JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2003**

Vol. 4, No. 7, September-October 2005

Confessions of a Service-Oriented Abuser

Mahesh H. Dodani, IBM Software, U.S.A.

1 MEET PAT TERNA BUSER

"*One comment I saw in a news group just after patterns started to become more popular was someone claiming that in a particular program they tried to use all 23 GoF patterns. They said they had failed, because they were only able to use 20. They hoped the client would call them again to come back again so maybe they could squeeze in the other 3.*

Trying to use all the patterns is a bad thing, because you will end up with synthetic designs—speculative designs that have flexibility that no one needs." – Erich Gamma, http://www.artima.com/lejava/articles/gammadp.html

Hello, my name is Pat, and I resemble the person that Erich is talking about in his quote above. I introduced my pattern abusing story to you in 1999 [Dodani, M., "Rules are for Fools, Patterns are for Cool Fools", Journal of Object-Oriented Programming, Vol. 12, No. 6, pp. 21-23, SIGS Publications, October 1999], and asked for help in getting over my abuse.

To summarize my abuses, I applied patterns mercilessly to any software development project, and in fact came up with a "pattern" to abuse patterns – which ensured that I could apply almost every pattern (yes, I was disappointed that I could not apply all 23 as well) to the implementation of a single class hierarchy (I assume that you are familiar with the Gang of Four Patterns):

- For a single class, I used the State pattern to implement the instance variables to facilitate flexibility in changing the behavior of the instance based on its state and the Stategy pattern to implement the methods to facilitate flexibility in the choice of algorithms. For the more complex methods, I used the Bridge pattern to ensure that the interface of the method was decoupled from the actual implementation, providing even greater flexibility. I used the Adaptor pattern to implement the interface defined by the class using existing legacy code.
- For a grouping of instances of a class, I applied the Chain of Responsibilitiy pattern to facilitate the flexibility of allowing the handler of the request to be

Cite this column as follows:Mahesh Dodani: "Confessions of a Service-Oriented Abuser", in Journal of Object Technology, vol. 4, no. 7, September-October 2005, pp 19-24, http://www.jot.fm/issues/issue_2005_09/column2

determined dynamically. Additionally, I used the Composite, Iterator and Visitor patterns to group the instances into a tree-like structure and to facilitate uniform access and treatment of the organized objects. For a hierarchy of classes, I used the Façade pattern to ensure that the entire set of classes could be accessed through a single interface which I could separate from the implementation. To create objects that are part of a family, I used the Abstract Factory, Builder and Factory Method patterns to facilitate flexibility in creating instances. In some cases, I achieved greater flexibility using the Prototype pattern to clone objects from existing ones rather than creating them as instances from a class.

- For the clients of objects, I used the Observer pattern to facilitate flexibility in how the client reacts to changes in the object. Furthermore, I used the Proxy pattern to completely decouple the methods provided by the object from the manner and location from which they need to be accessed.

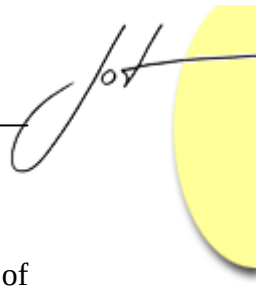
As I became more abusive, I requested help from you in resolving the following issues:

- How do I know when a pattern is really applicable, and will have the needed impact on the application?
- How do I know that the application of the pattern has made the application more flexible and reusable?
- Will the application become "better" after the patterns are applied? How do I define "better", and determine that the application has become "better".

2 REMISSION FROM ABUSING PATTERNS

The outpouring of support and help from the community was overwhelming. I got help that focused on several facets of my abuses – including, unfortunately, how to increase the number of patterns that I could use in my approach above! The key antidotes that I received that helped me overcome my sickness were the following:

- AntiPatterns which documents bad practices using the same template as patterns. It therefore identifies common problems with solutions, and then show how to refactor the solution to get rid of the problem. Several of these antipatterns deal with my problem of applying patterns just for the sake of applying patterns (e.g. BigDesignUpFront, DesignForTheSakeOfDesign, PolterGeists) and other (not really categorized as antipatterns) ways of handling "smells" in your design and implementation. I was able to leverage these antipatterns and their refactored solutions to bring back some sanity into my design and coding practices, and slowly wean myself off using patterns for the sake of patterns.
- Using larger grained assets through a gradual progression from design patterns to frameworks and finally to architectures. My first step was to move from design patterns to components, which allowed me to break away from the fine-grained confines of objects to the large-grained world of components, and from the



constricted inheritance-based flexibility to the more powerful world of composition. From components, it was easy to transition into architectures, which provided the most effective antidote to my design pattern abuses. Layered architectures allowed me to separate functionality and concerns between layers that have well defined interfaces (and consequently hidden implementations) to interact with – typically layers (from the outside in) include presentation, business logic, persistence, integration and enterprise applications. Within each layer, I can implement the required functionality (maybe through components, and the components in turn are implemented via objects.) The next step was to evolve to component architectures where the components themselves were considered first class citizens, and this led to using interacting components as the primary mechanism to build flexible systems.

- Agile methods facilitate individuals interacting with each other and in close collaboration with their customer to build incremental versions of the system and quickly respond to needed changes. Agile methods forced me to focus on particular parts of the system, and build flexibility for one piece of functionality at a time.

Using the above three antidotes, I was able to push myself into remission from abusing patterns. The architectures, frameworks, and components that I was having to deal with in addition to the design patterns made my design and development life complex and engrossing. These higher-level abstractions allowed me to use a model-driven approach to my development, so that by the time I came to implementing the components with objects, I knew which patterns would be applicable to provide the flexibility in the smaller context of implementing the needed interface. The agile methods gave me further focus by facilitating iterative and incremental feature/function driven design.

3 RELAPSING INTO ABUSING SERVICES

As you must be aware, a few years ago, the entire technical world was awash with services – here was the ultimate construct for building flexible systems. A Service is a discoverable software resource which has a service description. The service description is available for searching, binding and invocation by a service consumer. The service description implementation is realized through a service provider who delivers quality of service requirements for the service consumer.

I got introduced to services through web services. Web Services are self-describing, self-contained, modular applications that have to be described, published, found, bound, invoked and composed. Web services conform to the structure summarized in Figure 1.

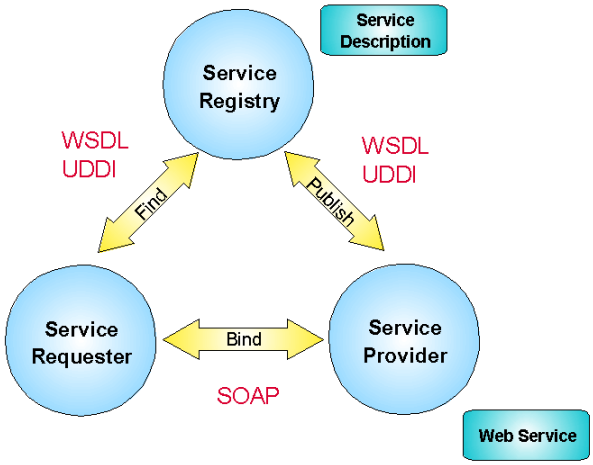
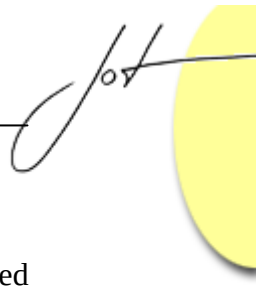


Fig. 1: The Web Service Structure

As shown in the figure, service providers create application functions that are available on disparate implementation platforms as web services and describe them using a standard definition language. These services are published in a service registry**.** Service requestors who need a particular type of service search the service registry and find the desired service. Once a service is found the requestor and the provider of the service negotiate to access and invoke the service.

The other beauty of web services is that they are standards based, implying that the underlying technologies supporting web services need to be platform and implementation neutral, and are therefore specified in XML. The core technologies include

- Simple Object Access Protocol (SOAP) is the XML based messaging specification that defines the message envelope content, the encoding rules for the data types, and conventions for defining requests and responses. SOAP is vendor neutral, and can support any language, programming model, and platform. Implementations of SOAP exist in Java, Perl, C/C++, and C#.
- Web Services Definition Language (WSDL) is the XML based description of a web service as a group of ports which are in turn defined by associating a network address with a reusable binding. Each binding defines a group of operations (port type) that is associated with a protocol. Each operation in turn is defined in terms of messages and types. WSDL facilitates the abstract definition of web services to be separated from their concrete implementations.
- The Universal Description, Discovery and Integration (UDDI) specifications define a way to publish and discover information about Web services. The core component of the UDDI is the UDDI business registration, an XML file used to describe a business entity and its Web services, and which are used to provide white pages (service provider), yellow pages business categories), and green pages (technical binding information.)



So, this was great – I was able to relapse right back to working with smaller-grained constructs (services), achieve great flexibility, and I had the added bonus of working in a standards-based environment! It was just too good to be true. I took to services, with the same excitement and fervor that I did with objects.

Very soon, I realized I had relapsed right back to abusing services. Here is what I was doing in my design and implementation:

- For every interface and API in my system, I implemented a web service. I replaced every (remote) call or point-to-point message with a SOAP message.
- I created these services for every application in my system, including for the non-business logic functionality that I had developed to support the application.
- I created UDDI based service registries to support the services that I was creating for each application, and used that to find the services that I needed.
- I created lots of fine-grained services to handle the interactions that I needed between the presentation and business logic functionality. So, for example, I would have individual services to get each of the attributes needed to populate information about a cutomer.
- I quickly found that I could convince customers who had spent all their time on putting in complex and expensive middleware to facilitate integration to change over to using point-to-point integration using services. Basically, I would wrapper the two applications that needed to be integrated as web services, and then invoke them using SOAP messaging. The fact that we were using standards convinced them that it would be cheaper and easier to maintain.
- I relied on web services to reach any part of the system that I wanted, and basically used that as the unifying structure across all applications. I argued that this made the systems simpler and therefore easier to change, manage and maintain.

Of course, I soon realized that I was back to my old pattern abusing days – taking advantage of my prowess with this new services construct and the hype surrounding it to brow beat any customer into agreeing that I was leaving them with a much more flexible, changeable and cheaper to maintain system. Of course, I used the metric of number of services as the key indicator of these “qualities”, along with the added “open standards” that were now an integral part of the system. Like before, the reality was far from what I was selling. The proliferation of services and registries, the unstructured nature of the system, the number of fine-grained services, and the point-to-point integration quickly led to systems that performed poorly, were difficult to maintain, and were expensive to make changes to.

4 RISING BACK FROM THE ABYSS

I quickly realized that I had relapsed so far into the dark side again that it was time for me to go back to my rehabilitation "pattern." Using the same techniques and approaches that I did to get out of my pattern-abusing nightmare, I was able to fight back from my services abyss. I got involved with Service Oriented Architectures and use those first to organize and structure the services that I would build and use and to define how these services could interact with another through the Enterprise Service Bus. I also determined how to handle qualities of service effectively through the layers of services. I used appropriate methods to help me identify services that are appropriate to the domain that I was involved with, starting first with modeling the business process, and then establishing the services needed to implement the process. I used "smell" tests to determine the appropriate validity and granularity of services. To provide the services, I used components to first pull together relevant functionality and then to make these available as services. I composed components and services to provide larger grained services.

Learn from my story, and heed my words – architecture is the salvation for your technical soul.

About the author

Mahesh Dodani is a software architect at IBM. His primary interests are in enabling communities of practitioners to design and build complex on demand business solutions. He can be reached at dodani@us.ibm.com.