ARCHITECTING FOR CHANGE

As an architect, you have probably heard at some point from the business "Our business is constantly changing to meet new demands of the marketplace", or "We need faster time to market to remain competitive", or even "Our plan is to engage heavily in mergers and acquisitions." What do all these statements have in common? Change.

It's a different world than it was many years ago. Both business and technology are in a constant state of rapid change. That means architectures have to sometimes change as well. However, the very definition of architecture is "something that is really hard to change." So how can we make our architectures respond faster and easier to change? In this article, I will explore the "architecting for change" meme and discuss several common techniques for ensuring that your architecture can properly adapt to change.

Architecture Agility

Simply put, traditional methods of architecture are not sufficient to meet the ever-changing demands of the marketplace. Business is ever changing through mergers, acquisitions, growth, increased competition, and regulatory changes. Technology is also ever-changing through new platforms, languages, frameworks, patterns, and products. Because of all this rapid change, we need to make our architectures more adaptable to change as well. Specifically, we need our architectures to be agile.

The term architecture agility means the ability to respond quickly to a constantly changing environment. Note the word "quickly" - this is the real key. All architectures can change; however, really successful architectures are ones which can quickly adapt to change.

There are many techniques you can use for ensuring your architecture can quickly adapt to change. In this article I will describe three of those techniques: abstraction, leveraging standards, and creating product-agnostic architectures.

Abstraction

The first technique, abstraction, involves decoupling architecture components so that components know less about each other, hence minimizing the overall impact of changes made to those components. However, what does it mean to apply abstraction to components?

Forms

It is important for architects to understand the various forms of abstraction and the various methods used to implement abstraction. There are five different forms of abstraction:

1. Location Transparency
2. Name Transparency
3. Implementation Transparency
4. Access Decoupling
5. Contract Decoupling

In this section I will describe each of these forms of abstraction, and also describe the methods used to apply these abstractions.

Location Transparency

The first form of abstraction, location transparency, means that the source component does not know or care where the target component resides. For example, the target component may reside on a server in Frankfurt, or even a server in Paris - it simply doesn't matter. This is the easiest form of abstraction to implement, with messaging, service locators, and proxies being the most common implementation methods.

Name Transparency

Name transparency means that the source component does not know or care about the name of the component or service method. For example, suppose you need to access a pricing server for stock prices. Name transparency means that you can call the service anything you

want (e.g. `GetLatestPrice`), whereas the actual name of the method you are invoking on the target component is `getSecurityPrice()`. The implementation name can continue to change, but the source component always refers to the service as `GetLatestPrice`. This form of abstraction is commonly found in messaging and service registries.

Implementation Transparency

Implementation transparency means that the source component does not know or care about what language or platform the target component is written in. It could be Java, C#, .NET, C++/Tuxedo, even CICS - it doesn't matter to the calling component.

Access Decoupling

Access decoupling means that the source component does not know or care about how the target component is accessed, whether it be RMI/IIOP (EJB), SOAP, REST, ATMI (Tuxedo), etc. Typically a source component standardizes on one access protocol (e.g. XML/JMS) and has a middleware component (e.g. integration hub or adapter) transform the protocol to that used by the target component.

Contract Decoupling

Finally, contract decoupling means that the contract advertised by the target component doesn't necessarily need to match the contract used by the source component. For example, let's say the source component uses a CUSIP (a security identifier) to check the price of a particular security, but the target component requires a SEDOL (another type of security identifier). A middleware component or adapter can perform a CUSIP to SEDOL conversion, thereby decoupling the contract of the target component from source components.

Methods

Now that you understand the five forms of abstraction, it's time to discuss the methods of applying abstraction, and what forms of abstraction those methods implement. The five methods I will be discussing in this article are:

- Messaging
- Adapters
- RESTful web services
- SOAP web services

- Message Bus (such as Apache Camel or Mule).

Messaging

Messaging as a method of implementation means that you place a queue in front of each target component to abstract that component from the source component. Therefore, the source components are communicating to a message broker with queues as opposed to communicating with the target components directly. Messaging implements location transparency, name transparency, and implementation transparency, but it does not implement access decoupling and contract decoupling. With pure messaging as a method of implementation there is only a message broker and a queue; there is no component you can write to handle the access and contract decoupling.

Adapter

The use of an Adapter as a method of implementation means that you place a custom build or vendor-supplied adapter between the source and target components. Because adapters can contain components (actual code) as well as another method of implementation (i.e. messaging), adapters have the capability to implement all five forms of abstraction. Notice I said "capability" – this is the neat thing about the use of adapters. You can start out with a simple adapter that uses messaging, and you automatically have the first three forms of abstraction. Then, you can evolve your system when needed to implement access and contract decoupling.

RESTful Web Services

RESTful Web Services as a method of implementation for abstraction means that you expose each of your target components as RESTful services, thereby accessing them though a URI rather than directly. RESTful web services implements the same forms of abstraction as messaging: location, name, and implementation transparency. The location transparency comes from the fact that you can use a proxy or DNS to hide the actual IP address of the target component itself. Name transparency is achieved by referring to the target component as a resource rather than by name. However, like messaging, RESTful web services does not implement access decoupling and contract decoupling because, like messaging, it is only a transport protocol. There is no component that exists to be able to write the code necessary for access and contract decoupling.

SOAP Web Services

SOAP Web Services used as a method of abstraction implementation mean that you expose

your target components as SOAP-based web services, thereby accessing them through a URI similar to the RESTful web services. Unlike RESTful web services, however, the SOAP-based web services implementation method does not support name transparency. You cannot refactor the method name of the target component exposed as a SOAP service without changing the WSDL contract. SOAP web services implements location and implementation transparency, but not name transparency, access decoupling, and contract decoupling.

Message Bus

Using a message bus (such as Apache Camel or Mule) as the implementation method for abstraction means that the middleware component (message bus) exposes centralized endpoints that provide an abstraction between the source components and target components. Since most message buses support messaging, they also support location, name, and implementation transparency. However, like the adapter, since it is a middleware component, it can also support access decoupling by exposing different endpoint protocols, and also contract decoupling by providing transformation functions and also message enhancement capabilities.

The level of abstraction you choose to implement is largely based on the trade offs you are willing to accept. For example, implementing abstraction through basic messaging automatically provides you with location, name, and implementation transparency. However, access and contract decoupling requires some sort of middleware component (like an enterprise service bus or custom adapters), both of which are expensive to implement and add a significant complexity to your application or system.

Leverage Standards

Another technique you can use to facilitate change within your architecture is to leverage standards. There are three types of standards you need consider as an architect:

- Industry Standards
- De-facto Standards
- Corporate Standards

Industry Standards

Industry standards primarily consist of protocols and payload formats defined as either universal or belonging to a particular business domain. XML and SOAP are examples of

universal industry standards, whereas SWIFT, FIX, and FpML are specific financial services domain standards. Sticking to industry standards, particularly domain-specific industry standards, allows the architecture to adapt to change more quickly by integrating better with other applications and systems within that domain. For example, by choosing SWIFT as your standard, you can pretty much integrate with any bank. However, if you have your own custom protocol and data format, integration will take significantly longer.

De-facto Standards

De-facto standards are those technologies and products that are so well known and widely accepted in the industry that they generally become a standard part of almost every technology stack. Hibernate, Struts, Apache Tomcat, and the Spring Framework are all good examples of de-facto standards. By leveraging these technologies and products, your architecture can adapt quicker to change primarily because the resource pool is large for these de-facto standards and the availability of documentation and references is widespread. For example, if you perform a Google search on a particular issue you are experiencing in Hibernate, chances are good that you will find a plethora of information to help you solve your problem. However, perform a Google search on a lesser-known persistence framework, and chances are you will be on your own to figure out the issue.

Corporate Standards

Corporate standards are the third type of standard and include those technologies, tools, and products that your particular company uses. Sometimes corporate standards include industry and de-facto standards, but usually include specific products and technologies like . NET, Java EE, JBoss, Eclipse, etc. Leveraging corporate standards is critical to achieving architecture agility. Change is quicker because the resource pool and skill set within the company for those products and technologies is widespread. For example, let's say you decided to break away from the corporate standards (say Java EE) and implement your architecture using Ruby on Rails. However, because the resource pool hired by the company is Java EE, changing the application will be difficult due to the lack of available Ruby on Rails resources to make those changes.

Product Agnostic Architecture

When designing an architecture for change, it is important to avoid what is known as vendor lock-in. While your architecture may need to be dependent on a particular product, the product should not be the architecture.

Creating a product agnostic architecture involves sufficiently abstracting the product using adapters, message bus technology, or messaging to form what my friend Neal Ford fondly refers to as an anti-corruption layer.

For example, let's say that you are using a major ERP product (e.g. Oracle, SAP, etc.), and you are asked to develop an architecture for the system. It is all-too tempting to just place the large ERP product in the middle of the architecture and leverage the many tools available from that product to develop and integrate various applications that use it. While this would certainly work, it is perhaps the hardest type of architecture to change. It would literally take years to swap out one ERP product for another or integrate other products into the architecture. A much better approach would be to create an abstraction layer around the ERP product so that product upgrades can be better isolated and product changes be made more feasible.

Conclusion

While there are many techniques you can use to create architectures that quickly adapt to change, it is important to remember using these techniques comes with a price. Applying abstraction and creating product-agnostic architectures usually decreases performance, adds complexity to the architecture, and increases development, testing, and maintenance effort and costs. As an architect you must analyze the trade-offs between these disadvantages and the advantage of an agile architecture that can quickly adapt to change. If you are in an industry that is frequently changing, the trade-off is easy - you must architect for change. However, just how much agility you apply to the architecture depends on the tradeoff's you are willing to accept.

About the Author

Mark Richards is a hands-on software architect with over 30 years of industry experience. Throughout his career he has performed roles as an application architect, integration architect, and enterprise architect. Although Mark works primarily in the Java platform, he also has experience in many other languages, technologies, and platforms. Mark is the author of *Java Message Service 2nd Edition* by O'Reilly, contributing author of *97 Things Every Software Architect Should Know* by O'Reilly, and most recently co-author with Neal Ford on a new video series by O'Reilly called *Software Architecture Fundamentals*. Mark has spoken at over 100 conferences worldwide, and is passionate about architecture, technology, and hiking.