Software Service Engineering – Architect's Dream or Developer's Nightmare?

Gregor Hohpe

Google, 1600 Amphitheatre Parkway, Mountain View, CA 94043
gregor@hohpe.com

Abstract. Architectural principles such as loose coupling are the key drivers behind the adoption of service-oriented architectures. Service-oriented architectures promote concepts such as composition, process modeling, protocol design, declarative programming, event-based programming, and object-document mapping. These architectural ideals can be fraught with challenges for developers who are faced with unfamiliar programming models and immature tools. This paper briefly reviews the service-oriented architecture concepts and highlights the most pressing challenges for developers. These challenges suggest several focus areas for tool builders and software service engineering researchers.

Keywords. Event-based programming, declarative programming, object-document mapping, patterns, process modeling, protocol design, service composition, software engineering, SOA

Introduction

Service oriented architectures provide many desirable properties. Solutions are composed from independent services, which communicate in a loosely coupled fashion through message exchanges. Meanwhile, an orchestration layer executes business processes, aided by explicit business rules. Stateful message exchanges are specified and tracked through choreographies. Application-specific data formats are hidden behind a transformation layer, which converts them into a canonical data format. Supporting systems provide an inventory of all services and their associated contracts. It's an architect's dream come true.

But building and maintaining such a system is far from trivial. Many of the key building blocks of an SOA present developers with new and potentially unfamiliar programming models [1]. For example, designing and implementing orchestrations requires a good understanding of process modeling, including long-running transactions. Asynchronous communication typically requires an event-based programming style with explicit state management. Business rules are declarative and can be challenging to test and debug. On top of all that, highly dynamic and recomposable systems can exhibit complex run-time behavior, which calls for new tools and languages. The architect's dream may end up becoming the developer's nightmare.

New Programming Models

The large majority of enterprise developers create business applications using an object-oriented programming model. Object-orientation provides rich constructs to express the relationships between objects, making it ideal for the modeling of complex business domains. However, service-oriented architectures present developers with different programming languages and models. While most of the models are not entirely new, they are mostly unfamiliar to today's enterprise developers, particularly considering high levels of proficiency, such as making design trade-offs, understanding common patterns and anti-patterns, refactoring, debugging, etc. It took the object-oriented community more than a decade to achieve widespread understanding and tool support, so we should carefully consider the challenges facing service-oriented developers.

The best way to illustrate how the new programming models come into play is by looking at a casual SOA architecture diagram, which we refer to as the *architect's napkin (see Fig. 1).*

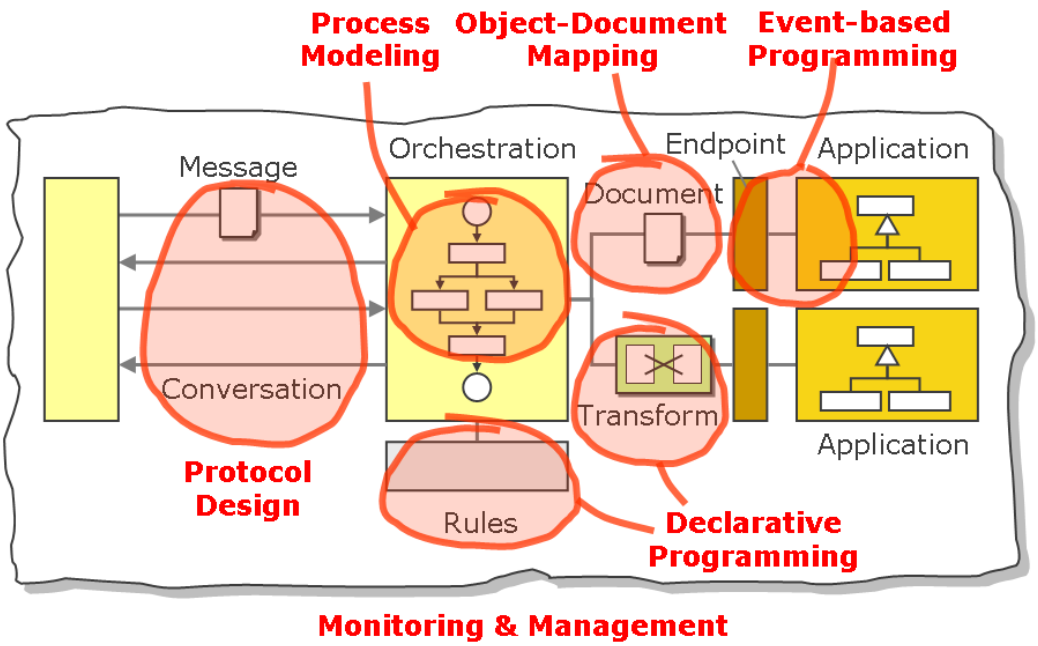

Fig. 1. Architectural building blocks and programming models in software service engineering

Let's briefly examine each of the programming models and its implications.

Composition

Service-oriented solutions are *composed* from individual services. This aspect of SOA is often not considered programming because it is usually concealed behind configuration files or visual editors [2]. However, this design activity is based on a programming model derived from the Pipes-and-Filters architectural style. Just like "normal" programming, composition requires coding, testing, and debugging.

Process Modeling

The separation of executable processes from the supporting functional assets (i.e. services) is one of the major benefits of SOA. In too many business environments, vital business logic is scattered across a mix of packaged and bespoke applications, with process state being stored in some application's database field or memory. Extracting this logic into a separate layer makes the processes visible and maintainable.

However, the vocabulary of process models is quite different from object-oriented programming: processes are composed of activities, branches, and synchronization points, process instances are identified through correlation sets, and state is persisted through checkpoints. Because processes are long-running, they have to manage consistency explicitly, e.g. through compensating actions, instead relying on ACID transactions. Process models also tend to have a large space state, which can make testing difficult.

Protocol Design

Many services engage in stateful conversations, especially in a business-to-business scenario. Also, even simple exchanges can become stateful as soon as they have to account for retries or time-out conditions. The state of such a conversation is often tracked inside a process model, but such a process model only represents a single participant's view of the conversation as opposed to the global state of the conversation. This discrepancy has been the source of a lengthy debate over choreography vs. orchestration. Suffice it to say that designing conversations can be at least as complex as designing a process: you have to account for the state of multiple independent participants, avoid deadlock

situations, etc. Designing conversations resembles protocol design, which is widely known as non-trivial and again unfamiliar to most enterprise developers.

Declarative Programming

The transformation and rules engines employed in service-oriented architecture, rely on a declarative style as opposed to the sequential-procedural style most developers are comfortable with. For example, many transformation tools are based on the XSLT[1] language, which matches incoming documents against templates and applies the corresponding transformation rules. Similarly, rules engines match incoming data against a set of declared business rules. 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 an incoming order or a low inventory alert. Programs react to incoming events as they occur, without assuming a particular execution order. Again, this style of development is quite different from traditional application development, which allows developers to control what happens when and in what order. This also means programming without the assistance of a call stack, which relies on linear execution flow through synchronous method calls and local variables. Instead, the programmer has to manage continuations and state explicitly. Many orchestration engines aid in the development of event-based processes, but the developers are still required to grasp concepts like correlation, sagas, etc.

Object-Document Mapping

Messages exchanged between services contain documents. Documents are subtly, but significantly different from objects. 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. As a result, documents tend to be more coarse-grained to aid in encapsulation between components and to reduce network "chattiness". Documents also have no inherent notion of inheritance or polymorphism. While these discrepancies are subtle, they resemble the impedance mismatch between objects and relation databases. O-R mapping has been an active topic for more than a decade.

Behold the Run-time

On top of these programming models, service-oriented solutions tend to have a more dynamic run-time than traditional systems: new services are provisioned, while other service retire, popular services may be spread across multiple instances, load balancing may come into play, and existing services may be proxied to satisfy new requirements. Service level agreements have to be monitored while vital metrics such as latency and throughput should trigger alerts if they fall outside acceptable ranges. Rule-based systems route such alerts to an appropriate party.

While many of these functions are a routine part of IT operations, the finer granularity and higher dynamicity of service-oriented solutions transports these topics into the realm of developers. They come with a new set of tools, again with new languages and programming models.

[1] eXtensible Stylesheet Language Transformation

Explicit better than Implicit

Do these concerns mean that you should avoid service-oriented solutions? Certainly not. They highlight two important facts, though: that transitioning from informal diagrams to implementation is likely more difficult and time-consuming than it may initially appear. Second, it will take some time (as in years) for common best practices to be established and widely understood.

One of the key benefits of SOA is that it makes implicit concepts explicit. For example, SOA relies on an explicit orchestration layer to represent business processes as opposed to a mix of business logic and private state. While designing, debugging, and testing such processes is far from trivial, having the processes scattered throughout application code and stored procedures is certainly worse. In the latter scenario, the processes are most likely completely untested because no one really understands what they are or where they are implemented.

Tooling Support Still Weak

SOA and ESB vendors like to point at their tool suites as the solution to the developer's conundrum. However, these tools often provide only a thin veneer over the programming models highlighted above. While tools can certainly increase developers' productivity by shielding them from technical and syntactic details, such as WSDL documents, they still require developers to understand the underlying model. A developer composing orchestrations using a visual tool still needs to deal with correlation sets, compensating actions, and sagas. Various attempts to hide the underlying model completely have so far mostly led to overly rigid and limiting development environments. Instead of trying to dumb down development, we should look for tools that embrace the programming model and make developers more productive, much like modern Java or C# IDE's embrace the object-oriented model and equip the developer with powerful tools such as refactoring and real-time code analysis.

These tools also need to support an iterative and incremental development cycle. Features like copy and paste, refactoring, step-by-step debugging, and integration with source code repositories are just a few of the required features. Ideally, the tools also support widely adopted techniques such as test-driven development, red-green-refactor cycles, and the like.

In these aspects, many SOA tools are still fairly limited. For example, when we worked with a popular orchestration designer, our "best practice" was to write each orchestration twice. First we would iterate until we obtained the desired behavior, then we would start all over, creating a clean implementation from scratch. This approach was more efficient than trying to clean up the first version.

Patterns

Design patterns have been instrumental in exposing developers to the subtle design trade-offs presented by object-oriented systems. But these patterns have limited applicability to the design challenges inherent in service-oriented solutions because the underlying programming models differ [3]. We will need new pattern languages to capture and distribute developer's knowledge in building such systems. These languages can be developed only over time, as the collective understanding in building such systems gradually solidifies.

Conclusion

Service-oriented architectures have demonstrated that they are the right answer to today's rapidly evolving and interconnected business environments. However, software service engineering involves a lot more than architecture diagrams, presenting developers with unfamiliar programming models and design trade-offs. We have to take this into account when estimating the effort to build SOA solutions and when identifying risks. After all, software schedules tend to slip one debugging cycle at a time.

References

[1] Hohpe G., Developing Software in a Service-Oriented World, in: 11. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme", GI Lecture Notes in Informatics P-65
[2] Hohpe G., Good Composers are Few and Far in Between, Gregor's Ramblings, http://www.eaipatterns.com/ramblings/19_composing.html
[3] Hohpe G., SOA Patterns – New Insights or Recycled Knowledge?, http://www.eaipatterns.com/docs/SoaPatterns.pdf