Design Patterns for Successful Service Oriented Architecture Implementation

G. M. Tere[1] and B. T. Jadhav[2]

Abstract - The successful implementation of Service Oriented Architecture (SOA) relies on a careful and holistic approach to business planning. One of the most important tools in the evaluation, purchase, and ongoing use of SOA is the best practices that vendors, consultants, and customers have developed and used. The promise of business agility, improved customer service, and competitive advantage with SOA is real. What varies most is the time, cost, and ease of SOA implementation. By learning from the experiences of those organizations that have been through the process and looking at the standard best practices of large-scale technology implementations, success can come at earlier stage. The Patterns for e-business are a group of proven, reusable assets that can be used to increase the speed of developing and deploying Web applications. This paper focuses how the Self-Service and Extended Enterprise business patterns, and the Application Integration pattern, can be used to start implementing solutions using the service-oriented architecture approach. Although the model of Web service interoperability is straightforward, it introduces new development practices and methodologies that can be difficult to learn. However, it can be successfully implemented if we recognize certain patterns to design issues.

Index Terms - Adapter, Controller, Design Patterns, Façade, Proxy, SOA.

1. INTRODUCTION

The role of architect is to evaluate business problems and build solutions to solve them. Architect begins by gathering input of the problem, outline of the desired solution, and any special considerations or requirements that need to be factored into that solution[1,2]. The architect then takes this input and designs the solution. This solution can include one or more computer applications that address the business problems by supplying the necessary business functions.

In the real world, web apps are complicated. A popular web site gets thousands of hits per day. To handle this kind of volume, most big web sites create complex hardware architectures in which the software and data is distributed across many machines. A common architecture is configuring the hardware in layers or tiers of functionality. Adding more computers to a tier is known as horizontal scaling and is considered one of the best ways to increase throughput. Most of the software for a big web application lives in either the web-tier or the business-tier. The web-tier frequently contains HTML pages, JSPs, servlets, controllers, model components, images and so on[13,23]. The business-tier contains EJBs, legacy applications, lookup registers, database drivers and databases. Many developers use J2EE containers to solve same problems. They found recurring themes in the nature of the problems they were dealing with, and they come up with the reusable solutions to these problems[4]. These design patterns have been used, tested and refined by other developers. A software design pattern is a repeatable solution for a commonly-occurring software problem.

[1]Department of Computer Science, Shivaji University, Kolhapur 416 004, India
[2]Department of Computer Science Y .C. Institute of Science, Shivaji University Satara - 415 001, India
E-mail: [1]girish.tere@gmail.com and
[2]btj21875@indiatimes.com

2. OVERVIEW OF SOA

SOAs provide modular services that can be easily integrated throughout an enterprise. They are flexible and adaptable to the current information technology (IT) infrastructure and investments. SOA implementations continue their emergence in business as a mechanism for integrating organizational operations in new and different ways and for promoting reuse while leveraging the existing value of legacy systems. In any business, the bottom line is the essential test of any technology. SOA can provide a significant return on investment (ROI) by integrating legacy and mixed technologies and maximizing the value of existing investments while minimizing risk. Promoting reuse through SOA also helps reduce overall development costs. If services and their data are generic enough, they can be accessed through a variety of interfaces. Decoupling services from their presentation reduces expenses and decreases the overall development time. Further, SOA makes IT consider the dynamic operations of an organization, not just a set of static requirements, thereby exposing information and data sharing across the organization and focusing development on the best ways to improve overall operations[3,7,9].

Although SOA brings significant business benefits, there are challenges to their implementation. As SOA services are typically coarse-grained and loosely coupled, their operations exhibit more latency than more tightly coupled implementations. This can be a challenge when implementing with real time requirements. SOA is designed to bring together legacy systems in heterogeneous IT environments. Standardization of naming, definitions, and identification can present implementation challenges. However, these challenges can be resolved by the implementation of identity and naming services. Finally, SOA is designed to cut through an organization horizontally and vertically, which presents many cultural, cooperation, ownership, and budget issues. Strong leadership must be in place, and executive support must be clear and evident in order for any SOA implementation to be a success[21].

Best practices suggest that there is an overall commitment to increase organizational efficiency. These practices must be

considered from the specific context of your organization. Although the notion of best practices is constantly evolving, it's clear that the following areas are critical:

a) Vision and leadership
b) Strategy and roadmap
c) Policies and security
d) Governance and acquisition
e) Operations and implementation

A key benefit of SOA is reuse of services. It's often tempting to build something from scratch instead of reusing what's already available. This can often happen for two reasons. First, developers may not be aware that a similar service already exists. Therefore, it's important to maintain a directory of available services that is readily accessible and uses the common vocabulary adopted across the organization.

Second, when designing and implementing services, their use outside traditional boundaries must be considered. Creating coarse□grained, modular services helps to promote their reuse in the organization. Organizations should not cringe from updating services that have already been deployed when there are additional needs demanding extension of functionality. For example, eventually, all the requirements should be factored into a single 'get Customer' service rather than having multiple services that get different subsets of customer information.

3. BRIEF OVERVIEW OF DESIGN PATTERNS

Patterns are defined as "an idea that has been useful in one practical context and will probably be useful in others"[19]. Patterns are good constructs for designing Web services. Design patterns are reusable solutions to common software design problems. Design patterns speed up the development process through the implementation of tried and tested solutions. They can play an important role in SOA implementation, especially in the standardization of service design. Since their introduction in the late 1980s, numerous patterns have been recognized and documented. Many SOA implementations use Web services[16]. It is important for architects of SOA implementations to have an understanding of the four primary design patterns for Web services:

1. Adapter: Promotes the reuse of existing technologies through wrappers, extending your existing investments
2. Façade: Used to reduce the coupling between the client and the server components—an essential technique for creating the appropriate level of granularity
3. Proxy: Provides an object surrogate, used to simplify the interaction between Web services components
4. Controller: A key component of the Model-View-Controller (MVC) architecture, used as an intermediary between the user interface (UI) and the data[5,7].

[A] Adapter

As previously discussed, promoting the reuse of existing technologies is essential for a successful—and profitable—SOA implementation. The Adapter design pattern, shown in Figure 1, allows compatible classes to work together by converting the interface of an existing class into an interface that clients expect. JCA is Java Connector Adapter[8,9].

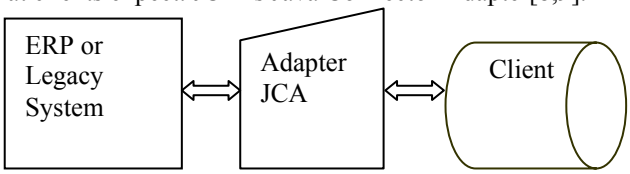

Figure 1: The Adapter design pattern

Organizations will look to reuse existing technologies in their SOA implementations; this is where the Adapter pattern is implemented. Typically, existing technologies provide interfaces that are incompatible with Web services. The Adapter pattern provides a bridge to the existing technology. You don't have to start from scratch when designing Web services: The Adapter pattern can leverage your existing investment and quickly get you started on the road to service implementation. However, it's important to realize that not every application may be a useful service. It's important to be judicious in your design.

Adapter or Wrapper: Used to expose internal application functionality with a different interface. In computer programming, the adapter design pattern (often referred to as the wrapper pattern or simply a wrapper) translates one interface for a class into a compatible interface. An adapter allows classes to work together that normally could not because of incompatible interfaces, by providing its interface to clients while using the original interface. The adapter translates calls to its interface into calls to the original interface, and the amount of code necessary to do this is typically small. The adapter is also responsible for transforming data into appropriate forms. For instance, if multiple boolean values are stored as a single integer but your consumer requires a 'true'/'false', the adapter would be responsible for extracting the appropriate values from the integer value. The Adapter pattern is used to convert the programming interface of one class into that of another. We use adapters whenever we want unrelated classes to work together in a single program.

[B] Façade

Providing the appropriate level of granularity is essential to service design. Services that are too fine grained can increase the overall network traffic as many service requests are made to perform an operation. More coarse-grained services can increase overall latency, but they help expose services that expose a business function. The façade pattern is a software engineering design pattern commonly used with Object-oriented programming. (The name is by analogy to an architectural facade.) Façade: Used to encapsulate complexity and provide coarse-grained services[26].

A façade is an object that provides a simplified interface to a larger body of code, such as a class library. A facade can make a software library easier to use and understand, since the facade

has convenient methods for common tasks; make code that uses the library more readable, for the same reason; reduce dependencies of outside code on the inner workings of a library, since most code uses the facade, thus allowing more flexibility in developing the system; wrap a poorly-designed collection of APIs with a single well-designed API (as per task needs).

An Adapter is used when the wrapper must respect a particular interface and must support a polymorphic behavior. On the other hand, a façade is used when one wants an easier or simpler interface to work with.

Façade defines a higher-level interface that makes the subsystem easier to use. The Façade pattern, shown in Figure 2, is often used to expose coarse-grained services. Instead of exposing the direct, one-to-one functionality of an existing software component or business function, the Façade pattern promotes encapsulation of these lower-level services to provide a single higher-level function

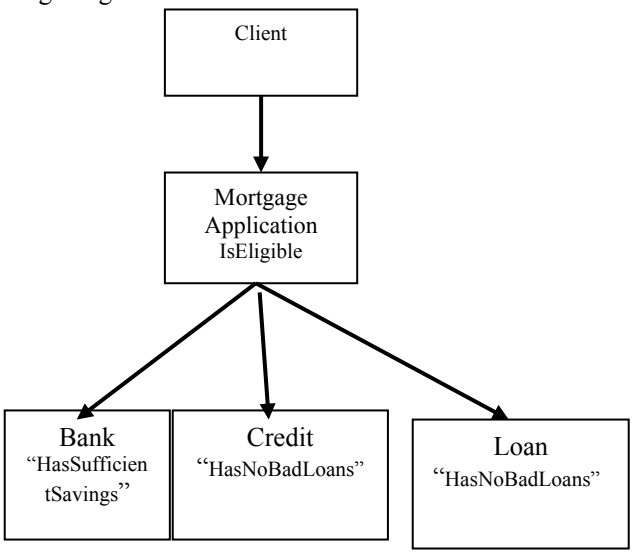

Figure 2: The Façade design pattern

The Façade pattern promotes consistent interfaces, abstracting clients from the implementation details of a service. Further, the pattern facilitates control and management of a service, providing a single entry point that simplifies elements such as security and transaction management. The Façade pattern is a familiar approach to building coarse-grained services. In J2EE, the Façade was represented by a session bean, while the fine-grained components were typically entity beans. For Web services, the same approach can be leveraged. The idea is to take existing components that are already exposed and encapsulate some of the complexity into high-level, coarse-grained services that meet the specific needs of the client. In using this approach, you can enhance overall performance of the Web services interactions and centralize infrastructure services such as security and transactions. Figure 3 shows the relationships between an application's presentation and business tiers using the Façade pattern.

The Façade pattern here is used between presentation and business tiers as a method of encapsulating application logic to address the specific needs of particular clients[3,16].

[C] Proxy

The Proxy design pattern, shown in Figure 4, provides a surrogate or placeholder for another object. It can be used to simplify the interactions among services. The proxy can serve as a

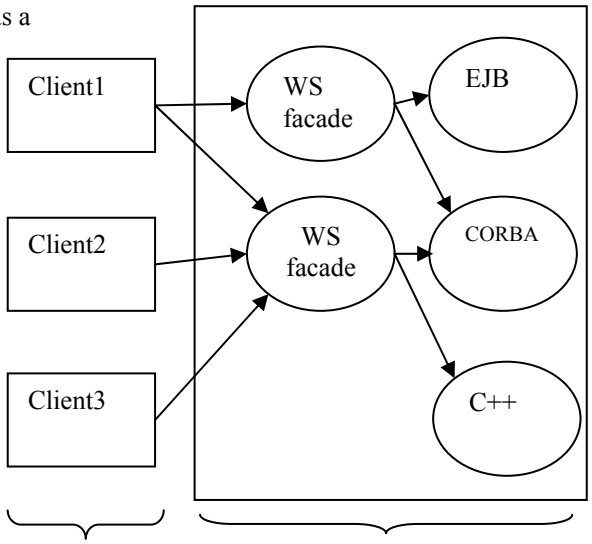

Figure 3: The Façade Pattern.

standardized interface for a collection of legacy back end services. In other words, instead of providing a service for each individual back end service, you can use the proxy to consolidate the messages into a single service, and then dispatch the request to the appropriate back end service, which simplifies the interaction with a collection of services. In computer programming, the proxy pattern is a software design pattern. A proxy, in its most general form, is a class functioning as an interface to something else. The proxy could interface to anything: a network connection, a large object in memory, a file, or some other resource that is expensive or impossible to duplicate. A well-known example of the proxy pattern is a reference counting pointer object.

In situations where multiple copies of a complex object must exist the proxy pattern can be adapted to incorporate the flyweight pattern in order to reduce the application's memory usage. Typically one instance of the complex object is created, and multiple proxy objects are created, all of which contain a reference to the single original complex object. Any operations performed on the proxies are forwarded to the original object. Once all instances of the proxy are out of scope, the complex object's memory may be deallocated.

A proxy is a stand-in for something or someone else. As an actor, you might hire a proxy to attend an autograph signing session. Your proxy is providing a layer between you and your fans, but can forward relevant messages as necessary. You want your proxy to behave as much like you do as possible, so

that the fans believe they are interacting with you directly. Proxy is used as a surrogate for another object or service.

The Wrapper pattern leverages the popular Adapter pattern. The basic idea is to convert a component's interface into another interface that the client expects. This would typically be used to provide some compatibility with the client. The adapter pattern can be used to expose existing technologies as Web services. For example, if you are running on a J2EE platform and have a need to interact with a C++ component, you may wrap the C++ component with JNI code, and then expose that Java interface as a Web service using the available Web services tools.

Proxies in software are similar to proxies in real life. You might create a distributed object proxy. Such a proxy is designed to make its clients think that they are directly interacting with the object, when in reality the object lives in a process on a remote machine. The proxy manages the intricacies of communicating with the distributed object while ensuring that the client remains blissfully ignorant of these details.

The Proxy design pattern can also be used for testing, especially when you are communicating with a third-party object that you do not control. The proxy would implement the same interface as the third party service and can stand in its place during the testing process. Again, this simplifies the testing process.

In the Proxy pattern one object can be used as a surrogate for another, often to offload processing from one component to another. This pattern has been frequently used to hide complexity of the SOAP messaging constructs. It can also be used in the development of mock objects, which have been around for quite some time.

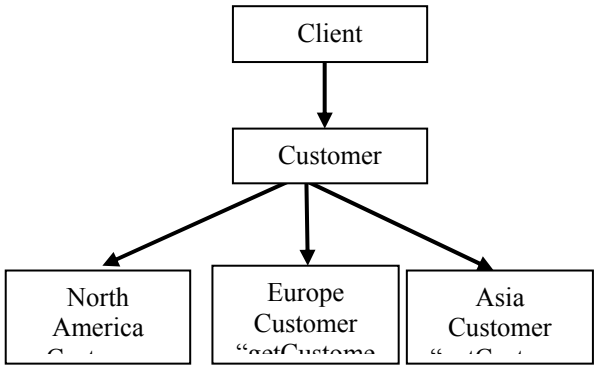

Figure 4: The Proxy design pattern

These are just a few of the design patterns that can readily be applied to Web service development. As you become more comfortable with Web service paradigms, you will find these patterns can guide you just as they do with other types of object-oriented design.

[D] Controller

The Controller design pattern, shown in Figure 5, is probably best known from the MVC application architecture. In the MVC architecture, the model contains the data that the application requires; the view manages the user UIs; and the controller provides the logic and serves as the interface between the model and the view. The Controller design pattern is used to separate the presentation and data layers[5].

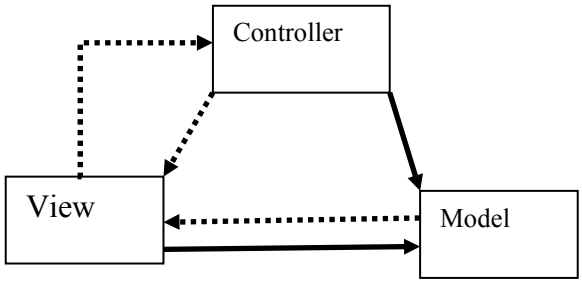

Figure 5: The Controller design pattern

The Controller design pattern can be used in SOA architectures to leverage existing application MVC design architectures and encapsulate the business logic of the service.

4. CONCLUSIONS

SOA best practices are constantly evolving. However, efforts must be made in each of the areas discussed: vision and leadership, strategy and roadmap, policies and security, governance and acquisition, and operations and implementation. Having a skilled professional who has a good understanding of SOA and can communicate that vision to all the stakeholders is essential to a successful implementation.

Look for the easy and achievable goals as you begin your SOA implementation. Establish success with a project; learn from your mistakes as well as from your success. An incremental and agile approach will be essential.

Whether design patterns are a familiar tool or a new concept, an understanding of the four key design patterns—Adapter, Façade, Proxy, and Controller—will be essential to SOA implementation. Begin with indentifying the services you need to implement, and then look to see how they fit into one of these patterns.

REFERENCES

[1]. Booch, Gary, SOA Best Practices, Software architecture, software engineering, and Renaissance Jazz blog www-03.ibm.com/developerworks/blogs/page/gradybooch.

[2]. Craig Larman, Applying UML and Patterns - An Introduction to Object-Oriented Analysis and Design, 2nd Ed, Prentice Hall, 2001.

[3]. Dirk Krafzig, Karl Banke, Dirk Slama, Enterprise SOA: Service-Oriented Architecture Best Practices, Prentice Hall PTR, Nov 2004.

[4]. Deepak Alur, John Crupi, Dan Malks, Core J2EE Patterns: Best Practices and Design Strategies, 2nd Ed., Prentice Hall / Sun Microsystems Press, 2003.

[5]. Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates, Head First Design Patterns, O'reilly, 2008.
[6]. Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1995.
[7]. Erl, Thomas, SOA: Principles of Service Design, Upper Saddle River, Prentice Hall, 2007.
[8]. Evdemon, John, ''Principles of service design: Service patterns and antipatterns.'' MSDN, August 2005. http://msdn2.microsoft.com/en-us/library/ms954638.aspx.
[9]. Fitts, Sean. ''When exceptions are the rule: Achieving reliable and traceable service oriented architectures.'' SOA/WebServices Journal, September 2005. http://webservices.sys-con.com/read/121945.htm.
[10]. Herr, Michael and Uwe Bath, Paper: The business-oriented background of Service Backbone. http://www.servicebackbone.org/, January 2004.
[11]. Herr, Michael and Ursula Sannemann, Paper: The Architecture of Service Backbone. http://www.servicebackbone.org/. October 2003.
[12]. Hohpe, Gregor, and Bobby Woolf. Enterprise Integration Patterns:Designing, Building, and Deploying Messaging Solutions. Boston: Addison-Wesley, 2004.
[13]. IBM Patterns for e-business http://www.ibm.com/developerWorks/patterns/.
[14]. Ivar Jacobson, Object-oriented Software Engineering – Approach, Addison Wesley, 1992.
[15]. Jonathan Adams, Srinivas Koushik, Guru Vasudeva, George Galambos, Patterns for e-business: A Strategy for Reuse, IBM Press, 2001.
[16]. Kanthi Hanumanth, and Alasdair Nottingham. ''Patterns: Implementing an SOA using an Enterprise Service Bus in WebSphere Application Server V6.'' IBM Redbooks, 2005.
[17]. Keith Levi, Ali Arsanjani, A goal-driven approach to enterprise component identification and specification, Communications of the ACM Volume 45 Issue 10, 2002.
[18]. Krafzig, Dirk, Karl Ganke, and Dirk Slama. Enterprise SOA: Service Oriented Best Practices, Prentice Hall, 2005.
[19]. Martin Fowler, Analysis Patterns, Addison-Wesley, 2004.
[20]. Mark Endrei, Jenny Ang, Ali Arsanjani, Sook Chua, Philippe Comte, Pål Krogdahl, Min Luo, Tony Newling, Patterns: Service-Oriented Architecture and Web Services, IBM, April 2004.
[21]. Mike Rosen, Boris Lublinsky, Kevin T. Smith, Marc J. Balcer, Applied SOA: Service-Oriented Architecture and Design Strategies, Wiley Publishing, Inc., 2008.
[22]. Olaf Zimmermann, Mark R Tomlinson, Stefan Peuser, Perspectives on Web Services; Applying SOAP, WSDL and UDDI to Real-World Projects, Springer, 2003.
[23]. Oracle's SOA Resource Center at http://www.oracle.com/technologies/soa/center.html.
[24]. Paul C. Brown, Implementing SOA: Total Architecture in Practice, Addison Wesley Professional, April 2008.
[25]. Towards a Pattern Language for Service-Oriented Architecture and Integration, Part 2: Service Composition.'' IBM developerWorks, December 2005. www128.ibm.com/developerworks/webservices/library/ws-soa-soi2.
[26]. Web Service Façade for Legacy Applications, Microsoft patterns & practices Developer Center, June 2003. http://msdn.microsoft.com/en-us/library/ms979218.aspx.

Continued from page no. 244

[4]. Peter Guller, "Integration of Transport and Land- use planning in Japan: Relevant finding from Europe," Workshop on Implementing sustainable Urban Travel Policies in Japan and other Asia-Pacific Countries, Tokyo, 2-3, March 2005. http://www.internationaltransportforum.org/europe/ecmt/urban/Tokyo05/Gueller.pdf
[5]. Mukti Advani & Geetam Tiwari, "Does high capacity means high demand?," Conference Proceeding- Future Urban Transport-2006, held at Gotenburg, Swedan. http://web.iitd.ac.in/~tripp/publications/paper/planning/mukti_FUT06.pdf
[6]. Data from Economic Survey of Delhi, Delhi Planning Dept., Delhi.
[7]. Results of C.R.R.I Study 2002.
[8]. S. Rajasekaran – *Neural Networks, Fuzzy Logic, and Genetic Algorithms*; Prentice Hall of India, 2003.

Continued From page no. 216

Tools/Attributes	Glossary & Ontology	Checklist	Templates	Use Case Modeling	Prototyping & Audit	TRS	Scalability	External Interface
RequisitePro	X	X	√	√	√	√	X	√
CaseComplete	√	X	√	√	√	√	X	√
Analyst Pro	X	X	X	√	X	√	√	√
Optimal Trace	X	X	√	√	√	√	X	√
DOORS	X	X	√	√	√	√	√	√
GMARC	X	X	√	X	√	√	X	√
Objectiver	√	√	√	X	√	√	X	√
RDT	√	X	√	X	√	√	√	√
RDD-100	√	X	√	X	X	√	X	√
RTM	X	X	√	X	X	√	X	√
Reqtify	X	X	X	√	√	√	X	√
TcSE	X	**X**	X	√	X	√	X	√
Code Assure	X	X	X	X	X	√	X	√
IRqA	X	√	X	√	X	√	X	√

Table 1: Software Functional Requirements

Tools/Attributes	Fair Exchange	Non-repudiation	Rbac	Secrecy & Integrity	Authenticity	Secure Information Flow	Guarded Access	Freshness
RequisitePro	√	X	X	X	X	X	X	√
CaseComplete	√	X	X	X	X	X	X	√
Analyst Pro	√	X	√	X	√	X	X	X
Optimal Trace	√	X	X	X	X	X	X	X
DOORS	√	X	√	X	√	X	√	√
GMARC	X	X	X	X	X	X	X	√
Objectiver	X	X	X	X	X	X	X	√
RDT	X	X	X	X	X	X	X	√
RDD-100	√	X	X	X	X	X	X	√
RTM	X	X	√	√	√	X	√	√
Reqtify	√	X	X	X	X	X	X	√
TcSE	X	√	√	√	√	√	√	√
Code Assure	X	√	√	√	√	√	√	√
IRqA	X	X	√	X	X	X	X	X

Table 2: Software Security Requirements

Note: √: Means satisfies the criterion
X: Means does not satisfy the criterion