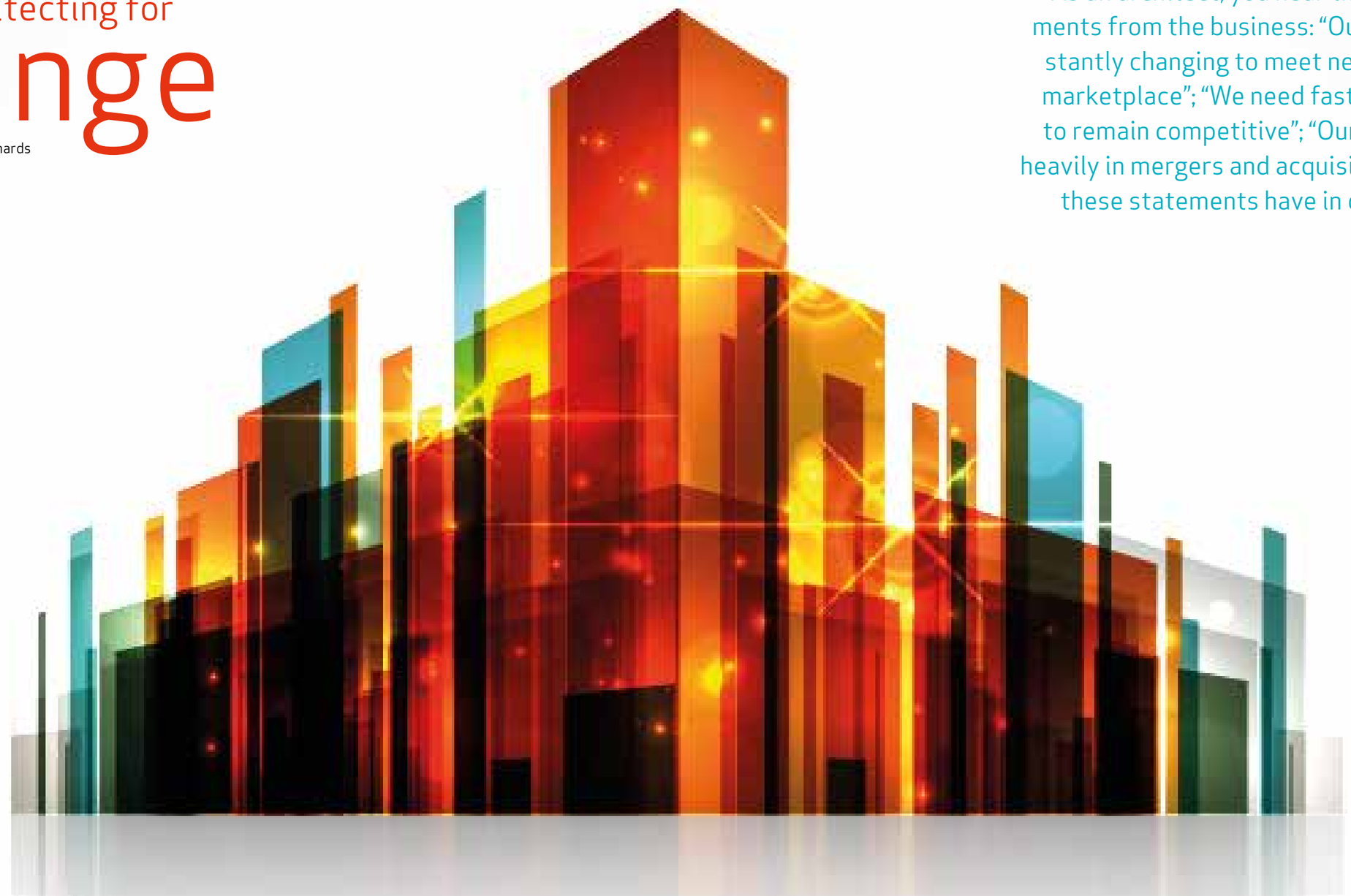
Architecting for change

By Mark Richards

As an architect, you hear the following statements from the business: "Our business is constantly changing to meet new demands of the marketplace"; "We need faster time to market to remain competitive"; "Our plan is to engage heavily in mergers and acquisitions". What do all these statements have in common? *Change*.



© zagats-and-zagats/ Shutterstock

It is 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 better to change?

Architecting for change is one of several memes Neal Ford and I introduce in our Software Architecture Fundamentals video series from O'Reilly and conference workshops (soon to be held at the Architecture Day in Oslo (www.programutvikling.no/kurs/architecture-day/4353). In this article, I will explore the architecting for change meme and discuss several 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. There are five main forms of abstraction: location transparency, name transparency, implementation transparency, access decoupling, and finally contract decoupling (the hardest form to implement). In this section I will describe each of these forms of abstraction.

The first form, 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 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 dictates that you can choose to 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 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 simply doesn't matter. Again, messaging is a common way to implement this form of abstraction.

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.

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, lets 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.

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, and corporate 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 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 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 wide-spread. 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 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 standard 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 significant complexity to the architecture, and increases development, testing, and maintenance 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 (and most are), the trade-off is easy - you must architect for change. However, just how much agility you apply to the architecture depends on the trade-off's you are willing to accept.

Mark Richards is a hands-on 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 and technologies. 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.