2006

A pattern language for service-oriented architecture

Haritha Kilaru
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_theses

Recommended Citation

Kilaru, Haritha, "A pattern language for service-oriented architecture" (2006). *Master's Theses*. 2894.
DOI: <https://doi.org/10.31979/etd.nuag-642x>
https://scholarworks.sjsu.edu/etd_theses/2894

This Thesis is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Theses by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

NOTE TO USERS

Page(s) missing in number only; text follows. Page(s) were scanned as received.

174

This reproduction is the best copy available.

UMI[®]

A Pattern Language for Service-Oriented Architecture

A Thesis

Presented to

The Faculty of the Department of Computer Engineering

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Engineering

by

Haritha Kilaru

May 2006

UMI Number: 1436920

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI[®]

UMI Microform 1436920

Copyright 2006 by ProQuest Information and Learning Company.

All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

© 2006

Haritha Kilaru

ALL RIGHTS RESERVED

APPROVED FOR THE DEPARTMENT OF COMPUTER ENGINEERING

Dr. Mohamed E Fayad, Academic Advisor

Dr. Weider Yu

Dr. Prem Malhotra, BlueRoads Corporation

APPROVED FOR THE UNIVERSITY

ABSTRACT

A PATTERN LANGUAGE FOR SERVICE-ORIENTED ARCHITECTURE

by Haritha Kilaru

The “Pattern Language (PL) for Service-oriented Architecture (SOA)” is a family of patterns that includes the core knowledge of SOAs. This PL is based on the Software Stability Model (SSM) for generalizing the main concepts of SOAs independent from any particular application area.

Four stable design patterns, AnyService, AnyType, AnyInformationHiding, and AnyCoupling, as part of the PL, have been developed and documented. Two stable design patterns, AnyService and AnyInformationHiding, have been implemented. The extension points called hooks are used to generate multiple applications. This ensures that the PL can be used in any domain and encompasses all the essential features to get the system working using SOAs.

The thesis also discusses SOA definitions and compares SOA with the existing software architectures, such as product-line architectures and model-based architectures. The thesis also covers the business impact of using SOA with SSM for generating stable business rules.

Acknowledgement

I would like to thank my advisor Dr. M.E Fayad for his excellent guidance all through my work on the thesis. His valuable feedback and insight into the architectural aspects of systems has shaped my understanding of the foundations for this thesis. His dedication to work countless hours with students to impart critical thinking has helped me learn new things all through this period. Thanks to my reviewers, Dr. Weider Yu from the Computer Engineering Department and Dr. Prem Malhotra from BlueRoads Corporation, for providing valuable feedback on my work. I would also like to thank members of Patterns Research Group for their companionship with research for the last two years. Special thanks to my parents, Satyanarayana Kilaru, Swarajyam Kilaru, and to my husband, Ratna Kavuri, for everything they have done for me.

Table of Contents

	<i>Page</i>
CHAPTER 1	1
INTRODUCTION	1
1.1 Overview of Pattern Language for SOA Issues	1
1.1.1 SOA in the Business Context	3
1.2 Problem	4
1.3 Solution	5
1.4 Research Methodology	5
1.5 Conducting the Research Methodology	8
1.6 Thesis Contributions	9
1.7 Thesis Layout.....	10
CHAPTER 2	11
COMPARATIVE STUDIES	11
2.1 Introduction.....	11
2.2 Software Architectures	12
2.2.1 Product-Line Architectures (PLAs)	12
2.2.2 Model-Driven Architectures (MDAs)	13
2.2.3 Pattern-Based Architectures	14
2.2.4 Aspect-Oriented Architectures	16
2.3 Comparative Studies.....	17
2.3.1 Evaluation Criteria	17
2.4 Comparative Analysis.....	20
CHAPTER 3	27
CHARACTERISTICS OF EFFECTIVE SOA	27
3.1 Characteristics of Effective SOA.....	27
3.1.1 Generating Stable Business Rules	28
3.1.2 A Base for Business Communication	28
3.1.3 A Foundation for Business Modeling and Process Design	29
3.1.4 Information Integration	29
3.1.5 Provide Security and Privacy	30
3.1.6 Deals with Continual Business Optimization.....	30
3.1.7 Business Standards.....	31
3.2 Effective SOA Designs Using SSM	31
3.2.1 Integrate Applications Through Services	32
3.2.2 Adapt Application to Change in Technology	32
3.2.3 Capitalize on Legacy System	33
3.2.4 Create New Business Processes Whenever You Desire	34
3.2.5 Generating Stable Business Rules	34
3.3 Business Rule Impact of Service-Oriented Architecture.....	34
3.3.1 Example Use of Business Rules in a Software Stability Model.....	35
3.4 Mapping to Service-Oriented Architecture.....	39
CHAPTER 4	43
GOALS.....	43
4.1 Introduction.....	43
4.2 Primary Goals (EBTs)	44
4.2.1 Goal: Secrecy.....	44
4.2.2 Goal: Fulfillment	48
4.3 Secondary Goals (EBTs)	49
4.3.1 Goal: Improvement.....	49

4.3.2 Goal: Delivery	49
4.3.3 Goal: Service	50
CHAPTER 5	51
CAPABILITIES OF SOA.....	51
5.1 Introduction.....	51
5.2 Business Objects or Capabilities	52
5.2.1 Capability: <i>AnyCoupling</i>	52
5.2.2 Capability: <i>AnyInformationHiding</i>	54
5.2.3 Capability: <i>AnyService</i>	55
5.2.4 Capability: <i>AnyType</i>	56
CHAPTER 6	61
KNOWLEDGE MAP AND STABLE ARCHITECTURE PATTERNS.....	61
6.1 Introduction.....	61
6.2 Knowledge Map Generation.....	62
6.2.1 Layout of KM	62
6.3 KM Properties	63
6.3.1 Partition.....	64
6.3.2 Creating Architectures from the KM.....	66
6.3.3 Directions	67
6.3.4 Rearrangement.....	67
6.3.5 Number of Applications.....	67
6.4 Conclusion.....	68
CHAPTER 7	69
DEVELOPMENT AND IMPLEMENTATION.....	69
7.1 Introduction.....	69
7.2 Implementation Issues.....	72
7.2.1 Type vs, Class in the Type-Oriented Paradigm (TOP)	72
7.2.2 Use of Hooks	76
7.2.3 Unique Segments of Code	81
7.2.4 Aggregation	85
CHAPTER 8	88
DEPLOYMENT AND VERIFICATION.....	88
8.1 Introduction.....	88
8.2 Patterns for Deployment and Verification.....	90
8.2.1 Deployment Patterns.....	91
8.2.2 Verification and Validation Factors.....	93
8.2.3 Quality Factors	95
8.3 Verification and Validation	98
CHAPTER 9	103
FUTURE WORK AND CONCLUSIONS	103
9.1 Introduction.....	103
9.2 Challenges.....	104
9.3 Future Work.....	106
9.4 Conclusions	107
REFERENCES.....	109
APPENDIX A: PATTERN TEMPLATES.....	114
1. AnyInformationHiding: A Stable Design Pattern	114
1.1 Pattern Name: <i>AnyInformationHiding</i> Design Pattern	114
1.2 Known As	114
1.3 Context	115
1.4 Problem.....	116

1.5 Constraints and Challenges	116
1.6 Solution.....	117
1.7 Pattern Structure and Participants	118
1.8 Detailed Model	120
1.9 Description of Class Diagram.....	120
1.10 Consequences	121
1.11 Usage Notes.....	122
1.12 Applicability With Illustrated Examples	122
1.13 Related Patterns	127
1.14 Conclusion	127
2. AnyType: A Stable Design Pattern	128
2.1 Pattern Name: <i>AnyType</i> Design Pattern	128
2.2 Known As	128
2.3 Context	128
2.4 Problem.....	129
2.5 Challenges and Constraints	129
2.6 Solution.....	132
2.7 Pattern Structure and Participants	133
2.8 Detailed Model	134
2.9 Description of Class Diagram.....	134
2.10 Consequences	135
2.11 Usage Notes.....	136
2.12 Applicability With Illustrated Examples	136
2.13 Related Patterns	140
2.14 Conclusion	141
3. AnyService: A Stable Design Pattern.....	142
3.1 Pattern name: <i>AnyService</i> Stable Design Pattern	142
3.2 Known As	142
3.3 Context	142
3.4 Problem.....	143
3.5 Constraints and Challenges	143
3.6 Solution.....	146
3.7 Description of Class Diagram.....	146
3.8 Pattern Structure and Participants	147
3.9 CRC- Cards.....	149
3.10 Detailed Model.....	151
3.11 Consequences	152
3.12 Usage Warning	153
3.13 Applicability with Illustrated Examples	154
3.14 Related Patterns and Measurability.....	159
4. AnyCoupling: A Stable Design Pattern.....	160
4.1 Pattern name: <i>AnyCoupling</i> Stable Design Pattern	160
4.2 Know As	160
4.3 Context	160
4.4 Problem.....	161
4.5 Constraints and Challenges	161
4.6 Solution.....	163
4.7 Description of Class Diagram.....	164
4.8 Pattern Structure and Participants	164
4.9 Detailed Model	166

4.10 Consequences	166
4.11 Draw Backs	167
4.12 Applicability with Illustrated Examples	168
4.13 Related Patterns and Measurability.....	172
APPENDIX B: SAMPLE CODE.....	175
1. AnyInformationHiding Pattern	175
2. AnyService Design Pattern	182
APPENDIX C: SAMPLE APPLICATIONS.....	188
1. Setup Instructions	188
1.1 Install JDK	188
1.2 Install Tomcat	189
1.3 Copy the Sample Applications	189
1.4 Copy the Sample Database Files	189
1.5 Setup ODBC Entry for Database	189
2. Running the Sample Applications.....	190
2.1 Applications for <i>AnyService</i> Pattern	190
3. Sample Screen Shots.....	191
3.1 Screen Shots for <i>AnyService</i> Pattern	191
3.2 Screen Shots for <i>AnyInformationHiding</i> Pattern	198

List of Figures

Page

1.1 High-Level View of Service-Oriented Architecture	2
1.2 Steps in Research Methodology	7
3.1 Business Rules in Core Pattern for Secrecy	36
3.2 Specializing Business Rules for Service	37
3.3 Traditional Model for Navigation System	40
3.4 SOA Model for Navigation System using SSM	41
4.1 Model for Secrecy Analysis Pattern	45
5.1 Model for AnyCoupling Design Pattern	53
5.2 Model for AnyInformationHiding Design Pattern	55
5.3 Model for AnyService Design Pattern	56
5.4 Model for AnyType Design Pattern	57
6.1 A Portion of the Knowledge Map for SOA	63
6.2 Partitioned Knowledge Map	64
6.3 Remote Partition for Knowledge Map	65
6.4 Generating Stable Architecture from KM	66
7.1 Expanded Model (Second Layer) for AnyActor (First Layer) Pattern	71
7.2 Hook for BO AnyService	79
7.3 Hook for Creating a Subclass for AnyActor	80
7.4 Aggregate Relationship in AnyService Design Pattern	86
8.1 KM Partition for Deployment and Verification	90
8.2 Model for AnyService Design Pattern	99
8.3 StockQuote Example	101
8.4 Book Borrowing Example	102
A.1.1 AnyInformationHiding Design Pattern	118
A.1.2 Detailed model for Information Hiding	120
A.1.3 Information Hiding in Bank Account	123
A.1.4 Sequence Diagram for Banking Account	124
A.1.5 Hiding Implementation Details	126
A.1.6 Sequence Diagram for Hiding Implementation Details	126
A.2.1 AnyType Design Pattern	132
A.2.2 Detailed Model for AnyType Design Pattern	134
A.2.3 Classification of BirthDate Example	137
A.2.4 Sequence Diagram for BirthDate Classification	138
A.2.5 Classification of Laser Printer Example	139
A.2.6 Sequence Diagram for Laser Printer Classification	140
A.3.1 AnyService Design Pattern	146
A.3.2 Detailed Model for AnyService Design Pattern	151
A.3.3 Forex Stock Quote Service Example	155
A.3.4 Sequence Diagram for Forex Stock Quote Service example	156
A.3.5 Book Barrowing Service Example	157
A.3.6 Sequence Diagram for Book Barrowing Service Example	158
A.4.1 AnyCoupling Design Pattern	163
A.4.2 Detailed Model for AnyCoupling Design Pattern	166
A.4.3 AnyCoupling for Electronic Circuits	169
A.4.4 Sequence Diagram for Coupling in Electronic Circuits	170
A.4.5 AnyCoupling for StockFeedService	171
A.4.6 Sequence Diagram for StockFeedService	172

C.1 Stock Quote Service	191
C.2 Foreign Exchange Service	192
C.3 Stock Quote Service in Foreign Currency	193
C.4 Database Tables Used for Library Service	194
C.5 Book Status Service	195
C.6 Book Finder Service	196
C.7 Combined Book Status Service	197
C.8 Banking Application as Seen by Bank Clerk	199
C.9 Banking Application as Seen by Account Owner	200
C.10 Weapons Application as Seen by a Major	201
C.11 Weapons Application as Seen by a Colonel	202
C.12 Weapons Application as Seen by a General	203

List of Tables

	<i>Page</i>
2.1 Comparison of Architectural Styles	24
5.1 SOA Capabilities	58
7.1 AnyService Type Relations for Stock Quote and Library Examples.....	74
7.2 AnyInformationHiding Type Relations for Banking and Military Examples	75
7.3 Hook Template Details	76
7.4 Hook for Creating a New Algorithm IO	77
7.5 Aggregation Hook for AnyService	78

List of Listings

Page

7.1 AnyServiceHook	82
7.2 BO AnyService using the AnyServiceHook	83
7.3 The Class Accessing IOs through the BO AnyService	84
7.4 The StockQuoteService IO Accessing AnyAlgorithm IO through the BO	84
7.5 Aggregation Relationship for AnyClass	87

CHAPTER 1

INTRODUCTION

This chapter briefly describes an overview of Service-Oriented Architecture (SOA) and all the related issues introduced in this thesis. This chapter covers the research methodology and how it will be conducted throughout the thesis. This chapter also highlights the contributions of the thesis.

1.1 Overview of Pattern Language for SOA Issues

Service-Oriented Architecture (SOA) was first described in 1996 by Gartner (Schulte and Natis, 1996; Schulte, 1996b). The Service-Oriented Architecture (SOA) is a modular software architecture that enables building complex software by breaking down a bigger problem into simpler problems that are either already being solved by existing components or can be built using simpler designs. SOA accomplishes this by enabling the software components to be published, discovered, and invoked by each other. SOA empowers the programmers to model complex problems in terms of services offered by components to anyone anywhere on the network. The fundamental building block of SOA, as a service, is a well-defined, self-contained function that does not depend on the state or context of other components of the system (Di, 2004).

Figure 1.1 High-Level View of Service-Oriented Architecture

SOA can be better explained with the main entities that participate in SOA and how they interact. As shown in Figure 1.1, the service providers build and define the service in a standard service-description-language (Mahmoud, 2003). Once the provider is ready to serve, the provider will publish the service description, which contains the contract for the service along with the details of how to access the service. The service registry maintains the services available with various Service-Providers for the consumers. The service consumer will look up the service from the registry. Once they find the service they are looking for, they will bind the service with a provider and use the service.

This thesis covers SOA definitions and compares SOA with the existing software architectures, such as product-line architectures (PLAs), model-driven architectures (MBAs), and aspect-oriented architectures. The thesis also provides a pattern language for SOAs. The developed pattern language can be used in any domain and encompasses all the essential features to get the system work using SOA. It ensures that the pattern language is flexible enough to represent solutions using SOA in many domains.

1.1.1 SOA in the Business Context

SOA enables enterprises to architect solutions focused on business requirements unlike the other technology-centric approaches. SOAs start with services, which are groups of software components that carry out business processes, for example, verifying a credit card transaction or processing a purchase order (Datz, 2004a). In other words, a Service-Oriented Architecture is a business-led approach to creating software that encourages the use of technology to model and automate the high-level services that a business delivers to its customers and suppliers (O'Toole, 2004).

The main problem with the current developed systems is their focus on available technology and their complexity. There is a great deal of technology involved in providing those services to the customer. A great many different systems need to be connected together to enable the service. The services created and delivered to the customer were limited by the available technology, thereby limiting the fulfillment of what a service can deliver (Datz, 2004a; O'Toole, 2004).

SOA promises to reverse that trend by starting with the service in the business context, as it is defined by the people who want to provide it and worry about the technology needed to deliver that service (O'Toole, 2004). That is the whole point of calling it: A *Service-Oriented Architecture*.

This is unlike many previous attempts at SOA that are very technology-centric. A lot of literature available about SOA today repeats the same mistake by defining and focusing SOA on technology, rather than the service itself. Designers need to separate

the functional definitions of services and SOA from the technology that enables it. This is one of the biggest gaps in people's understanding of SOA and making it work.

By focusing on technology people end up building systems that are limited by technology in what it can do, eventually delivering a technology-centric system that fails to deliver the promise of the original services. SOA will succeed by constantly focusing on the services designers wanted to provide in terms of the business consumers of services and make them work with other components in the systems without tight dependency on them. This in turn will lead to the fulfillment of the business value of SOA.

1.2 Problem

Though SOA tries to address the many problems in designing scalable and reusable applications, there are many pitfalls in applying it effectively to a problem domain. One of the main problems is that, SOA-based architectures are domain-specific. The SOA architecture in itself does not guide the designers to implement services in a generic and reusable fashion. This results in the system design being very specific to the problem at hand. This kind of design will yield services that are not reusable. Another problem area is with the design methodology and the representation of services in terms of data-flows. Because of this reason, the designs inherit all the disadvantages of the Data-Flow-Diagrams (DFD) that have been used since the early 70's, effectively making the architecture backwards in evolution and time.

SOA-based designs also repeat the common mistake of designing the solution focusing on the technological aspects of it, rather than the business aspects of it. This makes systems lose focus on the most important component of an SOA, the service.

1.3 Solution

This thesis proposes SOA architecture strategies, based on the software stability model (Fayad and Altman, 2001). This addresses the problems mentioned in the above section.

The software stability model focuses on extracting the common occurring themes into core knowledge of the system (Fayad and Altman, 2001). This helps the designers to focus on generic components of themes, which can be translated into services that are generic and reusable. SSM makes it easy to identify services from systems in a generic way. SSM legacy systems can be reused and extended simply by plugging in new application objects into the model. The core knowledge itself will remain stable across all the extensions. This shifts the focus of application designs from the old data-flows into the more important areas of service definitions. Chapter 3 presents a solution to convert the designs into more stable business themes, which is the main value proposition for an SOA.

1.4 Research Methodology

The solution is presented as a pattern language for SOA. A pattern language is a set of patterns and their relationships within a given context. This represents a set of known

recurring problems along with their solutions in that context. The concept of PL is a methodology for finding the patterns, their relationships and architecture and documents them in the form of a “map of patterns” for a given system. This map in turn will help us in finding the correct patterns and architectures for other systems. The methodology followed to develop the pattern language is illustrated (Figure 1.2) and outlined below. The first step in the PL methodology is identifying the goals/missions of the system. These goals must fit into the overall system goals. They must be quantifiable, meaningful and measurable in that problem domain. In the software stability model, these goals are known as Enduring Business Themes (EBT) or Stable Analysis Patterns (SAP) (Hamza, 2002; Hamza and Fayad, 2002a).

The next step is identifying capabilities of the system to fulfill their established goals and purposes. The capabilities in the patterns language are called Stable Design Patterns (SDP) and are formed with Business Objects (BOs).

Once the goals and capabilities of the system are identified, the thesis moves on to the architecture design. The architecture is a combination of goals and capabilities and provides the knowledge map for the given domain. It consists of both analysis and design patterns fulfilling a specific functionality within that domain.

The next step is to identify the application scenarios/implementation. Since the designers have the core knowledge to represent the specific problem, they need to identify application elements that change from application to application. These application elements are called the Industrial Objects (IOs) in stability model.

Designers then move on to the testing phase. This step conducts functional, operational, and interoperability testing. The testing should be done in every phase. The necessary recommendations, based on the outcome of the testing will be made in order to make the software work. The Verification and Validation (V&V) tests are used to see how this system can be used and deployed for actual use.

Figure 1.2 Steps in Research Methodology

1.5 Conducting the Research Methodology

To conduct the research methodology as described above, the following methods have been followed:

- a) Exploration:** Numerous design methodologies like “Procedure-Oriented Architecture,” “Object-Oriented Architecture,” and “Product-line Architecture” along with their advantages and disadvantages have been explored.
- b) Study:** Different approaches for SOA architectures have been studied in different trade publications and websites. Many articles regarding the methodology and patterns, and several papers on stability have been followed. The author has followed through a many of sites on SOA methodology. Some good sites include the BEA Weblogic’s SOA section (Weblogic soa section, 2003) and the Weblogic developer’s journal, Microsoft Architect journal, Web Services journal, and several articles and white papers on the IBM developer works site.
- c) Discussion:** Apart from following a lot of discussion groups on patterns on the Internet, had extensive discussions with Dr.Fayad. The models have been taken from the repositories of Dr.Fayad and discussed with members of the patterns research group to come up with sample applications for the models.
- d) Writing:** The author has participated in discussions and learnt from Research group members who have published papers on this topic. The author has also published a paper in *the 2005 IEEE International Conference on Information Reuse and Integration (IEEE-IRI 2005)*.

1.6 Thesis Contributions

The contributions of this thesis include various aspects of the pattern language for Service-Oriented Architectures. They include:

- 1. Comparative Studies:** The second chapter discusses the definitions of SOA and compares it with various other software architectures such as product-line architectures (PLAs), model-based architectures (MBAs), business models and so on.
- 2. Characteristics of Effective SOA:** In the third chapter, the characteristics of effective SOA are discussed along with guidelines for designing effective SOA using SSM. The mechanisms for mapping traditional architecture to SOA are also discussed.
- 3. Software Analysis Patterns (SAP):** The goals of SOA form the Enduring Business Themes (EBTs) of stability model. These are known as “Stable Analysis Patterns.” In the goals chapter, four stable analysis pattern models have been provided, such as Service, Secrecy, Improvement, and Delivery.
- 4. Software Design Patterns (SDP):** The capabilities of SOA form the Business Objects (BOs) of the stability model. These are known as “Stable design Patterns.” Some of the capabilities of SOA are explained in the capabilities chapter with example applications. Pattern templates have been provided for *AnyService*, *AnyType*, *AnyInformationHiding*, and *AnyCoupling* design patterns in Appendix A.
- 5. Software Architecture Patterns:** The thesis maps the goals and capabilities into a single architecture that represents the overall picture of the pattern language for SOA. These form the Knowledge Map (KM) of SOA. The knowledge map chapter discusses these architecture patterns.

6. Development and Implementation: The thesis develops and implements the *AnyService* design pattern and *AnyInformationHiding* design pattern along with the inner layers for each of the EBTs and BOs. The extensibility of these models has been demonstrated by generating sample applications using the software hook utility. The hook utility will enable generation of other applications using this model.

1.7 Thesis Layout

The rest of the thesis is organized as follows.

Chapter 2 provides comparative studies between the SOA and various other software architectures.

Chapter 3 focuses on describing what makes an effective Service-Oriented Architecture.

Chapters 4 and 5 provide the goals and capabilities of Service-Oriented Architecture.

Chapter 6 provides the description of architecture on demand for Service-Oriented Architecture.

Chapter 7 describes the development and implementation.

Chapter 8 provides the deployment and verification details.

Chapter 9 describes the conclusions and future work.

CHAPTER 2

COMPARATIVE STUDIES

2.1 Introduction

Whenever designers are faced with choosing among multiple choices, they instinctively compare them in terms of some criteria. Comparative study is scientifically done by measuring the attributes and behavior of different options that serve a common purpose and by comparing them in terms of identified criteria by measuring different aspects of them. This comparison would help us to identify the most suitable option among others. The same could be applied to when people have to choose one approach among several, to solve a particular problem. When designers set out to solve a problem, they need to understand the problem domain and the criteria to use for determining the right mechanism to solve the problem.

Software architects have devised several architectural styles for designing systems that solve complex problems. The designers have been using all these techniques for solving different types of problems. The comparative studies of various software architecture styles will give the designers the mechanism to choose one of the styles for solving their problems in a most efficient way.

This chapter briefly explains some of the software architectures. Section 2.2 gives details of the criteria to be used for evaluating the suitability of an architectural style of a given problem domain and compare them in terms of the identified criteria. The chapter mainly focuses on comparing the proposed SOA with the other architectures.

2.2 Software Architectures

The architecture of a software system defines the inherent structure and organization of the structural elements and their interfaces of which the system is composed. The software architecture document is used for efficiently communicating the design between stakeholders/developers/users for efficient reuse and further development. Architecture of large and complex systems can be explained by the behavior of various structural elements as specified in the collaborations among those elements and the composition of these structural and behavioral elements into progressively larger subsystems (Lil et al, 2003; How do you define software architecture, 2001).

2.2.1 Product-Line Architectures (PLAs)

A product-line is defined as a group of products sharing a common, managed set of features that satisfy the needs of a selected market. A PLA captures the essential commonality in the product-line and derives a common architecture with flexibility and adaptability to specific requirements of the product line (Judy, 2000; Product Line Systems Program, 2000). PLA relies heavily on reuse of existing components for enhancing existing products or adding a new product in the product line, as opposed to creating a totally new product. Different products in the product line depend on a common core platform with various options and upgrades differentiating them. Development involves domain analysis, and design and implementation of components required for the architecture.

By definition itself, the PLAs are domain-specific, as they try to extract commonality in related products from the same domain. Some of the disadvantages of PLAs include the need for upfront investment of resources and complexity of initial design. PLAs also limit the amount of diversification of product line, as all products rely on the same common core architecture. For companies with existing systems migrating their models into a PLA, without impacting existing deployments may not be easy to.

2.2.2 Model-Driven Architectures (MDAs)

MDA has been proposed by the Object Management Group (OMG) to address preserving investments in existing software components as the technology in various areas like operating systems, middleware, and programming languages keep changing. OMG aims to do this by proposing industry standards that do not change for a while. For the developers of the system, MDA comes with tools to support development, using MDA (Soley, 2000). The developers initially design the platform independent model (PIM) and model it with UML, which is described in generic terms. The MDA tools then can generate one or more platform-specific model (PSM) from this, using standard OMG mappings. The MDA tools can further generate the platform-specific implementations for these models. This way once the generic UML model is designed, everything else can be taken care of the tools. The tools can also take care of generating bridges that facilitate interoperability between PSMs from different platforms. The MDA methodology supports the full lifecycle of the system like analysis, design, implementation, deployment, maintenance, and integration with other systems. This

process also ensures interoperability and portability of the systems and lowers the initial cost of maintenance and maximizes the ROI. MDA has been successfully demonstrated using proof of concepts for enterprises in different domains and standards in various verticals that are well in progress.

Though MDA has been touted as the solution for building reusable systems, in practice it is not suitable for many applications because of its limitations. The UML notation, on which the MDA is based, supports multiple models like static and behavior models, but it does not support multiple views of the system. Architecture should have multiple views for extensibility and ease of maintenance of the system. It is also not possible to break down the systems built using MDA and reshuffle it and build a new architecture to incorporate new requirements. The MDA standards for various verticals proposed by OMG have a very large core component, making it very difficult to manage them. For these reasons, the MDA does not support all the needs of all customers. Integration for a large-scale application becomes too heavy using the MDA systems and the OMG-based platform bridges.

2.2.3 Pattern-Based Architectures

Design patterns are standard solutions for recurring problems within a given context. Effective software design requires considering issues, which may not become apparent until later in the implementation. Design patterns address this problem by capturing solutions for similar problems occurring in similar contexts and reusing them in building systems. This process of using design patterns can expedite the design process

by providing tested, proven development paradigms. They deal with the important issues between the high-level view of software system architecture and low-level coding. It focuses on a finer-grain view of a system — the details of interfaces and implementations of individual components, sets of components, and how the components compose and interoperate with each other.

Though design patterns have their uses, there are several limitations to using them for architecting systems across multiple domains. Some of them are discussed in the following:

- Though design patterns represent solutions for recurring problems, they mainly address the problems for a single domain. In other words, the patterns are domain-specific. One outcome of this is the lack of pattern reusability across domains. Another drawback is, one needs to be familiar with the domain to understand a pattern.
- The design patterns come from the experience and discovery, with no scientific basis for the solutions and reusability. Each of these experiences have to be gained by going through painful discovery which is very difficult and oftentimes very costly.
- There are multiple design patterns addressing the same design problem but there are no clear guidelines for choosing the right pattern for a given scenario.
- The design patterns are closer to and more useful in the implementation of a system than in architecting it. They do not address the requirements and analysis of the systems, they only implement the architecture.

2.2.4 Aspect-Oriented Architectures

Aspect-oriented architecture (AOA) is a way to decouple facets of application functionality, so that they are easier to design and implement (Hollander, 2005). AOA aims to separate program code related to the main purposes of the application (core concerns) from code related to secondary purposes (crosscutting concerns) (Aspect-oriented programming, 2003). For example, in a telecom application routing calls would be the main purpose or the core concern of the application, whereas logging the time duration of calls for billing purposes may be a secondary or crosscutting concern. In an AOA system, the designer defines and separates various concerns in the module and the developers implement each of the concerns. Then the AOA system will weave the concerns to form a combined system at runtime. Using AOA the business logic related to the main objective of a module can be kept clean from secondary and more generic activities like logging, debugging, call interception, etc. Because it separates the concerns clearly into different modules, it leads to designs that are easy to understand, implement and maintain.

One of the biggest problems with AOA is the complexity of debugging applications built with AOA, although concerns are clearly separated at design and implementation phases during runtime the aspects are woven together. Another issue is, concern-weaving in AOA can become unpredictable if it is not made clear which aspect should dominate (Aspect-oriented programming, 2003). Though AOA aims to maintain the concerns cleanly and independently of each other, maintenance becomes difficult over a period

when code like method signatures needs to be changed forcing some aspects to not work or worse still, work in a wrong way. Another issue with AOA is that it is a very early concept with nonstandard tools implementations making it difficult to use for complex systems.

2.3 Comparative Studies

The following sections explain the evaluation criteria for comparing different types of software architectures. It then uses these criteria to show the software evaluation and comparison between service-oriented architecture (SOA) and other architectures such as product-line architecture (PLA), model-driven architecture (MDA), pattern-based architecture (PBA), and aspect-oriented architecture (AOA).

2.3.1 Evaluation Criteria

Architects can use several criteria for comparing these software architectures based on the structural, computational, quantitative, and qualitative aspects of the system. For simplicity this chapter will cover the qualitative criteria in this section.

The Qualitative criteria used for evaluating software architectures are explained in this section.

- **Scalability:** Scalability is a very important criterion for a business system if it needs to be successfully used while meeting the increasing load from growth in business. Good software architecture should address the scalability requirements of applications. It should also be easy to design systems under different load

conditions with the architecture. Applications designed with good software architecture should scale under different load conditions. The features like loose coupling, location transparency, and reliability in the architectures make them scalable.

SOA is a very scalable architecture compared to other architectures such as PLA, MDA, Aspect-Oriented architectures, and domain-specific architectures. SOA supports the stateless services, which can be used to scale the system, by distributing load across multiple systems, as the clients are not required to talk to the same service for all requests. This along with the location transparency will enable the system to scale by deploying across multiple systems. Close coupling has been a major reason the other architectures are not scalable whereas, the SOA has very loose coupling.

- **Reusability:** Reuse enables the developers to build systems without repeating similar tasks, thereby building the system with a small number of components. This reduces a lot of cost in developing a system and maintaining it through its life-cycle.

When designers compare the reusability of components in SOA with other architectures, they can clearly see that components can be reused more efficiently in SOA. This is because SOA supports the concept of services, which are independent components of the system. By designing the system with services, they can identify the commonly occurring components as services that can be utilized from multiple modules of the system. Most of the older

architectures do not have that desired level of reusability either because of close coupling within the components or because the components are usually stateful.

- **Extensibility:** Change is omnipresent in any real deployment environment in which the system needs to function. Those changes need to be supported with corresponding changes in the underlying system along with service providers, consumers, and the messages they exchange in addition to adding new services and new message types.

In SOA the concept of services enforces loose coupling and the definition of messages has been made flexible to extend by changing/adding new parts. If the messages are not extensible, the service providers and consumers are locked into the only existing features of the system and cannot even support any changes in the deployment environment for existing services. The other architectures have limited extensibility because of the tight coupling or because they are not truly component-based architectures.

- **Adaptability:** Adaptability is very crucial to the success of any system. Without user adaptation and integration in the user environment, even the best system would be of no use to the business. This is because all the best features of the system could be used to the advantage of the business only once the users are comfortable using it. This would include the adaptability from the development, deployment, and administration teams, and most importantly the users.

The adaptability for developers depends on the ease of development, the availability of development tools along with good testing and debugging facilities. Though SOA has been around shorter time than many of the other architectures, it has excellent support in tools and ease of development. The ease with which different components in the system could be developed, deployed, and used in diverse environments has been a strong asset of SOA. The other architectures have limited adaptability as they lock the users into tightly coupled components, which are not easy to extend and customize to different user group requirements.

2.4 Comparative Analysis

This chapter compares SOA with other well-known architectures such as PLA, MDA, AOA, and PBA. Though most of these architectural styles promote reuse and increase productivity, they address different types of problems in different domains.

The PLAs are commonly arrived at by observing the commonality in functionality between different products, whereas the SOAs are mainly based on the principles of loose coupling. The many advantages of loose coupling include interoperability and high availability. Another clear advantage of the SOA is the separation of concerns between the client and server, by hiding the details of the server implementation from the clients. Both the PLAs and SOAs have the disadvantage of being domain-specific. PLAs have other drawbacks like multiple versions of assets, dependencies between assets, tight

coupling, and poor documentation, whereas the SOAs have the drawbacks of unclear security strategy and low-level of support.

The MDAs have long been promoted as the best way towards building business solutions, whereas the SOAs have been recently gaining prominence with loose coupling and good interoperability as the main advantages. In an MDA, the system is properly tested before it is even constructed, while in the SOA the services can be defined independently from the consumers because of loose coupling. The MDAs incorporate tight coupling, whereas the SOAs' main strength is their loose-coupling nature. The SOAs already have good tool support from major vendors whereas the MDAs lack much needed support tools. This makes it difficult to build large-scale systems with MDAs, whereas there is overwhelming support for SOAs from tool vendors and third party components to easily build real-world systems.

SOAs have been gaining momentum recently with the quick adoption of the concepts of Aspect-Oriented Programming (AOP) with its use in emerging frameworks like Spring, JBoss AOP, and C# AOP. But the main disadvantages with AOP are that it is still an emerging technology and therefore there are no proven large-scale projects implemented with it. Whereas SOA has wide support in the technology community, although it is a new technology, the underlying concepts have been in use for a while. The tools support for AOA is very limited with most of them in their early versions. The support for modeling AOA architectures in modeling languages like UML is non-existent. Modeling relationships between different concerns is not natural and leads to very ugly and hard-to-manage UML models for AOA-based systems. But modeling

SOA systems is built in the UML modeling as the concepts of modularity. The specification is very vague about the interaction between different concerns when multiple aspects are used in a system. This increases the complexity of designing and understanding systems with multiple concerns. In contrast, SOA systems are based on multiple services interacting with each other to deliver a business function, so it handles systems with multiple services clearly. Though reuse of concerns is one of the stated advantages of AOAs, in practice it is not very commonly seen as very few real-world examples of reuse exist. SOA on the other hand has numerous examples of service reuse, as service reuse is the basis for SOA. Though AOA and SOA seem like different choices for architectural style, in practice it is quite possible to use the concepts of AOA in some modules while architecting the system with SOA. This way the system will take advantage of salient points in both architectures.

Pattern-based architectures have been in use for a while in the software community. Though there are extensive catalogues of patterns developed by the designers, it still remains up to the intuition of the designers to find the suitability of using a pattern, at best to manually match them from a pattern catalogue or from prior experience of the designer. SOA incorporates the concept of service registry, which is much more than a repository as it provides discovery and description of services (UDDI) to make it easier to find a suitable service. PBAs proclaim the use of reusable patterns, often without any implementation except for the architectural style, which leaves out the implementation, but SOA promotes the use of services, which are readymade components, ready to be used. This leads to greater reuse and reduced implementation efforts for the system. The

tool support for PBA is limited as it is difficult to match a pattern for use in an architectural scenario. SOAs have better support from tool vendors which is getting even better with increased adaptation from the technical community. The existence of multiple patterns for solving the same problem without clear criteria for choosing the most suitable pattern among them, leads to even more confusion for the designers. SOA addresses this problem by introducing parameters in service descriptions from the service registries, making it easier to choose among multiple services. As stated with the AOA, it is still possible to use patterns for solving problems with well-known solutions while designing new services in an SOA give the designers the best of both worlds but the emergence of more services by vendors makes that a rare necessity.

The following table summarizes the comparison for the architectural styles discussed.

Table 2.1 Comparison of Architectural Styles

	Advantages	Side effects	Limitations
SOA	1. Message-based system 2. Loose coupling 3. Interoperability 4. Increased reuse 5. Location transparency 6. High availability 7. Clear separation of concerns between client and server (MSDN, 2004)	1. Domain Specific 2. Unclear security strategy 3. Lack of standards covering phases in system life-cycle. (Hollingsworth and Weide, 1995) 4. Low tool support	1. Lack of architectural patterns for SOA. 2. Cost of mistakes in design is much higher, as it impacts more systems.
PLA	1. Captures commonality between products 2. More productivity 3. Increased reuse among product line modules (Judy, 2000)	1. Domain Specific 2. High degree of erosion over time 3. High maintenance overhead 4. Limit diversification of product lines	1. High upfront investment of resources (Judy, 2000) 2. Complexity of initial design

MDA	<ul style="list-style-type: none"> 1. Supported by OMG standards. 2. Ability to generate platform-specific models from generic models. 3. Model is main system of record, and is easy to manage (Soley, 2000). 	<ul style="list-style-type: none"> 1. Does not offer multiple views of system required for business perspective. 2. Lack of extensibility and ease of maintenance. 3. High cost of testing: implementation gets regenerated with every change 	<ul style="list-style-type: none"> 1. Inability to break down components for reuse (Soley, 2000). 2. Complexity of core components proposed for verticals 3. Low industry adoption rate
PBA	<ul style="list-style-type: none"> 1. Capture solutions for recurring problems. 2. Promise of reusability 3. High adoption rate for patterns in the industry 	<ul style="list-style-type: none"> 1. Pattern solutions are domain-specific. 2. No scientific basis for validity and suitability for a scenario. 3. Needs costly discovery through trials 	<ul style="list-style-type: none"> 1. Patterns are closer to low-level implementation than at a grand level view needed for Architecture. 2. Do not address requirement and analysis of systems

AOA	<p>1. Separation of concerns at various levels.</p> <p>2. Keep the main concerns of system clean from crosscutting concerns.</p> <p>3. Designs are clean and easy to understand</p>	<p>1. Complexity of end application with all concerns is high.</p> <p>2. Difficulty in debugging final applications.</p> <p>3. Unpredictable weaving of concerns.</p>	<p>1. Costly maintenance: complex weave of concerns</p> <p>2. Fragile systems that have low tolerance for change.</p> <p>3. Very early concept in experimental stage.</p>
------------	---	---	---

CHAPTER 3

CHARACTERISTICS OF EFFECTIVE SOA

This chapter discusses issues relating to systems design using Service-Oriented Architectures and how to make effective system architectures that are extensible and easy to maintain. The effective characteristics are briefly introduced along with details and advantages of each of these characteristics. This chapter elaborates one of the most important characteristics of effective SOA: Generating stable business rules, in the following sections. The chapter also discusses how these characteristics can be more effective using SSM methodology. SSM methodology can address some of the difficulties in designing systems using SOA, particularly in identifying candidates for services in a system. The chapter further outlines the advantages and how SSM can help in designing an effective SOA.

This chapter further demonstrates how a legacy system can be converted into SOA-based design by illustrating it with an example. The discussions also compare and contrast the two designs and highlight the advantages of a system based on SOA.

3.1 Characteristics of Effective SOA

This section goes briefly over the characteristics of effective designs using Service-Oriented Architectures. These characteristics highlight how SOA can provide business value to a system. It gives the characteristics along with the description of how they impact the design and the benefits of each of them.

3.1.1 Generating Stable Business Rules

Business rules capture the core knowledge of how a business runs in terms of business processes. An effective SOA should support incorporating these common business rules in multiple systems across the organization (Johnston, 2001). The SSM models make this possible by capturing stable business rules in the core-models. As the core-models remain stable for multiple domains, these stable business rules will apply in all these domains. This also makes it easier to implement business rules that span across multiple systems by using the flexible integration capabilities of SOA.

3.1.2 A Base for Business Communication

Business communication mechanisms are essential for consolidating expertise across various departments to come up with a coherent business strategy and the effective execution of the strategy. An effective SOA should facilitate easy integration of systems from multiple departments of an organization along with partners and suppliers. Models designed with SSM make easy integration possible with the flexibility and extensibility inherent to the SSM methodology. The stable interfaces generated by the SSM enable easy communication of service interfaces avoiding the intricacies of complex object interactions. An effective SOA should be easy to understand and build applications to address all business communication requirements of an organization.

3.1.3 A Foundation for Business Modeling and Process Design

The business strategy modeling has been useful in documenting the aspects of strategy such as tracing cause-and-effect relationships between strategic decisions and their outcomes (Adler, 2004). A successful SOA should model the business strategy with sufficient detail and structure to allow quantitative analysis. At the same time it should be simple enough to support qualitative and visual analysis to bridge the gap between the science of business strategy and executing it successfully. SSM enables this by facilitating easy modeling of business themes and verifying their applicability in different business scenarios across multiple functional areas in the organization.

3.1.4 Information Integration

Business information drives any successful business decision. Information integration aims to provide a set of services to deliver the right information to decision makers at the right time, with appropriate context, so that decisions can be made more quickly and with more insight. An effective SOA should facilitate the easy integration of the information from multiple sources along with automated reasoning systems and human analysis to present coherent and flexible analytic services that are easily accessible across the entire organization. It should also be scalable with the increasing amount of both structured and unstructured information. The salient features of SSM like the extensibility for scalability and ease of adaptation for easy integrations, makes it easy to achieve this in the system.

3.1.5 Provide Security and Privacy

Security and privacy are very important in gaining the trust of the customers and business partners and maintaining competitive advantage over others. An effective SOA should bring the understanding of security required across multiple systems at an enterprise level, enabling enterprises to manage its trust relationship with its key stakeholders. It should support adding any security-related features across systems with minimal changes to the individual components. An effective SOA should support the emerging security standards in the services technology. With the extension capability of SSM-based models and the use of hooks it becomes easier to add in any new security enhancements to the system. With the use of security-based core models like *AnyInformationHiding* and secrecy, the same security strategy can be implemented across different systems in the organization.

3.1.6 Deals with Continual Business Optimization

Continual business optimization (CBO) applies rigorous mathematical models and algorithms to business data to support decisions in areas such as forecasting, supply planning, scheduling, and pricing. A successful SOA should facilitate effective collaboration with common language, standards, and understanding among numerous CBO participants. It should be flexible enough to facilitate changing the business processes without disrupting the existing deployments and with minimum changes to the individual components. Optimizing the system often involves changes to multiple components in the system, which makes it difficult to manage the change. The SSM

models adapt to change easily by extending the business objects with IOs representing the changed behavior. The SSM patterns make it possible to track effects of changes and ensure enhancements throughout the system.

3.1.7 Business Standards

Collaboration between partners and suppliers is very crucial to responding quickly to market conditions and executing the business strategy effectively. An effective SOA should support easy integration of systems across the partners and suppliers. Though the emergence of services and SOA has helped in easily reaching this goal, a lot of work remains to be done to integrate systems in specific domains. The key to that level of easy integration had been the adoption of standards in various industries like electronics (RosettaNet), insurance (Accord), banking (SWIFT), and healthcare (HL7). An effective SOA can make it easier to integrate with any of these standards by facilitating best practices in business standards adoption, and creation of key performance indicators. Using SSM methodology makes it easier to adapt existing components to newer business standards as they emerge over time.

3.2 Effective SOA Designs Using SSM

The characteristics of SOA can be brought out more effectively by using the stability model. The following points outline how architects can make an effective SOA design using SSM (Fayad and Altman, 2001; Hamza, 2002).

3.2.1 Integrate Applications Through Services

The point of integration in SOA is to combine all the services into a service base and provide the services based on consumer requests (Kidd, 2004). Though this may sound easy enough, the difficulty lies in identifying portions of the system that are candidates for a service. SSM performs the task of identifying the common recurring themes among systems as possible candidates for services.

For example, let us take two systems and designers who need to integrate those two systems. Based on SOA rules, designers place the services of those two systems into a service base and retrieve the services. Before they can do this, they need to identify the system components that can be designated as services. To do this more effectively, designers can use software stability methodology. According to this methodology, designers take the core knowledge of these two systems and they derive services and deposit them into the service base.

3.2.2 Adapt Application to Change in Technology

Whenever designers need to adapt an application for any new technology, they need to change the underlying model based on SOA. By using software stability model, they can adapt new components of the application without changing the core model of the SOA system. That way they can provide services that are stable over change, easy to maintain and use the system more effectively.

For example, let us take a stable service interface like AnyMedia. It may be initially implemented as a phone in the backend. If someone needs to change the media to email, they need to change the underlying model to adapt the new media into the model. But, by using SSM (Hamza, 2002; Hamza and Fayad, 2002a), they can adapt email as AnyMedia without changing the base model. This way they can adapt not only email, as many media as they want such as air, TV, etc.

3.2.3 Capitalize on Legacy System

Why should architects always throw out a legacy system whenever there is a new system? Rigid system designs result in throwing away investments in existing infrastructure, resulting in expensive systems. Using SSM methodology for building a system will enable designers to reuse existing components, so that they do not need to throw existing components. They can always integrate the business with the legacy system or they can always extend the business system to meet the new requirements.

This brings out one of the strong advantages of SOA. "That's the sweet spot for SOA—leveraging your existing portfolio," says Tim Bass, president of Silk Road, an IT consultancy (Datz, 2004b). One does not need to rip out and replace those systems with brand-new ones. By identifying the capabilities of existing systems and leveraging them, you maximize the value of your IT investments while minimizing your risk (Datz, 2004a).

3.2.4 Create New Business Processes Whenever You Desire

Adding new services should not result in expensive redesigns of the systems, or ripping apart existing components that are already working. Using SSM, one can add new services to the system without affecting the existing infrastructure. Whenever they need to create new services to the business, they will just have more participants to add into the service base.

3.2.5 Generating Stable Business Rules

Using the software stability model (SSM) it should be possible to capture the common recurring themes across the enterprise to capture the knowledge as stable business rules. This way the business objects are going to be the same for that system and it is not changeable externally, so you can add as many processes as you want to that business object.

3.3 Business Rule Impact of Service-Oriented Architecture

A business rule is a statement that defines or constrains some aspect of the business. It is intended to assert business structure, or to control or influence the behavior of the business. In business terms, a business rule is a guideline that shows there is an obligation concerning conduct, action, practice, or procedure within a particular activity or sphere (Defining business rules, 2003). A typical example of a business rule for a video rental store is, “a maximum of three videos can be rented at a time by any customer.”

3.3.1 Example Use of Business Rules in a Software Stability Model

The software stability model (SSM) can improve the capture of the business rules for a commonly recurring theme for different domains by providing the stable business rules. In the software stability model, the core model captures core knowledge of the central theme in any problem domain. The effectiveness of business rules can be enhanced by capturing them as part of the core model. The core model can be adapted to fit any specific application domain by extending it with application-specific industrial objects. By encoding the business rules into the core model, the extended models for each domain will automatically inherit these business rules. This way the business rules that are common to the underlying problem can be enforced uniformly across all domains.

The use of business rules in a core model can be illustrated with a sample model from the knowledge map presented in Chapter 6. If they take the secrecy analysis pattern from the knowledge map, they can extract the model associated as follows.

Figure 3.1 Business Rules in Core Pattern for Secrecy

In the above model (Figure 3.1), they can see the core model represented by the secrecy pattern along with the important participants of this model, Anything, AnySecret, AnyParty, AnyPrivilege along with others. Looking closely at this model illustrates a broad business rule, “*AnyParty should have the necessary privileges to access the secret about anything.*” This business rule is nicely encoded in the core model by using the participants Anything, AnySecret, AnyParty, and AnyPrivilege. The other components like AnyCondition, AnyMechanism, and AnySecrecyLevel can be participating in another such business rule.

Figure 3.2 Specializing Business Rules for Service

The extensibility of the business rule comes into play when they extend this core model to suit a specific application scenario (Figure 3.2). The core model itself only enforces that the party accessing secret information should have necessary privilege. But, it does not enforce the specific details of secret information and privileges. This leaves it flexible for application-specific extensions to customize the application-specific business rules in the contract. For instance, the application for a payroll system may stipulate, “The person accessing the salary details about a payroll record should have salary-viewing privileges.” This could be enforced with the condition that the accessing person is a manager. The mechanism to verify this could be by checking with the company organizational chart; whereas, a system administration works in similar ways by ensuring

that the person issuing the command has the super-user privileges required to do sensitive tasks. Both of these business rules have one thing in common, that the party should have necessary privileges to access secret information about anything.

Designers can represent those business rules like this.

An example of a POLICY for our payroll application might be:

- *"Only managers can see the salaries of reporting employees."*

An example of a POLICY for the system administration might be:

- *"Only super users can view the system activity logs".*

Using the SSM, they can derive these business rules from core knowledge and can represent it this way.

Business Rules:

AnyParty <DepartmentHead> can see AnySecret <SalaryInfo> about

Anything<PayRollData> if he has <ViewSalary> privilege.

AnyParty <SuperUser> can see AnySecret <ActivityLogs> about

Anything<System> if he has <ViewLogs> privilege.

By capturing the stable business rules into the core model of SSM, they can avoid the repetitive effort to enforce common business rules across multiple domains. It will also ensure that the business rules are enforced uniformly for similar business processes. By delivering the business rules out of the box, the software stability model brings an excellent business value proposition to any solution.

3.4 Mapping to Service-Oriented Architecture

This section discusses different techniques for converting traditional models into SOA models using SSM. It provides an example design using a traditional model and shows how it can be converted into an SSM-based Service-Oriented model. Later it compares the models and highlights the advantages of using the Service-Oriented model. Component-based modeling systems have delivered business value for some time (Adler, 2004; Saleh, 2004), this chapter tries to show how an SOA can deliver better value with their services, which are abstractions of components.

A mobile navigation system is used as an example. The mobile navigation system implements services for different users to provide the shortest route, map for hotspots and specific destination locations relative to the current position of the vehicle. The specific system discussed here involves a customer-facing interface, which includes a touch screen for user input, a graphical display for displaying maps and directions, and an audio interface for speech output. This application is to provide a mechanism for a user to request and receive driving directions. The navigation system will take care of getting

the directions and plotting the route on the map. The GPS system will get the current location and use that data to display a real- time map.

Proceeding to design a traditional model (Figure 3.3) for this system, designers start thinking in terms of objects, the interfaces they use for talking with each other, and the methods used on those interfaces to accomplish functionality. Objects are passed as arguments and returned as results with these methods. This way the system is coupled through the interfaces and the objects are passed during various interactions between these objects.

Figure 3.3 Traditional Model for Navigation System

In contrast to the more granular approach to designing in terms of interfaces and methods, the Service-Oriented Architectures focus on a coarser level of services offered

by the system to other entities. So, in the Service-Oriented model designers concentrate on message exchanges rather than interfaces and objects passed. The SSM model makes this more reusable and applicable across different domains by abstracting the core knowledge and facilitating easy extension mechanisms. The core knowledge along with stable business rules will be extracted into the EBTs and BOs forming the core model in SSM. This core model can be extended to suit individual applications by adding IOs specific to each domain. This enables the core model to be reusable just by adding different IOs suitable to individual applications in different domains. This is illustrated in the following model (Figure 3.4) for the mobile navigation system.

Figure 3.4 SOA Model for Navigation System Using SSM

As can be seen from the model presented in Figure 3.4, the core pattern is cleanly separated from domain-specific extensions. For instance, in our model the specific services like RealTimeMapService and RoutePlottingService are extensions of

AnyService and are domain-specific. When it comes to designing another application that uses the concept of guidance like a “weapons guidance system,” the core model can be reused without changing anything. The only new effort would be in identifying and implementing the IOs specific to this application. The same method can be followed to develop any number of applications that use the same concept of guidance; whereas, in the case of traditional methodology the designers would be forced to implement different models for each of the applications, resulting in a lot of duplicate effort and cost.

CHAPTER 4

GOALS

4.1 Introduction

Goals are very important for understanding the purpose of any system. Before designers set out to achieve anything they need to have a clear idea of their goals. Without this understanding it is very difficult to plan on how to achieve anything. In the software development domain goals are also very important as a measuring stick to understand the user requirements of the system and to ascertain that both the users and developers of the system have the same objectives in mind. Identifying the goals is the first step in understanding the system and its core knowledge. Stable and scalable systems are built on this knowledge.

The goals of a system can be identified by thinking of “What is the system for?,” in SOA, “what is the purpose of SOA?” This section tries to identify the important goals of SOA and describe their models. Stability model defines goals as the EBTs (Enduring Business Themes) (Fayad and Altman, 2001). These represent the core knowledge of the system and are stable and non-tangible. In pattern languages these goals are represented as the stable analysis patterns (Fayad and Pradeep, 2003).

4.2 Primary Goals (EBTs)

The goals of the system can be classified into primary and secondary goals. The primary goals of the system are established to directly satisfy the system requirements. These are the outcomes of the requirement gathering. As part of analyzing the primary goals, designers can identify other goals that will help in attaining the primary goals. In this way the secondary goals attain the system requirements indirectly through the primary goals. This section describes the primary goals of the system as the EBTs.

4.2.1 Goal: Secrecy

Information is the most valuable resource for an enterprise. Making information accessible to the authorized entities enables the proper use of that information to deliver useful business services. Protecting the information from improper access is equally important, as improper access to information could lead to loss of competitive advantage, and improper handling. Thus, maintaining the secrecy of information is identified as a goal of SOA. SOA can achieve the secrecy of information as it is based on services. Services offer a coherent interface to the functionalities of multiple backend systems thereby shielding the service consumers from the internal details. This characteristic of services could be used to enforce the secrecy of information by using different mechanisms. Using this pattern the information can be classified into several secrecy levels and control the access based on different privileges assigned to service consumers.

Model: The secrecy has relevance in any system that manages information. This would mean secrecy applies to most of the systems in use across multiple domains. Thus it makes sense to capture the principles of managing secrecy and the associated mechanisms for reuse in all these domains. This can be accomplished by using the guidelines of SSM (Fayad and Altman, 2001) to arrive at a stable design pattern as outlined below.

Figure 4.1 Model for Secrecy Analysis Pattern

Name: Secrecy analysis pattern

Secrecy analysis pattern deals with the concept of securing information using any mechanism and the associated parties and privileges.

Context: Information is power for business systems. The information has to be easily accessible to the concerned parties in the system, but it should be protected against unauthorized access. To control access to the information it is essential to maintain a control layer that can properly authorize access to the data, based on the roles of the users and the sensitivity of the data requested.

Problem: Most of the systems require the information to be secured, but they greatly differ in the types of information, the sensitivity of it, the parties needing access to it, their roles and the different mechanisms used to secure the data. The ideal analysis pattern should bring out these common underlying themes while supporting maximum flexibility in the types of information, sensitivity levels and the users, roles, and privileges. The problem at hand is to design an analysis pattern that captures the underlying theme of secrecy and information hiding with flexibility to support diverse types of information, sensitivity levels, users, roles, privileges, and different mechanisms to secure the information.

Solution and Participants: The solution model is presented in Figure 4.1. In this model the secrecy protects sensitive information about anything using any mechanism from

anybody who does not have specific privileges. The participants of this pattern are described below.

Classes:

Secrecy: Secrecy is the EBT of this model. This represents the secrecy maintained within the system as the main driving force behind protecting classified information. It will also grant visibility of the protected information based on the privileges of the users.

Patterns:

AnyPrivilege: This pattern-BO represents the privileges associated with AnyParty and the privileges required for accessing AnySecret.

AnyParty: This represents the party requesting access to hidden information related to Anything. There can be one or more privileges associated with AnyParty that can aid in determining the accessibility of sensitive information.

AnySecrecyLevel: This represents the sensitivity level of information. The greater the secrecy level, the more privileges AnyParty would need to access it.

AnyMechanism: This represents the mechanism used to protect the secret from unauthorized access.

AnyCondition: This is a means to qualify and measure the state of AnySecret. The condition can be used to verify that the user satisfies specific criteria required to access sensitive information.

AnySecret: One of the characteristics or data about Anything, which is classified as sensitive. The Secrecy EBT protects the identified secret by using sensitivity level and privileges for the user requesting information.

Anything: This is the main entity that the pattern is trying to protect with data that can be classified with different sensitivity levels.

4.2.2 Goal: Fulfillment

Description: Satisfaction of user requirements and fulfillment of business expectations is for successful deployment and acceptance of any system. The fulfillment of service is an essential component of successful SOA systems that lead to user satisfaction. Fulfillment may also include coordinating with other interacting entities to deliver a functionality that is part of business requirements. In SOA-based systems fulfillment may involve one or more services and the workflow involved between service interactions. The fulfillment goal enables the designers to look at systems purely in terms of the necessary functionality and behavior. As the satisfaction of user requirements is the primary goal for any system, the fulfillment goal is very important in any system design. The fulfillment goal can be used as the yardstick to identify the capabilities required in the system and later used to identify possible enhancements to system behavior.

4.3 Secondary Goals (EBTs)

The secondary goals of the system are more definitive and help the system reach the primary goals. These can be viewed as indirect requirements and are presented below.

4.3.1 Goal: Improvement

Description: One of the key advantages of SOA is its ability to model critical business processes without affecting existing components, thereby supporting continuous business process improvement. Therefore, improvement has been identified as one of the goals of SOA. An SOA allows a clear representation of process flows identified by the order of the components used in a particular business service (Channabasavaiah, Holley, and Tuggle, 2004). This provides the business users with an ideal environment for monitoring business operations as process modeling is reflected by the business services and their order of execution. Process manipulation is achieved by reorganizing the business services in the process flow. This would further allow for changing the process flows while monitoring the effects, and thus facilitates continuous improvement (Channabasavaiah et al., 2004).

4.3.2 Goal: Delivery

Description: Service delivery is one of the most important aspects of a business system's usefulness to any enterprise. Making the services of a business system like an ERP system available to the users and offering a coherent interface that integrates the services

from multiple backend systems has been a major success factor for SOA. Thus delivery is identified as one of the goals for SOA. An important aspect of SOA is the emerging area of the service delivery platform (SDP) (Morris, 2005; Rich, 2004). The Service delivery capabilities of SOA aim to integrate disparate service elements encompassing the total system assets of the corporation and make them available to users over multiple access devices. This enables the users to take advantage of a fully integrated service platform leading to the best utilization of enterprise resources.

4.3.3 Goal: Service

Description: Service is a central concept around which SOA is built. Thus provision of service is identified as one of the goals of SOA. A service in SOA is an application function packaged as a reusable component for use in business processes (Colan, 2004). These are the building blocks for an SOA. Services are self contained, maintain their own state, and provide a loosely coupled interface. They encapsulate the functionality of one or more systems to provide a specific functionality to the service consumers thereby isolating them from the internal details of backend systems. This enables designers to look at systems in terms of services provided by them instead of getting bogged down by the internal details of individual systems, making it easier to integrate disparate systems to offer a unified view to the users of the system.

CHAPTER 5

CAPABILITIES OF SOA

5.1 Introduction

The goals of any system are achieved through a combination of capabilities of the system. So, identifying capabilities required for reaching all the goals is critical for designing a successful system. In the software stability model these capabilities are identified by looking at the driving forces necessary to achieve the specific EBT requirements identified in Chapter 4. If the goals of the system answer the question of “What can the system do?”, the capabilities will address “How are these goals achieved?”. These capabilities are the workhorses of the system and would form the business objects (BOs) in the SSM models. Each of these BOs is mapped to one of the EBTs. Though the BOs are internally stable, they can be externally adapted with IOs specific to different application domains.

In the stability model for pattern Languages (Fayad and Altman, 2001), capabilities are represented as the Business Objects (BOs). The process of determining the BOs for pattern language follows the guidelines of SSM and qualifies each BO with the following criteria:

- **Stability:** The BOs should be internally stable and should not change based on the application scenario. Each of the BOs should interact with EBTs and other BOs in the model.

- **Applicability:** The BOs should not be applicable only to a specific application context. They should be generic and applicable for all applications dealing with the same underlying theme.
- **Granularity:** The BOs identified should be clearly distinguishable from the other entities in the stable model such as other BOs, EBTs, and IOs.
- **Extensibility:** The BOs should be externally adaptable. This means the BOs should be extendable with IOs for specific application domains to suit variations in behavior. This makes the core-pattern of the SSM flexible and adaptable to different applications.

5.2 Business Objects or Capabilities

This section describes some of the important capabilities of SOA along with their models.

5.2.1 Capability: *AnyCoupling*

Description: Coupling measures the number of dependencies between modules of a system. Based on the number of dependencies between modules, the systems are classified into two types as loose coupling and tight coupling. Tight coupling integrates the modules of a system very closely and a change in one module can possibly impact all other modules and the system hugely. This makes it very difficult to change modules and maintain the system over time. SOA promotes loose coupling between service-

consumers and service-providers, by exposing a very few well-known dependencies among service consumers and providers. Loose coupling is an important capability of SOA (Hamza and Fayad, 2002a), as this reduces dependencies among services and service consumers making it easy to maintain and change systems. The *AnyCoupling* design pattern template [Appendix A] provides the details of using the concept of coupling in different domains.

Model: Using the SSM guidelines (Fayad and Altman, 2001), the following model has been designed for the *AnyCoupling* design pattern, which can be used in multiple domains.

Figure 5.1 Model for *AnyCoupling* Design Pattern

5.2.2 Capability: *AnyInformationHiding*

Description: Security is a very important aspect of any system. Sensitive information should be secured by the system from unauthorized access. System details that are likely to change over time need to be exposed only through controlled interfaces to avoid percolating these changes to other modules. Information hiding allows managing the security of information using different mechanisms of controlled access to the information. Any program that needs details of the hidden information needs to go through these mechanisms to gain access. This way the information hiding can enforce different security mechanisms to protect the secrecy of the data. As the security mechanisms offer a stable interface to the data, the service consumer does not have to know any implementation details of the service, thereby reducing complexities between service-consumer and service provider.

Hence information hiding is an important capability of SOA.

AnyInformationHiding design pattern template [Appendix A] provides the details of using information hiding in different domains.

Model: Using the SSM guidelines (Fayad and Altman, 2001), the following domain-independent model has been designed for *AnyInformationHiding* design pattern.

Figure 5.2 Model for *AnyInformationHiding* Design Pattern

5.2.3 Capability: *AnyService*

Description: Service is a function that is well-defined, self contained and does not depend on the context or state of the service (Service-oriented architecture definition, 2003). This enables developers to work with them without affecting other components promoting loose coupling and greater component reuse. SOA derives its strengths from the concept of services (Baresi, Heckel, Thone, and Varro, 2003) that can be reused for different purposes across an enterprise system. The capability of service is very important in SOA, as it enables the architecture to be flexible and extensible. The *AnyService* design pattern template [Appendix A] provides the details of using service in different domains.

Model: Using the SSM guidelines (Fayad and Altman, 2001), the following domain-independent model has been designed for *AnyService* design pattern.

Figure 5.3 Model for *AnyService* Design Pattern

5.2.4 Capability: *AnyType*

Description: Type identifies the nature of any entity in terms of behavior and the data representation. The significance of type comes into play when designers try to classify entities in a problem space, by grouping the entities with similar characteristics and

behavior. Type has applicability in various fields such as information technology, data warehousing, and inventory systems.

In SOA, type is used as one of the main criteria when searching for a service in a service-registry. The classification of services into proper types, and identifying the interfaces they offer is an essential part of any service-registry. It is also important to identify the types of arguments and return values of services when describing a service interface. So, the concept of type is a very important capability of SOA. The *AnyType* design pattern generalizes the concept of classifying entities into types in different domains. The *AnyType* design pattern template [Appendix A] provides the details of using type in different domains.

Model: Using the SSM guidelines (Fayad and Altman, 2001), the following domain-independent model has been designed for *AnyType* design pattern.

Figure 5.4 Model for *AnyType* Design Pattern

The following table explains some more of the SOA capabilities briefly.

Table 5.1 SOA Capabilities

Term	Type	Definition
Cohesiveness	BO	The measure of how well the pieces of a subsystem work together to produce the view of a single system.
AnyServiceProvider	BO	The party or software component that provides one or more services in a system.
AnyServiceConsumer	BO	The user or software component that consumes the service in the process of providing a system functionality.
AnyServiceResponse	BO	The message sent by the service provider in response to a request by the service consumer.
AnyConstraint	BO	Any condition that could be used to validate any observable behavior of an entity in the system.
AnySession	BO	A representation for a sequence of interactions between one or more components in a system.
AnyInterface	BO	The specification of the behavior of an entity with respect to its external interactions with other entities in the system.

Hook or ExtensionPoint	BO	A well-defined part of an interface that is intended for changing or extending the behavior of a system.
AnyActor	BO	A participant in the system with a clearly identified role in the system.
AnyMedia	BO	The media or mechanism through which the interactions between different systems take place.
AnyFormat	BO	The clearly defined arrangement to organize and present any information.
AnyRequest	BO	Represents a message encapsulating a request for a service.
AnySystem	BO	An autonomous body with well-defined behavior that can interact with other entities independently.
AnyInterpreter	BO	An entity that can consume information in a predefined format and take action on it.
AnyImplementation	BO	The practical application of an algorithm to fulfill a specified purpose.
AnyClass	BO	A collection of data and methods representing the behavior associated with the data.
AnySequence	BO	An arrangement of items in a linear fashion specifying an order among the represented entities.

AnyPeriod	BO	A representation for the duration of time elapsed from the beginning of an event and the end of it.
AnyAlgorithm	BO	A finite set of instructions for accomplishing a predefined task, which will change the state of a system in a predefined way.
AnySignature	BO	A specification for one of the interaction points for an entity containing method name along with types of input parameters it expects along the return values of that interactions.
AnyCategory	BO	A grouping of entities together based on well-defined criteria.

CHAPTER 6

KNOWLEDGE MAP AND STABLE ARCHITECTURE PATTERNS

6.1 Introduction

A knowledge map (KM) is a summary representation of a pattern language representing all the patterns and the interconnections between them. This will describe all the EBTs and BOs presented in the pattern language along with the relations between them. This can be used to generate multiple valid architectures for any system.

The KM for pattern language can be thought of as a subset of a universal KM that contains all the patterns sharing the common knowledge within the stable software development community. The use of KM to develop architectures ensures that they can be deployed across any system without depending on domain-specific details. KM promotes reuse by using systematic approaches of software stability. Many stable architectural patterns can be derived by the KM by combining the EBTs and BOs presented in the KM (Fayad and Hamza, 2003a; Fowler, 1997). Using the KM, architectures can be built by interconnecting patterns with any route combination of stable patterns to establish a solution for a new system, based on the capability requirements identified.

This chapter presents a portion of the KM for the pattern language for Service-Oriented Architectures. The KM properties are explained along with examples of building sample architectures from the KM using valid connections in the KM.

6.2 Knowledge Map Generation

The first step in generating a knowledge map for a pattern language is to identify all the EBTs and BOs. This involves understanding all the goals and capabilities of the system. EBTs and BOs for the pattern language are identified in Chapters 4 and 5.

The knowledge map can be generated, by considering the goals and capabilities along with the deployment and development scenarios of the system. The related EBTs and BOs are connected with arrows to indicate their relationships. In Figure 6.1 the solid lines connect the valid paths to follow to form a valid architecture. Multiple architectures can be generated from the knowledge map by following any combination of valid routes. Based on the requirements of any system, a set of valid routes will be activated to identify parts of the deployment, development, goals, and capabilities of the knowledge map with valid routes to form a valid architecture.

6.2.1 Layout of KM

Using the relationships established between the EBTs and BOs, the knowledge map can be designed. The EBTs and BOs can be presented along with their relationships in a knowledge map diagram summarizing the pattern language knowledge. Figure 6.1 below shows the layout of the partial KM for the pattern language.

Figure 6.1 A Portion of the Knowledge Map for SOA

6.3 KM Properties

The knowledge map contains all the EBTs and BOs of a pattern language to enable valid architectures using them and relationships between them. To extend the scope and generation of stable architecture patterns the KM should have the following properties that support better reuse for the patterns presented.

6.3.1 Partition

To better organize the EBTs and BOs represented, the KM can use partitioning. The KM can be partitioned in two ways. The first type involves dividing the KM into several smaller KMs. Each of these KMs can have some EBTs/BOs in common. For instance in the KM, designers can take one EBT along with the associated BOs to form one sub-KM.

Figure 6.2 Partitioned Knowledge Map

In the Figure 6.2, the patterns related to Development are presented along with their relations in the Development partition of the KM. Similarly the patterns specific to SOA are presented in the SOA partition. The intersection shows the common entities used by both the Deployment and SOA partitions. These are relevant to both deployment and SOA knowledge and are shared by both KM segments. It is also possible for multiple KM partitions to have common entities shared between all of them. From this diagram

stable architectural patterns can be derived from the EBTs and BOs that have common relationships. An alternative way of partitioning the knowledge map is to partition into sub-KMs that deal with a specific functional module. For instance in a sample KM for the development of web application that deals with database interactions and report generation, each of these functional areas can be represented as a different KM partition.

Figure 6.3 Remote Partition for Knowledge Map

The other form of partitioning is called remote partitioning where there are no common relationships shared between two KM partitions. For instance, consider patterns relating to the web services security standards such as OASIS, X.509 token profiles. Though these patterns are not part of the KM, they share some entities with the patterns covered in the KM such as Delivery as shown in Figure 6.3.

6.3.2 Creating Architectures from the KM

As the KM captures all the core knowledge of EBTs and BOs along with their relationships, stable architectures can be extracted from the KM by taking the related EBTs and BOs with valid connections. The following Figure 6.4 illustrates generating a stable architecture from the knowledge map. The generated architecture is highlighted and marked. It covers the related entities starting with Secrecy and following valid routes identified in the KM. The entities Fulfillment and Secrecy are included in the KM as there are valid routes from AnyParty to Fulfillment.

Figure 6.4 Generating Stable Architecture from KM

6.3.3 Directions

One important part of KM is the associations represented between various EBTs and BOs. The architectures derived from the KM use these associations to extract reusable patterns. Therefore, the validity of the architectures largely depends on the association being represented accurately. It is important to check the validity of all the associations and their directions in the KM for it to be usable. It will be also helpful to generate a few architectures from the KM and ensure their validity to check that the KM represents the knowledge accurately. This thesis has verified the association and validity of the KM by generating a few architectures with sample scenarios.

6.3.4 Rearrangement

The position of various EBTs and BOs in the KM need not be fixed in a KM. They can be rearranged to arrive at a different organization of the KM as long as they retain valid connections to the associated entities in the KM. This makes the design of the KM more flexible to accommodate different entities and their associated relationships without making it appear too complex.

6.3.5 Number of Applications

As the core knowledge represented in the KM is stable and can be applied to any domain where a similar problem reoccurs, it is possible to have an unlimited number of applications designed based on this KM. The sample architecture extraction presented

in section 6.3.2 illustrates the use of the KM to derive patterns that are applicable for different applications. For instance the sample architecture presented using Service can be used for any reservation system for resources. This can be also extended to any service request being performed by using other services, which happens in most of the service-based applications.

6.4 Conclusion

A knowledge map is a summary document for a pattern language. As it represents all the reusable parts of a pattern language such as goals, capabilities, and their relationships, the KM can be used to generate reusable architectures. The use of the software stability model (Fayad and Wu, 2002) as a foundation of the pattern language makes it easier to build stable patterns that encapsulate the core knowledge. This promotes stability and reuse across multiple systems. Once the KM for a pattern language is constructed it becomes easier to construct valid architectures by using systematic methods to extract EBTs and BOs from the KM. This way every different route in the KM results in a valid architecture that can be reused across multiple systems.

CHAPTER 7

DEVELOPMENT AND IMPLEMENTATION

7.1 Introduction

Implementation of sample patterns from a pattern language is important to establish the practicality of these patterns for different application domains. Once the goals and capabilities are identified the core layer of the patterns are well defined in terms of EBTs and BOs. When designers set out to implement the patterns the first step is to implement the core layer (first or top layer) with the clear relations between EBTs and BOs that form the stable model.

The next step is to demonstrate the applicability of the core patterns to a specific application domain. This involves first identifying any domain-specific extensions as Industrial Objects (IOs). Designers should identify a set of IOs specific to this domain corresponding to the BOs of the pattern and establish the relationship between them. For instance, the specific services available and the interactions between producers and consumers in an application is going to vary based on the specific application. These application-specific behavior variations are captured in the IOs. Identification of all the IOs and the relationships between IOs and corresponding BOs would complete the design of the aGpplication.

The author has implemented the core entities and sample applications for two important patterns from the pattern language. The *AnyService* pattern demonstrates the use of the Computation EBT and the BOs *AnyAlgorithm*, *AnyService*, *AnyClass* for two

applications. The first application demonstrates the use of services in a library system and the second one for a stock quote application. The *AnyInformationHiding* pattern demonstrates the use of the Hiding EBT and the BOs AnyClass, AnyActor, AnyInterface for two applications. The first application demonstrates the use of information hiding for a banking application and the second application demonstrates its use in a military application. This thesis has used hooks extensively to weave the BOs and IOs into the application. The next sections discuss the details of development and implementation of these patterns.

The first or top layer shows the abstract model. To complete the prototype of this model, this section further elaborates the top layer of the architecture. This is accomplished by showing the inner layer by expanding the top-level pattern-EBTs and pattern-BOs. This involves taking pattern-EBTs and pattern-BOs from the core layer architecture and designing a model around the unexpanded pattern-EBTs and pattern-BOs. The expanded models for one of the important entities is shown in Figure 7.1 below, showing AnyActor expanded for the *AnyInformationHiding* design pattern. When the top layer AnyActor entity is clicked, the expanded model for AnyActor will be visible as shown in Figure 7.1 below.

Figure 7.1 Expanded Model (Second Layer) for AnyActor(First Layer) Pattern

Similarly the other pattern-EBTs and pattern-BOs for *AnyService* and *AnyInformationHiding* design patterns can be expanded. For the *AnyInformationHiding* design pattern, designers can expand the pattern-EBT Hiding and the pattern-BOs

AnyImplementation, AnyInterface, AnyClass. For *AnyService* design pattern designers can expand the pattern-EBT Computation and the pattern-BOs *AnyType*, *AnySignature*, *AnyAlgorithm*, *AnyClass*, *AnySequence*, *AnyPeriod*, and *AnyMedia*. The implementations for these inner layer patterns are presented in Appendix B.

7.2 Implementation Issues

7.2.1 Type vs, Class in the Type-Oriented Paradigm (TOP)

The Type-Oriented paradigm is very useful concepts used in modern architectures such as Object-Oriented systems. Though these concepts have been used for a long time in the design community, domain analysis by software stability model emphasized significance of types even more for identification of entities and services for SOA-based design (Fayad and Wu, 2002).

In Type-Oriented systems the types define interfaces capturing the common behavior for a group of entities that share some common characteristics (Fayad, 2002). The type names an interface and its specification includes a set of methods and contract for corresponding behavior by the interface. This includes the complete set of services and requests supported by that interface. Each of the methods has a signature indicating the name of the method, the expected input arguments and the return value. The various elements of a signature include: intended purpose of the operation name, parameter names, what the interface promises to do when invoked, what the interface does in exceptional circumstances and the return value for the method. For instance the Integer

interface may have methods like `int add (int)`, `int subtract (int)`, `int multiply (int)`, and `int divide (int)`.

Classes implement types by implementing the behavior for all methods defined in the interface as per the contract. An interface could be implemented by multiple classes honoring the underlying contract for each of the methods defined in the interface. Similarly a class can implement more than one interface as long as it honors contracts for methods in all the interfaces it implements. The type definition needs to follow a prescribed convention to achieve accurate communication of the contract.

One of the major advantages of a type/interface is that once the clients use the interface they can replace one class with another one implementing the same interface. For this substitutability to work the clients are advised to always use interfaces where possible. Substitutability can be summarized as “requiring no more and promising no less”, which means one class cannot impose extra requirements on the client beyond those defined by the interface itself. The behavior of the class in cases where the client failed those requirements should be adhered to as defined by the interface.

The concept of type and classification of entities into types has been captured by the *AnyType* design pattern. This involves identification of a set of constraints for testing the qualification of entities for each of the types supported. Based on the criteria as defined by the types, each entity may belong to one or more classes. This includes capturing the behavior of a type into an interface making it easier to shield the implementation details from the clients. This offers a good amount of flexibility for the developers to change implementation details of a class without affecting the clients as long as they continue to

honor the contract of the interface. In this way types encapsulate the design whereas classes encapsulate the implementation of those types. The pattern template for the *AnyType* design pattern is presented in Appendix A and explains it in more detail along with two sample applications.

For instance, let us take the example of an inventory system maintaining the list of products. The product list is a public interface that needs to be exposed to the clients that need to know about the products in the inventory. This is extracted as a type like a List. Internally the type List can be implemented as a LinkedList, Vector or a Set. But the client need not be exposed to these details. This enables the system implementers to choose a suitable/more efficient implementation at a later date as long as they adhere to the same contract as the List type. This is illustrated in the example given for the *AnyType* design pattern in the Appendix A. The participating classes in the example are illustrated in the following Table 7.1 and Table 7.2.

Table 7.1 *AnyService* Type Relations for Stock Quote and Library Examples

Super Class	Sub Class
<i>AnyService</i>	StockquoteService ForexService BookFinderService BookStatusService

AnyAlgorithm	ForexAlgorithm StockquoteAlgorithm BookFinderAlgorithm BookStatusAlgorithm
AnyClass	ForexStockquote BookSearchClass

Table 7.2 *AnyInformationHiding* Type Relations for Banking and Military Examples

Super Class	Sub Class
AnyActor	AccountOwner BankClerk Captain Colonel General Major Sergeant
AnyClass	CheckingAccount Weapons

7.2.2 Use of Hooks

Hooks are a very useful construct in making the pattern easy to use in deploying and making it easier to develop extensions to the pattern (Fayad, Johnson, and Schmidt, 1999). Use of hooks can be simplified even more with a hook tool that can automatically generate hook code for attaching the various components for an application. A hook encapsulates a set of extension points in the original pattern that can be changed / enabled / disabled or augmented by the developers to produce a deployable application using the original design pattern (Froehlich, Hoover, Liu, and Sorenson, 1997, 1998).

The hook description includes details of the type of hook, intended usage, and various classes it will affect. This documents all the details of the hook and the affected portions of the design clearly for the developers. The various parts of a hook template and their meanings are presented in the following Table 7.3.

Table 7.3 Hook Template Details

Name	A unique name to identify the hook.	
Requirements	A description of the problem the hook is intended to solve.	
Type/Level	Method of adoption used.	Amount of support provided for the problem by the framework.
Area	Parts of the framework effected by the hook.	
Uses	The other hooks required for using this hook.	
Participants	The components of the pattern that participate in this hook.	
Changes	The main section outlining the changes to interfaces, associations, and control flow among the components from the participants section.	
Constraints	Limitations imposed on the hook.	
Comments	Any additional description on usage of the hook.	

The implementation has used hooks extensively to relate the components of various design patterns in the sample implementation. This made it easy to extend the design to implement various sample applications based on our patterns. The sample hook templates used for the *AnyService* pattern are presented here as examples (Table 7.4).

Table 7.4 Hook for Creating a New Algorithm IO

Name	Create New Algorithm	
Requirements	A new algorithm has to be implemented to accomplish a custom task.	
Type/Level	Augmenting behavior	Open-ended
Area	Services	
Uses	GetUserInput	
Participants	AnyAlgorithm	
Changes	subclass NewAlgorithm of AnyAlgorithm NewAlgorithm.init specializes AnyAlgorithm.init new operation addQuote:symbol:price: repeat as needed symbol = GetUserInput ['symbol'] price = GetUserInput ['price'] NewAlgorithm.addQuote (symbol, price) NewAlgorithm.performTask specializes AnyAlgorithm.performTask **AddCode**	
Constraints		
Comments	This creates a new algorithm to be used by a custom service, as part of <i>AnyService</i> pattern.	

As the hook description language is very powerful, it is possible for the hook tool to implement a great deal of functionality without requiring large amounts of code written by the developers. For instance in the following hook (Table 7.5) developed for AnyClass, the participant's sections indicate that the IOs use an aggregation relationship with the BO.

Table 7.5 Aggregation Hook for *AnyService*

Name	Aggregation Hook for <i>AnyService</i> Pattern	
Requirements	All the IOs for AnyClass need to be hooked up to the BO AnyClass	
Type/Level	Adding a feature	Supported Pattern
Area	Services	
Uses	GetUserInput	
Participants	anyservice.AnyClass;stockquote.ForexStockQuote;library.BookSearchClass	
Changes		
Constraints		
Comments	This hooks all the IOs available for the BO AnyClass to be hooked up. All the users of AnyClass implementation will lookup the various IOs from AnyClass instead of creating them directly.	

The hook tool uses this information to generate the hook to attach the IOs with the BO (Figure 7.2).

Figure 7.2 Hook for BO AnyService

Another hook is used for generate IOs extending the BOs and specializing specific behavior. The following hook (Figure 7.3) creates a subclass of BO AnyActor and overrides the init() method. The name of the new class will be taken from the user input.

Figure 7.3 Hook for Creating a Subclass for AnyActor

A hook makes the design of a pattern very structured and easy to use and extend. Using hooks, the design can be easily modified to suit the application requirements. The different ways a hook can alter a design include “enable,” “disable,” “replace,” “augment,” and “add a feature.” The different mechanisms used for hooks may include “aggregation,” “association,” “inheritance,” etc. The hook may be as simple as altering the behavior of a single component or it may involve changing interactions between several components in a design. It is also possible for one hook to use another hook making the use of hooks more modular.

7.2.3 Unique Segments of Code

The *AnyService* pattern and *AnyInformationHiding* pattern have been implemented along with sample applications for them. This will illustrate the usefulness of the patterns in designing real-world applications based on the patterns. The implementation also has used hooks in providing extension points and various places the design can be modified to suit the specific application scenario.

One of the common hooks used in the implementation is the aggregation hook, which manages the aggregation relationship between the BO and the various IO extensions for it. For instance, in *AnyService* implementation the BO *AnyService* has multiple IOs that extend its behavior such as *ForexService* and *StockQuoteService* for the stock quote application and *BookFinderService* and *BookStatusService* for the library application. The *AnyServiceHook* manages the relationship between the BO and its IOs. The hook template in Figure 7.3 above will generate the hook code for *AnyServiceHook* (Listing 7.1) as follows:

```

public class AnyServiceHook extends AnyService
{
    public BookStatusService bookstatusservice;

    public BookFinderService bookfinderservice;

    public ForexService forexservice;

    public StockQuoteService stockquoteservice;

    public AnyServiceHook()
    {
        super();
        this.bookstatusservice = new BookStatusService();

        this.bookfinderservice = new BookFinderService();

        this.forexservice = new ForexService();

        this.stockquoteservice = new StockQuoteService();
    }
}

```

Listing 7.1 AnyServiceHook

Here, the various IOs are managed within the hook and are made available to the main BO. This hook manages the IOs by instantiating them and making them available to the BO. The hook tool has used the dependencies section in the hook template to identify the components and their associations. The *AnyService* is the base BO to be extended, whereas all the other IOs are related to it with an aggregation relationship because of the “;” symbol. Based on that, the tool has generated the hook class with the IOs as members along with the constructor creating the IOs. All the other components

needing to use any of the IOs will request the BO for getting access to the IO. The corresponding code in the *AnyService* BO for managing the access to IO is listed below (Listing 7.2):

```
/**
 * ##### Hook Methods #####
 */
private static AnyServiceHook hook = null;

public static AnyService getHook(String IO) {
    if (hook == null) {
        hook = new AnyServiceHook();
    }
    if (IO == null) {
        return null;
    }

    try {
        String ioName = IO.toLowerCase();
        Field iof = hook.getClass().getField(ioName);
        if (iof == null) return null;
        return (AnyService)iof.get(hook);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return null;
}
```

Listing 7.2 BO *AnyService* Using the *AnyServiceHook*

Any Component in the design that needs to use the IOs will go through the BO *AnyService* for accessing them. For instance the *ForexStockQuote*, which implements the combined functionality of two services will need to access these two services for

sequencing them together. The corresponding code in the actual class will be as follows (listing 7.3):

```
/* Get each of the services and add them to the sequence with actions */  
AnyService stockQuoteService = AnyService.getHook("StockQuoteService");  
this.addService(stockQuoteService,"O0");  
  
AnyService forexService = AnyService.getHook("ForexService");  
this.addService(forexService,"P1,O1");
```

Listing 7.3: The Class Accessing IOs Through the BO AnyService

The hooks can be further used by IOs to access other IOs required to perform their functionality. This way all the associations are managed by the hooks and can be configured easily without manual code changes that are prone to errors. In the *AnyService* pattern a service in turn is going to use an algorithm to perform its task. The IO *StockQuoteService* will get the specific algorithm IO *StockQuoteAlgorithm* from the *AnyAlgorithm* hook, which in turn will use the *AnyAlgorithmHook* to get the IO. This is illustrated in the following code segment (Listing 7.4) from *StockQuoteService* class.

```
/* Get the algorithm IO from the hook and set it in our parent BO */  
AnyAlgorithm myAlgorithm = AnyAlgorithm.getHook("StockQuoteAlgorithm");  
super.setAlgorithm(myAlgorithm);
```

Listing 7.4 The StockQuoteService IO Accessing AnyAlgorithm IO Through the BO

This way the various IOs will interact with the other BOs, IOs or other classes through the hooks to make the whole design modular and easy to be configured per the application requirements. This will also streamline the way in which the design can be extended and modified ensuring that only the legal modifications are made to the design.

7.2.4 Aggregation

Aggregation is one of the fundamental mechanisms used to build complex infrastructure from simple components by composing them together. Aggregation differs from simple composition in that there is no implication of ownership for the aggregation relationship, and the enclosed components have their own life-cycle apart from the enclosing component. For instance an online shopping service may be an aggregation of multiple services such as product catalog service, credit verification service, and billing service. If the online shopping service is discontinued, it would still make sense to use the other services such as credit verification service on their own, and they will continue to exist independently.

Example Use of Aggregation: The *AnyService* design pattern [Appendix A] illustrates the use of aggregate relationship between entities in the pattern. The *AnyService* design pattern deals with the concept of service and computation. In this pattern *AnyClass* BO provides the functionalities to users using one or more *AnyService* instances. This has been modeled as an aggregate relationship between the *AnyClass* BO and *AnyService* BO. The following model (Figure 7.4) highlights this relationship.

Figure 7.4 Aggregate Relationship in *AnyService* Design Pattern

This aggregate relationship is implemented as part of the pattern in the *AnyClass* implementation. In this example the relationship is represented as a java Collection holding references to instances of different services. The segments of code dealing with this relationship are presented below (Listing 7.5).

```

/** ##### Segment of code for AnyClass ##### */

public class AnyClass {
    protected List services;
    //.. other members ...

    public void init() {
        services = new ArrayList();
        //...other code...
    }

    public void addService(AnyService ser, String pmap) {
        services.add(ser);
        seq.addService(ser, pmap);
    }

    public Iterator listServices() {
        return services.iterator();
    }
}

```

Listing 7.5 Aggregation Relationship for AnyClass

Aggregation relationship is used predominantly throughout these pattern models and illustrated the use of aggregate relationship for hooks in the hooks section 7.2.2. The *AnyService* hook is an example of aggregate relationship and shown in Table 7.5 below. The implementation for this aggregate hook is shown in the “unique segments of code” section in Listing 7.1 and the use of aggregate hook in Listing 7.2.

CHAPTER 8

DEPLOYMENT AND VERIFICATION

8.1 Introduction

The deployment phase is one of the most important phases for systems based on the software stability model (SSM). A smooth and easy deployment of a system is very important in integrating the new system with existing systems, and in gaining the confidence of administrators of the systems and will go a long way to indicate easy maintenance of the system. After development of the model is complete, one needs to deploy the system into the target environment. This involves identifying, addressing, and managing the issues that arise during the deployment of stable patterns developed in the earlier phases.

In the complex systems of the real world, the deployment phase can take considerably more time than the initial development phase. So, it is even more important for the architects to design the patterns so that they can be easily integrated into the existing systems with automation. This enables the deployment team to introduce the patterns into use in production systems across the enterprise.

Based on the above requirements, the design patterns needs to address the following issues for easy deployment:

1. Validate and verify that the developed patterns would satisfy the requirements of the systems when used in production?

2. Verify that a composition of various patterns can be formed so that, regardless of the route taken within the knowledge map, a stable system is always achieved.
3. Ensure that no abnormalities are found while using the new patterns simultaneously across multiple existing and new systems. Consider how abnormalities result in additional time and cost for the organization? This issue is true for critical systems where guaranteed execution of service requests is crucial for the systems success and acceptance by the users. For example, in real-time financial systems such as stock trading systems, any failure within a micro second space can cost the customers possibly billions of dollars. The impact of a missed service request would greatly depend on the error tolerance level of the system and it can vary greatly between different application domains. How can designers make sure that reliability of the system is addressed with care?
4. Cover all aspects of the deployment process like the people who should be involved in the deployment of patterns, and the roles and responsibilities of the participants within this phase?

This chapter discusses these issues throughout the following sections and explains how the patterns address the above concerns. The next section discusses the patterns for deployment and verification for the pattern language based on the above concerns. The verification and validation section addresses the point-1 above by illustrating the validity with sample applications from different domains.

8.2 Patterns for Deployment and Verification

Based on the above concerns that are common within the deployment space of any software component, designers would need to determine the enduring business themes (EBTs) and the business objects (BOs) that are based on the goals of deployment and verification of the system. Some of these EBTs and BOs can also exist across different contexts within the universal knowledge map. Using these patterns can deliver huge cost and time savings for the organization to maximize return on investment.

The various factors required for a successful deployment of a system can be classified into three different categories such as deployment, quality factors, and verification and validation. The deployment patterns influence the ease of deployment and adaptability of the system (MSDN, 2004b). The quality factors will help in measuring the effectiveness of the system. The verification and validation patterns determine the ease of verifying the system behavior and the fulfillment of functional requirements for the business. The three types of patterns are presented in the following Figure 8.1.

Figure 8.1 KM Partition for Deployment and Verification

8.2.1 Deployment Patterns

The important EBTs and BOs for the successful deployment of systems based on Service-Oriented Architecture are discussed below.

- **Adaptability:** Most of the enterprises use numerous software systems to perform different types of business operations. It is rare that any of these systems can function in isolation and deliver the business requirement. For successfully fulfilling the business requirements a system needs to interact with other systems in the business to perform a greater business workflow. Adaptability of the system determines how well it can integrate with other systems under various deployment scenarios. Integrating a new system with the existing system infrastructure is one of the major problem areas that take up a great amount of time and are costly. The adaptability of the system will make this easier by cutting the time and cost of integration.

The pattern language establishes low coupling as one of the design patterns that influence the adaptability of the system. The use of services established by the *AnyService* pattern makes it easier to integrate diverse components without getting bogged down by the internal details of them. The sample application shown in section 8.3 below illustrates the ease of integrating multiple services to perform business functions.

- **Extensibility:** The extensibility of the system refers to the ease of extending the existing functionality of the system. With the increase in complex requirements of enterprises, most of the time it is difficult to find a system that can fulfill all the

requirements established. Even the currently deployed systems may need enhanced functionality because of changing business requirements and competition. Extensibility enables the system functionality to be enhanced without breaking and redesigning existing components in the system.

The pattern language has been designed based on the software stability model, which has proven mechanisms of extensibility. The core patterns remain stable over change and can be extended by adding new IOs to the system to enhance functionality. All the patterns presented in the pattern language are extensible. This has been illustrated with two applications for each pattern in Appendix A.

- **Customizability:** Customization refers to change in the way the system behaves in a way determined by the business. As each of the businesses behave in a different way and have their own established practices it is required to customize systems as per the individual requirements of the enterprises. If the system is designed in a closed and rigid way, it is going to be very difficult to customize it to suit individual requirements. A system that is highly customizable can save a huge amount of time and cost during the deployment of a system.

The patterns presented in the pattern language are easy to customize as they are based on the software stability model. In the SSM the core model remains unchanged while the IOs can be customized to suit specific application needs. This customizable behavior is inherent in the SSM making customizations easier for systems based on SSM.

8.2.2 Verification and Validation Factors

Verification and validation is a very important phase in determining the acceptability of the final product as a system. Verification involves testing the functionality of the individual components and their interactions with the user-facing services to ensure that the system has been built right. Validation involves validating the product behavior against the product specification established at the beginning of the project. For successful acceptance of the system it should have the following factors.

- **Testability:** Testing is an important quality factor in ensuring the correctness of the system and that the prescribed goals are being met by the system under specific constraints. In a typical application environment, testing would involve creating various test case scenarios based on the available knowledge at any given point of time. However, capturing every test combination would become cumbersome and impose high overhead in the overall deployment process.

Service-Oriented Architecture is convenient for testing as different service components are supposed to behave independently without relying on the state of other components. The pattern language highlights this advantage with the low coupling and improvement patterns. The Improvement EBT captures the behaviors of the system that need to be enhanced and enables the testing of the improvement.

- **Verifiability:** For a system to be accepted, it is necessary to verify its behavior against established product requirements. This requires that the system behavior

be easy to observe and measure. As the systems get more complex with increasing sophistication in modern software systems, it gets very complex and time-consuming to observe and measure the behavior. This makes it even more important to make it easy to verify the system behavior.

The pattern language establishes clear, measurable quality and performance factors for the system. The enhancement design pattern uses the measurable conditions of the system to verify the behavior of any system enhancements or any changes in general.

- **Maintainability:** Statistics show that in a typical system lifespan most (over 80%) of the effort goes into maintaining the system as opposed to designing, developing, and deploying the system. To get the maximum return on investment, the systems need to work reliably over long periods of time with few issues. Given that the existence of bugs in any systems is unavoidable, it is important that the system is amicable to frequent changes for fixing bugs or introducing new features to adapt to changing business requirements is very important.

The use of the software stability model and its capability of retaining the core pattern stable while adding new extensions makes it easy to design systems that are easy to maintain. All the patterns in the pattern language are designed based on the SSM model and therefore support easy extension capability. The core pattern components such as EBTs, BOs do not change, making it easier to test them and ensure low defect count and easy maintenance, thereby ensuring maximum ROI on the system.

8.2.3 Quality Factors

Quality factors mainly determine the usefulness of the system and how well the system can satisfy the requirements in general. These can be viewed as the yardsticks for purchase decisions for software systems by enterprises. The quality factors deal with system characteristics that influence the successful deployment of the system such as reliability, delivery, security, and scalability. The important quality factors used are outlined below.

- **Delivery:** The effectiveness of any system will depend largely on the availability of the system to the users and the usability of the system. Delivery of the available services to users will increase the usefulness of the system and in turn will show return on investment (ROI).

Service-Oriented architectures excel in this by making the backend system facilities available through various systems using the standard services interface. The Delivery EBT captures the dynamics of service delivery and the interactions with associated patterns. This will empower the designers to ensure the services are designed with ease of delivery in mind.

- **Reusability:** Reusability of the patterns is the core requirement of the deployment to get maximum benefit out of the investment. This would also have other profound advantages like reduced maintenance costs and more quality components as the same components are getting used in multiple places. This leads to a smaller number of components and more focus on them, making them

of higher quality and reducing the number of different behaviors the users need to learn about the system.

Loose coupling provided by the EBTs will make the components more independent and easier to use in different modules in the system. Designers also would need to ensure that the patterns would fit well within the context of the application, to integrate them seamlessly. The *AnyCoupling* BO captures the objective of achieving loose coupling between different components and guides the designers in achieving high levels of reusability in the system.

- **Security:** Security is a very important aspect in most of the business systems as they deal with business transactions. The InformationHiding EBT ensures that the basic security concerns are addressed in the architecture. The service standard is still in the early stages of evolving a standard security model to address more stringent security requirements for service delivery.

The Secrecy EBT from the pattern language addresses the issue of maintaining information accessibility in a controlled way. This EBT deals with classifying the data based on its sensitivity and makes it available to other components based on the roles and rights. This ensures that only components with sufficient privileges will get to see sensitive data and maintain restricted access to other users of the system.

- **Portability:** The pattern developed should be portable to ensure that the expected behavior from the system is the same regardless of location and platform under which the patterns are deployed. In SOA-based systems various participants

could be residing in different platforms and in different geographic locations introducing factors like language and localization. So, it is important to ensure that the behavior from the components is the same.

Service-Oriented architecture and services in general are cleanly separated from any platform dependency compared to their earlier counterparts like RMI, DCOM, and CORBA. The *AnyService* design pattern captures the concept of service as a component that does not expose any platform-specific behavior and independent of the deployment scenario. This enables the systems to be highly platform independent and interoperable.

- **Reliability:** Reliability is one of the critical goals of any business system. Unreliable systems are not acceptable in a real-world business scenario leave alone any mission-critical applications. Fault tolerance is another goal that is related to reliability of the system. Hardware or software failure is unavoidable in real world deployment, though the probability of failure is very low. But even those few instances of failure are not acceptable to business, so designing the system to recover from failure and using multiple systems to mitigate for failure is common in reliable systems. The patterns should support these underlying concepts by low coupling and transparent failovers with services.

The portability of services that are captured by the *AnyService* design pattern has other significant benefits like platform-location independence for the components. This along with the requirement for services to be state free enables the design of systems that can be scaled and to build redundancy into the system

along with failover mechanisms. All these features combined will lead to high reliability of the system and the disaster recovery mechanisms in the rare case of system failures.

- **Scalability:** Over the lifespan of a system it is expected that the load on the system will grow as the demand increases on the business. So, it is very important that the pattern support scaling the system over a period of time with little impact to the underlying architecture of the system.

The modularity and the on demand deployment of services ensure that the system scales well under load conditions. The Service and Delivery EBTs ensure that the services will support various deployment options and multiple service providers can serve the service requests. This ensures that the load can be shared among different service providers transparently, making the systems scalable to handle higher system loads.

8.3 Verification and Validation

Verification and validation involves establishing by objective evidence that the pattern meets all the goals and is extensible to meet different domain requirements. In case of this pattern language, designers need to extract sample architectures from the language and build applications around these architectures to establish the applicability. This thesis did the verification and validation to all the patterns provided in the knowledge map by applying different scenarios to each of the patterns to make sure that

they are modeled correctly. Two sample applications for each of the four patterns are provided in Appendix A.

Here one model (Figure 8.2) is taken from the pattern language and its applicability using two sample applications that are tested to verify the model is illustrated.

Figure 8.2 Model for *AnyService* Design Pattern

For this model, two application scenarios are attached to verify the model and establish its applicability. This establishes the validity of the model in different scenarios and its applicability in different application domains. This section will further show that the core pattern remains stable for both these diverse application scenarios, thereby establishing the usefulness of the software stability model to build stable systems. The

diverse natures of the applications highlight the extensibility and adoptability of the models.

Application #1: Stock Quote Service in Multiple Currencies

Getting stock quotes for trading is a very important and common service used in the brokerage systems. The service is generally provided by the stock exchange companies themselves or some network companies that provide feed to all brokerage firms. Though most of the stock quotes systems deliver the stock quotes in the local currencies of the markets, it would be nice to be able to get the stock quotes in different currencies with the growing number of global investors. This example (Figure 8.3) illustrates a sample class that combines the services of a stock quotes service along with that of a foreign exchange rate service to give stock quotes in different currencies. This application illustrates the use of AnySequence to orchestrate the execution of these two services and the shuttling of results between them. Both these services use their own AnyAlgorithms to perform their specific functionality and the IO ForexStockQuote provides the combined service, which is an extension of AnyClass BO.

Figure 8.3 StockQuote Example

Application #2: Book Borrowing Service

The most common task performed by patrons of a library is to search for a book in the catalogue and find if it is available. The book borrowing service (Figure 8.4) combines these two tasks to make the job of finding a book that is available for borrowing. This involves two services. The first service finds the books by a specific author, or by its title. This service has two signatures: one service that takes the author's name and another that takes part of the title. Both these signatures will find the books and return them with details like the serial number. The second service takes the serial numbers of a set of books and returns the availability of these books. The class `BookAvailability` will use a sequence to orchestrate the above two services to find books

that match the criteria along with their availability status. As can be noted from the following model that the core pattern has not changed; only new IOs are added to it to match this specific example.

Figure 8.4 Book Borrowing Example

As can be seen from both the above examples, the pattern has been used effectively to build two sample applications, both addressing a different domain problem. The core pattern remains the same; the IOs suitable for each application scenario are hooked to the BOs in both cases. This verifies the applicability and correctness of the pattern.

CHAPTER 9

FUTURE WORK AND CONCLUSIONS

9.1 Introduction

This thesis develops a “Pattern Language for Service-Oriented Architecture” or a “Knowledge Map for Service-Oriented Architectures (SOA)” as a system of patterns capturing the core knowledge for SOA. This pattern language (Coplien and Schmidt, 2002) uses the software stability model (SSM) to analyze and design more stable systems and to generalize the main concepts of SOA independent of any specific domain (Fayad and Altman, 2001; Fayad, 2002b).

Alternative architectural approaches, such as Model Driven Architectures (MDAs), Pattern Based Architectures, Model Based Architectures, Product Line Architectures, and Aspect-Oriented Architectures to design systems are studied and are compared with the SOA-based architectures. By using evaluation criteria such as scalability, adaptability, reusability, and extensibility, this thesis has shown that SOA has major advantages over the other architectures. The thesis has identified the specific characteristics for effective SOA architectures (Endrei, 2004) in Chapter 3, and the goals and capabilities required for designing good SOA architectures in Chapters 4 and 5. The knowledge map of the pattern language has been presented summarizing the various possible routes that can be taken with the EBTs and BOs to arrive at valid architectures for SOA-based systems (Fayad, 2002b).

9.2 Challenges

Though SOA-based architectures address many issues found in software systems, it is essential to follow established guidelines to come up with effective systems that can live up to the expectations of modern business systems. The thesis identifies the common issues designers face while developing systems and lays down guidelines for designing effective SOA architectures.

Identifying Services: The concept of Service is central to SOA. Oftentimes the most difficult part of the design starts with identifying the candidates for services in a system. The SSM makes it easy to identify the services by capturing the core concepts of any system. The problem at hand is to bring out a stable pattern that represents the common concerns behind services that is easily adaptable and easily extensible to any kind of service across multiple domains. This is addressed in the *AnyService* design pattern from the pattern language.

Adaptability and Extensibility: Another important issue in designing systems that can be reused and adapted easily with existing components is low coupling. Designing systems that are cohesive while keeping low coupling is a challenge faced by designers. The *AnyCoupling* design pattern from the pattern language generalizes the concept of low coupling to be used in a generic way across multiple domains.

Security: Security is an essential component in enterprise software systems. This is a reoccurring theme in multiple components of the enterprise architecture and it does not make sense to design different models dealing with the same concept. The *AnyInformationHiding* design pattern covers the concept of information hiding in a generic model applicable for different domains.

Type and Reuse: The concept of type and classification is central to identifying recurring behavior patterns and finding common solutions for them. Existing techniques used to classify items into types are specific to particular domains and are not formalized. The *AnyType* design pattern addresses the classification of entities into types in a generic way applicable across multiple domains.

Delivering Business Value: Though the current SOA-based systems deliver a good amount of functionality out of the box for easy deployment, they do not encapsulate any logic for business rules central to the business. For delivering real business value proposition the components delivered should support some level of business rules common to the business scenario at hand. The use of software stability model ensures that the core models capture the business rules around the underlying theme and deliver them consistently across multiple domains wherever the core-patterns are used and extended.

9.3 Future Work

Additional Patterns for SOA: With the explosive growth in the Service-Oriented architecture space, it will be important to identify and adopt patterns that yield effective SOA architectures. As the usefulness of these patterns and SSM for designing architectures that are applicable across domains gain more momentum, designers can look forward to seeing development of more patterns covering more areas of SOA. Combining these patterns with the flexibility and standardization of SOA will lead to a greater advancement of systems that deliver greater business value.

Patterns for All Stages of System Life-Cycle: Following the concepts of SSM, the patterns from the pattern language can be adopted to various application domains by extending the BOs with the IOs specific to the domain (Fayad, 2002b; Fayad, Schmidt, and Johnson, 1998). As SOA specifications and standardization cover more areas of the component interactions, it is possible to develop more patterns dealing with all aspects of the system life-cycle making it easier to design, develop, and deploy systems with greater ease.

Complete Knowledge Map: The knowledge map of the pattern language can be further enhanced to cover all the up-coming patterns and the possible interactions between them. Developing a few prototype applications, based on the patterns in the knowledge map would help the designers understand the concepts in a better way.

9.4 Conclusions

Though SOA is very effective in designing modular architectures that deliver effective business solutions, a lack of patterns facilitating stable and extensible designs is reducing the effectiveness and success of these systems. The pattern language for SOA presented in the thesis addresses the drawbacks of current design methodologies by covering different aspects of designing SOA-based systems. The knowledge map presented can be used to derive stable architectures for designing effective systems that can evolve based on expanding system requirements. The pattern language enables the system architects to design more effective SOA architectures using the guidelines provided. The pattern language covers all stages in the system life-cycle to enable the designers to deploy, extend, and maintain the systems through their life cycle.

Though SOA has been delivering better integration capabilities over traditional approaches, the real business value is achieved only through capturing stable business rules that can be reused across multiple applications. The SSM enables us to capture stable business rules encapsulated by the underlying theme into the core patterns. This makes the job of developers easier by providing common business rules out of the box and avoids repetition of costly mistakes.

Another major drawback of current design methodologies is the need for designing different systems for application scenarios dealing with similar issues. The inability to reuse parts of the architectures and the constant need to redesign systems to accommodate new changes escalates the cost of building and maintaining these systems. The pattern language has used the SSM methodology to take advantage of its stable core

models. This gives designers the distinct advantage of retaining the core pattern stable over changes to the system. The extensions or changes to the individual applications are supported through IOs that can be plugged into the core pattern to suit specific application scenarios. Customizing the core pattern for an application scenario can be further streamlined for the developers by using a hook tool (Fayad, Schmidt, and Johnson, 1998). The use of a hook tool and hooks make it easier for domain experts to design and plug IOs into the pattern, making the development and deployment process systematic and easy to manage.

In conclusion, the thesis PL for SOA provides a roadmap for building more stable business solutions in business prospects and framework to derive different architectures from the given knowledge map for supporting different domains. It has provided the hooks for the hook facility to generate a number of applications from different domains reducing repetitive work. A number of architectures can be derived from the KM and generate multiple applications using the hook facility to extend the core model. SSM has played a major role in developing all these patterns, and with proven usefulness designers can expect the SSM to play a major role in the software development lifecycle, providing elegant solutions in architecting systems.

REFERENCES

- Adler, M. (Jun, 2004). Component business modeling - mapping the way in insurance. *IBM Building an edge magazine*, 5(6).
- Allen, R., and Garlan, D. (1992). A formal approach to software architectures. *Proceedings of IFIP*. Sep, 1992.
- Baresi, L., Heckel, R., Thone, S., and Varro, D. (2003). Modeling and validation of service oriented architectures: Application vs. style. *Proceeding of ESEC/FSE*. 2003.
- Barry and Associates. (2004). Service-oriented architecture (SOA) definition. Retrieved on Jan, 2005, from Service architecture Web site: http://www.service-architecture.com/web-services/articles/service-oriented_architecture_soa_definition.html
- Bass, L., Clements, P., and Kazman, R. (2003). *Software Architecture in Practice*. 2nd ed. Addison-Wesley.
- Carnegie Mellon Software Engineering Institute, (2001). How do you define software architecture. Retrieved Nov, 2004 from: <http://www.sei.cmu.edu/architecture/definitions.html#Classic>
- Carnegie Mellon Software Engineering Institute, (2000). Product line systems program. Retrieved Nov, 2004 from: <http://www.sei.cmu.edu/productlines/index.html>
- Channabasavaiah, K., Holley, K., and Tuggle, E. (2004). The case for SOA. *Loosely Coupled*. Retrieved Dec, 2004 from Loosely Coupled Web site: <http://www.looselycoupled.com/opinion/2004/chann-case-infr0927.html>
- Colan, M. (2004). Service-oriented architecture expands the vision of web services: Part 1. Retrieved on Jan, 2005 from IBM developer works Web site: <http://www-128.ibm.com/developerworks/library/ws-soaintro.html>
- Coplien, O. J. and Schmidt, D. (2002). *Pattern languages of program design*. Addison-Wesley Professional. 2002.
- Datz, T. (Jan 15, 2004a). What you need to know about service-oriented architecture. *CIO Magazine*, Retrieved Nov , 2004, from <http://www.cio.com/archive/011504/soa.html>

- Datz, T. (Jan 15, 2004b). What you need to know about service-oriented architecture. *CIO Magazine*.
- Di, L. (2004). Distributed geospatial information services-architectures, standards, and research issues. 20th ISPRS Congress. July, 2004. Istanbul, Turkey.
- Endrei, M. (2004). *Service oriented architecture and web services*. IBM Redbooks Series. 2004.
- Fayad, M. E., and Pradeep, R. (2003). Evaluation Analysis Pattern. 8th Stable Patterns Workshop, 2003.
- Fayad, M. E. (2002a). Type oriented paradigm (TOP). Lecture notes for, *Software Design Methodologies*. Aug, 2002.
- Fayad, M. E. (2002b). Accomplishing software stability. *Communications of the ACM*, 45(1). 2002.
- Fayad, M. E. (2002c). How to deal with software stability. *Communications of the ACM*, 45(4). Apr, 2002.
- Fayad, M. E., Johnson, R. E., and Schmidt, D. C. (1999). Reusing hooks. *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. John Wiley & Sons. Sep, 1999.
- Fayad, M. E., Schmidt, D., and Johnson, R. (1998). Reusing hooks. *Building application frameworks: Object oriented foundations of framework design*. Wiley & Sons. New York: 1998.
- Fayad, M. E., and Altman, A. (2001). An introduction to software stability. *Communications of the ACM*, 44(9).
- Fayad, M. E., and Hamza, H. (2003a). Applying analysis patterns through analogy: Problems and solutions. *Proceedings of Tools USA*. 2003.
- Fayad, M. E., and Wu, S. (2002). Merging multiple conventional models in one stable model. *Communications of ACM*, 45(9). Sep, 2002.
- Fowler, M. (1997). *Analysis patterns: reusable object models*. Addison-Wesley, July, 1997.
- Fowler, M. (2004). "Patterns of Enterprise Application Architecture". Addison-Wesley Professional. Nov, 2004.

- Froehlich, G., Hoover, H. J., Liu, L., and Sorenson, P. G. (1997). Hooking into object-oriented application frameworks. *Proceeding of Int. Conference on Software Engineering*. Boston, Mass: May 17-23, 1997.
- Froehlich, G., Hoover, H. J., Liu, L., and Sorenson, P. G. (1998). Reusing application frameworks through hooks. *Object-oriented Application Frameworks*. John Wiley & Sons. Sep, 1998.
- Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994a). *Design patterns: Micro architectures for reusable object-oriented software*. Addison-Wesley. Reading, MA: 1994.
- Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994b). *Design patterns: Elements of reusable object-oriented software*. Addison Wesley Professional. 1994.
- Hamza, H. (2002). *A foundation for building stable analysis patterns*. M.S. Thesis, Lincoln, University of Nebraska.
- Hamza, H., and Fayad, M.E. (2002a). A pattern language for building stable analysis patterns. 9th Conference on Pattern Language of Programs (PLoP 02). Sep, 2002. Illinois.
- Hamza, H., and Fayad, M.E. (2002b). Model-based software reuse using stable analysis patterns. *Workshop on Model-based Software Reuse*, 16th ECOOP. Malaga, Spain: 2002.
- Hamza, H., and Fayad, M.E. (2003b). Stable analysis patterns: A true problem understanding with UML. *UML workshop*. 6th International Conference on UML. Oct 20, 2003.
- He, H. (2003). What is service oriented architecture? Retrieved on Mar, 2005, from Web services Web site: <http://webservices.xml.com/pub/a/ws/2003/09/30/soa.html>
- Hollander, T. (Fed, 2005). Introducing enterprise library. *Microsoft Patterns and Practices Digest*, 1(1).
- Hollingsworth, J. E., and Weide, B. W. (1995). Micro-architecture vs. macro-architecture. *Proceedings of the Seventh Annual Workshop on Software Reuse*, USA.
- Johnston, T. (2001). Stable business rules, meta data and late binding. *DM Direct Newsletter*. Retrieved Jan, 2005 from DM review Web site: http://www.dmreview.com/article_sub.cfm?articleId=3066

- Judy, K. (2000). Architectural evaluation for product line. *Workshop on Evaluating Software Architectural Solutions*, UC Irvine.
- Kidd, R. (Jul, 2004). SOA blueprints: An executive summary. Retrieved Dec, 2004, from SOA Blue Prints consortium Web site: <http://www.soacenter.com/>
- Larman, C. (2004). Applying UML and patterns: An introduction to object-oriented analysis and design and iterative development, 3rd ed. Prentice Hall Publications. Oct, 2004.
- Mahmoud, Q. H. (2003). Service-oriented architecture (SOA) and web services: The road to enterprise application integration (EAI). Retrieved Apr, 2005, from Sun Web site: <http://java.sun.com/developer/technicalArticles/WebServices/soa/>
- McNatt W., and Bieman, J. (2001). Coupling of design patterns: common practices and their benefits. *Proceedings of COMPSAC*. Oct, 2001.
- Microsoft Development Library [MSDN]. (2004a). Service oriented architecture. Retrieved Dec, 2004, from .Net architecture center Web site: <http://msdn.microsoft.com/>
- Microsoft Development Library [MSDN]. (2004b). Deployment patterns. *Microsoft Patterns & Practices*. 2004.
- Morris, S. (2005). Software futures: architecture. Retrieved from Informit Web site: <http://infomit.com>
- O'Toole, A. (2004). Service-oriented architecture: the right side up. Retrieved Nov. 14, 2004, from Cape Clear Web site: <http://www.capeclear.com/technology/clearthinking/rightside.shtml>
- Rich, R. (2004). Service delivery platforms: Moving toward a software-defined communications experience. *Communications Network Infrastructure*. Nov, 2004.
- Rogers, W. P. (2001). Encapsulation is not information hiding. *Java World Journal*. May, 2001.
- Rudzki J., and Hammouda, I. (2001). Pattern Classification. *Course Report*. Retrieved Feb, 2005 from Tampere University of Technology Web site: <http://www.cs.tut.fi/~kk/webstuff/PatternClassificationKalvot.pdf>
- SOA resource center, (2004). Understanding service oriented architecture. Retrieved Dec, 2004, from BEA Systems Web site: <http://soa.bea.com/>

- Saleh, T. (Jun, 2004). Component business modeling in retail banking. *IBM Building an edge magazine*, 5(6).
- Schulte, R. W. (1996). 'Service oriented' architectures: Part 2. *SSA Research Note SPA-401-069*. Apr 12, 1996.
- Schulte, R. W., and Natis, Y. V. (1996). 'Service oriented' architectures: Part 1. *SSA Research Note SPA-401-068*. Apr 12, 1996.
- Soley, R. (Nov, 2000). *Model driven architecture*. Object Management Group (OMG) White Paper.
- The Business Rules Group. (2003). Defining business rules. what are they really? Retrieved on April, 2005 from Business rules group Web site: <http://www.businessrulesgroup.org/brghome.htm>
- Weblogic soa section. (2003). Retrieved Jan. 12, 2005, from Dev2Dev online Web site: <http://dev2dev.bea.com/technologies/soa/index.jsp>
- Wikipedia online. (2003). Aspect-oriented programming. Retrieved on Mar, 2004, from Wikipedia Web site: http://en.wikipedia.org/wiki/Aspect-oriented_programming
- Wolski, R., Spring, N., and Hayes, J. (1999). The network weather service: A distributed resource performance forecasting service for meta-computing. *Future Generation Computer Systems*, 15(2). July, 1999.
- Zimmer, W. (1995). Relationships between design patterns. *Pattern Languages of Program Design*. Addison-Wesley. Reading, MA: 1995, pp. 345-364.

APPENDIX A: PATTERN TEMPLATES

1. AnyInformationHiding: A Stable Design Pattern

1.1 Pattern Name: *AnyInformationHiding* Design Pattern

Information hiding is an important principle in managing information for any system. The purpose could be to protect sensitive data in the application from unauthorized access or to make systems more stable by hiding implementation details that may change. This pattern tries to generalize the use of the information hiding principle for managing information access, based on the sensitivity level attached.

1.2 Known As

This pattern is similar to the AnyEncapsulation and AnyAbstraction patterns. Though “*information hiding*,” “*encapsulation*,” and “*abstraction*” may sound similar and are used interchangeably by some people, they represent different underlying principles (Fayad and Altman, 2001). “*Encapsulation*” refers to the bundling of data with the methods that operate on that data. “*Abstraction*” refers to the use of interfaces and subclasses to present a class to its users; whereas, “*information hiding*” is used for hiding data or managing information to protect. Information hiding is used for hiding complex implementation details or internal details that can change to protect the user from change or the need to know complex details to use the system.

1.3 Context

Information is a crucial component for any system to work properly. Information should be easily accessible for the systems to function effectively and interoperate with other systems. On the other hand, information that is exposed improperly can lead to serious system compromises. In high-level business systems like banking and financial applications, keeping information safe is very important. The *AnyInformationHiding* design pattern deals with the issue of keeping the information hidden and secure while exposing the business methods required for conducting the required tasks.

Information hiding can also be used to accomplish simplicity by hiding the complex details from the simple users of complex systems, leading to simple and effective user interfaces. For instance, the complexity of focusing and adjusting the lighting conditions in a camera is hidden by the auto focusing mechanism, just exposing the stable point-and-shoot interface. This enables simple users also to use the cameras effectively without having the knowledge of professional photographers.

The *AnyInformationHiding* design pattern can be applied to various applications in diverse domains such as software systems, complex applications like certain military projects, and so on. This pattern can be applied to different applications for information hiding for various reasons like, keeping information secret, hiding complexity or protecting systems from change by hiding design components that are most likely to change. Application scenarios are given to illustrate this in Figure A.1.3 and Figure A.1.5.

1.4 Problem

The *AnyInformationHiding* design pattern covers the concept of information hiding in the context of many domains that are different in nature and functionality. Therefore, it makes sense to model a generic design pattern that can be applied to all these domains. However, the requirement to hide information for each domain may vary, based on the information to be maintained along with users of the system and other participants. Moreover, the information that needs to be hidden comes from different domains (Hamza and Fayad, 2003b). The fact that information hiding is used in different ways in all these domains makes it difficult to come up with a generic model.

Using the software stability model (SSM) (Fayad and Altman, 2001), this problem is solved and a generic model is provided for different domains. The model (Figure A.1.1) is shown in the solution section.

1.5 Constraints and Challenges

Challenges:

- The concept of hiding and the use of restricted interfaces for information hiding have to be conveyed in the pattern along with the core concept of information hiding.
- The classification of different parts of information into “*visible*” or “*hidden*” will depend on application-specific domain issues. The pattern has to be defined in a generic way to apply the domain-specific hiding requirements and the mechanisms to guard the information.

- Some application domains have very stringent information hiding requirements for sensitive information; the pattern has to be flexible enough to support these requirements.

Constraints:

- All the information maintained in a system should be classifiable into different sensitivity levels based on the nature of information.
- Information hiding should hide all the sensitive information in the system from inappropriate access.
- Information hiding should use *AnyInterface* to expose only the needed information and hide the sensitive information from *AnyActor*.
- *AnyActor* should be able to access *AnyClass* only through *AnyInterface* as defined by the rules of *AnyInformationHiding*.
- *AnyClass* implements *AnyInterface* as defined by *AnyInformationHiding* and extends *AnyImplementation*. All other information and methods of *AnyClass* not present in *AnyInterface* are not visible to *AnyActor* and are protected.

1.6 Solution

The stable pattern in Figure A.1.1 is a proposed solution for handling the issues in hiding information. This concentrates on providing a generic pattern that can be applied to any domain, leaving out the domain specific features (Hamza and Fayad, 2003b). Figure A.1.1 shows the diagram of the *AnyInformationHiding* design pattern.

1.7 Pattern Structure and Participants

The participants of the *AnyInformationHiding* design pattern are introduced below.

Classes:

***AnyInformationHiding*:** This class represents a BO of information hiding which hides AnyImplementation to show only AnyInterface. This class is responsible for extracting AnyInterface from AnyClass and hides sensitive information.

Figure A.1.1 *AnyInformationHiding* Design Pattern
(From Dr. M.E. Fayad's Pattern Repository)

Patterns:

- **Hiding:** This class represents the EBT, the concept of hiding data from users of data.
The information is classified into sensitive and visible categories.
AnyInformationHiding is used to hide sensitive or internal data.
- **AnyClass:** This class represents the main class holding the information to be hidden.
Some of this information needs to be hidden while some of it may be exposed through *AnyInterface* to *AnyActor*.
- **AnyInterface:** This class represents the interface extracted from *AnyClass* by *AnyInformationHiding*. It is only through this interface that *AnyActor* can use *AnyClass*. *AnyImplementation* implements this interface, and the details of the implementation are hidden from *AnyActor*.
- **AnyActor:** This class represents a user of the information and the *AnyClass*, which is only accessible through *AnyInterface*.
- **AnyImplementation:** This class represents an implementation of *AnyInterface*.
This is the workhorse behind *AnyInterface* and is hidden from *AnyActor* by *AnyInformationHiding* using *AnyInterface*.

1.8 Detailed Model

The Figure A.1.2 illustrates the detailed model for the pattern.

Figure A.1.2 Detailed Model for Information Hiding

1.9 Description of Class Diagram

The class diagram graphically shows all the classes and their relationships with other classes in a model. In the above class diagram, **Hiding** is the EBT of this model and offers methods to find sensitivity levels allowed for the requesting user, **AnyActor**. This in turn is used by the *AnyInformationHiding* BO to extract methods allowed at that sensitivity level to be exposed to **AnyActor** through **AnyInterface**.

The information is only accessible to AnyActor through AnyInterface, thereby restricting access to information from AnyClass. AnyImplementation represents the implementation of the methods exposed in the AnyInterface. AnyClass has the information to be hidden and managed by *AnyInformationHiding* and offers methods, based on the sensitivity level associated with them.

1.10 Consequences

The *AnyInformationHiding* design pattern satisfies its objective and provides a base pattern that is adaptable to applications in different domains.

Understandability: The *AnyInformationHiding* design pattern presents the concept of hiding and uses information hiding in an easily understandable fashion. It accomplishes this through the Hiding EBT, and by using information hiding with interfaces.

Adaptability: The *AnyInformationHiding* design pattern is generic enough to be applicable in multiple domains that require information hiding. This is illustrated through the use of simple interfaces to allow access only to selected methods of AnyClass

Stability: The concepts of hiding and information hiding are described in generic terms without using any domain-specific industrial objects. Hence, it will be stable over multiple domains. Applications in different domains can use these concepts by substituting IOs(industrial objects) specific to the application. Examples of domain-specific IOs may be mostly different interfaces to restrict access to classes holding sensitive information that needs to be hidden in a domain.

Extensibility: The pattern can be extended by plugging in the application-specific context classes like instances of *AnyClass*, *AnyInterface*, and *AnyActor* into the pattern. Hence, the system provides a high-level of extensibility to suit applications in various domains.

1.11 Usage Notes

The designer has to understand the information in the application domain and the users of information along with details like sensitivity level associated with information and users of it in that domain in order to successfully use the *AnyInformationHiding* pattern.

1.12 Applicability With Illustrated Examples

This section provides an example to illustrate the use of the *AnyInformationHiding* design pattern (Figure A.1.1) for hiding information.

1.12.1 Application #1: Information Hiding in Bank Accounts

Hiding sensitive information is a very important factor in financial applications like banking systems. This application illustrates the classification of bank account information into various security levels and allowing access only to some data to authorized users, based on sensitivity of the information. The information maintained by the *CheckingAccount* class is extracted into special interfaces like *AccountSummary* and *AccountOwnerDetails* to be accessible to actors like *AccountHolder* and *BankClerk*. In this example, the account holder and the bank clerk can both access *AccountSummary*,

but only the account owner can see certain account details like SSN, and pin through AccountOwnerDetails.

1.12.1.1 Model

The model for this application is provided in Figure A.1.3 below.

Figure A.1.3 Information Hiding in Bank Account

1.12.1.2 Sequence Diagram

Figure A.1.4 Sequence Diagram for Banking Account

1.12.1.3 Use Case: Information hiding in bank account

1. AnyActor(BankClerk) asks *AnyInformationHiding* for access to the AnyClass (Checking Account information).
2. *AnyInformationHiding* will ask the Hiding pattern to check the sensitivity level for AnyActor (BankClerk).
3. Hiding will return the sensitivity level allowed for AnyActor (BankClerk)
4. *AnyInformationHiding* will request the methods from AnyClass (CheckingAccount) for methods at that sensitivity level.
5. AnyClass will return the list of methods for sensitivity level.
6. *AnyInformationHiding* will construct an AnyInterface (AccountSummary) from those methods.

7. The implementation for AnyInterface (AccountSummary) is provided through AnyImplementation (CheckingAccountSummary).
8. AnyActor(BankClerk) will get access to AnyInterface(AccountSummary), and makes a call to AccountSummary to find the balance.
9. The method will be handed over to AnyImplementation(AccountSummary), which may hand it down further to AnyClass (CheckingAccount).
10. AnyClass(CheckingAccount) will return the balance, which will be presented to AnyActor (BankingClerk).

1.12.2 Application #2: Hiding Implementation Details for a Data Structure

Hiding implementation details from the users of a system is an important aspect in building systems that are easy to manage. This application uses information hiding to simply expose a standard interface to the users of the data structure. Instead of trying to hide sensitive information, this application is trying to separate out the users of the information from the implementation details of how that data is managed. This demonstrates other uses of information hiding in a system. The information maintained by an inventory system is like a list of products; a HashSet is exposed to the inventory manager through a standard interface of a set. This way when the implementation is changed to another solid class, the inventory manager will not be affected.

1.12.2.1 Model

The model for this application is provided in Figure A.1.5 below.

Figure A.1.5 Hiding Implementation Details

1.12.2.2 Sequence Diagram

Figure A.1.6 Sequence Diagram for Hiding Implementation Details

1.13 Related Patterns

The Bridge pattern (Gamma, Helm, Johnson, and Vlissides, 1994b) mainly deals with hiding implementation details from the abstraction of a class using specialized bridge implementations, whereas the *AnyInformationHiding* design pattern deals with hiding any sort of information along with implementation details and methodology. The Factory design pattern and Abstract factory design pattern (Gamma, Helm, Johnson, and Vlissides, 1994a; Fowler, 1997) deal with hiding the information about the creational logic for implementations, whereas the *AnyInformationHiding* pattern hides both the creational logic and the implementation details from the user.

1.14 Conclusion

This paper proposes a stable design pattern for information hiding. The pattern has been developed to represent information hiding using software stability principles. The core knowledge of hiding has been extracted from different domains and presented as EBTs and BOs. This forms the core model that is applicable across multiple domains. This core model can be extended with industrial objects to design applications suitable to specific domains. This is illustrated with an application using the core model to implement information hiding, demonstrating the applicability and usefulness of the model.

2. AnyType: A Stable Design Pattern

2.1 Pattern Name: *AnyType* Design Pattern

Type signifies the classification of entities in a problem domain. It could be the classification of entities into various data types in a class diagram or various data items in a database. The *AnyType* design pattern generalizes the concept of classifying entities into types in different domains.

2.2 Known As

This pattern can also be known as AnyKind.

2.3 Context

Type identifies the nature of any entity in terms of behavior and the data representation. The significance of type comes into play when designers try to classify entities in a problem space, thereby grouping the entities that share similar characteristics and behavior. The type has applicability in various fields such as information technology, data warehousing, and inventory systems. For example, in programming languages the data entities are classified into various data types. Another example of type is the classification of entries in a data warehouse into different types of records like orders, customers, and contracts. Another use case for types is the classification of items in an inventory system into various types like office supplies, electronics, and furniture.

Two application scenarios are given to illustrate this in Figure A.2.3 and Figure A.2.5 for different domains.

2.4 Problem

Classification of entities into types is very useful in designing solutions across multiple domains. Existing techniques are used to classify entities into types are specific to a particular domain and are not formalized. So, it is not straightforward to determine the types and the classifications of entities in a given problem space. To be useful across multiple domains the *AnyType* pattern has to be generalized for use as a base of any classification. At the same time the pattern should be as specific as possible to make it adaptable and easy to extend the concept and apply it to any specific domain.

2.5 Challenges and Constraints

Challenges

- One must recognize the core concept of a type and must convey it clearly in the pattern.
- Type and classification of entities is a common concept that can be applied across multiple domains. The pattern has to be generic enough to be useful in various domains.
- There could be domains where some things can belong to more than one type, when they satisfy the criteria for multiple types. Typical use cases in real life involve situations such as when a person can be classified as a software-engineer and a

tennis-player at the same time. Another important use case is multiple-inheritance in object-oriented programming languages. This pattern addresses such scenarios by classifying some things into multiple type categories and types. However, this can increase the complexity of representing type relationships in a domain. In object-oriented terms the multiple types anything belongs to could be represented in terms of multiple-inheritance between anything and multiple types.

- As there are multiple levels of classification in the pattern from type category, type and further subtypes to that type, it gives the designer different levels of granularity. On the other hand deciding how many type categories and types should be used can have profound impact on the domain.
- As the classification concept is widely used, the pattern should support diverse use cases. Some domains may have complex criteria, based on a combination of characteristics of things. The pattern should allow such complex criteria.

Constraints

- A type category can be used to classify things in an application domain into different groups with common characteristics identified for that type category.
- A type category can have one or more types. This way things that belong to the same type category can be further classified into multiple types.
- Type can be defined as having one or more characteristics. When classifying anything into a type, it must meet all the criteria/characteristics defined for that specific type.

- Each thing in a type category can always be further classified into a type within that type category, based on some common characteristics.
- One or more things could be classified into the same type provided all of them satisfy the criteria for that type. However, the reverse may not always be true. Even when anything satisfies criteria for more than one type, it could be classified into the type with most criteria in certain domains.
- Each thing possesses one or more characteristics. This is what defines/influences the classification of it. Multiple parties can have none/some/all characteristics in common, depending upon the individual case. Based upon the level of commonality between them, they can be classified into the same type category and further into a type within that type category.
- Not all characteristics of anything have to be used for classifying it into a type category or a type. Based on the criteria for the type category, not all characteristics of anything have to be used to classify it. But, at least one of the characteristics of it is required for the classification.
- Based on the complexity of a type, it can be further classified into subtypes.
- The characteristics of any type can be extracted into one or more interfaces that are supported by that type. Anything that belongs to a type would be implementing all the interfaces identified for that type.

2.6 Solution

The Pattern presented in Figure A.2.1 is a proposed solution for the *AnyType* design pattern.

Figure A.2.1 AnyType Design Pattern
(From Dr. M.E. Fayad's Pattern Repository)

2.7 Pattern Structure and Participants

Classes:

AnyType: This class represents a type with specific properties and operations defined on it. Everything in the problem space will be eventually mapped into one of the types.

Patterns:

Classification: This class represents the concept of classification of Anything into a specific type. It is modeled as an EBT and conveys the core concept of classifying entities into type categories and types.

AnyInterface: This class represents the behavioral aspects of a type. This embodies the operations supported by a conceptual type. Each type can expose one or more interfaces, based on the properties of the type.

Anything: This class represents an entity that plays a role in the application domain. Everything has certain properties and characteristics that can be quantified and verified against any criteria.

AnyTypeCategory: This class represents a group of types that are related and the criteria that can be applied against one or more properties of Anything. Based on the properties, Anything would either pass or fail the given criteria, and be accepted or rejected into that AnyTypeCategory. Anything is further mapped into a certain type, based on whether it satisfies a combination of one or more criteria.

2.8 Detailed Model

The Figure A.2.2 presents the detailed model for this pattern.

Figure A.2.2 Detailed Model for *AnyType* Design Pattern

2.9 Description of Class Diagram

In the above class diagram, **Classification** is the EBT of the model, which uses **AnyTypeCategory** for qualifying **Anything** first into a type category, based on its attributes and behavior, and further qualify into *AnyType* of that category.

AnyTypeCategory has multiple types, which can be further fine grained into subtypes.

Each of the types is represented by one or more *AnyInterface* that describes the behavior for that type.

2.10 Consequences

The *AnyType* design pattern satisfies its objective and provides a base pattern that is adaptable to applications in different domains. The main benefits and usage notes for this design pattern are outlined below.

Understandability: The *AnyType* design pattern presents the type concept in an easily understandable fashion. It accomplishes this through the EBT, the concept of Classification.

Adaptability: The *AnyType* design pattern is generic enough to be applicable in multiple domains that use types.

Stability: The concept of type is described in generic terms, without using any domain-specific industrial objects. Hence it will be stable over time. Applications in different domains can use the concepts by substituting IOs(industrial objects) specific to the application.

Extensibility: The pattern can be extended, by plugging in the application-specific context classes like type category and types with different characteristics into the pattern. Hence, the system provides high-level of extensibility to suit applications in various domains.

2.11 Usage Notes

The designer has to understand the types and the granularity and criteria to classify things in that domain, for successfully using the *AnyType* pattern.

2.12 Applicability With Illustrated Examples

This section provides an example for illustrating the applicability of *AnyType* (Figure A.2.1) design pattern.

2.12.1 Application #1: Classification of Birth Date

When collecting patient data for a data warehouse application, the designer needs to classify the gathered data items into various type categories and types for qualifying their behavior. This application illustrates the Classification EBT qualifying the given patient birth date value into a *PersonalRecordType* category and into date data type. A standard interface for dates has been further derived from this type, which supports operations on date like *calculateAge()*, which calculates the years since the said date happened.

2.12.1.1 The Model

Figure A.2.3 Classification of BirthDate Example

2.12.1.2. Use Case: Classification of BirthDate

1. Anything(PatientBirthDate) is given to the EBT Classification for classifying.
2. Classification calls the AnyTypeCategory (PersonalRecordTypes) to qualify the value.
3. AnyTypeCategory(PersonalRecordTypes) qualifies the Anything(PatientBirthDate) as a personal record type.
4. AnyTypeCategory(PersonalRecordTypes) checks with AnyType(DateType) to see if the value qualifies.
5. The Anything(PatientBirthDate) is qualified as an AnyType(DateType).

6. AnyTypeCategory(PersonalRecordTypes) will extract AnyInterface(DataInterface) from the AnyType(DateType).

2.12.1.3 Sequence Diagram

Figure A.2.4 Sequence Diagram for BirthDate Classification

2.12.2 Application #2: Classifying an Item in Inventory (Laser Printer)

The second example illustrates the classification of products in an online storefront. The example discussed here is about classifying a specific model of HP laser jet printer into a peripheral devices category, and more specifically as a printer. The qualification used for that purpose is to check if it is a peripheral device, and further if it can print. This further extracts a standard interface of the printer, which has well established methods like print() and cancel().

2.12.2.1 The Model

Figure A.2.5 Classification of Laser Printer Example

2.12.2.2 Use Case: Classification of Laser Printer

1. Anything(HPLJ8000) is passed on to Classification .
2. Classification will request AnyTypeCategory (PeripheralTypes) to qualify Anything (HPLJ8000)
3. AnyTypeCategory(PeripheralTypes)will validate that Anything(HPLJ8000) is a peripheral device.
4. AnyTypeCategory(PeripheralTypes) will request *AnyType(Printer)* to validate Anything(HPLJ8000) for that type
5. *AnyType (Printer)* will pass Anything (HPLJ8000) as a valid printer type

6. *AnyType(Printer)* will extract *AnyInterfaces(PrinterInterface)* and return to the *AnyTypecategory(PrinterTypes)*
7. *AnyTypecategory(PrinterTypes)* will pass on *AnyInterface(PrinterInterface)* to *Anything(HPLJ8000)*.

2.12.2.3 Sequence Diagram

Figure A.2.6 Sequence Diagram for Laser Printer Classification

2.13 Related Patterns

Type is a common concept used in programming languages. Though it has been used extensively in most of the programming languages the type of an entity is always decided manually by the programmer. Although there are some rules and criteria governing how to guesstimate the type of an entity, it was neither formalized nor enforced by the compiler/runtime.

Jakub Rudzki and Imed Hammouda in *Pattern Classification* (Rudzki and Hammouda, 2001) discuss the concept of classifying the patterns themselves, based on the work of W. Zimmer (1995). But it only deals with classifying the patterns into various types, based on some manually applied criteria. The *AnyType* design pattern remedies this by formalizing the classification of things into types using the attributes and behavior of entities to be classified in any domain.

2.14 Conclusion

This paper presents the *AnyType* stable design pattern. Software stability concepts have been used to design the model to be applicable across different domains. Thus, the core model can be extended to suit applications from different domains that use type. The pattern makes it easy to design systems using the concept of type by simply writing industrial objects specific to that domain around the core model. It also makes it easier to understand the concept of type by presenting the core knowledge around type in a clear and easy to understand fashion. The applicability of the pattern has been illustrated using two applications from different domains.

3. AnyService: A Stable Design Pattern

3.1 Pattern name: *AnyService* Stable Design Pattern

The *AnyService* design pattern deals with the concept of service, computation and the interactions between various entities involved with services. The concept of computation and using a service to fulfill some functionality is used in multiple domains, although each of these domains uses different services and the parties dealing with services are unique to the domain. The *AnyService* design pattern makes it easier to capture underlying concept of service and related interactions in different application domains in a generic way using the software stability model. The pattern presents a generic model for services that can be extended to different domains, by capturing the core knowledge of services and assistance.

3.2 Known As

This pattern can also be known as *AnyAssistance* and *AnyAction*.

3.3 Context

Service is a well-defined unit of work done by a service provider to achieve desired end results for the service consumer (Di, 2004; He, 2003). Service has become a well-known concept with the increasing popularity of Service-Oriented Architectures. The concept of service is applicable in various domains such as information technology, financial systems, and health services. For instance, in the financial industry, examples of services include credit card processing, portfolio maintenance, etc. In information

systems, examples of services include providing security services like user authentication, encryption, website verification or other computational services like scientific calculations. Other examples of services in various fields include reservation services in travel, recreation industry, and diagnosis and treatment services in healthcare industry.

3.4 Problem

Due to wide applicability of services, it would be helpful to design a pattern that can be used across various service domains. To achieve this, the pattern aims at capturing the core concept of a service. The problem at hand is to bring out a stable pattern that represents the common concerns behind services, and which is easily adaptable and easily extensible to any kind of service across multiple domains. To be useful as a basis for designing applications in multiple domains, the extraction and definition of a service has to be formalized and generalized to be applicable across multiple domains. The pattern should make it easy to understand the concept of a service and to extract a service from the behaviors of an entity.

3.5 Constraints and Challenges

Constraints:

- A service is a well-defined unit of work that can be described in definitive terms.
- A service is triggered by a computation requirement as part of performing any functionality. This way computation signifies the core need for *AnyService*.

- A service can use one or more other services to fulfill its functionality. This is accomplished through *AnySequence*, which orchestrates the performances of services and shuttling of results between them.
- One or more *AnyClass* can offer a service. The specific class that would perform the service will depend on the *AnySequence* that is using the service.
- *AnyClass* can use zero or more *AnySequence* instances to perform any functionality. This is accomplished by adding the services of the class to *AnySequence* to orchestrate the execution.
- *AnyService* can have one or more signatures that indicate how that service can be requested and results returned from that service. Each of the signatures may differ in the number and type of arguments it takes, but the functionality provided by each of the signatures will be the same -- the service.
- *AnyAlgorithm* summarizes how the service is provided and represents the functionality provided by *AnyService*. The execution of this *AnyAlgorithm* will take some time, based on the inputs captured by *AnyPeriod*.
- *AnyService* uses *AnyType* to classify the arguments, the results of its execution and the many signatures through which it is available.
- *AnyService* can be accessed through *AnyMedia*. This represents the media and the service port for accessing service for the consumers.

Challenges:

- Service is a generic concept applicable in a variety of application domains that may have very little in common. For example, credit-card processing, automotive diagnosis, and reservation systems.
- The common concepts underlying service need to be captured and conveyed in an easily understandable way through the pattern.
- The design pattern should cater to different types of services used in various domains. For example, the pattern should be applicable for services where there is a payment component in financial services, as well as services without any payment component like a book loan service at a city library. It should also cater to services that do and do not require identification by the service provider.
- A service will execute in a specific medium, but the pattern should allow for different media types in order to support multiple services.
- The service may have more than one signature through which it could be accessed. This is because the input can be provided in different forms to convey similar information as needed for service execution. Hence, the pattern should accommodate the use of multiple signatures for the same service.

3.6 Solution

The solution model for this pattern is presented in Figure A.3.1.

Figure A.3.1 *AnyService* Design Pattern
(From Dr. M.E. Fayad's Pattern Repository)

3.7 Description of Class Diagram

The class diagram above graphically represents all the classes and their inter-relationships for the *AnyService* design pattern. In the above class diagram, *Computation* is the EBT—the underlying theme that triggers *AnyService* to function. *AnyService* is used to access the functionality of *AnyAlgorithm*. *AnyPeriod* represents the unit of time taken to execute an invocation of *AnyAlgorithm*. *AnyClass* offers one or more services and can string them together into *AnySequence*. *AnyService* can be delivered through *AnyMedia*

and can be classified with *AnyType* and *AnySignature*. The specific service functionality can be chosen by using the corresponding *AnySignature*.

3.8 Pattern Structure and Participants

Classes:

- ***AnyService***: This represents the service itself. The *AnyService* pattern is associated with the enduring business theme of Computation. It holds all the details of the service such as identity, service description, and the signatures it supports. It is offered by *AnyClass* and can participate in *AnySequence*.
AnyAlgorithm represents the functionality provided by the service.

Patterns:

- **Computation**: This represents the enduring business theme of this pattern. This EBT holds computation details such as the type and help provided along with methods to find matching services that can provide the computation. This is used by any entity that needs to use *AnyService*.
- ***AnySignature***: This represents a signature of the service. *AnySignature* offers a way of accessing the service along with details like the input and output parameters. *AnyService* can be accessed using multiple *AnySignature* classes.

- **AnyAlgorithm:** This represents the activity that the service implements for providing the functionality of the service. *AnyAlgorithm* is an important part of *AnyService* as it embodies the core functionality of the service.
- **AnyClass:** This represents the entity providing the service for others. *AnyClass* can provide one or more services. As part of performing its duties *AnyClass* can use one or more services as orchestrated by *AnySequence*.
- **AnySequence:** This represents the steps needed to perform any functionality. This may include invoking one or more services along with the details of how the results from one service are used for invoking other services. *AnySequence* orchestrates the invocation and funneling of results between various services to perform a unique functionality for *AnyClass*.
- **AnyType:** This represents the type of the functionality provided by *AnyService* along with the types of results for the service. The *AnyType* will classify the signatures of the service in terms of the results provided by the various signatures.
- **AnyMedia:** This represents the medium in which the service executes its functionality. This may include the media through which the service can be accessed for performing its functionality.
- **AnyPeriod:** This represents the time it takes to get the results from a service. This includes the time it takes for *AnyAlgorithm* to execute as well as the time it takes to execute all the actions in *AnySequence*.

3.9 CRC- Cards

Computation (Performer)		
Responsibility	Collaboration	
Performs the computation	Clients	Services
	AnyService	perform()

AnyType(Classifier)		
Responsibility	Collaboration	
Classifies signatures	Clients	Services
	AnyService AnySignature	typeName() classifySignature() classifyService()

AnyClass (Implementer)		
Responsibility	Collaboration	
Implements the functionality possibly using other services	Clients	Services
	AnyService AnySequence	addService() listServices()

AnySequence (Orchestrator)		
Responsibility	Collaboration	
Orchestrates the execution and flow of parameters between services	Clients	Services
	AnyClass AnyPeriod AnyMedia AnyService	addService() execute() listServices() findParaMap()

AnyMedia (Medium)		
Responsibility	Collaboration	
Facilitates a medium to access the service	Clients	Services
	AnySequence AnyService	identifyMedia() findMediaType()

AnyService (Service Provider)		
Responsibility	Collaboration	
Provides service	Clients	Services
	Computation AnyAlgorithm AnyType AnyClass AnySignature AnySequence	findReturnType() describeService() listSignatures()

AnySignature (Describer)		
Responsibility	Collaboration	
Describes service details along with arguments and return values	Clients	Services
	AnyType AnyService AnyAlgorithm AnySequence	addArgument() findMethodName() classifyReturnType() describeSignature()

AnyPeriod (Timekeeper)		
Responsibility	Collaboration	
Keeps track of time taken to accomplish activities	Clients	Services
	AnyAlgorithm AnySequence	calculateDuration() markStartTime() markEndTime()

AnyAlgorithm (Worker)		
Responsibility	Collaboration	
Encapsulates the work required to perform a service	Clients	Services
	AnyService AnyPeriod	performTask() identifyAlgorithm() findArguments()

3.10 Detailed Model

The detailed model for this pattern is presented in Figure A.3.2.

Figure A.3.2 Detailed Model for *AnyService* Design Pattern

3.11 Consequences

The *AnyService* design pattern satisfies its objective by providing a core pattern that is adaptable to all kinds of services. This enables the design and extraction of services from systems in a systematic way. The advantages and warnings of this pattern are outlined below:

Advantages:

- The *AnyService* pattern captures the core concept of service and makes it easy to understand components of a service like signatures and algorithms. The core knowledge is captured in the Computation EBT and highlights the underlying principle of a service.
- The *AnyService* pattern is designed independent of any specific service application and is generic enough to fit any application domain. Hence, the pattern is easily adaptable. This is achieved by extending the business objects (BOs) of the pattern as the application specific industrial objects (IOs).
- As the pattern itself does not depend on any application specific IOs, it is application neutral and does not change from application to application. The pattern is stable over time. For instance, the same core pattern can be used to design applications in the financial as well as other domains. This is demonstrated in the following sections and illustrated with examples.

- Applications can be designed to be useful in multiple domains by using this design pattern. This promotes high-level of reusability of software components and models. For instance, a service can be reused multiple times as part of performing the functionality of other services or classes.
- As the core components of the pattern do not change between applications, they can be designed to support high scalability. This is demonstrated by the use of BOs *AnySequence*, *AnyService*, and *AnyClass* in the section on examples.
- The pattern supports great flexibility in design to accommodate different scenarios where services are applicable. This is evident from its support of devising services in different types, reusing services and sequences to orchestrate the execution of services.

3.12 Usage Warning

Since the service has such wide usage scenarios in different categories and domains, it is difficult to model all the details of a service like the specifics of service security. For example, restricting service availability to specific service consumers and encryption of service requests and responses.

3.13 Applicability with Illustrated Examples

This section illustrates the applicability of the *AnyService* (Figure A.3.1) design pattern using two examples from different application domains.

3.13.1 Application #1: Stock Quote Service in Multiple Currencies

Getting stock quotes for trading is a very important and common service used in brokerage systems. The service is generally provided by the stock exchange companies or through network companies that provide the feed to various brokerage firms. Although most stock-quote systems deliver the stock quotes in local currencies of the markets, it would be nice to be able to get the stock quotes in different currencies, especially in light of the growing number of global investors. This example illustrates a sample class that combines the services of a stock-quote service along with that of a foreign exchange rate service to give stock quotes in different currencies. This application illustrates the use of *AnySequence* to orchestrate the execution of these two services and shuttling of results between them. Both these services use their own *AnyAlgorithm* to perform their specific functionality. The *IO ForexStockQuote* provides the combined service, which is an extension of *AnyClass BO*.

3.13.1.1 The Model

Figure A.3.3 Forex Stock Quote Service Example

3.13.1.2 Sequence Diagram

Figure A.3.4 Sequence Diagram for Forex Stock Quote Service Example

3.13.2 Application #2: Book Borrowing Service

The most common task performed by patrons of a library is to search for a book in the catalog and see whether it is available. The book borrowing service combines these two tasks to perform the job of finding a book that is available for borrowing. The first service locates books by a specific author or title. This service has two signatures, one

that takes the author name and another that takes part of title. Both these signatures will find books and return records with details like serial number. The second service takes the serial numbers of a set of books and returns the availability of these books. The class BookAvailability will use a sequence to orchestrate the above two services to find books that match the criteria along with their availability status. As can be inferred from the following model, the core pattern has not changed; only new IOs are added to it to match this specific example.

3.13.2.1 The Model

Figure A.3.5 Book Borrowing Service Example

3.13.2.2 Sequence Diagram:

Figure A.3.6 Sequence Diagram for Book Borrowing Service Example

3.14 Related Patterns and Measurability

Service is a common concept used in day-to-day interactions in many domains. In software systems, the earlier component models like COM+, DCOM, RMI, and CORBA are precursors to the later concept of web services. Although all the earlier models try to achieve modularity-platform-neutral way to access functionality from other components, they have the disadvantages of locking into specific technology aspects like programming language, the nonstandard communication mechanisms between the service provider and consumer. The use of protocols for defining the interaction between a client and server in client-server systems is analogous to the service provider and consumer, although many components like service definition and location transparency are not supported. There have been many design patterns around specific services like “*The network weather service*” by Rich W and Neil T. (Wolski, Spring, and Hayes, 1999) but they focus on a specific service rather than the general concept of service and the associated entities. *AnyService* design pattern focuses on the definition of services and the interactions between participating components in a systematic way in different domains.

4. AnyCoupling: A Stable Design Pattern

4.1 Pattern name: *AnyCoupling* Stable Design Pattern

The *AnyCoupling* design pattern deals with the concept of coupling, entities participating in the coupling, the constraints for the coupling and other related interactions. Coupling indicates the relationship and dependency between different modules of a system. Coupling serves to provide connection between two modules that are compatible with each other. The dependency could indicate that changes in one module may affect other modules using services from the changed module. Whether the change affects the system, and by how much, is decided by how tightly the modules are coupled in the system.

4.2 Know As

This pattern can be also know as AnyLink.

4.3 Context

A typical system with reasonable complexity is normally comprised of multiple modules that interact with each other to accomplish the functionalities of the system. Coupling is defined as the relationship between various modules of a system. It has applicability in various fields like information technology, transportation, electronics, and physics. For example, in computer programming coupling indicates the degree to which one module depends on other modules in the system. In physics two systems are coupled

when they are interacting with each other, such as vibratory systems and magnetic fields. Another example of coupling is in transportation systems such as coupling between railway wagons, and hose coupling between two fire hoses. In all these contexts coupling may have different uses while the theme of coupling is common across these domains.

4.4 Problem

Coupling is an important characteristic of a system because it measures how multiple modules in a system depend on each other. Though it is a very commonly occurring theme in systems, there have not been formal methods to measure/qualify its characteristics. Using it as a guideline to understand the system behavior is difficult. For example, in computer systems low coupling means one module does not have to be concerned with the internal details of other modules it interacts with. Low coupling is a sign of well-structured systems. However, tight coupling is needed for efficient implementation of systems. To make proper design decisions as to the amount of coupling for a system, one needs to have standard procedures to identify the coupling with guidelines. The *AnyCoupling* design pattern generalizes the concept of coupling across multiple domains to be used in a generic way.

4.5 Constraints and Challenges

Constraints:

- The core concept of coupling is to provide a connection between components of a system.

- The coupling has to establish a clear relation between two entities in a system. This may include details on how the entities are related along with the characteristics used in the relation.
- There should be at least two entities participating in a coupling relation. The relation between the two entities may be bi-directional or unidirectional. The coupling is unidirectional when only one of the entities depends on the other entity.
- The coupling should be measurable with criteria and can be classified into AnyDegreeOfDependency between the participating entities. This would mean that any changes to the other entities would require corresponding changes in the dependent entity.
- The participating entities in the coupling must satisfy the criteria established by AnyCriteria to work with each other and to participate in the coupling.

Challenges:

- The core concept of coupling and measuring dependencies between components has to be communicated in the pattern.
- Coupling of entities is a generic concept that is common across multiple domains. The pattern should be generic enough to be useful in all these domains by capturing the commonly occurring themes in coupling.
- Coupling between two entities can be characterized by measuring the degree of dependency and compatibility constraints between them. The pattern should have

mechanisms to capture these details about the couplings between various components.

- As the coupling and dependency relation between entities could be transitive, the pattern should make it easier to present the relationship between multiple components and their interdependencies easily.

4.6 Solution

The solution model for this pattern is presented in Figure A.4.1.

Figure A.4.1 AnyCoupling Design Pattern
(From Dr. M.E. Fayad's Pattern Repository)

4.7 Description of Class Diagram

The classes from the model along with their inter-relationships are represented in the above class diagram. The Measurement represents the EBT of this model, which represents a measure of dependency encapsulated by *AnyCoupling* relationship. AnyEntity represents the participants in the coupling in which the system is interested. The relationship between the entities is checked using the terms of AnyCriteria that checks the relationship between two or more entities. Two or more entities participate in *AnyCoupling*, which introduces a dependency between these AnyEntity encapsulated by AnyDegreeOfDependency. This enables the model to represent the coupling between entities in terms of their compatibility and their dependencies.

4.8 Pattern Structure and Participants

Classes:

***AnyCoupling*:** This represents the coupling relationship between two or more entities.

The *AnyCoupling* pattern is associated with the enduring business theme of Measurement that qualifies the degree of dependency between entities. It holds the coupling characteristics like the participating entities and the relationships.

Patterns:

Measurement: This represents the enduring business theme of this pattern. This encapsulates a measure of dependency between entities participating in *AnyCoupling*. This qualifies the relation between one or more AnyEntity instances.

AnyCriteria: This represents the criteria used to identify and measure the dependency between two or more entities participating in *AnyCoupling*. There can be multiple criteria, possibly independent of each other, can be used to identify a coupling relationship.

AnyEntity: This represents a participant in the *AnyCoupling*. An entity can participate in relationships with multiple other entities and exhibit different degrees of dependency as measured by the Measurement EBT.

AnyDegreeOfDependency: This stands as a representation for the degree of dependency between entities as qualified by the Measurement EBT.

4.9 Detailed Model

The detailed model for this pattern is presented in Figure A.4.2.

Figure A.4.2 Detailed Model for *AnyCoupling* Design Pattern

4.10 Consequences

The *AnyCoupling* pattern abstracts the common principles of coupling and serves as the base pattern for coupling adaptable to applications across multiple domains. The benefits and drawbacks of this pattern are explained in the following sections.

Benefits:

Understandability: The *AnyCoupling* design pattern presents the concept of coupling in an easily understandable fashion. It accomplishes this through the EBT, the concept of Measurement of dependency between multiple components of a system.

Adaptability: The *AnyCoupling* design pattern is generic enough to be applicable in multiple domains that use modular design of components with coupling between them.

Stability: The concept of coupling is described and modeled in generic terms without using any domain specific industrial objects by the *AnyCoupling* pattern. Hence, this pattern will be stable over time. Applications in different domains can use these concepts by substituting their own IOs (industrial objects) specific to the application.

Extensibility: The pattern can be extended by plugging in the application-specific context classes like entities and dependencies into the pattern. Hence, the system provides high-level of extensibility to suit applications in various domains.

4.11 Draw Backs

As explained in the challenges, the transitive relations between entities are not covered by the *AnyCoupling* pattern. This could mean that the *AnyCoupling* pattern may not capture all the dependencies between components in certain systems. This can be resolved by developing a notation system to calculate all transitive relations and the indirect dependencies they create in the system.

4.12 Applicability with Illustrated Examples

4.12.1 Application #1: Coupling for an Electronic Circuit

Any simple electronic circuit has numerous components interacting with each other in different ways. The relationship between these components can be electrical, mechanical or of another type such as sonic or optical. The following example models a simple electronic circuit board with IC-Slots that can be used to hook IC-Chips. The different criteria to consider would include pin-compatibility between an IC-Chip and IC-Slot. Another consideration may be the size-compatibility that determines if a chip width/depth would fit into slot. The criteria are represented in terms of driving current as required by each chip. The coupling for these components is both mechanical and electronic, which means that both these criteria need to be satisfied by the components to work properly.

4.12.1.1 The Model

The model for this application is presented in Figure A.4.3.

Figure A.4.3 *AnyCoupling* for Electronic Circuits

4.12.1.2 Sequence Diagram

The sequence diagram is presented in Figure A.4.4

Figure A.4.4 Sequence Diagram for Coupling in Electronic Circuits

4.12.2 Application #2: Coupling for Stock Feed Service

In communication systems, the correct working of two components depends on them talking in terms that both can interpret. This coupling is represented in the following example as the DataCoupling and CommunicationCoupling. In this example, the StockFeedService has communication coupling as it needs the communications with other components for it to work properly. Similarly the PortfolioService has DataCoupling as it needs the data feed. The criteria required for them to participate in the coupling are represented in the DataFormatCompatibility and ProtocolCompatibility both of which are crucial for the entities to work together.

4.12.2.1 The Model

The model for this application is presented in Figure A.4.5.

Figure A.4.5 *AnyCoupling* for *StockFeedService*

4.12.2.2 Sequence Diagram

The model for this application is presented in Figure A.4.5.

Figure A.4.6 Sequence Diagram for StockFeedService

4.13 Related Patterns and Measurability

Coupling is a concept well-known to system designers. There have been numerous papers discussing the effects of coupling on a system, but none that focus on measuring the coupling of a system. The design pattern “*Low-coupling design pattern*” (Larman, 2004) talks about how to reduce the coupling between the components of a system. In general, low coupling promotes low dependency, minimizes change impact and increases reuse of components. However it can cause unnecessary complexity in the system by introducing too many generic interfaces. Many of the design patterns aim to increase component reuse and high cohesion by reducing coupling. Some of these well-known

patterns are Singleton pattern, Mediator pattern, Factory pattern, and Creator pattern (Gamma, Helm, Johnson, and Vlissides, 1994b). McNatt and Biemann present the coupling between design patterns (McNatt and Bieman, 2001) in their paper and analyze the benefits and costs of pattern coupling. The *AnyCoupling* design pattern mainly focuses on mechanisms to bring out the dependencies and the coupling between components of a system in a systematic way.

APPENDIX B: SAMPLE CODE

Sample code segments for few of the patterns are listed here. In the interest of keeping things brief some of the obvious classes are not shown.

1. AnyInformationHiding Pattern

Code snippets from the *AnyInformationHiding* design pattern are presented here:

```
/* *****  
 *----- AnyInformationHiding -----  
 */  
  
/**  
 * Created On Jun 25, 2005  
 * Created By Haritha Kilaru  
 *  
 * Copy Right 2005  
 * Unauthorized use in any form is prohibited  
 * with out prior written permission of Author.  
 */  
package aih;  
  
import java.util.List;  
import java.util.Map;  
  
/**  
 * AnyInformationHiding Pattern  
 */  
public class AnyInformationHiding {  
    private List classes;  
    private Map interfaceMap;  
    private AnyClass clazz;  
    private Hiding hd;  
  
    private static AnyInformationHiding inst = null;  
  
    public AnyInformationHiding() {  
        hd = new Hiding();  
    }  
}
```

```

public AnyInterface accessInterface(AnyActor client) {
    if (clazz == null) {
        return null;
    }
    List methods = clazz.findMatchingMethods(client);
    return AnyInterface.constructInterface(clazz,
hd.calSensitivity(client.getRole()), methods);
}

public AnyImplementation findImplementation() {
    return null;
}

public void hideClassInformation(AnyClass clz) {
    this.clazz = clz;
    clazz.registerRoles(hd);
}

public int findSensitivityLevel(String role) {
    return hd.calSensitivity(role);
}

public void registerRole(String role, int sense) {
    hd.addRole(role, sense);
}
}

```

```

/*****
 *----- AnyInterface ----- *
 */
public class AnyInterface {
    private int visibilityLevel;
    private List clients;
    private List methods;
    private AnyClass implementor;

    public static AnyInterface constructInterface(
AnyClass mentor, int visibilityLevel, Collection methods)
    {
        AnyInterface iface = new AnyInterface();
        iface.implementor = mentor;
        iface.setMethods(methods);
        iface.setVisibilityLevel(visibilityLevel);
    }
}

```

```

        return iface;
    }

    public boolean allowClient(AnyActor user, AnyClass clz) {
        return (visibilityLevel <=
            clz.findSensitivityForRole(user.getRole()));
    }

    public List findAllowedClients() {
        return null;
    }

    public Object performOperation(String method, Map args) {
        return this.implementor.performOperation(method,
            args);
    }

    /**
     * @return Returns the methods.
     */
    public List getMethods() {
        return methods;
    }

    /**
     * @param methods The methods to set.
     */
    public void setMethods(Collection methods) {
        this.methods = new ArrayList(methods);
    }

    /**
     * @return Returns the visibilityLevel.
     */
    public int getVisibilityLevel() {
        return visibilityLevel;
    }

    /**
     * @param visibilityLevel The visibilityLevel to set.
     */
    public void setVisibilityLevel(int visibilityLevel) {
        this.visibilityLevel = visibilityLevel;
    }
}

```

```

/*****
 *----- AnyClassHook ----- *
 */
public class AnyClassHook extends AnyClass
{
    Map ioMap = new HashMap();

    public AnyClassHook()
    {
        super();
        ioMap.put("CheckingAccount", new CheckingAccount());

        ioMap.put("Weapons", new Weapons());
    }

    public Object getIo(String ioName) throws Exception
    {
        if(ioMap.containsKey(ioName))
        {
            return ioMap.get(ioName);
        }
        throw new Exception(" IO with name " + ioName
                               + " not found.");
    }

    public void removeIo(String ioName) throws Exception
    {
        if (ioMap.containsKey(ioName))
        {
            ioMap.remove(ioName);
        }
        throw new Exception(" IO with name " + ioName
                               + " not found.");
    }
}

```

```

/*****
*----- AnyClass ----- *
*/
public class AnyClass {
    private String name;
    private List attributes;
    private Map methodMap;
    protected AnyInformationHiding aih;

    public AnyClass() {
        attributes = new ArrayList();
        methodMap = new HashMap();
        //aih = AnyInformationHiding.getInstance();
        aih = new AnyInformationHiding();
    }

    //Subclass should over-ride this
    public Object performOperation(String method, Map args) {
        return null;
    }

    public AnyInformationHiding getClassProtector() {
        return aih;
    }

    public int sensitivityForMethod(String mName) {
        Integer sens = (Integer)methodMap.get(mName);
        if (sens != null) {
            return sens.intValue();
        }
        return -1;
    }

    public List findMatchingMethods(AnyActor act) {
        String role = act.getRole();
        int sense = findSensitivityForRole(role);
        List allowed = new ArrayList();
        for (Iterator methods =
            methodMap.keySet().iterator();
            methods.hasNext();) {
            String mName = (String) methods.next();
            if (this.sensitivityForMethod(mName) <= sense)
            {
                allowed.add(mName);
            }
        }
        return allowed;
    }
}

```

```

    }

    public int findSensitivityForRole(String role) {
        return aih.findSensitivityLevel(role);
    }

    public void addAttribute(String aname) {
        attributes.add(aname);
    }

    public List getAttributes() {
        return this.attributes;
    }

    public void addMethod(String mName, int sense) {
        methodMap.put(mName, new Integer(sense));
    }

    public Collection listMethods() {
        return methodMap.keySet();
    }

    /**
     * @return Returns the name.
     */
    public String getName() {
        return name;
    }

    /**
     * @param name The name to set.
     */
    public void setName(String name) {
        this.name = name;
    }

    /**
     * @param hd
     */
    public void registerRoles(Hiding hd) {
        //
    }

    /**
     * ##### Hook Methods #####
     */
    private static AnyClassHook hook;

    public static AnyClass getHook(String IO) {

```



```

        if (hook == null) {
            hook = new AnyClassHook();
        }
        try {
            return (AnyClass)hook.getIo(IO);
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
        return null;
    }

    public static void removeHook(String IO) {
        try {
            hook.removeIo(IO);
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}

```

2. AnyService Design Pattern

Code snippets from the *AnyService* design pattern are presented here:

```
/*
*****
*----- AnyService -----*
*/

/*
 * Created on May 23, 2005
 * Created by Haritha Kilaru
 */
package anyservice;

import java.lang.reflect.Field;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;

/**
 * @author Haritha Kilaru
 */
public class AnyService {
    private String name;
    private String description;
    private AnyAlgorithm algorithm;
    private Map signatures;

    public AnyService() {
        signatures = new HashMap();
        init();
    }

    public void init() {
    }

    public Object executeSignature(AnySignature sig,
                                   List args)
    {
        return new Computation().perform(algorithm,
                                           sig, args);
    }

    public String describeService() {

```

```

        return description;
    }

    public String findReturnType() {
        return getSignature().classifyReturnType();
    }

    public List listSignatures() {
        return new ArrayList(signatures.keySet());
    }

    /**
     */
    public AnyAlgorithm getAlgorithm() {
        return algorithm;
    }

    /**
     * @return
     */
    public String getName() {
        return name;
    }

    /**
     * @return Service
     */
    public AnySignature getSignature() {
        Iterator it = signatures.entrySet().iterator();
        if (it.hasNext()) {
            return (AnySignature)it.next();
        }
        return null;
    }

    public AnySignature getSignature(String name) {
        return (AnySignature)signatures.get(name);
    }

    /**
     * @param algorithm
     */
    public void setAlgorithm(AnyAlgorithm algorithm) {
        this.algorithm = algorithm;
    }

    /**
     * @param string
     */

```

```

public void setName(String string) {
    name = string;
}

/**
 * @param signature
 */
public void addSignature(AnySignature signature) {
    signatures.put(signature.findMethodName(),
        signature);
}

/**
 * @param string
 */
public void setDescription(String desc) {
    description = desc;
}

public String toString() {
    return name;
}

/**
 * ##### Hook Methods #####
 */
private static AnyServiceHook hook;

public static AnyService getHook(String IO) {
    if (hook == null) {
        hook = new AnyServiceHook();
    }
    if (IO == null) {
        return null;
    }
    try {
        String ioName = IO.toLowerCase();
        Field iof = hook.getClass().getField(ioName);
        if (iof == null) return null;
        return (AnyService)iof.get(hook);
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
    return null;
}
}

```

```

/*****
 *----- AnyServiceHook -----
 *
 */
package anyservice;

import anyservice.AnyService;
import library.BookStatusService;
import library.BookFinderService;
import stockquote.ForexService;
import stockquote.StockQuoteService;

public class AnyServiceHook extends AnyService
{
    public BookStatusService bookstatusservice;
    public BookFinderService bookfinderservice;
    public ForexService forexservice;
    public StockQuoteService stockquoteservice;

    public AnyServiceHook()
    {
        super();
        this.bookstatusservice = new BookStatusService();

        this.bookfinderservice = new BookFinderService();

        this.forexservice = new ForexService();

        this.stockquoteservice = new StockQuoteService();
    }
}

```

```

/*****
 *----- AnySequence ----- *
 */
public class AnySequence {
    private List services = new ArrayList();
    private List paraMap = new ArrayList();

    public void addService(AnyService ser, String pmap) {
        services.add(ser);
        paraMap.add(pmap);
    }
}

```

```

public String findParaMap(int index) {
    if (index > paraMap.size()) {
        return null;
    }
    return (String)paraMap.get(index);
}

public List listServices() {
    return services;
}

public Object execute(List args) {
    Iterator svcs = services.iterator();
    Iterator pmaps = paraMap.iterator();
    List myArgs = new ArrayList(args);
    Object ret = null;
    while(svcs.hasNext()) {
        AnyService svc = (AnyService)svcs.next();
        String pmap = (String)pmaps.next();

        StringTokenizer stok
            = new StringTokenizer(pmap, ",");
        while(stok.hasMoreElements()) {
            String tok = stok.nextToken();
            if (tok.length() > 1) {
                String numeral = tok.substring(1);
                int index = Integer.parseInt(numeral);
                if (tok.startsWith("O")
                    && args.size() > index) {
                    myArgs.add(index, args.get(index));
                } else {
                    myArgs.add(index, ret);
                }
            }
        }

        AnySignature sig = null;
        ret = svc.executeSignature(sig, myArgs);
        myArgs.clear();
        myArgs.add(ret);
    }
    return ret;
}
}

```

```

/*****
 *----- AnyClass ----- *
 */
public class AnyClass {
    protected String className;
    protected List services;
    protected AnySequence seq;

    public AnyClass() {
        init();
    }

    public void init() {
        services = new ArrayList();
        seq = new AnySequence();
        className = this.getClass().getName();
    }

    public void addService(AnyService ser, String pmap) {
        services.add(ser);
        seq.addService(ser, pmap);
    }

    public Iterator listServices() {
        return services.iterator();
    }

    public AnyService getService(String name) {
        Iterator it = services.iterator();
        while (it.hasNext()) {
            AnyService ser = (AnyService)it.next();
            if (name.equals(ser.getName())) {
                return ser;
            }
        }
        return null;
    }

    public Object getResult(AnyService ser, AnySignature sig,
                           List args) {
        if (ser != null) {
            Object res = ser.executeSignature(sig, args);
            if (res != null) {
                if (res instanceof Double) {
                    NumberFormat nf
                        = new DecimalFormat("0.000");
                    Double d = (Double)res;
                    return nf.format(d);
                } else {

```

```

        return res;
    }
    }
    return null;
}
return getResult(args);
}

public String getResult(List args) {
    return "";
}

/**
 * ##### Hook Methods #####
 */
private static AnyClassHook hook;

public static AnyClass getHook(String IO) {
    if (hook == null) {
        hook = new AnyClassHook();
    }
    try {
        return (AnyClass)hook.getIo(IO);
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
    return null;
}

public static void removeHook(String IO) {
    try {
        hook.removeIo(IO);
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
}

}

```


APPENDIX C: SAMPLE APPLICATIONS

Sample applications have been developed to demonstrate the applicability for some of the patterns from the pattern language. This involves implementing the core patterns and developing applications in different domains using these patterns. The following sections give the details for setting up the deployment environment for the sample applications and a few screen shots.

1. Setup Instructions

The application has been developed using the Java development environment and uses web-based user interface with JSPs. So, the following steps are required for preparing the systems for installing the sample applications.

1.1 Install JDK

The applications are developed in Java. The Java runtime environment needs to be installed to run these applications. Install Java Development Kit (JDK) 1.5 on the target system. The JDK can be downloaded from Sun's website at the following URL:

<http://java.sun.com/j2se/1.5.0/>

1.2 Install Tomcat

The applications use web-based user interfaces using Java Server Pages (JSPs). A JSP-compatible web server is needed to run them. Install Tomcat 5.5.9 on the target system. This uses the JDK already installed in step 1.1 above. The latest version of Tomcat server can be downloaded from Apache website at:

<http://jakarta.apache.org/tomcat/index.html>

1.3 Copy the Sample Applications

The sample applications are bundled as web archive (WAR) files. These need to be copied into the applications folder of the Tomcat server. For instance, if Tomcat is installed at [C:\SOA\jakarta-tomcat-5.5.9], then the WAR files need to be copied in to the webapps folder under Tomcat at [C:\SOA\jakarta-tomcat-5.5.9\webapps]. All the applications for a specific pattern are bundled into a single WAR file.

1.4 Copy the Sample Database Files

The applications use sample databases to store data for simulating application scenarios. The sample data is bundled into a single Microsoft Access database file. The file "ProjectDB.mdb" needs to be copied into a folder on the target system.

1.5 Setup ODBC Entry for Database

For the application to access the database, the ODBC system needs to be setup to create a data source using the sample database file above. This can be accomplished by

- a. On the windows system go to the “Control Panel” / “Administrative Tools” / “Data Sources (ODBC).”
- b. Here under the “System DSN” tab click the “Add” button to create a new data source.
- c. In the new dialog select “Microsoft Access Driver” and click “Finish.”
- d. In the next dialog enter the “Data Source Name” as “HarithaDB.”
- e. Enter some value for description.
- f. In the database section, click the “Select” button and choose the file copied in the step 1.4 above.
- g. Click “OK” to finish adding the new data source.

After this step the target system should have all the software needed for the applications. Now the target system is ready to run the applications.

2. Running the Sample Applications

The Tomcat server can be started by running a batch file (startup.bat) from the “bin” folder under your Tomcat server.

2.1 Applications for *AnyService* Pattern

The webapp for *AnyService* pattern implementation is AnyService.war. This has two sample applications.

- a. StockQuote: This application can be accessed using URL
<http://localhost:8080/AnyService/jsp/StockQuote.jsp>

3. Sample Screen Shots

3.1 Screen Shots for *AnyService* Pattern

The *AnyService* design pattern has been illustrated with two applications. The stock quote service application combines the functionalities of a stock quote service and a foreign exchange service to give a stock quote service in foreign currency.

i. Stock Quote Service

Figure C.1 Stock Quote Service

Mozilla Firefox

File Edit View Go Bookmarks Tools Help

http://localhost:8080/AnyService/j

Stock Quote Services

Service Type:	Foreign Exchange Service
Choose Currency:	Euro
Enter Amount:	1947.815	
Service Request:	Get the Forex value for	
	ServiceChoice	
Result:	1546.370	

Figure C.2 Foreign Exchange Service

Mozilla Firefox

File Edit View Go Bookmarks Tools Help

☐ http://localhost:8080/AnyService/jsp/StockQuote.jsp Go

Stock Quote Services

Service Type:	Stock Quote in Foreign Currency
Choose Stock Symbol:	DELL
Choose Currency:	Yen
<input type="button" value="ServiceChoice"/>		
Result:	The price for DELL in Yens is 4319.228	

Figure C.3 Stock Quote Service in Foreign Currency

ii. Library Service

The screenshot shows the Microsoft Access interface for a database named 'HarithaDB'. The 'Tables' list on the left includes 'authors' and 'books'. The 'authors' table is displayed in a datasheet view with columns 'ID' and 'name'. The 'books' table is also displayed in a datasheet view with columns 'ID', 'title', 'isbn', 'authid', and 'status'. The 'ID' column in the 'books' table is marked as an AutoNumber.

ID	name
1	J.D. Salinger
2	Mark Twain
3	Charles Dicker
4	Jane Austen

ID	title	isbn	authid	status
1	The Catcher in the Rye	0316769487	1	Available
2	Nine Stories	0316769509	1	Lost
3	Adventures of Huckleberry Finn	0520228383	2	Available
4	The Adventures of Tom Sawyer	0520235754	2	Barrowed
5	A Tale of Two Cities	0451526562	3	Available
6	Great Expectations	0486415864	3	Damaged
7	David Copperfield	0140434941	3	Available
8	The Pickwick Papers	0140436111	3	Barrowed
9	Oliver Twist	0612580036	3	Available
10	Persuasion	0553213105	4	Damaged
11	Pride and Prejudice	0553213105	4	Available
12	Emma	0553212737	4	Barrowed
13	Mansfield Park	0451526295	4	Available
*(AutoNumber)			0	

Figure C.4 Database Tables Used for Library Service

Figure C.5 Book Status Service

Figure C.6 Book Finder Service

Mozilla Firefox

File Edit View Go Bookmarks Tools Help

http://localhost:8080/AnyService/jsp/library.jsp

Sunnyvale Public Library

Catalog Services

Service Type: Find Book Details with Status

Choose Author: Mark Twain

ServiceChoice

	Title	Author	ISBN	Status
Result:	Adventures of Huckleberry Finn	Mark Twain	0520228383	Available
	The Adventures of Tom Sawyer	Mark Twain	0520235754	Barrowed

Figure C.7 Combined Book Status Service

3.2 Screen Shots for *AnyInformationHiding* Pattern

The *AnyInformationHiding* design pattern has been illustrated with two applications. The banking application illustrates the use of information hiding with a checking account as viewed by two actors, the account owner and a bank clerk. As can be seen, both of them get different services, based on their privileges. The military application deals with secure information about weapons that are available to personnel, based on their ranking and privileges. As can be seen from the screen shots, some of the powerful and highly sensitive weapons are only available to high ranking personnel.

i. Banking Application

Mozilla Firefox

File Edit View Go Bookmarks Tools Help

http://localhost:8080/AIH/jsp/BankAccount.jsp Go

Bank of America Higher Standards

Welcome to Bank Of America

Login As:	BankClerk
Sensitivity Level:	5
Choose Service:	<ul style="list-style-type: none">Get Owner NameGet Owner NameGet AddressGet Account NumberGet Customer SinceGet BalanceGet Allowed Credit

Figure C.8 Banking Application as Seen by Bank Clerk

Mozilla Firefox

File Edit View Go Bookmarks Tools Help

http://localhost:8080/AIH/jsp/BankAccount.jsp Go

Bank of America Higher Standards

Welcome to Bank Of America

Login As:	AccountOwner
Sensitivity Level:	9 :
Choose Service:	Get Social Security Number
<input type="button" value="ServiceChoice"/>		
Result:	123-45-6789	

Figure C.9 Banking Application as Seen by Account Owner

ii. Military Application

Mozilla Firefox

File Edit View Go Bookmarks Tools Help

http://localhost:8080/AIH/jsp/Military.jsp

Go

Weapons Command Central

Your Designation is:	Major
Sensitivity Level:	4 :
Choose Weapon:	<div>Get Mk82 General Purpose Bomb</div> <div>Get GAU-4 Vulcan Machine Gun</div> <div>Get AIM-9 Sidewinder Missile</div> <div>Get F16 Fighter Plane</div> <div>Get Mk82 General Purpose Bomb</div>

Figure C.10 Weapons Application as Seen by a Major

Figure C.11 Weapons Application as Seen by a Colonel

Figure C.12 Weapons Application as Seen by a General