ThoughtWorks®
The art of heavy lifting.℠

Developing Software in a Service-Oriented World

Whitepaper

January 2005

Gregor Hohpe

Summary

The software development community is once again at an interesting inflection point. Distributed and interconnected systems have become the norm for new development efforts to the extent that the word "application" itself might have to be redefined. Independent variability and evolution of the interconnected components are enabled through loosely coupled solutions, such as Service-oriented Architectures (SOA). At the same time, Model-driven Architectures (MDA) aim to simplify development by abstracting and automating large portions of the coding effort.

This leaves us wondering what it will mean to develop in a service-oriented world. Will business analysts wire together components in visual tools? Will developers program using meta-meta-models and domain specific languages? What new types of programming paradigms will developers use? How can we get ready?

Table of Contents

1 Right-Click, Make Web Service

There is no doubt that the advent of Web services standards has made previously tedious programming tasks significantly easier and more efficient. For example, I recently developed a simple application that retrieves the Amazon sales rank of my book and e-mails it to my mobile phone. Naturally, this application needs to retrieve data from an external organization over the Internet. Still, developing the application could not have been easier. I pointed my IDE to the WSDL provided by Amazon and without much ado the tool generated a client proxy class for me. I added a few lines of code, and my application was up and running. Not only is this application fully buzzword compliant – I used Web Services, XML, WSDL, SOAP, you name it -- but it actually does something useful (bolster an author's ego).

Exposing existing functions as Web services has become similarly simple. Most development tools are not far from allowing the developer to highlight a method and to select "Make Web Service" from the pop-up menu. It would be almost trivial to expose an enhanced version of my little program as a public Web service, for example allowing other users to submit a list of ISBNs and an e-mail address to receive periodic sales rank updates via e-mail.

2 Web or Service or Architecture?

However, just like one swallow does not make summer, one service make does not make a service-oriented architecture. My simple solution is based on the technologies that collectively define Web services but it hardly qualifies as an example of a service-oriented architectural style. Many people before me (e.g., [1]) have discussed the difference between Web services (a collection of technologies) and Service-Oriented Architectures (an architectural style) so I will not reiterate that. Let's however, have a closer look at what makes an architecture service-oriented.

Most popular architectural styles seem to be a product of the weaknesses of their predecessors. After all, systems development styles will always be incremental as today's systems are tomorrow's legacy. This phenomenon is no different in case of SOA. Some of the main drivers behind SOA were to cure the struggles developers and architects had with the prevalent prior style, namely distributed component (DC) architectures. In particular, architects wanted to remedy the following problems they experienced with existing distributed architectures:

- **Vendor lock-in.** Many DC architectures were based on proprietary protocols and implementations. In contrast, standards-based protocols support (at least in theory) free interoperability between multiple vendor products.

- **Tight coupling.** DC architectures typically link components directly to one another, making the solutions brittle. An asynchronous, document-oriented style of interaction allows more independent variability between components [3].

- **The Transparency Illusion.** Distributed components promised to hide remote communication from the developer by making the remoteness "transparent". While the basic syntactic interaction between remote components can be wrapped inside a proxy object, it turned out that dealing with partial failures, latency, and remote exceptions could not be hidden from the developer. It turned out that 90% transparency was actually worse than no transparency because it gave developers a false sense of comfort.

- **Complexity.** DC architectures extend the rich, but complex interaction between objects across the wire. One object can control the lifetime of remote objects, pass references, and rely on polymorphism and inheritance. This model turned out to be too complex given the inherent complexity that distributed architectures add into the mix.

- **Call Stack.** While a call stack is an enormous convenience for monolithic applications it can quickly turn into a liability for loosely coupled, distributed applications. Waiting for a remote component to complete a task before continuing execution at the source tends to make distributed architectures unresponsive and brittle.

- **Connectivity.** Most DC architectures require reliable, permanent connectivity between components. They do not work well over wide-area, intermittent networks such as the Internet or mobile networks.

A service-oriented approach aims to address these issues by using a simplified, document-oriented interaction model that is based on technology-neutral, standardized protocols. Coupling between interacting services is minimized through the use of flexible document formats, explicit contracts, service registries, and an asynchronous interaction style.

Even my simple example was able to benefit from some of these new SOA properties. For example, even though my program was written in C# I had no problem accessing Amazon's Web service, which was likely not written in the same language. I also did not have to create a remote object using some remote factory in order to execute my query. Instead, I could simply pass an XML document (generated by the stub) to the remote service.

But other subtle, yet important aspects did not yet come to play in this simple example. For example, the client program holds a standing TCP connection to the remote service while the request is being processed. The proxy object also waits synchronously for the results, preventing the client application from performing any other tasks while it is waiting for the results. This simplistic programming model has a definite remote-procedure call flavor to it. While such a model is convenient, it is not very suitable to remote service-oriented interactions.

What does it take to graduate from building trivial Web service demo apps to becoming a serious developer of loosely-coupled, service-oriented architectures?

3 Architect's Dream – Developer's Nightmare?

In Patters of Enterprise Application Architecture [2], Martin Fowler warns that loosely coupled, distributed system architectures that look great on a whiteboard can easily become "An architect's dream and a developer's nightmare". There is little doubt that making a system distributed adds a whole new level of complexity. For example, configuration and deployment become more difficult, and you have to deal with latency and new failure situations such as network interruptions or version mismatches between components.

As mentioned above, the premise of a loosely coupled architecture is to reduce the complexity of the interaction between components and reduce the coupling between them. Coupling can roughly be measured by the number of assumptions communicating parties make about one another. For example, loosely coupled architectures typically do not make any assumptions about which technology was chosen to implement the other endpoint. Likewise, these architectures are more robust against minor changes in the data formats, such as the addition of new fields. It is the reduction in the number of assumptions that enable independent variability of the conversation partners – one of the key motivations behind loose coupling.

But fewer assumptions also mean a thinner safety blanket for developers. Strong coupling allows the compiler to catch many mismatches between communicating parties. Misspelled method names, missing parameters, mismatching data types can all be flagged at compilation time. In a loosely coupled architecture this is no longer the case. As a result, debugging a tightly coupled architecture with a call stack and shared memory space is a lot easier than debugging heterogeneous, asynchronous, distributed solutions.

4 The Human Side of Service-orientation

Many people have argued that Web services are over-hyped and that it was possible to build fine service-oriented applications with CORBA or similar technologies. While there is certainly some truth in the argument it appears to miss the point slightly. Changes in architectural styles happen as much in the developers' head as they do in the available technologies and APIs. Architectural styles are based on shared intent, correct trade-offs, and consistency of vision. Choosing the right technologies can make accomplishing the desired architectural style easier but it is by no means sufficient.

The nice thing about history is that it tends to repeat itself, especially in systems architecture. Essentially, we can compare the transition from distributed components to service-oriented architectures to the transition from procedural to object-oriented programming or the transition from text-based user interfaces to graphical user interfaces (GUIs). The initial adoption of new technologies tends to be based on using the old architectural style in the new technology or API, often referred to as the "lipstick on a pig approach". For example, using a Web service as an RPC replacement is reminiscent of running a text-based application in a window-based graphical user environment. Yes, technically speaking, the text-based application uses the new GUI technology. You can resize the window and move it around. However, the application hardly takes advantage of the true capabilities of new environment. Learning how to develop great GUI applications required new tools, a new mindset, and new patterns and guidelines. Interestingly, the new APIs were a rather small part of this transition in user interface style.

Likewise, I believe that SOAP will turn out to be a relatively small part of the transition to a service-oriented development style. Especially in the near term, service-orientation will be more dependent on conventions rather than technology. Not using proprietary data types as parameters in methods exposed as a service operation is one such convention. Not making assumptions about the location of a service is another one. How can we avoid making services too coarse grained or too fine grained? How about deciding how much logic to embed inside the service and how much to leave for the orchestration layer? Currently, no tool can help us decide or enforce these guidelines. They are a mere result of the team's shared understanding and agreement on a specific architectural style and intent.

The reliance on agreement and convention seems to be particularly pronounced in loosely coupled architectures. As discussed above, loose coupling means a reduction in assumptions between components. Fewer assumptions mean less compile-time validation and enforcement by tools. Instead, the validations have to be implemented via human communication and agreement.

What does this mean for developers of loosely coupled, service-oriented architectures? It means that human-to-human communication is more critical than ever. Documenting the big picture and the intent is critical to preserving the architectural integrity of these solutions. In the near term this likely means that life for developers gets harder rather than easier. It also means that in the near future one of the most useful tools for service-oriented architectures will be Word and PowerPoint (or OpenOffice Writer and Impress).

5 New Tools

New tools will be enormously useful in the transition to a service-oriented development model. Interestingly, for every major new programming paradigm or technology shift, new tools seem to appear in a number of specific phases. I have generally observed three distinct generations of tools that support new technologies:

First Generation: Retrofit. The first generation of tools typically retrofits the old to make it look like the new. These tools turn green-screen mainframe interfaces into GUIs or any good old COBOL program into a Web service. While this is useful, the catch is that the tool converts simply the technology but not the architectural style or intent.

Second Generation: Simple Tools for Simple Problems. The next generation of tools enables developers to quickly build solutions using the new technology and the new style. However, these tools are targeted at simple problems that can be solved with simplistic approaches. For example, many early GUI tools generated the GUI directly from the database schema. This approach works well if the application is simple (e.g., a CRUD-style application), but is not suitable for applications with complex business logic.

Third Generation: Efficient Tools for Complex Problems. The next wave of tools goes beyond simple problems. While they cannot magically trivialize complex problems they do provide efficient tools for developers to deal with the complexity. Good examples of this league of tools can be found in the world of Java IDEs, namely Eclipse and IntelliJ IDEA.

It generally takes a significant amount of time for third generation tools to arrive in the market place, for a combination of reasons. First, building complex tools naturally takes more time. More importantly, though, in order to build an efficient tool for complex problems one has to first observe how developers deal with these complex problems. That takes time and experience and requires a hard core of developers who are willing to work on the bleeding edge, solving complex problems with inadequate tools.

Most Web service tools today fall into the first or second category. We can find a lot of "Web service façade" tools or tools that make simple service development easy, à la "Right click, make Web service". What will the third generation of tools look like? I believe that we will see much more sophisticated testing and debugging tools along the lines of SOAPscope. We will also observe a rapid evolution of monitoring and management tools. These tools can discover services and the communication between them at run-time, and render visual images of service dependencies. They also allow us to apply policies to multiple distributed services at once.

I believe that these discovery and visualization tools will play an important role in the third generation of SOA tools. A basic tenet of loose coupling is to enable independent variability and evolution. This implies that we cannot (and do not want to) impose an *a priori* top-down structure onto the solution. Instead, we allow the solution to evolve over time in ways that might not have been anticipated. In order to track the evolution of such a system we need tools that can detect the current state and present it in human-friendly form, for example as a graph [4]. These types of tools are now starting to appear in the marketplace and I think we will see more sophisticated varieties emerge over the next year or two.

6 New Programming Paradigms

Learning new tools, though, is only part of the transition to developing service-oriented architectures. We already discussed how embracing SOA requires a new way of thinking about the interaction between components. It also requires developers to become comfortable with new programming models and paradigms. I want to highlight a few of the major models here – the list is by no means complete.

6.1 Object-Document Mapping

First, a document-centric approach is subtly, but critically different from an object-oriented approach. An object-oriented system typically benefits from the interplay between fine-grained, highly interconnected object instances that reference each other with object references (or pointers). Document-centric interfaces represent documents, not behavior, in tree-like structures. As documents have to be passed around multiple systems, managing references can be difficult and cumbersome. Documents tend to be more coarse-grained to support proper encapsulation between components and to reduce network "chattiness". They also have no inherent notion of inheritance or polymorphism.

These differences are subtle but important for successful development of a maintainable SOA. They are somewhat reminiscent to the impedance mismatch between OO systems and relational databases. The O-R mapping problem is subtle as well and has been around for a while. Nevertheless new mapping layer implementations still arrive at a notable pace. Many of the current Web services tools are analogous to 2-tier DB development tools that can generate GUIs from a database schema. These tools work well if the application maps very closely to the back-end model but they generally do not support development of complex business applications. The same is true with many of the tools that expose existing methods in object-oriented application as services. These tools will quickly reach the limits of their usefulness once you try to expose complex application components as well-defined services on a large scale. As a result, developers will have to create their own object-to-document mapping layers or wait for more sophisticated development tools.

6.2 Declarative Programming

Two core functions that are commonly bundled into services architectures or a so-called Enterprise Service Bus (ESB) [5] are transformation and rule engines. The programming models used to describe these functions are typically declarative as opposed to the sequential-procedural style most developers are comfortable with. For example, most transformation tools are based on XSLT, which is a based on a pattern-matching approach. The XSLT processor matches incoming documents against a collection of rules and selects the most specific rule to execute. Some tools might hide the actual XSLT document behind a drag-drop style user interface but the programming model remains essentially the same. If you have made the switch from object-oriented programming to XSLT you will likely notice two things. Your first XSL stylesheet is likely to either return everything or nothing because you got the pattern matching wrong. Also, your subconscious will tempt you to use a lot of <xsl:call-template> elements because that construct most closely resembles traditional functional call semantics. It also generally makes for the worst style sheets.

Declarative programming systems can result in very compact solutions. But they also require a different thought pattern and can be harder to debug because the execution path is determined at run-time instead of design time. As transformations and rules engines become more common, developers need to learn how to design and debug such declarative subsystems.

6.3 Event-based Programming

Most business processes react to events, such as incoming orders, payments, or customer calls. This implies that the execution is no longer based on a linear sequence but often has to react to incoming events as they occur. This style of developing is very different from traditional application development that allows the developer to control what happens when and in what order.

Therefore, forming an event-driven mindset will take some getting used to for most developers. Essentially, being event-driven means abolishing the call stack. Without a stack, execution flow is no longer controlled by push-and-pop, synchronous method calls and local variables. Instead, the programmer has to manage continuations and state explicitly as events arrive. Many process and orchestration tools provide mechanisms such as "correlation sets" and "convoys" to help developers deal with these scenarios but still require the developer to make a mental shift in order to use these constructs effectively.

6.4 Exceptional Exception Handling

While developing software that can respond to events may seem challenging enough, what happens if something goes wrong? The traditional notion of a two-phase commit transaction is inherently based on a predictive, relatively tightly coupled model. It also requires a sophisticated 2-phase interaction model between the transaction initiator, the individual resources and a transaction coordinator. In service-oriented, loosely coupled environments this style of interaction is often unavailable or undesirable.

Instead, new error handling strategies like retry or compensation will be much more common in the service-oriented world [6]. Once again the concepts are not new (for example, we have known about Sagas [7] for over 15 years) and many tools such as BPEL engines have built-in support for long-running transactions and compensating actions. However, patterns and best practices for the effective use of these new types of exception handling are not yet widely known in the developer community. Similar to pattern-based design of object-oriented solutions it will likely take many years for the developer community to embrace these new interaction models.

6.5 Doodleware

Another programming style that has been at the (literal) forefront of distributed system development is the visual programming environment, sometimes disparagingly referred to as "doodleware". Visual representations are enormously useful to display parallel activities over time or dependency graphs and are widely used for process and orchestration models. Programming in a visual environment takes some getting used to, though. For example, it is often more difficult to scale to large solutions. Some developers have already suggested that visual process modeling tools should ship with a second monitor to provide the necessary screen real estate. Also, common text processing utilities like "diff" or even "find-replace" typically do not exist in the visual world. Debugging can also be harder, even though I have seen some nice visual debuggers appear recently. I think it will take quite some time until we will see mature visual development environments that are on par with the likes of IntelliJ IDEA or Eclipse in the text-based world.

7 Conclusion

Developing in a service-oriented world will remain interesting for quite some time to come. The advent of Web services has certainly taken some of the bitter flavor out of "EAI" and has made distributed system development a mainstream activity. However, the collective understanding of a new architectural style and the evolution of appropriate tools will occur only gradually over the next years.

A lot of the first generation of SOA tools can be deceptive. Developing a successful SOA is not a simple matter of dragging and dropping a few icons. For developers, service-orientation means learning new programming models and techniques. Changing developers' mindsets to effectively use these new models might be the biggest hurdle for successful adoption of SOA yet.

About the Author

Gregor Hohpe leads the Enterprise Integration practice at ThoughtWorks, Inc., a specialized provider of application development and integration services. Gregor is a widely recognized thought leader on asynchronous messaging architectures and co-author of the seminal book "Enterprise Integration Patterns". Gregor speaks regularly at technical conferences around the world and maintains the Web site www.eaipatterns.com.

References

[1] Web Services are not Distributed Objects, Werner Vogels, IEEE Internet Computing, Nov-Dec 2003

[2] Patterns of Enterprise Application Architecture, Martin Fowler, 2002, Addison-Wesley

[3] Enterprise Integration Patterns, Gregor Hohpe, Bobby Woolf, 2003, Addison-Wesley

[4] Visualizing Dependencies, Gregor Hohpe, http://www.eaipatterns.com/ramblings/11_dependencies.html

[5] Enterprise Service Bus, David Chappell, 2004, O'Reilly

[6] Starbucks Does Not Use 2-phase Commit, Gregor Hohpe, http://www.enterpriseintegrationpatterns.com/ramblings/18_starbucks.html

[7] Sagas, Hector Garcia-Molina, Kenneth Salem in Proc. ACM Sigmod, 1987

[8] Production Workflow, Frank Leymann, Dieter Roller, 2000, Prentice-Hall PTR