play. Since I had a smartphone too, and technical pride was on the line, I decided to find or build a similar application.

On the internet, I found an app for my phone. It was visually attractive and had many features even though it was an early release. However, when I ran it, I found some problems. Initially, the app would not talk to my media center and it was hard for me to diagnose why. And navigating through the pages of the app was slow – even backing up a page caused a noticeable lag.

After looking at the code, I realized that these problems could be fixed by rewriting the communication library it was using. To solve the problems, the library needed to cache results and report more details about connection errors. In a minor way, the application had an unsuitable architecture.

It is worth relating this story for three reasons. First, it helps to demonstrate the value of an architectural perspective that goes beyond just features. The agile movement has been enormously helpful in that it has cleared out much of the process cruft that I have seen building up in large IT shops. However, some agile advocates recommend a pure focus on functionality and no up-front design at all. I believe that goes too far, and this story shows how a small amount of architectural consideration can be helpful.

Second, it helps to show that architecture is about modeling and abstractions. The essence of a problem can become clearer when using a model that contains only relevant details. In this case, if I didn't already know it before, a simple model annotated with timing information would convince me that no amount of speedy routines on the client side is going to yield lower latency user interaction until the number and frequency of server requests is reduced.

Third, it helps to reveal that architecture does not come from the problem domain. If we agree that the problem domain here is music, then it is clear that knowing domain facts (e.g., artists produce albums containing songs; playlists are ordered lists of songs) does not lead us to choose a client-server architectural style.

I'll discuss each of these in the sections that follow, using the example of the phone app.

2 How do we approach a problem from an architectural perspective?

It is a fair assumption that the phone app focused more on features than architecture. For the purposes of comparing feature-centric and architecture-centric perspectives let's assume the guess is accurate. It is true that I have the benefit of a bit of hindsight, but I believe that if either you or I had sat down and considered the architecture before starting development of this app we would have come to the same conclusions I discuss here.

Engineering successful systems means anticipating failures and avoiding design alternatives that are likely to fail. Henry Petroski, a historian of engineering, said it like this: "Every solution of every design problem begins, no matter

how tacitly, with a conception of how to obviate failure in all its possible and potential manifestations.” (Design Paradigms: Case Histories of Error and Judgment in Engineering, 1994).

Petroski’s observation seems almost tautological: success means not failing. However, we can choose to spend time anticipating failures, or we can choose to do something else with our time, like building more features. From that perspective, we are doing something less than engineering if we are not considering different design options and anticipating how each could fail. Perhaps software is special in this regard, but before discarding the engineering tradition of considering failures and choosing the best design alternatives, we should give pause and consider its merits.

So, what kinds of failures can we anticipate for a simple phone app that controls a media center? Certainly it could fail to function, i.e., it could fail to control playback of music on the media center. But there are many more ways to fail. As mentioned earlier, it could be annoyingly slow or users could have a hard time diagnosing connectivity problems. Or it could allow anyone at the party to control the media center: a security failure. We can broadly group things like latency, usability, and security into what architects call quality attributes.

2.1 Architecture options

Once we have thought of ways the system could fail, we can consider architectural options. The obvious question is “will a particular architecture option avoid the failures?” Some architectures have already been analyzed and are known to promote or inhibit certain qualities, so it may be possible to choose an architecture off-the-shelf that is a good antidote for the failures you anticipate. Beyond these off-the-shelf architectures, you will have to generate your own architecture options and analyze them with respect to the possible failures you anticipate.

Some architectures will be easier to analyze than others. For example, an Enterprise Java Bean (EJB) server has the job of concurrently running many EJB’s. It constrains the EJB’s by prohibiting them from starting their own threads. This architectural constraint makes it easier for the server developers to analyze the concurrency problems, which would undoubtedly be harder if EJB’s were allowed to create new threads.

Engineers anticipate failures and choose designs that are most likely to succeed. Unfortunately, up-front analysis does not guarantee success. You could weigh the risks incorrectly or be blindsided by an unforeseen risk. That means that even if you do some architecture design up-front, the agile advice to build a running system ASAP still applies. To get the balance of up-front vs. emergent design right, you can follow the Risk-Centric Model. As general rule, putting some thought up-front into the architecture is prudent but spending a lot of time is a bad idea.

2.2 Design options to avoid latency

Let’s consider the architecture of the phone app. The first failure I considered was the high latency between user gestures (e.g., button press) and our app’s response. To reduce latency, I had to either tighten up the code so it runs faster or change the kind of work the app was doing. The time taken for a server query is likely the biggest contributor, so I started looking at that first. One architecture option was to query the server whenever the app needs data. Another option was to cache some of that data so the app can reduce the number of server queries. Going to the extreme, the app could even cache all of the music metadata (e.g., song names, album art, album lists), which would eliminate server queries while the user browses the metadata. But since it is a phone, storage is limited, so I would have to trade off local storage space on the phone with latency reductions.

While it is not yet obvious what the best option is, the solution space is starting to become clearer. Perhaps I could even graph latency vs. amount of metadata cached and make a principled choice, or give the user control over the cache size.

2.3 Design options to enable connection debugging

The other failure I considered was that it is hard to diagnose what is wrong when the phone cannot communicate with the media center. Looking into the problem more, I discovered at least the following cases could cause failure:

1. Phone internet turned off

2. Phone unable to connect to internet/wifi
3. Phone can contact internet but not media center
4. Wrong IP address for media center
5. Media center not enabled for remote control
6. User login credentials to media center rejected
7. Phone-server protocol error

Looking at the source code, I discovered that the library that was used to communicate with the server failed to report many of these options. Instead, it lumped many of them together as a single error, which made it impossible for the user interface to provide helpful feedback beyond, “connection problem”.

It seems useful to separate the concerns of communication with the server separately from the rest of the app, so it is best if it is a separate module. The app must have access to detailed information about the connection failure so that it can give that feedback to the user, so that information must be present in the module interface. That interface should probably hide technical details of how the server is contacted (currently it is HTTP). However, any other module using this interface would need to know that this is a remote call since they take orders of magnitude longer than local calls.

2.4 Other architectures

So far we have kept the original client-server architecture, where the phone is the client and the media center is the server that responds to commands. However, it is worth considering other architectural styles and their implications. In my house, there are often song files on my server, on my phone, or on my laptop. With a peer-to-peer architecture, media would exist in the peer-to-peer cloud, perhaps discovered by UPnP, and any peer could play the music. In fact, we could imagine streaming the music from the “cloud” to any device, including back to the phone. Even if we decide not to act on this option right now, it opens up our thinking about our options and enables us to design interfaces that enable us to switch to this option later without a major refactoring.

2.5 Approach

Looking back, how did I approach the problem from an architectural standpoint? First, I explicitly considered failures – specifically failures related to quality attributes like latency, security, and modifiability. Second, I generated design options and evaluated them with respect to the failures. For the latency failure, it was even possible to start understanding the solution space and its general tradeoffs. Finally, I looked at the overall architectural style (client-server) and considered if it was matched to the problem at hand.

The order of these activities is less important than the thinking that accompanies them. That is, do not treat this as a process to be copied. Instead, notice the attention paid to quality attributes, failures, design options, and architectural styles in this architectural approach compared to purely feature-focused development.

3 Architecture is about modeling

Architecture is about modeling, but modeling does not mean CASE tools and binders full of diagrams. Models are what we use to understand and express the essence of something. Models are by definition abstractions and they necessarily omit details. When I think about the phone app being a client-server application, I am mentally using a model that has just a client and a server, a model that elides details like what programming language either is written in.

Developers who are sensitive to architectural concerns can look at problems and extract a model that contains the essential details needed to solve the hard problems. They use models as a tool to make their analysis capabilities more effective. Without a good model, the details would overwhelm their ability to see the problem clearly and design a solution.

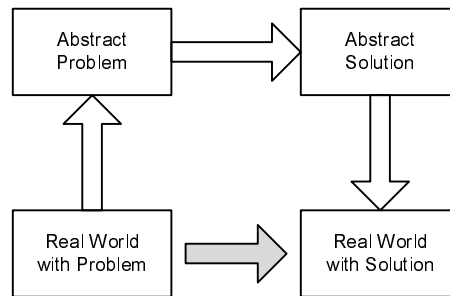


Figure 1: Commuting diagram

3.1 Commuting diagram

Mary Shaw likes to show a commuting diagram (see Figure 1) to illustrate how engineers use models to solve problems. What is needed is to produce a real world solution to a real-world problem. Sometimes engineers can do this directly, as is indicated in the gray arrow on the bottom. However, when the problems are large or complex, engineers solve the problem in the modeling domain. To do so, they must first create a model of the problem (the Abstract Problem), then, when they have created a solution (the Abstract Solution), translate that model solution back into the real world. Failures can occur across any of the arrows in the diagram. For example, if the model is missing relevant details then the solution will be wrong, or at least incomplete.

The alternative to abstraction is the mastery of details. This is appealing to many software developers and has the advantage that it can sometimes lead to simpler solutions. However, no matter how talented you are, you have limits. It is unlikely that anyone can successfully follow this path with millions of lines of code.

3.2 Standard architectural styles, elements, and relationships

So, to think architecturally means to think about models of systems. The bad news is that we can create bad models and therefore fail to solve problems. The good news is that there are standard models that can be learned. For example, architectural styles are models with predefined elements and constraints. There are also standard model elements, so new abstractions need not be invented for each project. Examples of model elements include modules, dependencies, components, connectors, ports, and environmental elements. And there are standard relationships between models, including views, refinement, and designation. As a result, becoming good at modeling architecture is easier than you might think.

3.3 A guide for thinking

These standards for creating models can guide your thinking about problems. In physics classes, students are taught to understand many problems using a Free Body Diagram that helps them to reason about the forces acting on an object. This guide to thinking transforms the way students understand how the world works and it persists even after they have forgotten the exact equations.

In architecture, the standard models guide thinking too. Once you are familiar with the standard models and elements, when you see a new software system you will, perhaps unconsciously, fit what you see into the abstractions you have learned. You will wonder what standard architectural style, if any, best describes the system. You will think about the properties of connectors and how they influence the ways in which you can compose the components. You will look for constraints in the architecture that simplify your analysis and guide the system's evolution. The standard architectural models enable you to understand more and understand it faster, since you are better at categorizing what you are seeing.

4 Architecture does not come from the problem domain

It is a common misconception that a system's architecture is derived from its domain. While it is true that some domains have presumptive architectures, for example the use of 3-tier architectures in Information Technology systems, it is usually possible to implement any system using any architecture. Developers must evaluate the problem and choose an appropriate architecture.

4.1 Wikipedia: domain and architecture

Here is an example. Wikipedia is a large, online, editable encyclopedia. Its domain consists of things like articles, editors, edits, and versions. Wikipedia is implemented using a Content Delivery Network (CDN) that consists of geographically distributed read-only replicas of its database and a large memory cache of web pages. We could imagine using its CDN design in other contexts, perhaps to serve images for a photo sharing site, or even Linux ISO images.

The use of a CDN is a good choice for the risks Wikipedia faces. There are lots of readers, so it must serve up lots of web pages. However, its funding is limited and there are not many system administrators. And there are few writers, so biasing the system towards serving up lots of infrequently changing web pages seems like a good idea.

Thinking hard about Wikipedia's domain – articles, editors, edits, versions, etc. – does not lead us to its architecture. But thinking about failure risks, in this case how serving up billions of web pages could go wrong, does help. The architecture is chosen to avoid those failures.

The domain of the phone app might include things like songs, playlists, artists, and albums, none of which inspired me to think about the failure risks. However, thinking about failure risks did lead me to consider caching results and reporting connectivity failures.

4.2 Domain and architecture are different kinds of models

We can build a model of the phone app's domain. If we did it in the style of object-oriented analysis, we would see types representing the songs, playlists, artists, and albums. These types would be linked by associations, for example an association between an album and the many songs on it.

The architecture model for the phone app is different than its domain model. From a runtime viewtype perspective, the architecture model includes a client (the phone app) and a server (the media center). They communicate using HTTP over an internet connection. From a module viewtype perspective, it includes the modules for the media center (presumably these are provided as binaries) and the source code for the phone app. From an allocation viewtype perspective, there is phone hardware that runs the app and a desktop machine that runs the media center software. This model could be elaborated further, but it is good that describes the three primary viewtypes: the runtime, module, and allocation viewtypes.

Perhaps your definition of the problem domain is broader than described here. It could include the idea that the phone is controlling the media center and therefore the problem would open up to include what I have defined as the architecture model.

However, there are advantages to defining things as I have done. As discussed for Wikipedia's CDN, we can reuse the same CDN architecture on different domains, like newspapers. Similarly, the same domain could be paired with different architectures, like using a peer-to-peer architecture for the music domain. Broadening what we include in our domain models means this would no longer be possible.

5 Conclusion

It is easy to focus on functionality. Indeed, many agile software development advocates suggest that you should. Software architecture researchers were not the first to suggest that quality attributes (or extra-functional requirements, or the "-ities") are worthy of your attention, but they have reinforced that message and have connected quality attributes

with architecture choices. The architecture of a system can mitigate risks that the system faces, primarily quality attribute risks.

When thinking about a system's architecture, you should consider the failure risks that the system faces. You can choose a design option that reduces those risks. Some architectures come with constraints that help you reason about the problem. Standard architecture styles, such as client-server, pipe-and-filter, map-reduce, and peer-to-peer have already been studied, so you can match the quality attribute risks you face with a suitable architectural style.

Models are the key to architecture. The essential nature of a model is that it abstracts away some details, so to build a useful model you must make sure it includes the right details. Creating a good model is still an art, but the job is made easier because of the existence of standard architectural styles, elements, and relationships. Once you understand these standards, analyzing a system will be easier, much like how the idea of Free Body Diagrams helps you understand physics.

While you can model an architecture and you can model a domain, the two models are of different things. You can mix and match the architecture and the domain, so for example you could have written the music controlling phone app either as a client-server architecture or as a peer-to-peer architecture.

This article has discussed how to look at systems from an architectural perspective and, along the way, mentioned many key architectural ideas. Architecture should not be equated with Big Design Up Front and, as seen here, some time spent thinking about the architecture can help you choose designs that address failure risks.