SOA Patterns – New Insights or Recycled Knowledge?

Gregor Hohpe, Google

gregor@hohpe.com

Abstract

Design Patterns have enjoyed enormous popularity in the software community and have become somewhat of a fad. This can make it challenging to distinguish patterns that convey new knowledge from recipes or tutorials that have been cast into patterns form. This article examines the role patterns play in the adoption of new technologies and architectural styles. In particular it describes why the shift to service-oriented architectures requires us to discover, document, and share new patterns.

Biography

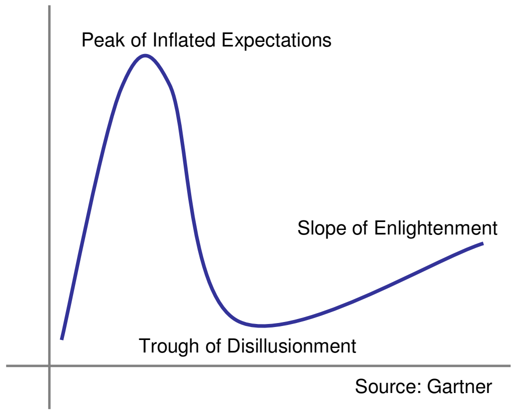
Gregor Hohpe is a software architect with Google, Inc. Gregor is a widely recognized thought leader on asynchronous messaging architectures and service-oriented architectures. He co-authored the seminal book "Enterprise Integration Patterns" and speaks regularly at technical conferences around the world. Find out more about his work at www.eaipatterns.com.

Software's Fascination with Patterns

Design Patterns have enjoyed enormous popularity in the software community. A recent search for the keyword "patterns" on an on-line computer bookstore yielded 135 current titles. This phenomenon naturally begs two questions: "why are patterns so popular in the software community?" and "do patterns really provide value in all of these scenarios?"

Patterns encapsulate knowledge. Ward Cunningham has often described them as "mind-sized chunks of information". This makes the wide adoption of patterns in the knowledge-intensive IT industry unsurprising, especially given the short half life of knowledge. However, many other science and engineering disciplines are too based on constant knowledge creation and dissemination yet rarely use patterns for knowledge capture. So what makes patterns so appealing in the software industry? Unlike most other engineering disciplines the software industry is still characterized by the lack of a precise vocabulary. While computer science has established solid theoretical foundations, designing complex software systems tends to be a much less structured activity than designing buildings or machines. In the absence of an industry-accepted vocabulary patterns fill in a critical gap by establishing a common language. Since patterns are not meant to be precise definitions and do not have to map into an overarching metamodel, they can also be "soft" around the edges, conveying knowledge without pretending to be engineering specifications. The ability to document knowledge without having to create a definitive step-by-step reference is particularly valuable in a field that uses a "soft" medium. Software developers usually start with a blank slate and are constrained by very few factors beyond the language syntax. There are no electric codes that have to be met or standard measurements to be used. In such an environment patterns guide the designer as opposed to prescribing a specific solution.

The early successes of software patterns induced a network effect and a bit of a fad. Software professionals started to document patterns in many related areas, ranging from managing projects, holding stand-up meetings, or writing more patterns. Soon, the publishing industry began to use the word "pattern" to label books that offer design guidance as opposed to describing the latest programming interface, indicating that patterns are not immune to the often cited "hype cycle" of new technologies [1]. Once patterns reached the Peak of Inflated Expectations they were frequently hailed as the silver bullet that will make software cheaper, better, and less buggy. Since then patterns have slid somewhat into the Trough of Disillusionment – we once again realize that silver bullets may work in the world of mythology but not in software engineering. But entering the Trough is by no means a sign of failure but rather indicates widespread acceptance; it simply indicates



that the community uses patterns more extensively and realizes that they cannot fulfill all (inflated) expectations.

Dissecting Knowledge

The popularity of patterns leaves many developers to decide which patterns document new useful knowledge versus which ones just use the pattern format to increase readership. The situation is complicated by the fact that patterns present solutions that may be applicable in a wide range of domains. For example, one can find patterns such as *Correlation Identifier* [2] in Web services specifications or business protocols as well as in low-level communication layers such as TCP. Did the pattern author really discover new insights or simply recycle existing knowledge?

Let's address this question by coming back to the original purpose of patterns: to chunk up and convey knowledge. Having chunked up knowledge is particularly useful if you are working with new and less familiar technologies. But as patterns are applicable across technologies (for example, most patterns in [3] can be implemented in both Java and C#) a new technology alone does not imply a new set of patterns. To better understand this phenomenon it helps to categorize our knowledge of programming and systems building into distinct levels:

Syntax specifies the mechanics of making the machine understand the solution the developer in his or her head. One could posit that syntax is an artifact of our crude input methods; if we didn't have to type the solution into lines of characters we might have to worry less about where the curly brace goes or why variable names cannot start with a digit. Some programming models such as visual process modelers shield the developer from having to understand the syntax or defer the syntax question to XML files. We quickly realize though, that understanding a system's syntax is a necessary but by far not a sufficient condition to creating viable solutions.

Constructs are the elements of the programming model. In object-oriented languages we use constructs such as classes and the relationships between them, for example inheritance or association. Constructs gives us the framework for expressing our solutions, such as dividing responsibilities across a set of classes. Understanding the constructs of a programming model or an architectural style typically requires a certain amount of abstract thinking but the definitions often fit on just a few pages of text. The main reason lies in the fact that the constructs, just like the syntax, only describe legal system configurations but give no guidance on how to come up with a suitable solution or how to distinguish a good solution from a bad one.

Principles help us come up with good solutions that take advantage of the underlying constructs. For examples, the principle of *Separation of Concerns* teaches us to separate our logic across classes (or services) instead of sticking it all into a single element. The *Open-Closed* principle [4][5] reminds us that a class should be open for extension but closed for modification. These principles are designed to hold almost universally true, which turns out to be their biggest strength and weakness at once. Applying all principles at once can result in overly complex and sometimes even contradictory solutions. The key is to find the right *balance* between these principles to arrive at a sensible solution.

Patterns apply the principles to a specific context and present a concrete solution that balances often contradictory forces. For example, *Two-step View* [3] applies *Separation of Concerns* to rendering user interface screens: it instructs us to separate the generation of logical data from the actual rendering of the screen. Still, the pattern tells us a lot more than the principle: it draws a specific context, explains the forces at work, spells out the specific concerns we want to separate, provides a concrete solution, and points us to related patterns that might be of interest. Thus, patterns guide developers to a solution that balances design principles to address a concrete problem.

The Language of Patterns

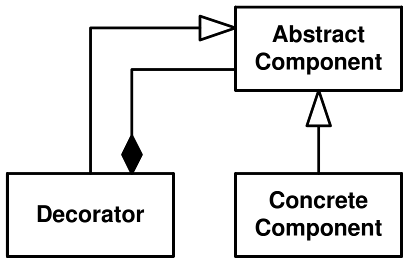


Figure 1: Decorator Pattern for Object-oriented Systems

Patterns are characterized by a strong solution focus [6]. To describe the solution, patterns use the language of the underlying constructs. For example, the *Decorator* pattern [7] describes how to attach additional responsibilities to an object by creating another class that implements the same interface and delegates calls to the original instance by keeping a reference (see Figure 1). Clearly, the solution relates to objects, classes, interface implementation, delegation – the constructs of object-oriented programming.

The ability to express patterns using programming constructs, as opposed to code, explains why many patterns apply across programming languages. For example, C# and Java share the same object-oriented programming model and thus the same vocabulary to express patterns. This means that in order to discover entirely new sets of patterns we need to look not only for new technologies but also for a shift in programming model or architectural model.

SOA with an Uppercase A

The biggest shift in mainstream programming or architectural model today is toward Service-oriented Architectures (SOA). After much confusion about the relationship between Web services and SOA, promoters of service orientation place the emphasis of the acronym on the letter "A" to highlight the fact that SOA is an *architectural style* as opposed to collection of specific technologies. According to Garlan and Shaw [8] an architectural style determines the vocabulary of components and connectors that can be used together with a set of constraints on how they can be combined. Common examples of architectural styles include *Pipes-and-Filters*, *Object-oriented Organization*, or *Layered Systems*.

Each architectural style defines a set of constructs and therefore provides a different foundation for solution patterns. New architectural styles also change the way developers think about solutions, highlighting the need for new guidance in form of patterns. For example, [2] presents a language of 65 patterns based on the *Pipes-and-Filters* architectural style.

If SOA is truly a new architectural style we should expect it to form the basis for many new patterns. However, opinions often diverge on this matter: some herald SOA as the biggest shift in thinking about systems in the past decade while others challenge it as simply recycling connected systems knowledge from about 20 years ago. As in most cases, there is a bit of truth on both sides of the argument. Despite the rapid rate of innovation, or maybe because of it, the software world is prone to rediscover many techniques over and over, and SOA may be no exception. However, there is a big step from some people possessing knowledge of connected systems architecture to millions of developers constructing systems in a consistent and well understood way. In my opinion it is this shift into the mainstream that makes SOA significant. And it is also the reason we can expect design patterns to play a significant role in SOA adoption: they package up knowledge for consumption by a wide range of consumers, i.e. developers. Reading a WS-* specification is a lot less relevant to most developers than understanding the common usage patterns and how they can be mapped onto the available technologies.

SOA Patterns

Giving SOA the benefit of the doubt as a new architectural style, what categories of new patterns should we expect to see? First, we need to understand that SOA impacts system design and development at many different levels. The "napkin sketch" in Figure 2 [9] depicts a typical interaction between services in a service-oriented architecture.

In this scenario the service on the left accesses the service registry to discovers another service and interacts with it by sending a message, for example placing an order. This message is often part of a longer conversation between the services, for example an order, followed by an acknowledgement, an invoice, payment notice, etc. Some of these messages might be encrypted, or require authentication. In this example the service provider is a composite service, i.e., a service that uses other services to fulfill its responsibilities. For example, an order service might need to access inventory or pricing services in order to accept an order or issue a quote. The implementation of such a composite service is frequently performed by an orchestration engine [10], an element optimized to execute a multi-step process, which includes interactions with other services. A rules engine may guide the orchestration engine in the execution of the process by incorporating business rules, such as orders over a certain amounts being handled with priority. Figure 2 assumes that the composite service interacts with stands-alone services, i.e. services that do not require other services. These services are implemented by custom or packaged applications exposed as service endpoints. These endpoints manage the translation between the asynchronous world of messaging and the synchronous, oriented-oriented application program. Because different applications and services tend to use incompatible data formats a translation of the message is often required along the way.

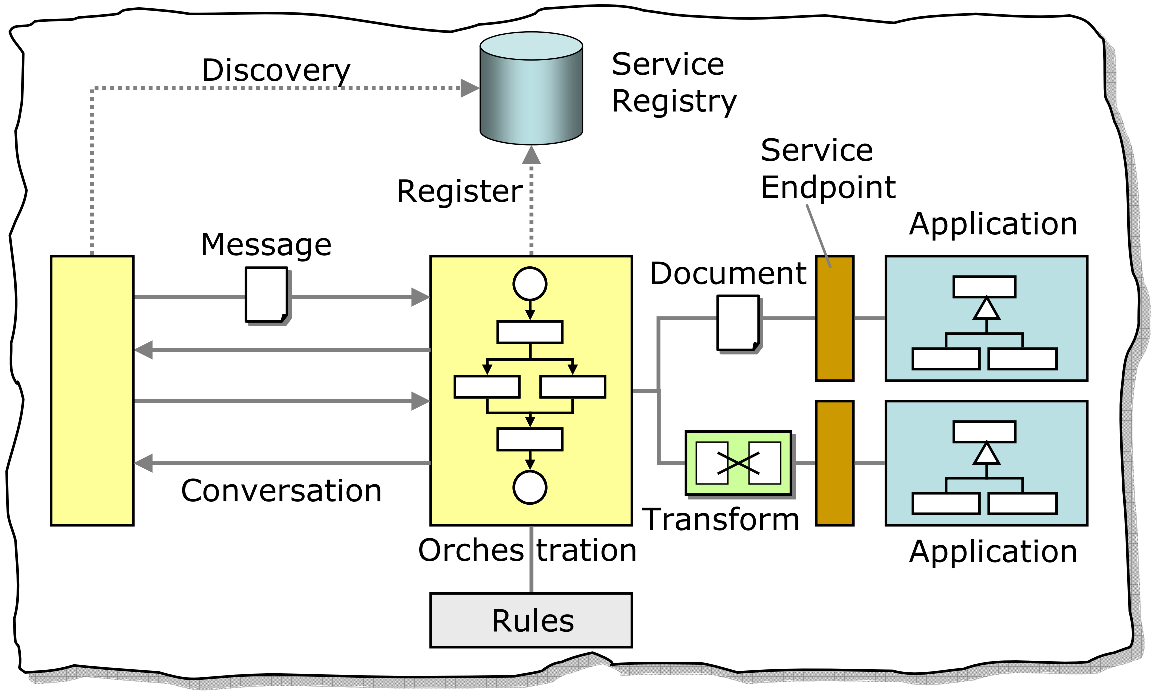


Figure 2: "Napkin Sketch" of an SOA Implementation

This (over-)simplified overview of an SOA implementation nevertheless exposes new and unfamiliar programming models. These models require a different way of programming, a different way of thinking, and also new guidance and patterns.

Programming Models

The above example highlights the following new programming models, which are different from traditional object-oriented application development:

- Composition
- Process models
- Declarative programming
- Event-driven programming

Composition

Service-oriented solutions are *composed* from individual services. This aspect of SOA is often not considered programming because it is concealed behind configuration files or visual editors. However, this activity has an underlying programming model, the Pipes-and-Filters style, and requires encoding, testing, debugging just as any other programming language [11]. Because the composition is often done at (or after) deployment time a disciplined development, test and release cycle is essential for this programming activity.

Process Modeling

As the previous example showed, composite services use orchestration to coordinate the interaction across multiple services. In fact, the separation of process from the functional assets (i.e. services) is one of the major benefits of SOA. The vocabulary of process models is quite different from object-oriented programming as it includes tasks, branches, and synchronization points. The long running nature of these processes also requires a very different approach to exception handling than typical application programming.

Declarative Programming

The transformation and rules engines in the above example share a similar programming model, albeit mapped into very different syntax. Both components are programmed in a declarative style as opposed to the sequential-procedural style most developers are comfortable with. For example, most transformation tools are based on XSLT, which matches incoming documents against templates and outputs a transformed document. Similarly, rules engines match incoming data against a set of rules. Often, this type of programming is hidden behind visual editors, but the challenges of the declarative model usually remain. Because the execution order of these declarative "programs" is not specified inside the code but is determined at run-time, testing and debugging tend to be more difficult. In general, declarative solutions are elegant to read, difficult to program and incredibly hard to debug.

Event-based Programming

Services communicate through messages, which are often the result of business events, such as a new order or a low inventory alert. Programs handling these messages cannot assume a predefined execution order but have to react to incoming events as they occur. Once again, this style of development is very different from traditional application development, which allows the developer to control what happens when and in what order [12]. Event-driven programs essentially abolish the notion of a call stack, which prescribes the execution flow through synchronous method calls and local variables. Instead, the programmer has to manage continuations and state explicitly as events arrive not necessarily in the anticipated order. Many orchestration engines aid in the development of event-based processes, but the developers are still required to grasp concepts like correlation, convoys etc.

Each of these programming models brings its own constructs, principles, and patterns. While none of these programming models are entirely new (a theme common with SOA), the widespread adoption will make it easier to harvest patterns from actual use. It'll also increase developers' collective appetite for design guidance in form of patterns. Hence we should expect to see new publications in these areas.

New Patterns

Because SOA is more than a new programming paradigm we should expect to see new patterns and pattern languages beyond the programming models above.

We can, for example, expect patterns that describe the **operational infrastructure of SOA**, such as service registries, orchestration engines, routing services and the like. While many SOA frameworks and products will have most of the patterns incorporated into the platform it is still valuable for developers to understand the "why" in addition so simply following the "how" prescribed by the vendors.

Service-oriented architectures are large, dynamic, distributed systems that need to be managed actively. Patterns can help system architects and operational engineers manage complex system configuration and also document insights on the **tuning an SOA** for different parameters, such as throughput, latency, resiliency etc.

Because service-oriented architectures form highly interconnected systems security plays a significant role. We should therefore expect to see a renewed focus on **security patterns**. While security is not a new concern, SOA provides a new context in which many of the general techniques apply.

Last but not least the shift towards SOA is not merely about technology. Converting a whole enterprise to a new architectural model requires change management and a clear strategy on how to change the proverbial wheels on a moving car. SOA **organizational patterns** should be document the evolving body of knowledge in this area.

Current Patterns Work

Trying to catalog ongoing work in patterns related to SOA would be an enormous task and would likely not be much more useful than a few Web searches. Therefore, I want to highlight just a few efforts that are currently under way or have been published recently.

Architectural Patterns guide architects in the overall composition of a service-based solution:

- SOA Patterns [10] and [13] guide architects in the overall composition of a service-based solution.
- Service discovery patterns [14] describe the motivation behind service registries and the trade-offs between different mechanisms for service discovery.
- Security patterns [15]are not limited to SOA but are particularly relevant in that context.

- Asynchronous Messaging Patterns [2] describe how to design asynchronous messaging solutions.
- Conversation Patterns [16] describe ways multiple system communicate using messages.
- Service Interaction Patterns [17] describe the basic forms of inter-service communication. Conversation patterns are composed of service interaction patterns.
- Workflow Patterns [18] describe the common elements of constructing processes and orchestrations.
- Orchestration Patterns [10] describe the key components that power orchestration engines.
- A Pattern Language for Distributed Computing [19] ties together many of the patterns that underlie distributed systems.

Usage before Patterns

With such a flurry of activity one has to wonder about the timeline and quality of the patterns documentation. Patterns authors are sometimes criticized for delivering useful information a few years too late. However, patterns are not created by a few authors but instead *harvested* or *mined* from actual usage in the community. Therefore, patterns cannot be documented before actual usage has built up enough experience and knowledge to be worth documenting. Furthermore, good pattern descriptions are typically the result of community collaboration. Many pattern authors share their work in progress with the community to solicit feedback and refine their work. This implies that pattern documentation can lag a shift in thinking by a few years.

Knowing that patience is not a particular virtue of the software industry we can expect to see a few false starts in SOA patterns as well. Adoption of a new architectural style can take considerable time. Arguably, object technologies required a decade or more to go main stream. This poses a challenge for tools and platform vendors who are expected to have quarterly releases introducing new features. They are therefore tempted to provide guidance that is more strongly influenced by their specific tools as opposed to common usage in the developer community. The popularity of the pattern format might entice some of them to cast rather specific usages into the form of a common pattern, potentially misleading developers.

How can developers tell the "good" patterns from the "bad" patterns? A good pattern encapsulates actual knowledge and design guidance as opposed to just being an example. A good pattern describes more than a solution, but discusses forces and explains the "why" in the addition to the "how." Patterns do use specific technologies as an example implementation but the solution itself should focus on the architectural concepts, not the implementation technology. Lastly, most good patterns do not apply universally but describe when they are (and when they are not) applicable.

Patterns and Standards

So far I have focused on SOA ads an architectural style and have largely ignored the Web services standards. However, because the wide adoption of standards and specifications drive the adoption proliferation of service-oriented architectures one should not dismiss Web services as just another technology. Likewise, the definition of these standards has an enormous impact on the development of SOA solutions as they shape the vocabulary, tools, and often the thought patterns of SOA developers.

Unfortunately, standards committees are often tasked with creating specifications before the usage patterns are well known or understood. Because standards specifications have to remain static to be consistently adopted, many resulting standards can make the implementation of popular patterns cumbersome or even impossible [18].

Incorporating common patterns into the standards development process can help ensure that the standards are not purely based on architectural models but also on actual usage in the community. Because good patterns come only with actual usage this has likely to be done in an iterative process.

Conclusion

Patterns have gone mainstream and have occasionally become the victims of their own success. As pattern books and papers flood bookstores and the Web we should keep an eye out for many new useful patterns that teach us how to effectively design, implement and manage service-oriented architectures. We have to be patient, though, as good patterns can only come from real experience.

Bibliography

[1] *Hype Cycle*, Gartner, http://www.gartner.com/pages/story.php.id.8795.s.8.jsp

[2] *Enterprise Integration Patterns*, Hohpe, Woolf, Addison-Wesley, 2003

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

[4] *Object-oriented Software Construction*, Meyer, Prentice Hall, 1997

[5] *Open-closed Principle*, Martin, http://www.objectmentor.com/resources/articles/ocp.pdf

[6] *Writing Software Patterns*, Fowler, http://martinfowler.com/articles/writingPatterns.html

[7] *Design Patterns*, Gamma et al, Addison-Wesley, 1995

[8] *An Introduction to Software Architecture*, David Garlan and Mary Shaw, 1994, CMU-CS-94-166

[9] *Developing in a Service-Oriented World*, Hohpe, January 2005, GI Lecture Notes in Informatics P-65

[10] *SOA Enterprise Patterns*, Lubinsky, Manolescu, http://orchestrationpatterns.com

[11] *Good Composers are Few and Far in Between,* Gregor's Ramblings, http://www.eaipatterns.com/ramblings/19_composing.html

[12] *Programming Without a Call Stack – Event-driven Architectures*, Gregor Hohpe, http://www.eaipatterns.com/docs/EDA.pdf

[13] *SOA Patterns*, Rotem-Gal-Oz, http://www.soapatternsbook.com

[14] *Pattern Language for Service Discovery*, in Pattern Languages of Program Design 5, Manolescu, Voelter, Noble, 2006, Addison-Wesley

[15] *Security Patterns*, Schumacher et al, 2006, Wiley

[16] *Workshop Report: Conversation Patterns*, Hohpe, June 2006, Dagstuhl Seminar Proceeings 06291

[17] *Service Interaction Patterns: Towards a Reference Framework for Service-based Business Process Interconnection*. Barros, Dumas ter Hofstede: Technical Report FIT-TR-2005-02, Faculty of Information Technology, Queensland University of Technology, Brisbane, Australia, March 2005.

[18] *Workflow Patterns*, Aalst et al., http://is.tm.tue.nl/research/patterns/patterns.htm.

[19] *Pattern-Oriented Software Architecture, Vol. 4*, Buschmann et al, 2007, Wiley