Design Pattern as a Service for Service-Oriented Systems

by

Eltaher Mohamed El-Shanta

Masters of Computer Science, UNB, 2003

A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

in the Graduate Academic Unit of Computer Science

Supervisor: Weichang Du, PhD, Computer Science

Exchanging Board: Scott Bateman, PhD, Computer Science, Chair
Mohamed Drira, PhD, Business Administration
Huajie Zhang, PhD, Computer Science

External Examiner: Wendy MacCaull, PhD, Computer Science, St. Francis Xavier

This dissertation is accepted by the Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

September, 2019

© Eltaher Mohamed El-Shanta, 2020
Abstract

Software systems nowadays face many more challenges than ever before due to the intrinsic high complexity of systems and increasing demands from organizations. While patterns enable reuse of abstract design and architectural knowledge, abstractions documented as patterns do not directly yield reusable code. Software design patterns in particular are essential building blocks for almost any software system. In an effort to enable the use and reuse of implemented software design patterns, we propose a methodology to implement software design patterns as pattern services to make building pattern-based software applications considerably easier and faster. We describe the conceptual architecture and steps of the proposed methodology, and then we explain the implementation stages of a pattern as a service. After that, we demonstrate how the proposed methodology can be applied to Service-Oriented Architecture (SOA) patterns. To create a platform for managing pattern services, we design a Pattern as a Service (PaaS) system that functions as the platform for developing, storing, integrating, deploying, and managing pattern services and pattern-based applications. Furthermore, we describe a prototypical implementation of the PaaS system and the implementation of two case study applications, namely, an Online Discussion Group (ODG) and Online Stock Market Ticker (OSMT) that make use of the Observer pattern service and use the prototypical PaaS system as their platform. Then we perform some evaluation procedures on the proposed methodology both analytically and experimentally, and we give some concrete test results. Finally, we attach an appendix to this thesis in which we apply the methodology to the 23 design patterns introduced by Gamma et al. (1995). In it, we describe the important contents of each resulting pattern service.
Dedication

To every member of my family, your patience and support made it happen.
Acknowledgements

First and foremost, I would like to thank my country Libya and the Libyan people for their generosity in granting me the scholarship to pursue my studies and earn this degree. I would also like to thank my supervisor Dr. Weichang Du and the advising committee Dr. Andrew McAllister, and Dr. Scott Bateman for their guidance and valuable advice throughout the research period. I would like to thank the UNB Writing Center and Dr. Richard Spacek in particular for his valuable help. Finally, I would like to thank all those who helped me in any way to finish this research.
# Table of Contents

Abstract ..................................................................................................................... ii

Dedication .................................................................................................................... iii

Acknowledgements ........................................................................................................ iv

Table of Contents .......................................................................................................... v

List of Tables .................................................................................................................... xi

List of Figures .................................................................................................................... xii

List of Symbols, Nomenclature or Abbreviations .......................................................... xvi

Chapter 1 ......................................................................................................................... 1

Introduction ....................................................................................................................... 1

1.1 Motivation .................................................................................................................. 1

1.1.1 Why Pattern Implementation Reuse? ................................................................. 3

1.1.2 Why Design Patterns as Pattern Services? ......................................................... 5

1.2 Research Questions ................................................................................................... 8

1.3 Overview of Thesis Solution ..................................................................................... 12

1.4 Contributions ........................................................................................................... 14

1.4.1 Systematic Methodologies for Developing Design Pattern Services ............. 14

1.4.1.1 Conceptual Methodology for Design Pattern Services .................................. 15

1.4.1.2 Implementation Methodology for Design Pattern Services ........................... 15

1.4.1.3 Evaluation Methodology for Design Pattern Services ....................................... 16

1.4.1.4 General Methodology Idea for Software Pattern Services ....................... 17
1.4.2 Complete Design of all 23 GoF Design Patterns 18

1.4.3 PaaS Platform 19

1.5 Thesis Structure 19

Chapter 2 22

Background and Related Work 22

2.1 Background 22

2.1.1 Patterns 22

2.1.1.1 Software Patterns 25

2.1.1.2 Patterns for Software Development 27

2.1.2 Design Patterns 30

2.1.3 Service-Oriented Systems 32

2.1.3.1 What Is a Service? 32

2.1.3.2 Service-Oriented Computing and SOA 34

2.1.3.3 What Is a Service Pattern? 35

2.2 Related Work - Design Pattern Implementations as Object-Oriented Frameworks 36

Chapter 3 42

Motivating Example 42

3.1 Introduction 42

3.2 Implementing the Observer Design Pattern as a Pattern Service 46

3.3 General Pattern Service Methodology Overview 55
3.4 Summary ........................................................................................................... 68

Chapter 4 .................................................................................................................. 70

Conceptual Methodology for Design Patterns Services ........................................ 70

4.1 Identifying Pattern Layers .................................................................................. 77

4.2 Specifying Required Interfaces ......................................................................... 86

4.3 Classification of Pattern Services Based on Their Interface ......................... 95

4.3.1 Classification Based on Interface Type ......................................................... 95

4.3.2 Classification Based on Application Dependency of the Interface .......... 101

4.3.3 Classification based on Application-specific Service Interface ............. 108

4.4 Conceptual Pattern Service Realization through Interfaces ......................... 112

4.5 Summary ............................................................................................................. 117

Chapter 5 .................................................................................................................. 120

Implementation Methodology for Design Patterns Services .............................. 120

5.1 Application-specific Service and Pattern Service Binding ............................. 122

5.2 Pattern Service Runtime Configuration ............................................................. 126

5.3 Design Pattern Service Implementation Reference Architecture ............. 131

5.4 Implementation Level Pattern Service Realization ........................................ 136

5.5 Application-specific Pattern Service Generation ............................................ 140

5.5.1 Generation Methodology ............................................................................... 142

5.6 Application-specific Pattern Service Configuration File ............................ 145
Chapter 7

7.2 Functionality and Use Cases .............................................. 190
7.3 System Architecture Design.............................................. 203
7.4 PaaS System Case Studies .............................................. 216
  7.4.1 Online Discussion Group ........................................... 218
  7.4.2 Online Stock Market Ticker ........................................ 224
7.5 Summary .......................................................................... 229

Chapter 8

Evaluation Methodology and Experiments .................................. 231

8.1 Analysis-based Evaluation .............................................. 232
  8.1.1 Development Effort Evaluation .................................... 232
  8.1.2 Knowledge Skillset Requirements Evaluation ................. 242
    8.1.2.1 Design Pattern Knowledge Requirement for PSCs ......... 243
    8.1.2.2 Pattern Service Knowledge Requirement for AppSSCs .... 243
    8.1.2.3 Pattern Service Knowledge Requirement for AppSPSCs .... 246
  8.1.3 Performance Overhead Evaluation ................................. 251
8.2 Test-based Evaluation .................................................. 254
  8.2.1 Test Case Design for Functionality .............................. 257
  8.2.2 Test Case Design for Conformability ............................ 263
  8.2.3 Test Case Design for Performance and Scalability .......... 264
8.3 Experiments and Results .................................................. 265
8.4 Summary .................................................................................. 274

Chapter 9 ....................................................................................... 275

Conclusion and Future Work ............................................................ 275

9.1 Summary .................................................................................. 275

9.2 Discussion ................................................................................ 279

9.3 Conclusion ................................................................................ 282

9.4 Future Work .............................................................................. 283

Bibliography .................................................................................... 285

Glossary ......................................................................................... 298

Appendix A ..................................................................................... 301

Catalog of Design Pattern Services ................................................. 301

Curriculum Vitae
List of Tables

Table 4.1: PS classification based on interface types 96
Table 4.2: PS classification based on application dependency of interface 102
Table 4.3: PS classification based on AppSS interface requirement 109
Table 6.1: List of SOA patterns that apply design pattern ideas 180
Table 7.1: Functionalities supported by PaaS system 190
Table 8.1: Comparison of the PS and SOA development effort 240
Table 8.2: Estimated percentage of PS knowledge required for AppSSCs 246
Table 8.3: Knowledge percentage required for the AppSPSCs 250
Table 8.4: Estimated percentage of performance overhead incurred 253
Table 8.5: SOA and PS impl. using 1 Subject and 1 Observer 267
Table 8.6: SOA implementation using 1 Subject and many Observers 268
Table 8.7: PS implementation using 1 Subject and many Observers 269
Table 8.8: SOA impl. using many Subjects and many Observers – test1 270
Table 8.9: PS impl. using many Subjects and many Observers – test1 271
Table 8.10: SOA impl. using many Subjects and many Observers - test2 272
Table 8.11: PS impl. using many Subjects and many Observers – test2 273
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Patterns in software development</td>
<td>30</td>
</tr>
<tr>
<td>2.2</td>
<td>Observer design pattern</td>
<td>32</td>
</tr>
<tr>
<td>2.3</td>
<td>Observer design pattern with layers identified</td>
<td>41</td>
</tr>
<tr>
<td>3.1</td>
<td>A simplified general view of a stock data inquiry application</td>
<td>43</td>
</tr>
<tr>
<td>3.2</td>
<td>Service-oriented stock inquiry using Observer design pattern</td>
<td>45</td>
</tr>
<tr>
<td>3.3</td>
<td>Overview of stock inquiry using Observer pattern service</td>
<td>49</td>
</tr>
<tr>
<td>3.4</td>
<td>AppSS to Observer pattern service registration process</td>
<td>52</td>
</tr>
<tr>
<td>3.5</td>
<td>Stock inquiry system using the Observer pattern service</td>
<td>54</td>
</tr>
<tr>
<td>3.6</td>
<td>Interactions in stock inquiry system using the Observer pattern service</td>
<td>55</td>
</tr>
<tr>
<td>3.7</td>
<td>First step of the general methodology idea</td>
<td>58</td>
</tr>
<tr>
<td>3.8</td>
<td>Second step of the general methodology idea</td>
<td>58</td>
</tr>
<tr>
<td>3.9</td>
<td>General structure of a generic pattern service</td>
<td>59</td>
</tr>
<tr>
<td>3.10</td>
<td>First step of converting a generic PS into an AppSPS</td>
<td>62</td>
</tr>
<tr>
<td>3.11</td>
<td>Second step of converting a generic PS into an AppSPS</td>
<td>63</td>
</tr>
<tr>
<td>3.12</td>
<td>General structure of an AppSPS</td>
<td>64</td>
</tr>
<tr>
<td>3.13</td>
<td>PS and application binding at integration time</td>
<td>65</td>
</tr>
<tr>
<td>3.14</td>
<td>PS instance and application instance binding at configuration time</td>
<td>66</td>
</tr>
<tr>
<td>3.15</td>
<td>PS instance and application instance binding at runtime</td>
<td>68</td>
</tr>
<tr>
<td>4.1</td>
<td>Algorithm showing the conceptual methodology process</td>
<td>73</td>
</tr>
<tr>
<td>4.2</td>
<td>Observer design pattern</td>
<td>82</td>
</tr>
<tr>
<td>4.3</td>
<td>Layer separation of the Observer design pattern</td>
<td>85</td>
</tr>
<tr>
<td>4.4</td>
<td>Layers of the Adapter design pattern</td>
<td>86</td>
</tr>
</tbody>
</table>
Figure 4.5: Interfaces provided by the Observer PS ................................. 94
Figure 4.6: Interfaces provided by the Adapter PS .................................. 97
Figure 4.7: Interfaces provided by the Mediator PS ................................. 99
Figure 4.8: Interfaces provided by the Decorator PS ............................... 100
Figure 4.9: Interfaces provided by the Template Method PS ....................... 103
Figure 4.10: Interfaces provided by the Flyweight PS ............................... 105
Figure 4.11: Interfaces provided by the Command PS .............................. 107
Figure 4.12: Interfaces provided by the AppSSs of the Strategy PS ............... 110
Figure 4.13: AppSSs of the Singleton PS provide no runtime interfaces ......... 111
Figure 4.14: Runtime realization of the Decorator PS .............................. 115
Figure 4.15: Runtime realization of the Mediator PS ............................... 116
Figure 4.16: Runtime realization of the Singleton PS ............................... 117
Figure 5.1: AppSS to PS binding process ................................................. 125
Figure 5.2: Pattern service runtime configuration processes ...................... 130
Figure 5.3: Reference Implementation Architecture of Design Pattern Services .. 133
Figure 5.4: XML schema specific for the Observer PS ............................. 148
Figure 5.5 (Cont.): XML schema specific for the Observer PS ..................... 149
Figure 5.6 (Cont.): XML schema specific for the Observer PS ..................... 150
Figure 5.7 (Cont.): XML schema specific for the Observer PS ..................... 151
Figure 5.8 (Cont.): XML schema specific for the Observer PS ..................... 152
Figure 5.9 (Cont.): XML schema specific for the Observer PS ..................... 153
Figure 5.10 (Cont.): XML schema specific for the Observer PS ................. 154
Figure 5.11 (Cont.): XML schema specific for the Observer PS ................. 155
Figure 5.12 (Cont.): XML schema specific for the Observer PS ................. 156
Figure 5.13: Example XML document for configuring Observer PS

Figure 5.14 (Cont.): Example XML document for configuring Observer PS

Figure 5.15 (Cont.): Example XML document for configuring Observer PS

Figure 5.16 (Cont.): Example XML document for configuring Observer PS

Figure 5.17 (Cont.): Example XML document for configuring Observer PS

Figure 6.1: Chess Game Application Design

Figure 6.2: Chess game using Undoable Command PS – first design

Figure 6.3: Chess game interactions using Undoable Command PS – fist design

Figure 6.4: Chess game using Undoable Command PS – second design

Figure 6.5: Chess game interactions using Undoable Command PS – second design

Figure 6.6: The primary influences of SOA design patterns

Figure 6.7: Request/Reaction SOA pattern

Figure 6.8: Request/Reaction pattern service

Figure 6.9: Concurrent Contracts SOA pattern

Figure 6.10: Façade to Core Service Logic mapping step

Figure 6.11: Concurrent Contracts SOA pattern using Façade PS

Figure 7.1: PaaS system activities to manage pattern services

Figure 7.2 (Cont.): PaaS system activities to manage pattern services

Figure 7.3: High-level architecture of the PaaS system

Figure 7.4: PaaS support for PS creation and conversion into AppSPS

Figure 7.5: PaaS support for PbApps creation and PbAppIs deployment

Figure 7.6: Screenshot of the PaaS System’s main page.

Figure 7.7: Screenshot of the application instance deployment process.

Figure 7.8: Case study implementation of the Observer PS
Figure 7.9: ODG registration and setting processes with the Observer PS 219
Figure 7.10: ODG runtime interactions with the Observer PS 220
Figure 7.11: GUI screenshot of the administrator of the ODG 223
Figure 7.12: GUI screenshot of an ordinary member of the ODG 224
Figure 7.13: OSMT registration and setting processes with the Observer PS 225
Figure 7.14: OSMT runtime interactions with the Observer PS 226
Figure 7.15: GUI screenshot of an anonymous user of the OSMT 227
Figure 7.16: GUI screenshot of a registered user of the OSMT 228
Figure 8.1: Direct SOA implementation of the Observer PS 234
Figure 8.2: PS implementation of the Observer design pattern 236
# List of Symbols, Nomenclature or Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>App</td>
<td>Application</td>
</tr>
<tr>
<td>AppC</td>
<td>Application Creator</td>
</tr>
<tr>
<td>AppDSM</td>
<td>Application-dependent Service Method</td>
</tr>
<tr>
<td>AppDST</td>
<td>Application-dependent service Template</td>
</tr>
<tr>
<td>AppISM</td>
<td>Application-independent Service Method</td>
</tr>
<tr>
<td>AppS</td>
<td>Application-specific</td>
</tr>
<tr>
<td>AppSPS</td>
<td>Application-specific Pattern Service</td>
</tr>
<tr>
<td>AppSPSC</td>
<td>Application-specific Pattern Service Creator</td>
</tr>
<tr>
<td>AppSPSR</td>
<td>Application-specific Pattern Service Repository</td>
</tr>
<tr>
<td>AppSS</td>
<td>Application-specific Service</td>
</tr>
<tr>
<td>AppSSM</td>
<td>Application-specific Service Method</td>
</tr>
<tr>
<td>AppSSR</td>
<td>Application-specific Service Repository</td>
</tr>
<tr>
<td>AppSSRef</td>
<td>Application-specific Service Reference</td>
</tr>
<tr>
<td>FoD</td>
<td>Field of Discussion</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>ODG</td>
<td>Online Discussion Group</td>
</tr>
<tr>
<td>OSMT</td>
<td>Online Stock Market Ticker</td>
</tr>
<tr>
<td>PaaS</td>
<td>Pattern as a Service</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>PbApp</td>
<td>Pattern-based Application</td>
</tr>
<tr>
<td>PbAppC</td>
<td>Pattern-based Application Creator</td>
</tr>
<tr>
<td>PbAppI</td>
<td>Pattern-based Application Instance</td>
</tr>
<tr>
<td>PbAppIC</td>
<td>Pattern-based Application Instance Creator</td>
</tr>
<tr>
<td>PbAppIEU</td>
<td>Pattern-based Application Instance End User</td>
</tr>
<tr>
<td>PbAppR</td>
<td>Pattern-based Application Repository</td>
</tr>
<tr>
<td>PS</td>
<td>Pattern Service</td>
</tr>
<tr>
<td>PSC</td>
<td>Pattern Service Creator</td>
</tr>
<tr>
<td>PSR</td>
<td>Pattern Service Repository</td>
</tr>
<tr>
<td>SOA</td>
<td>Service-Oriented Architecture</td>
</tr>
<tr>
<td>ST</td>
<td>Stock Ticker</td>
</tr>
<tr>
<td>STF</td>
<td>Stock Ticker Feed</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

Fundamental to any science or engineering discipline is a common vocabulary for expressing its concepts, and a language for relating them (Appleton, 2014). The goal of patterns within the software community is to create a body of literature to help software developers resolve recurring problems encountered throughout many software development projects. Patterns help create a shared language for communicating insight and experience about these problems and their solutions (Appleton, 2014). As such, software patterns are considered an important tool in the analysis and design of a software solution. They give the software architect the ability to conceptualize a solution at different levels (Hofstader, 2014).

In their well-known book Design Patterns, Gamma, Helm, Johnson, and Vlissides (1995) explain that design patterns make it easier to reuse successful designs and architectures. They argue that expressing proven techniques as design patterns makes them more accessible to developers of new systems. They further add that design patterns help in the choice of design alternatives that make a system reusable and avoid alternatives that compromise reusability. Another point they state is that design patterns can even improve the documentation and maintenance of existing systems by furnishing an explicit specification of class and object interactions and their underlying intent (Gamma et al., 1995).
Since the publication of the design pattern book, the design patterns it introduced have become the standard that most of the software community adopted in designing and building software systems. They found their way to academia and became an essential part of software design and architecture courses taught all over the world.

Back in the early 1990s, software development was – and still is to a certain extent – dominated by the Object-Oriented (OO) programming paradigm and principles. Technologies were in place to support OO systems and facilitate their interactions. As a result, most of the software patterns that appeared in that time were OO software patterns. Back then, Service-Orientation had not been developed. Service-oriented software systems are systems that use software components implemented in software services as their building blocks. Nowadays, service computing is a considerably mature software development paradigm; it stands to reason that that set of useful OO software patterns can be adapted in such a way that they can be effectively used in service-oriented software systems.

Given the fact that design patterns are relatively low level patterns that describe how smaller components of a software system can be designed, they are particularly important building blocks that service computing can significantly benefit from. A design pattern can be implemented as a service, or a service can consist of one or more design patterns. This gives the individual services of a service-oriented system the qualities that come with design patterns such as the integrity and reliability of well proven solutions to design problems. As a matter of fact, several SOA patterns stem from design patterns, especially those dealing with aspects that service-oriented systems tend to share with OO applications and systems. For example, the Edge Component
pattern (Rotem-Gal-Oz, 2012) is actually derived from the Façade and Proxy design patterns Gama et al. (1995). State Repository of the Inventory Implementation patterns (Erl, 2009) is similar in its functionality to the Memento design pattern of Gamma et al. (1995). Moreover, the Service Façade pattern (Erl, 2009) is the service-oriented version of the Façade design pattern. Another SOA pattern that is derived from design patterns is the UI Mediator pattern (Erl, 2009). This UI Mediator pattern performs the same mediation task the Mediator design pattern does. Furthermore, a considerable number of service and SOA patterns such as Command Invoker, Service Connector, Service Controller, Request Mapper (Diagneau, 2012), and many more use one or more design patterns to accomplish their goals. Along with some architectural patterns such as Domain Model, Data Mappers, and Lazy Loading (Fowler, 2003), to name a few, design patterns such as Command, Factory Method, Façade, Proxy, Builder, and Observer are used to implement service and SOA patterns.

Since this thesis focuses mainly on software design patterns and how to implement them as pattern services, in the rest of this thesis, unless we specifically mention the precise name of a group of patterns, when the terms “software pattern” or “pattern” are mentioned, we are actually referring to software design patterns in particular.

**1.1.1 Why Pattern Implementation Reuse?**

To practically answer the question of “why do we need to reuse pattern implementations?” we discuss software reuse and highlight its advantages. The purpose or advantages of software reuse can be classified under two main categories, namely, lower cost, and better quality software products. Reused products are cheaper because of their shorter development time, increased productivity of the development staff and
software experts, decreased testing requirements, and easier maintenance. As for the better quality attribute, it is proven that software code that is written for reuse usually has better specifications and it is more thoroughly tested. This implies that reused software artifacts are more dependable, require less maintenance, and incur lower maintenance costs (Sommerville, 2011). Reusing reliable and well standardized software components improves the resulting system’s specification, standardization, and organization.

Sommerville lists the benefits of software reuse as follows (Sommerville, 2011):

1- Increased dependability – reused software that has been tried and tested in working systems should be more dependable than new software.

2- Reduced process risk – the cost of existing software is already known whereas the costs of development are always a matter of judgment.

3- Effective use of specialists – instead of doing the work over and over again, application specialists can develop reusable software that encapsulates their knowledge.

4- Standard compliance – some standards, such as user interface standards, can be implemented as a set of reusable components. When reused, all applications present the same UI formats to users. Consequently, users make fewer mistakes because of the familiarity with the interface.

5- Accelerated development – bringing a system to market as soon as possible is often more important than the overall development costs. Reusing software can speed up system production because both development and validation time may be reduced.
Putting the benefits of software reuse in the perspective of software pattern implementation, it can be argued that a software system that is built by reusing software pattern implementation achieves the maximum of all the benefits of software reuse mentioned above. This is due to the fact that software patterns offer the best solutions to many software development problems. Further, software reuse has been known to have many benefits some of which are listed above. Combining the advantages of software reuse and software patterns in a well implemented and tested form that can be readily reused to build software systems is like having the best of both worlds. On the one hand, you have software patterns that can be trusted to deliver the best solutions as the core of the reuse material. On the other hand, having them implemented and ready for reuse brings in software reuse benefits and makes building new trustworthy software systems a much simplified and accelerated task.

Moreover, reusing software patterns as readily implemented components puts them under intense scrutiny and testing in different situations and use cases. This process makes software pattern implementation more and more robust, correct, scalable, among other quality attributes. Of course, development efficiency is one attribute of software pattern implementation reuse that cannot be neglected. And by development efficiency we mean saving software developers’ effort and time by relieving them from having to repeat the process of implementing different software patterns from scratch over and over again.

1.1.2 Why Design Patterns as Pattern Services?

Service-Oriented software development is centered on many advantages that make service computing and service-based software systems among the best in terms of
modularity, composability, scalability, and availability (Erl, 2005). Qualities such as improved integration, ease of reuse, platform and vendor independence, loose coupling, and code mobility are inherent to service-oriented computing, not to mention non-technical benefits such as better Return on Investment (ROI) and reduced IT burden (Erl, 2005). Logically, software components built as services strictly adhering to service-orientation rules inherit these qualities, which make such services highly reliable and excellent candidates for reuse.

With the advent of cloud computing, service computing has become even more appealing. Cloud computing with its variety of delivery models and the abundance of resources creates a perfect medium for developing and deploying service-oriented software systems and components. Platform as a Service (PaaS) and Software as a Service (SaaS) are particularly useful models to develop and host service-based software systems. The former provides a complete platform with all needed infrastructure, networking, operating system, and software development packages ready for use by service-based software developers to develop and deploy their own services. The latter model can be thought of as a PaaS offering plus the provider takes care of the development of the needed service-based software systems on the consumers’ behalf so they can offer them to their end users.

The abundance of resources provided by the cloud computing paradigm through the use of data centers and server virtualization and quick disaster recovery procedures it adopts, makes service-based software even more scalable, reliable, and resilient. Further, service-based software systems offered over the cloud have global availability;
all that is needed is an internet connection. Lately, there has been a good market for both models among businesses and organizations, thanks to the advantages they bring.

Going back to the question posted above, the benefits of implementing software patterns as services as opposed to ordinary code implementation include inheriting the benefits and advantages of the service-oriented software systems we mentioned. Dynamic service integration, composition, and generation are other valuable qualities that the success of patterns as services relies heavily upon. Another advantage that service-based software development has at its disposal is cloud computing, which removes a lot of the obstacles that usually hinder or at least slow down software development and deployment processes. Cloud computing provides service-based software developers with all necessary hardware and software infrastructure and platforms, equipment, and tools to develop, test, and deploy SaaS components. Also, implementing software patterns as services makes them easily accessible to anybody anywhere in the world. This advantage promotes the idea of crowd sourcing, through which people can be invited to participate in enriching the library of software pattern services (See Glossary for definition) by creating their own services and helping in the testing and fine tuning process of the software pattern services contained in the library.

To sum up, the goal of this research is to propose a methodology for implementing software patterns in a way that achieves the following benefits:

a) No redevelopment of pattern implementations is needed.

Once the patterns are implemented as pattern services, software developers will not need to re-implement each pattern from scratch; rather, they will only need to add application-specific logic and attach it to the subject pattern service.
b) **Standardized and correct implementation of the patterns reduces pattern ad-hoc implementation.**

The methodological implementation of the patterns reduces or may eliminate ad-hoc re-implementation of such patterns, which is inherently error prone.

c) **Quality control that comes with tested and reused pattern services.**

The recurring usage of the implemented pattern services by users building different software applications puts the pattern services under continuous test which may result in fixing any bugs and fine tuning of the pattern service implementation.

d) **Improved development productivity for service-oriented systems.**

Having software patterns implemented as readily available services will make building service-oriented software systems a lot faster than if the developers had to implement everything from scratch.

e) **No deep technical knowledge of the patterns is needed for developing applications.**

Service-oriented system developers will still need to have a certain level of understanding of the software patterns they intend to use in their systems; however, with patterns implemented as services, such a level of understanding does not have to be deep and comprehensive.

### 1.2 Research Questions

The problem statement that summarizes the problem this research tries to solve can be stated as follows: “*Software patterns have well-known and largely recognized benefits in developing reliable software systems. The problem with the current state of software*
patterns is that, while they enable reuse of abstract design and architecture knowledge, abstractions documented as patterns do not directly yield reusable code. In the era of everything as a service, and the feasibility of developing service-oriented systems, can those documented abstractions of software patterns be implemented as reusable service components? Also, can these reusable pattern implementations be developed separately from the development of application services, and if yes, how?"

As part of the solution to this problem, we determine that the main objective of this research is to put together a methodology for implementing software design patterns from different fields with different granularity as pattern services that can be used in the development of service-oriented systems. Although in this thesis work we are focusing on design patterns in particular, the methodology may be generic enough to be valid for use with other kinds of software patterns.

In the process of developing such a methodology, we need to investigate the answers to a few questions related to design patterns and service-oriented computing and where the common ground between them lies.

The first question that we need to answer is “What are the pros and cons of using design patterns in service-oriented systems?”

In this research, we need to study design patterns to understand their interactions, collaboration, strengths, weaknesses, and the consequences of using those design patterns. Then we compare and contrast their benefits and compromises if they are used in a service-oriented environment. This also includes studying the possibility of
introducing some adaptations to the original structure of the design pattern and what the implications of such adaptations may be.

The second question that we need to answer is “How can design patterns be used in service-oriented systems, and can they be implemented as services to be used in the construction of service-oriented systems?”

As mentioned in the motivation above, design patterns were originally introduced to solve design problems and recommend good practices for the OO design and programming paradigm. Although service-computing shares some internal design principles with OO programming, the relationships, cooperation, and interaction between the services that make up a service-oriented system are different. For this reason, and to answer the first half of the question, we need to study and understand those differences, their extent, and how they may affect the overall design and implementation of the services that implement design patterns.

The answer to the second half of the question is an important and fundamental one. If the answer is no, then there is no point in pursuing this research any further. On the other hand, if the answer is not a resounding no, then still there may be some design patterns that may not be easily implemented as services, because of their structure or behavior that may not align well with the Service-Orientation paradigm or the result of implementing such patterns as services does not justify the implementation and runtime complexities involved. So what we really need to find out is the feasibility of enclosing the implementation of a design pattern inside a service or split it between two or more cooperating services. We also need to make sure that the proposed service design, whether it be a single service or a composition of services, works within the limits of
service-orientation principles and does not break those of the subject design pattern at the same time.

The third question we need to investigate and answer is “Can the development of reusable pattern services be separated from the development of concrete application-specific services?”

This question is related to the fact that design patterns, and patterns in general, must have application-specific logic added to their generic parts in order to be used in a certain application or system. This research needs to figure out the best way to design a pattern service that implements all the generic aspects of a design pattern and leaves the application-specific parts to be implemented by application developers so that they include their application-specific implementation details in those parts. The generic pattern service (See Glossary for definition) part must, of course, provide all the necessary means to integrate with the application-specific services (AppSSs; See Glossary for definition). All of the above mentioned questions need answers in order to make the implementation of design patterns as services a possibility.

Once we build a methodology for implementing patterns as services, we need to consider ways of putting such a methodology to work. Given the fact that a pattern service that implements a design pattern for service-oriented systems is – most likely – going to be made up of a few services, a system or a platform that facilitates the integration, composition, and deployment of service-oriented systems made up of groups of such pattern services is also needed. A part of this research is dedicated to the design of a comprehensive system or platform that supports all the steps of creating, integrating, and deploying pattern services and Pattern-based Applications (PbApps;
See Glossary for definition). To test the designed platform, we are planning to build what we call a Pattern as a Service (PaaS) system (See Glossary for definition). This prototypical system functions as a platform for managing and deploying PbApps built using software pattern services, pattern service compositions, AppSSs, and other application services.

1.3 Overview of Thesis Solution

The solution presented by this research thesis is to put together a methodology for implementing software design patterns from different fields with different granularity as pattern services. In addition to design patterns, the methodology may be applied to other types of software patterns such as architecture and analysis patterns.

In the process of devising, applying, and testing the methodology for implementing software design patterns as pattern services we perform the following tasks:

1- We present a conceptual analysis of the proposed methodology and how to identify pattern layers and the players involved in realizing the pattern. The conceptual analysis is done specifically on the field of software design patterns, and we use the group of 23 Gamma et al. (1995) design patterns to demonstrate how such a conceptual process can be applied in practice.

2- We describe the implementation methodology for the design pattern as pattern services. After conceptualizing the steps of how a design pattern can be implemented as a pattern service and classifying the different types of design patterns based on their layers of abstraction, interfaces, and players involved, this stage looks at how the conceptual model of a design pattern as a pattern service can
actually be implemented and configured to work in the context of specific software applications.

3- After providing the conceptual structure of the proposed methodology and the implementation steps involved in realizing the pattern as a service target, we investigate the relationship between OO design patterns and SOA patterns. We give examples of how the methodology can be applied to some SOA patterns and describe the structure of some SOA patterns implemented as pattern services. Then, we turn our attention to the platform that supports the creation, storage, configuration, deployment, and management of the resulting pattern services. The pattern as a service (PaaS) system acts like the pattern as a service platform that provides the essential services to manage pattern service repositories and help integrate pattern services with each other and with application-specific components to help solve software problems as pattern-based service-oriented software systems.

4- Finally, we evaluate the proposed pattern service methodology and measure the development effort, pattern knowledge required by all of the participants in this process, and the overhead incurred by applying the methodology in developing pattern services. We also propose the test case and what goes in each test case to validate and test pattern services before they are used in production software applications.

The importance of this research lies in the development of a set of methodologies and sub-methodologies to guide the process of implementing software design patterns as reusable pattern services to be used in service-oriented systems. Ideally, the proposed methodologies are used to implement software design patterns as reusable pattern
service that are stored in some widely accessible repository. Software application designers and developers should be able to access that repository, search for reusable pattern services, select the ones they need to reuse in their applications, add their application-specific logic and necessary configuration to the reusable pattern service, and finally use the resulting configured pattern service in their software applications. Having a repository of software design patterns implemented as reusable pattern service saves software application designers and developers from having to implement the design patterns from scratch, every time they need to use them in a different software application.

Due to the nature of software design patterns and their level of abstraction, they are considered design building blocks for the components and sub-systems. This means that pattern services built using the proposed methodologies are still design building blocks, in the form of reusable services, and can be used as building blocks for any service-oriented system regardless of its magnitude, small applications as well as enterprise scale ones.

1.4 Contributions

The main contributions of this thesis work are briefly described in the following subsections.

1.4.1 Systematic Methodologies for Developing Design Pattern Services

In this thesis work, we presented methodologies for developing design pattern services. The methodologies proposed by this work give detailed descriptions of the conceptual,
implementation, and evaluation aspects of the process of implementing software design patterns as pattern services. In fact, it also shows how to generalize these methodologies and apply them to implement almost any software pattern as a pattern service.

In the following subsections we list the conceptual, implementation, and evaluation methodologies presented by this thesis work, and, where appropriate, we highlight their impact on software engineering research and software development practice.

1.4.1.1 Conceptual Methodology for Design Pattern Services

The first methodology for creating design pattern services, contributed by this research work, is the conceptual. The conceptual methodology presented helps build the conceptual model of how a design pattern can be implemented as a pattern service. It describes how the design pattern at hand can be divided into layers of abstraction, and then how to extract the generic logic from these layers and use it to construct the generic pattern service layer. Thirdly, it gives guidelines on how to identify the application-specific logic and how to integrate it with the generic pattern service later on.

Having such a design pattern service conceptual methodology increases the domains in which design patterns can be used. The conceptual methodology presented by this thesis work adds a very useful set of design patterns to the tools that software engineers can use to design service-oriented software systems.

1.4.1.2 Implementation Methodology for Design Pattern Services

The conceptual methodology of design pattern services was the first step in the contributed sequence of methodologies to implement design patterns as pattern services. While the conceptual methodology enables software engineers to conceptualize and use
design patterns in their service-oriented system designs, the implementation methodology helps them add the implementation puzzle pieces to the overall design of the system. The implementation methodology can also be used to guide the implementation stages of the pattern service, adaptation and configuration to the application-specific needs, and the process of integrating it with the service-oriented system at hand.

1.4.1.3 Evaluation Methodology for Design Pattern Services

The third methodology contributed by this thesis work is the methodology for evaluating design pattern services to measure their development effort and performance efficiency. The benefit of the evaluation methodology is that it prescribes a way to test the worthiness of implementing a design pattern as a pattern services: what are the costs in terms of effort, time, and pattern expertise, and how efficient will the implementation of the design pattern as a pattern service be. In fact, the evaluation methodology is twofold. The first fold is the analytical component of the methodology. The second is the experimental.

a) Analytical Methodology for Evaluating Design Pattern Services

As the name suggests, the analytical component of the design pattern service evaluation methodology is concerned with the evaluation measures that can be used to assess the feasibility and effectiveness of implementing a design pattern as a pattern service. Using the conceptual methodology to identify the pattern artifacts, the mathematical formulas proposed by the analytical evaluation methodology can be used to assess the development effort, degree of pattern knowledge required by the different participants in the design and implementation of the pattern service,
and the overhead penalty that may be incurred. The application of this aspect of the evaluation methodology helps software engineers decide what pattern services they should use in their systems based on the analytical assessment they perform on those pattern service before deciding whether to implement them or not. Also, this analysis software engineers perform helps them decide the type and nature of resources, computational and human, they need to allocate to the development of the subject pattern service.

b) **Experimental Methodology for Evaluating Design Pattern Services**

The second part of the evaluation methodology contributed by this thesis comes after the pattern service has been developed. The experimental evaluation methodology proposes a battery of tests and guidelines to build the suitable test cases that software engineers and developers can run on the pattern service implementation to assess its efficiency. The methodology suggests a group of tests that can be run on every part or component of the pattern service. These include tests for the generic part of the pattern service, the application-specific services, and the fully configured application-specific pattern service. This part of the evaluation methodology complements the analytical one and both comprise a valuable asset for software engineers and developers to help them with the design decisions of service-oriented systems and the testing procedures for the pattern services they developed and intend to use in their service-oriented systems.

1.4.1.4 **General Methodology Idea for Software Pattern Services**

The fourth contribution of this thesis comes in the form of a potential generalized methodology for implementing software patterns in general as pattern services.
Although the main focus of this thesis work is on developing methodologies to implement software design patterns as patterns services, including methodologies for the conceptualization, implementation, and evaluation of design pattern services, the proposed methodologies may be generalized and applied to other types of software patterns, i.e. analysis, architectural, and enterprise patterns. It is important to mention that, even though we did not conduct enough research and we did not provide sufficient examples, and tests to fully support this conjecture, however, there is some evidence presented in Chapter 6 that shows that the proposed methodology can be used with software patterns other than design patterns. In Chapter 6 we explained how the proposed methodology can be used to implement SOA patterns as pattern services.

1.4.2 Complete Design of all 23 GoF Design Patterns

Attached to this thesis is an appendix that contains a catalog of the design pattern services for all 23 GoF design patterns. Each entry in the catalog gives the design of the pattern service that can be implemented based on one of the GoF design patterns. It also contains a brief description of the subject pattern service interfaces and the purpose the service method behind those interfaces serve. Further, the collaboration between the components of the pattern service is also presented in a collaboration diagram to further explain the sequence of interactions that realize the pattern service. Finally, a sample configuration XML file and the XML schema to validate it are listed for each of the 23 design pattern services. This pattern service catalog may incite software engineers to opt for implementing one or more of the GoF design patterns as pattern services, given that the design and collaboration of the design pattern services is already provided. This
makes implementing GoF design patterns as pattern services easier and faster because of the catalog contributed by this thesis work.

### 1.4.3 PaaS Platform

Last, but not least, is the design of the PaaS system that this thesis presents. The PaaS system is a platform for creating, storing, configuring, composing, managing, and deploying pattern services. The proposed system design takes care of all aspects of managing pattern services and pattern-based applications, applications that use pattern services to deliver part or most of their functionalities. The design of the system, presented in chapter 7, enables pattern-based application creators (PbAppCs) to search for pattern services in its pattern service repository, select specific pattern services, integrate them with proper application-specific services, apply needed configuration to the pattern services, and integrate them in their pattern-based service-oriented applications. Such a system promotes pattern service use in service-oriented applications and formalizes the process of creating and using pattern services.

### 1.5 Thesis Structure

The rest of this thesis is divided into 8 chapters. Chapter 2 gives the background material about software patterns with more emphasis on design patterns – since they are the focus of this thesis research. There is also a section that covers service computing and SOA. The third part of Chapter 2 is dedicated to the related work.

Chapter 3 uses a motivating example to describe the steps lead to implementing a design pattern as a pattern service. These steps are later generalized to formally specify the steps of the pattern service methodology.
Chapters 4 and 5 propose a design pattern specific realization of the proposed general methodology idea. Although the proposed methodology is applicable to any OO design pattern, in this thesis we use the group of 23 design patterns introduced in Gamma et al. (1995) as an example to describe the process of applying the methodology.

Chapter 4 conceptually explains how the proposed methodology can be applied to implement software design patterns as pattern services. Then it classifies the 23 Gamma et al. (1995) design patterns into groups based on certain criteria. Finally, it explains how each classified group of pattern services can be realized and what the involved actors and interactions are.

Chapter 5 focuses on the implementation of the proposed methodology in regard to design patterns. It describes the steps to be followed in implementing design patterns as pattern services. It also shows how the resulting pattern services can be configured and adapted to become application-specific pattern services that can work according to the rules and requirements of specific software applications.

Chapter 6 of this thesis discusses some of the use cases of the proposed design pattern service methodology. It starts by showing an example of how two or more pattern services can be composed to work together to solve some software design problem. Then it demonstrates how to apply the pattern service methodology to SOA patterns. It also classifies SOA patterns based on their relationship to some of the OO design patterns.

Chapter 7 introduces the Pattern as a Service system or platform. The first section explains the functionality of the system and its use case. The second section describes
the architecture and design of the system. The third section gives a brief description of the prototypical implementation of the system. The last part of Chapter 7 describes two use case studies that were developed to test the prototype implementation of the PaaS system, namely, an online discussion group, and an online stock market data ticker.

The first few sections of Chapter 8 present the analysis-based evaluation of the methodology. They explain the evaluation procedures and provide some comparison results in terms of development effort, pattern knowledge requirement, and performance overhead of applying the proposed methodology compared to the direct implementation using the SOA paradigm. The other sections describe how to test the different components of a pattern service and how to build the proper test cases to test each component. The last section of Chapter 8 gives some concrete performance test results for the Observer design pattern implemented as a pattern service, and as a direct SOA service.

Chapter 9 concludes this thesis with a summary and proposes some future work.
Chapter 2

Background and Related Work

2.1 Background

2.1.1 Patterns

The *Oxford Dictionary* has several definitions of a pattern; one of those definitions states that “A pattern is a regular and intelligible form or sequence discernible in the way in which something happens or is done” (Oxforddictionaries.com, 2014).

Another definition for patterns from a business perspective is “A pattern is a consistent and recurring characteristic or trait that helps in the identification of a phenomenon or problem, and serves as an indicator or model for predicting its future behavior” (Businessdictionary.com, 2014).

*Wikipedia*’s definition of a pattern is “A pattern, apart from the term’s use to mean a template, is a discernible regularity in the world or in a man-made design. As such, the elements of a pattern repeat in a predictable manner” (Wikipedia, 2014).

From the definitions above, we can infer that patterns are repetitions of some kind that take place usually in a systematic way. A pattern may describe shape, structure, and/or behavior.

Patterns can be found in almost every field or discipline, and they are very useful to us. We use patterns in our thinking, behavior, relationships, and almost every aspect of our daily lives. We use patterns to solve problems. When we see a solution to a problem or
situation that works, we tend to use it, most times without even noticing it, every time we are faced with the same problem or situation.

The deliberate first use of patterns as solutions to recurring problems is attributed to the architect Christopher Alexander who defines them in these terms: “The pattern, in short, is at the same time a thing, which happens in the world, and the rule which tells us how to create that thing, and when we must create it. It is both a process and a thing; both a description of a thing which is alive, and a description of the process which will generate that thing” (Alexander et al., 1977, p. 72).

James Coplien (1996), relates this definition to dress patterns. He writes:

I could tell you how to make a dress by specifying the route of a scissors through a piece of cloth in terms of angles and lengths of cut. Or, I could give you a pattern. Reading the specification, you would have no idea what was being built or if you had built the right thing when you were finished. The pattern foreshadows the product: it is the rule of making the thing, but it is also, in many respects, the thing itself. (p. 3)

The important points of the discussion above can be summarized in Alexander’s more concise definition of the pattern: “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice” (Alexander et al., 1977, p. x).

The above discussion of patterns by Alexander et al. can be divided into three main parts as follows: A pattern describes a generic solution to a recurring problem, within a
defined context (Alexander et al., 1977). The three parts of the pattern can be further explained as follows:

1- **Generic Solution**: This means that a pattern does not define a specific solution. Rather it identifies the “class” of a problem and how that problem might be solved.

2- **Recurring problem**: It means that patterns are useful when the problem is not unique and most useful when the problem occurs often.

3- **Defined context**: This means that bounds have to be put on the generic solution because there are no universally true solutions. So you have to understand the circumstances in which the generic solution is valid, and hence how to elaborate on it to create your own specific design (Alexander et al., 1977).

For most patterns, there are some trade-offs that are inherently associated with their application. It is important to note that context, problem, and trade-offs are important aspects of a pattern; consequently, they should be studied and understood prior to pattern application (Stepan, 2011).

In general, it can be stated that the purpose of patterns is to provide the following:

1- Give vocabulary to a problem, context, solution, and consequences;

2- Catalog architectures and designs that have proved successful in past systems and formalize their elements, relationships, interactions, etc;

3- Provide enough information about trade-offs and consequences to allow intelligent design decisions to be made about applying a given solution.

Notwithstanding all the well-deserved praise for patterns, it is important to know that patterns are not complete solutions to the problem at hand; rather, they can be described as solutions that need to be contextualized (Blaimer, Bortfeldt, and Pankratz, 2010).
This means that a pattern needs to be elaborated on and adapted according to the requirements in order for the solution that uses the pattern to be complete.

Designers have used patterns in a wide range of fields including “development organization and process, exposition and teaching, and software architecture” (Coplien, 1996). As a matter of fact, patterns exist in all mature engineering disciplines, including software engineering (Hofstader, 2014). Enterprise and business patterns share the same pattern properties. A business pattern describes a re-usable approach to the solution of a particular business problem, usually giving it parameters by business process requirements. It offers a solution based on previous success in defining solutions to the same, or similar, business problems. A business pattern is characterized by being instructive, structural, reusable, and proven, and lastly by making business sense (Blaimer, Bortfeldt, and Pankratz, 2010).

2.1.1.1 Software Patterns

From a software perspective, patterns are a software engineering problem-solving discipline that emerged from the object-oriented community. Patterns have roots in many disciplines, including literate programming, and most notably in Alexander's work on urban planning and building architecture (Coplien, 1996). The software community saw the potential of Alexander’s idea of patterns in the software design and development processes. They eagerly welcomed patterns as a valuable asset in building software systems and applications. In 1987, Kent Beck and Ward Cunningham began experimenting with the idea of applying patterns to programming and presented their results at the OOPSLA conference that year (Kent and Cunningham, 1987; Wikipedia, 2014). Since then, the use of patterns became a cornerstone in software development
and many new software patterns, at different levels of granularity, have been introduced. They have given software architects and developers the ability to conceptualize a solution at different levels. Software patterns are also a valuable way of communicating concepts between software professionals (Hofstader, 2014). For more than a decade, design-focused patterns, and other software patterns for that matter, provide a vocabulary for expressing analytical and architectural visions; this leads to clear, concise representative designs and detailed implementations (Buschmann, Henney, and Schmidt, 2007).

A popular form of technical writing is the cookbook style, where a book describes recipes as solutions. There is a lot of similarity between cookbooks and pattern books. Both emphasize a problem-solution style. The big difference between the two, though, is in the notion of building vocabulary. Recipes tend to be more particular, usually tied to a particular programming language and platform. Even when patterns are tied to a platform, they try to describe more general concepts. As a consequence of this, recipes have a stronger problem focus than the solution focus in patterns (Fowler, 2006).

Software systems now face many more challenges than ever before due to the intrinsic high complexity of systems and increasing demands from organizations. Moreover, customers now demand shorter and shorter time-to-market (Chang, Lu, and Hsiung, 2010). This set of requirements can be made easier to satisfy by adopting patterns in building software systems.

Formally, it is proven that non-pattern forms of software design are less efficient compared to pattern form in terms of design metrics (Chen and Chen, 1994). Pattern use
in the software development cycle has proven to be an effective way of reducing cost, effort, and time. The effectiveness of pattern use does not stop at the design and development of a software system; rather, it extends to the testing and maintenance stages. Zhu (2009) shows that software development and maintenance based on software patterns can reduce cost and time, and improve reusability, maintainability, scalability, and other quality attributes.

According to Jacobsen, Kristensen, and Nowack (1997), pattern use can affect another dimension of the software development process, namely, consistency. They argue that if patterns are used in the analysis, design, and implementation phases of system development, it would be possible to develop systems that follow the characteristics of those patterns, which can reduce the effort of checking consistency between the software development phases.

Software patterns are the common stories the software community culture shares. They can be viewed as stories about commonly occurring problems in software architecture and design and their solutions. As young children learn about good and evil from fairy tales, beginning software engineers learn about good design “design patterns” and bad design “anti-patterns” (WikiBooks, 2014).

2.1.1.2 Patterns for Software Development

In software design and implementation, there are three main types of patterns: analysis and architectural patterns, design patterns, and idioms. Each of these three types should be engaged at a certain level of abstraction. Buschmann, Meunier, Rohnert, Sornmerlad,
and Stal (1996) explain the nature and use of each of these three types of patterns as follows:

1- Analysis and Architectural patterns can be used at the beginning of coarse-grained design, when specifying the fundamental structure of an application.

2- Design patterns are applicable towards the end of coarse-grained design, when refining and extending the fundamental architecture of a software system, for example deciding on the basic communication mechanisms between subsystems. Design patterns are also applicable in the detailed design stage for specifying local design aspects, such as the required support for multiple implementations of a component.

3- Idioms are used in the implementation phase to transform a given software architecture into a program written in a specific programming language.

Zhu (2009) expresses the difference as follows: domain models or architectural patterns provide a structured framework, but implementation of the elements in the architecture need to be achieved based on design patterns and idioms. In general, the quality of architecture directly determines the quality of software systems, and design patterns can help build high-quality architecture. Similarly, idioms provide support for implementing design patterns.

The classic Gang of Four design patterns (GoF; Gamma et al., 1995) are the foundation for patterns as solutions to recurring software design problems. They consist of patterns that mostly deal with the low level design of the software. These patterns are solutions or in fact templates for making the code – that results from implementing the patterns – more and more reusable and maintainable.
Martin Fowler and various other prominent software professionals introduced another set of patterns at a rather different level of abstraction. This set of patterns is mainly applicable at an earlier stage of software development than the design stage. These patterns do not necessarily deal with how to design the classes and interfaces; rather, they are based on how to compose software system components in a broader sense. They defined Enterprise Analysis and Architectural Patterns. These patterns deal with commonly used solutions across enterprises. They represent another set of patterns which deal with common solutions to recurring problems in enterprise applications (Fowler, 1997; Fowler, 2003).

Martin Fowler’s definition of a pattern is as follows: “A pattern is an idea that has been useful in one practical context and will be probably useful in others” (Fowler, 1997, p. 8).

Fowler’s definition makes clear that he puts great emphasis on the tried-and-tested quality of his patterns. For him, the key element of a pattern is that it was discovered during the everyday development process and is not an academic invention. Most of the patterns he presented were developed in one or several projects that he supervised.

Figure 2.1 shows a simple classification of software patterns, their level of abstraction, and the phases of software development in which each kind of patterns is used. As we noted earlier, in the rest of this thesis we focus mainly on design patterns and how to implement them as pattern services that are suitable for use with service-oriented software systems.
2.1.2 Design Patterns

The most well-known group of software design patterns is the one introduced by the Gang of Four (GoF) in their famous book *Design Patterns, Elements of Reusable Object-Oriented Software* (Gamma et al., 1995). In their book, the GoF introduced a set of 23 software design patterns. They divided this group of design patterns into three subgroups, namely creational, structural, and behavioral design patterns. The GoF definition of the design patterns they introduce in their book is as follows: “Design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context” GoF (p. 3). Buschmann, Meunier, Rohnert, Sornmerlad, and Stal (1996) emphasize structure but at a lower level than the architectural: “A design pattern provides a scheme for refining the subsystems or components of a software system, or the relationships between them; it describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context” (p. 221).
Design patterns support the creation of the design model. They provide solutions for recurring problems that occur during the transition from the analysis and architectural model to the design model. As distinct from architectural patterns which are closely related to the overall architecture of the software system and its constituent sub-systems, design patterns contain more detailed model fragments which are dedicated to rather technical or implementation-related issues. Design patterns describe the respective solution in terms of classes and their relationships, i.e. they are used to specify components and sub-systems in detail (Blaimer, Bortfeldt, and Pankratz, 2010). They provide a clear concept of design structure by describing the relationships of inheritance and reference between components of the software system (Chang, Lu, and Hsiung, 2010).

Moreover, design patterns are medium-scale patterns. They are smaller in scale than architectural patterns, but, contrary to idioms they tend to be independent of a particular programming language or programming paradigm. The application of a design pattern has no effect on the fundamental structure of a software system, but may have a strong influence on the architecture of a subsystem (Buschmann, Meunier, Rohnert, Sornmerlad, and Stal, 1996). Thus design patterns are clearly distinguished from architectural patterns, which describe solutions from a high-level perspective, i.e. in terms of components and sub-systems.

Similar to other software patterns, design patterns require some adaptations and modifications when being integrated into the design model at hand (Blaimer, Bortfeldt, and Pankratz, 2010). In fact, Fant (2011) argues that applying design patterns in practice
can be difficult since design pattern descriptions are general and can be applied at multiple levels of abstraction.

One of the interesting design patterns, which we use as demonstrative example throughout this thesis, is the Observer design pattern introduced by the GoF. We picked the Observer design pattern is because it represents a good model that combines all structural and behavioral aspects that need to be described during the process of implementing a design pattern as a pattern service. Figure 2.2 shows the structure of the Observer design pattern.

![Observer Design Pattern](image)

Figure 2.2: Observer design pattern after GoF (p. 294)

2.1.3 Service-Oriented Systems

2.1.3.1 What Is a Service?

The term “service” has existed for thousands of years. When a person or a group performs some work to benefit another, it becomes a service (Zhang, Zhang, and Cai, 2007). Many versions of definitions exist for the term “service”. For example, James
Fitzsimmons (2005) defines a service as follows: “A service is a time-perishable, intangible experience performed for a customer acting in the role of co-producer” (p. 94). Christian Gronroos (1990) defines a service from the perspective of management and marketing as follows: “A service is an activity or series of activities of more or less intangible nature that normally, but not necessarily, take place in interactions between customer and service employees and/or physical resources or goods and/or systems of the service provider, which are provided as solutions to customer problems” (p. 27). Focusing on IT-enabled business services, Zhang et al. (2007) define the term “services” as follows: “Services represent a type of relationship-based interactions (activities) between at least one service provider and one service consumer to achieve a certain business goal or solution objective” (p. 3).

Although these definitions look slightly different, they all indicate the fact that each service involves two essential sides: service provider and service consumer. A service provider offers the service, and a service consumer utilizes the service. The interaction between a service consumer and a service provider may happen in real-time or off-line (Zhang et al., 2007). A service provider commits to complete the tasks and provide value to a service consumer during the service’s lifecycle. Both sides share a common goal of keeping a healthy, long-term trust with efficient and valuable services (Zhang et al., 2007).

Services can be realized in different ways, represented by corresponding service operation models. Traditionally, services are provided in an end-to-end service operation model, meaning that service providers deliver services directly to their end users (i.e., service consumers). Leveraging the recent Information Technologies,
services can now be delivered in several novel approaches, such as hosted service model, business process outsourcing, data-centered outsourcing, cloud computing outsourcing, and services through online broker agencies. These approaches intend to enhance customers’ service experiences and enhance service providers’ productivity (Diagneau, 2012).

From a technical perspective, the term service has been used to refer to any software function that carries out a business task, provides access to files, or performs generic functions like authentication or logging (Diagneau, 2012).

In modern software engineering, service-orientation is about grouping a company's capabilities into well-defined and scoped services. A service should consist of a collection of capabilities that are grouped together because they relate to a functional context established by the service (Erl, 2008).

Chengjun (2008) explains that a service has two levels of granularity: composite service and simple service. A composite service contains a set of services and specifies the choreography in which constituent services are “choreographed” to deliver business processes. A simple service only contains a business goal.

2.1.3.2 Service–Oriented Computing and SOA

Newcomer and Lomow (2005) describe Service-Oriented Architecture (SOA) as follows: “A Service-Oriented Architecture is a style of design that guides all aspects of creating and using business services throughout their lifecycle (from conception to retirement)” (p. 13). Fernandez and Yoshioka (2011) define SOA as “an architectural style in which a system is composed from a set of loosely coupled services that interact
with each other by sending messages (packets of data)” (p. 1). SOA establishes an architectural model that aims to enhance the efficiency, agility, and productivity of an enterprise by positioning services as the primary means through which solution logic is represented in support of the realization of strategic goals associated with service-oriented computing (Erl, 2008).

As a form of technology architecture, an SOA implementation can consist of a combination of technologies, products, APIs, supporting infrastructure extensions, and various other parts. The actual form of a deployed Service-Oriented Architecture is unique within each enterprise (Erl, 2008). Moreover, service-orientation is about composing services to perform complex tasks required by service consumers. A common practice in service-orientation is the creation of a composite service by combining a set of other services. The orchestration of services to construct a new service requires several service interactions. This means that the construction of a composite service can be a complex and time-consuming task (Monsieur, Snoeck, and Lemahieu, 2009).

2.1.3.3 What Is a Service Pattern?

Fki, Jmaiel, and Dupuy (2012) define a service pattern as an abstract service which features a generic and reusable description. It presents the typical ways of composing services to achieve certain goals. Also, a service pattern generalizes the commonalities of a group of concrete services and provides a certain level of abstraction over these services.
Fki et al. (2012) go on to describe the similarities and differences between service patterns and design patterns, explaining that both are proven solutions to recurring problems and are designed to reuse previous experience. But design patterns are focused on reusing expert experience in the design phase, while service patterns are designed to facilitate service compositions in the processing phase (Fki et al. 2012).

Service patterns can be composed to build complex processes. They are design-time components that are predefined by a domain expert whose knowledge is key to the design of service patterns (Fki et al. 2012).

2.2 Related Work - Design Pattern Implementations as Object-Oriented Frameworks

There are several definitions of a software framework. Some of these definitions describe the structure of a framework, while the others describe its purpose. GoF define a framework as “a set of cooperating classes that make up a reusable design for a specific class of software” (p.26).

Ralph Johnson (1997) gives two definitions of a framework; the first states that “A framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact” (p. 39). The second one is that “a framework is the skeleton of an application that can be customized by an application developer” (p. 39). Johnson’s first definition describes frameworks from a structural viewpoint, while his second one sheds some light on their purpose.

Clearly, software patterns and frameworks go hand in hand in the design and development of software applications. They have proven to have most of the good
qualities that make software applications flexible and reusable. Levine and Schmidt (2019) argue that while developing software is hard and developing reusable software is even harder, proven solutions include patterns and frameworks. Levine and Schmidt proceed to compare patterns and framework in terms of their support for reuse; they state that patterns support reuse of software architecture and design, while frameworks support reuse of detailed design and code (Levine and Schmidt, 2019).

In the context of object-oriented programming, a framework is a set of abstract and concrete classes that collaborate with each other, and contain their interface definition of instances (Chen and Chen, 1994). These classes are constructed as reusable components for some domain. Designers and developers can customize the framework to a particular application by inheriting from these instances of classes to create new ones and composing instances of these classes (Chang, Lu, Chu, Hsueh, and Koong, 2009; Jacobsen, Kristensen, and Nowack, 1997). In short, a framework strives to provide an environment to support all activities involved in software development.

In addition to the abstract extendable group of classes, an OO framework usually comes with a component library that contains concrete subclasses of the abstract classes in the framework (Johnson, 1997). This allows all generic reusable functionality of a framework to be grouped together in one place. Such a generic reusable functionality of frameworks is represented by the unrealized, or partially realized, group of abstract super classes. Software applications add concrete classes to realize these unrealized abstract super classes of a framework.

Regarding the reusability of frameworks, Srinivasan states that Object-Oriented frameworks provide an important enabling technology for reusing both the architecture
and the functionality of software components (Srinivasan, 1999). Srinivasan proceeds to say that frameworks are particularly important for developing open systems, where both functionality and architecture must be reused across a family of related applications. The reusability aspect of frameworks is a cornerstone of their design and implementation. In fact, a framework can be said to be a software concept that contains both software component reuse and design reuse (Chang, Lu, Chu, Hsueh, and Koong, 2009).

By contrasting the contexts and uses of frameworks and patterns, it is clear that frameworks are usually used on problems within a specific domain, while patterns are used to solve general problems (Chang, Lu, Chu, Hsueh, and Koong, 2009). Further, patterns facilitate design reuse, while frameworks are an intermediate form, that enable both code and design reuse (Johnson, 1997). So, it can be stated that patterns are more abstract than frameworks and that frameworks are at a different level of abstraction than software patterns. The role that patterns play in the reusability of software systems and applications is well known. The reusability requirement of frameworks dictates that for a framework to be successful, it has to be built using software patterns. In general, a single framework will contain many software patterns (Johnson, 1997).

Consequently, the use of mature software patterns in the analysis and design can lead to an easier maintainable, flexible, and extensible framework. A framework that addresses the application using software patterns is far more likely to achieve a higher level of design and code reuse than the one that does not (Shalloway and Trott, 2001; and Yixing and Yaowu, 2007). At the same time, the users can study the framework better with the use of patterns (Shalloway and Trott, 2001). In fact, framework designs can be
discussed in terms of design pattern concepts, such as participants, applicability, consequences, and trade-offs, before examining specific classes, objects, and methods (Srinivasan, 1999). Levine and Schmidt conclude that sophisticated frameworks typically embody dozens of patterns (Levine and Schmidt, 2019). On the other hand, the implementation of software patterns using a framework makes it clear that those patterns work together, and that patterns really define roles (Christensen, 2004). In fact some software patterns have already been used as the main skeleton for OO software development frameworks. Take, for instance, the Model/View/Controller software architectural pattern. Popular programming languages like Java, C#, Python, Ruby, and PHP have MVC frameworks that are used in web application development (Wikipedia-MVC, 2019).

Software design patterns, on a rather smaller scale, can be implemented as frameworks. Such a framework consists of the abstract classes of the design pattern and some of the design pattern’s concrete classes that implement some generic logic. The rest of the concrete application-specific logic can be introduced by some concrete classes that the application developers implement to realize the abstract classes provided by the framework, thus realizing the overall functionality of the subject design pattern. As an example of how such a category of frameworks may be built, we use the GoF Observer design pattern. The reason we chose this specific design pattern is because it consists of some classes that contain both some generic and some application-specific logic, plus another set of classes that should contain logic that is purely application specific. Figure 2.2 shows the structure of the Observer design pattern as introduced by the GoF. Before we explain how the Observer pattern might be implemented as a reusable OO
framework, we will give a brief description of the purpose that this design pattern serves and the way it works.

The Observer design pattern is usually used whenever the state of one or more objects depends on the state of another object. The Observer design pattern provides a good design for the situation when one or more objects – we will call them the first set of objects – depend in their state and behavior on one or more other objects – we will call them the second set of objects. This means that the first set of objects needs to observe the second set and whenever the state of the second set of objects changes, the first set of objects changes its state accordingly. The Observer design pattern calls the first set of objects the Observers, and the second set of objects the Subjects. The Observer objects register with one or more Subject objects so that every time the state of the Subject object changes, its registered Observers are notified so that they can update their states accordingly. Figure 2.2 shows the inner details of the Observer design pattern. It is worth mentioning that in most cases the second set of objects is actually one Subject object, but that does not mean that the Observer design pattern limits the number of Subject objects to only one object.

A possible implementation of an OO design pattern as an OO framework would be to first identify the abstract and concrete layers of the design pattern. This step enables the separation between the group of classes, methods and attributes that belong to the framework from those that are application specific and therefore should be left to the observer design pattern framework users to implement according to their application needs. In the context of the Observer design pattern this separation step results in the abstract class “Subject” and the interface “Observer” being identified as the abstract
Figure 2.3 shows how the two layers, abstract and concrete, are identified and separated by the dashed line.

By inspecting the methods defined by the abstract “Subject” class, we can tell that the programming logic in the three methods, Attach, Detach, and Notify, can be considered generic and can be reused regardless of the actual application that is using this framework. The same applies to the Update method defined by the “Observer” interface. The Observer design pattern dictates that each observer class must implement an Update method to enable their state data update every time the state of the Subject class changes. Therefore, it is safe to say the implementation of the Observer design pattern as an OO framework should consist of the implementation of the above listed methods. Users of such a framework need to realize the functionality promised by the Subject abstract class and the Observer interface by extending the framework and implementing their concrete application-specific Subject and Observer classes.
Chapter 3

Motivating Example

As stated in Chapter 1, the main objective of this research is to put together a general methodology idea for implementing software patterns from different fields and having different granularity as pattern services. We must take into account that there are several types of software patterns, e.g. OO design patterns, architectural patterns, analysis and enterprise architectural patterns, and SOA patterns. An idea of a general methodology that will help turn this disparate set of patterns into services must accommodate all the differences this set of pattern types have. Differences include patterns having different granularities, and abstraction, application, or domain specificity levels.

In this section we are going to explain the proposed methodology for implementing software patterns as pattern services. We will first go through the steps of applying the methodology using a simple example application that uses the Observer design pattern. Later we will generalize the steps we performed to implement the Observer design pattern as a pattern service and explain how the idea of a general methodology works.

3.1 Introduction

To make it easier to explain the stages of implementing a software pattern as a pattern service and to build towards an idea of a general methodology for doing so, we adopt a step-by-step approach of showing how a software pattern is gradually implemented as a pattern service.
We use a stock market trading application as a motivating example. We are not interested in the trading mechanisms and algorithms of the application; rather, we are interested in the component that receives stock market data and dispatches it to the Graphical User Interface (GUI) component of the application. The data obtained from the stock market may be displayed in several formats and may be used for different purposes. For the purpose of this demonstration, in our example application we assume that there are two types of users of the stock data. The first user displays the data in the form of tables, while the second user displays the data as charts.

Figure 3.1 shows a very basic and simplified general view of what a stock market data inquiry application may look like.

![Stock Data Inquiry Application Diagram]

**Figure 3.1: A simplified general view of a stock data inquiry application**

As can be seen from Figure 3.1, the application pulls stock data from a Google service we call *Google stock ticker service* (See Glossary for definition). The stock data manager component is the part that periodically pulls stock data from the Google stock ticker service and dispatches it to the stock data displayers.
The implementation of the stock data manager and the stock data displayers are a good fit for the Observer design pattern introduced by the GoF (See Figure 2.2 and Figure 2.3). The Observer design pattern is usually used whenever the state of one or more objects depends on the state of another object.

In our example application we have two stock data displayers that depend on the data provided by the stock data manager, which in turn gets them from the Google stock ticker service. When we superimpose the Observer design pattern on our example application we can see a correspondence between the stock data manager in our example application and the Subject component of the pattern. Furthermore, the stock data displayers correspond to the pattern’s Observer component. Now we know which part of the example application can be the Subject and which ones can be the Observers.

If we are to implement the example application as a service-oriented system we will turn the stock data manager into a service that represents the Subject, and the two stock data displayers will be the services that represent the Observers. Then we establish the interaction between the components of the pattern in the form of service calls between the Subject service and the Observer services. Figure 3.2 shows how the example application can be implemented as a service-oriented system.

As depicted in Figure 3.2, the processing starts with step (0) which occurs at the initialization stage. In step (0), stock data displayer services “Observers” register themselves with the stock data manager service “Subject”.

44
This is done by the two Observers invoking the “Attach” service on the Subject. The rest of the operations take place during runtime. They are executed in the following sequence.

Step (1): The stock data manager pulls the stock market data from Google stock ticker service. The “SetState” service of the stock data manager is to be programmed to periodically pull the stock data from the Google stock ticker service.

Step (2): Every time the stock data manager pulls some stock data, it compares it to its current state data. If the pulled stock data are different from the current state data of the stock data, the stock data manager first updates its state data, then it invokes the notify service to alert the registered Observers of the recent stock data update.
Step (3): The notification action, originated from the stock data manager, triggers the “Update” services on the stock data displayers. This results in a service call to the “GetState” service – provided by the stock data manager. The service call to the “GetState” service returns stock data of interest to the stock data displayers. The stock data displayers will each manipulate and display the data according to the way they are implemented.

While the registration process for each Observer is usually done once at the start of their interaction with the subject, the rest of the interaction typically continues to occur in the same sequence as long as the service-oriented example system is still running.

3.2 Implementing the Observer Design Pattern as a Pattern Service

One of the essential qualities of a pattern is to have a generic part or aspect that can be applied in different contexts. Its general applicability is really what makes it a pattern. A complementary part of the pattern is its context or application-specific part. This application-specific part enables the same pattern to be used in disparate contexts and applications without breaking the pattern’s generic part and purpose.

What can be noticed from the structure of the Observer design pattern, and the majority of software patterns for that matter, is that it consists of two layers. The first and top layer contains abstract classes that define the behavior of the pattern and the generic functions that the pattern can or should perform regardless of the context or application that uses it. This layer consists of one or more abstract classes that guide the implementation of the pattern and define the relationships and interactions between its
objects. The second layer of the Observer pattern shown in Figure 2.3 depicts the application-specific part of the pattern. This part usually consists of one or more concrete classes that implement the application-specific behavior of the pattern. Concrete classes usually inherit their relationships and interaction rules from their parent – abstract – classes defined in the top layer of the pattern.

Building on this dissection of the common structure of most software patterns, we will start demonstrating the steps of implementing the Observer design pattern as a pattern service. Then we will generalize the approach used to propose the general methodology idea for implementing software patterns as pattern services.

Before we delve into the details of implementing a software pattern as a pattern service, let us first define what a pattern service is.

**Definition 3.1: Pattern Service**

Given a software pattern, a pattern service is the implementation of the subject software pattern and all functionalities of the subject software pattern in the form of a reusable and configurable software service.

As can be seen in Figure 2.3, the Observer pattern consists of two layers: the top layer which contains the abstract classes and the bottom layer which consists of the concrete classes. The first step in implementing the Observer pattern as a pattern service is to implement the generic classes in the abstract layer and their relationships and interaction rules as the major part of a generic pattern service. This part will enforce all pattern rules and coordinate all the interactions between its classes and objects.
Next we give a formal definition of the generic pattern service, also referred to – in this thesis – as the generic part of a pattern service.

**Definition 3.2: Generic Pattern Service**

Given a software pattern, a generic pattern service is the implementation of the application-independent functionalities of the subject software pattern in the form of services provided by the pattern service that implements the subject software pattern.

Since the concrete layer of the Observer pattern, and most patterns for that matter, will naturally be made up of classes that have application-specific logic, the implementation of the concrete classes is left to the application developers, who will use the Observer pattern service as part of their application. The application developers will need to develop the group of concrete classes that belong to the pattern as a set of services. We give the name “Application-specific Services” (AppSSs) to this set of concrete class services.

Next we formally define the Application-specific Service.

**Definition 3.3: Application-specific Service**

Given a software pattern with application-dependent functionalities and the generic pattern service implementing the software pattern, an application-specific pattern service extends the generic pattern service by implementing interfaces of application-dependent functionalities of the software pattern in the form of services, to form the overall pattern service that implements the software pattern.
Figure 3.3 gives the general structure and interaction sequence between the generic Observer pattern service and the concrete application-specific services in the context of our stock data inquiry example.

The Observer pattern service shown in Figure 3.3 implements the abstract layer of the Observer design pattern. This includes the abstract subject class and the Observer interface shown in Figure 2.3. It also provides services to facilitate the interaction between the Observer AppSSs “Stock Data Displayers” and the Subject AppSS “Stock Data Manager.”

As explained earlier in this section, the interaction begins with the registration process. The registration process is executed during the application initialization phase. It is denoted by step (0) in Figure 3.3.
Step (0): In the proposed Observer pattern service, both the Subject and the Observer AppSSs need to register with the generic Observer pattern service. The pattern service takes care of the association between the Observers and the Subjects. In our example case there is only one Subject, but in a generic Observer pattern service, provision for multiple Observers observing multiple Subjects must be made.

Step (1): After the registration process, the stock data manager starts pulling data from the Google stock ticker service.

Step (2): When the stock data manager’s state changes because of the data it pulled in step (1), it notifies the Observer pattern service of the new data.

Step (3): The Observer pattern service calls the Update service of each stock data displayers to trigger its state change.

Step (4): The invocation of the Update services results in the stock data displayers asking the Observer pattern service to get the required data on their behalf from the stock data manager.

Step (5): Upon the stock data displayers’ request, the Observer pattern service calls the Get State service provided by the stock data manager to get the data for the Observers. The Observer pattern service then forwards the data it got from the stock data manager to the stock data displayers. That is basically how the interactions in Figure 3.3 are performed.

In addition to the generic part, the AppSSs are basically an important part of the pattern. This means that the generic part must be able to interact and integrate with the AppSS part of that pattern. Therefore, our generic Observer pattern service must provide the
means for integrating all the required AppSSs with its generic part. This can be done by equipping the generic pattern service with a registration capability that application developers can use to register their application-specific services with the generic pattern service.

**Definition 3.4: Application-specific Service (AppSS) Binding**

Given a generic pattern service, AppSS binding is the establishment of the integration between the generic pattern and the application-specific services that implement the application-specific logic pertaining to the subject software pattern. It is done by the generic pattern service providing AppSS registration services and data structures.

Figure 3.4 shows how the binding of the Observer pattern AppSSs is done. As can be seen in Figure 3.4, two registration lists are added to the generic Observer pattern service. One of these lists stores service references to the subject component of the pattern service, which is the stock data manager in our case. The second registration list holds service references to the Observer components of the pattern service, which – in our example – are the stock data displayers. Once the registration process is put in place, it is time for the next important step in implementing the Observer design pattern as a pattern service.

The next step, of course, is to enable the communication between the generic Observer pattern service and the AppSSs registered with it.

In order for the generic Observer pattern service to be able to interact with the registered AppSSs, an interface and some delegation code need to be created. This can be done through configuring the pattern service. Using a configuration file a service can be
invoked to generate the necessary interfaces and delegation code that will enable the interaction between the generic pattern logic and the registered AppSS components.

Next we define the configuration step of a generic pattern service to convert it into an application-specific one.

**Definition 3.5: Application-specific Service Configuration**

Given a generic pattern service and configuration information on required application-dependent service interfaces, application-specific service configuration is the process to
generate the corresponding application-specific pattern service including application-
specific service interfaces and supporting service implementation components.

The processes of integrating the required AppSSs, and applying the needed
configurations, to make the generic pattern service work according to the needs of the
software application at hand, actually convert the generic pattern service into an
Application-specific Pattern Service (AppSPS). The resulting pattern service is no
longer generic because it is now tailored to work in the specific context of the
application at hand

**Definition 3.6: Application-specific Pattern Service (AppSPS)**

The Application-specific pattern service is the pattern service created as a result of
binding application-specific services and applying application-specific configurations to
a generic pattern service.

With the generation of the delegation code and the application of any other needed
configurations, the Observer pattern service is now ready to be used in our example
service-oriented software system. Figure 3.5 puts the Observer pattern service in the
context of the service-oriented stock data inquiry system and shows how the data will
flow between the components within the Observer pattern service. Notice the presence
of the necessary interface and delegation code that make the communication between
the generic pattern service part and its complementing AppSSs possible.
Figure 3.5: Stock inquiry system using the Observer pattern service

The sequence diagram in Figure 3.6 further explains the sequence of the interactions inside the service-oriented stock market inquiry system.
3.3 General Pattern Service Methodology Overview

Based on the common generic characteristics that software patterns share and by applying the steps demonstrated in the motivating example section above, we conclude that these same steps can be adopted to form a general methodology idea for implementing the majority of software patterns, design, architectural, analysis, and SOA, as pattern services. These pattern services can be used as building blocks for service-oriented software systems.

In accordance with the example explained above, the steps of the proposed methodology to create a generic pattern service from a software pattern can be summarized in the following list of actions:

1- Study the subject software pattern and identify the abstract and generic layer that can be implemented as a generic pattern service.
2- Implement the structure of the pattern and the generic functionalities, and provide for AppSS registration, lists, interfaces, and registration service implementations.

The creation of the generic pattern service is the first main step in the proposed methodology. The second main step consists of the processes involved in implementing the resulting generic pattern service as a pattern service tailored and configured for use with a certain application. The processes to be performed as part of this second main step can be summarized as follows:

1- Study the structure of the generic pattern service and identify the AppSSs that need to be added and the configuration that needs to be applied to the generic pattern service in order to adapt it to work with the target application.

2- Convert the generic pattern service into an AppSPS.
   a) Implement the AppSSs identified in the last step and bind them to the generic pattern service by registering them using the provided AppSS registration services.
   b) Prepare and run a configuration file that adds any configuration to the generic pattern service and helps create the necessary application-specific interface and delegation service code to facilitate interaction between the generic pattern service and the added AppSSs.

3- Create two application services that facilitate the interaction during runtime between the application and the now application-specific pattern service (See Glossary for definition). The first application service acts as the client that takes care of accessing the pattern service on behalf of the software application, while the second application service enables setting and switching between the AppSSs attached to
the pattern service. Accomplishing this last requirement should make the application ready for instance creation and deployment.

4- The Pattern-based Application Instance Creators (PbAppICs; See Glossary for definition) study the released software application, prepare configuration files that configure the application to their needs, create required instances of the application, and run the configuration files against the created application instances.

The deployment of the application instances to the Pattern-based Application Instance End Users (PbAppIEUs; See Glossary for definition) marks the final stage of the proposed methodology.

To describe the general methodology idea, we adopt the same approach that we used in explaining the steps of implementing the Observer design pattern as a pattern service. The general methodology idea we propose starts with a software developer with deep technical knowledge of software patterns selecting a candidate pattern that he or she wishes to implement as a generic pattern service. We call this software developer a Pattern Service Creator (PSC; See Glossary for definition). After selecting a software pattern, the next step the PSC takes is to extract the abstract and generic logic and structure of the software pattern and implement them as a generic pattern service. Figure 3.7 depicts the first step of the general methodology idea to implement a software pattern as a generic pattern service. Once the abstract and generic part of the pattern has been implemented, the PSC needs to provide the capability of attaching or registering AppSS components to the generic pattern service. This is done by providing a registration service that facilitates AppSS registration and a registration list – as many lists as needed – that stores references to the registered AppSSs. Also, in the case that
more than one alternative AppSS is registered, an interface that allows for the selection between available AppSSs is required. The PSC needs to implement all of this so that the stages of pattern service creation that follow can be performed. Although the role of the added service capability is to deal with AppSS components, it is still considered to be part of the application-independent logic because it is an essential part that enables the integration between the generic pattern service and its AppSSs.

Figure 3.7: First step of the general methodology idea

Figure 3.8 shows the addition of the registration and setting interface, its implementation, and the registration list to the generic pattern service.

Figure 3.8: Second step of the general methodology idea

With the completion of the second step, the creation of a generic pattern service is complete. The resulting generic pattern service now contains implementation of the
pattern structure and generic logic that enforce the rules of the pattern, plus all the necessary functionality to attach and select from the AppSSs that are used by the pattern’s generic structure, logic and rules to perform its task.

At this point, the necessary documentation that explains how the generic pattern service can be used is released along with the generic pattern service implementation to the pattern service users.

Figure 3.9 gives the complete general structure of a generic pattern service.

![Figure 3.9: General structure of a generic pattern service](image)

So far we have shown the general methodology idea to create the generic part of the pattern service out of a software pattern. Now is the time to add the application-specific parts to the generic pattern service, which converts it into an application-specific pattern service. An application-specific pattern service is basically a generic pattern service with added application-specific components, logic, and configuration.
As we mentioned earlier, a software pattern is made up of two parts, a generic or application-independent part and a non-generic or application-specific one. Due to the variety of the applications that might use a pattern service, it is neither logical nor possible to implement application-specific logic in a generic pattern service. That is why those who use a generic pattern service in their applications are required to implement the portion of the pattern that is deemed application specific.

The attachment of application-specific services (AppSSs) and the generation of necessary interfaces and delegation code turn a generic pattern service into an application-specific pattern service. That is because the addition of the application-specific material and the application of any configuration to the generic pattern service makes it no longer generic. We refer to the application developer, or team of developers, that convert a generic pattern service into an Application-specific Pattern Service (AppSPS) as the Application-specific Pattern Service Creator (AppSPSC). While we refer the developer or team of developers that create the necessary application-specific Services (AppSSs) as the Application-specific Service Creator (AppSSC). The developer or team of developers that use the readily configured AppSPSs in their applications are referred to as the Pattern-based Application Creator (PbAppC). It is important to note that the PbAppCs can assume the roles of other developer roles such as the AppSSC and AppSPSC. That is why throughout this thesis we may assign the task of AppSPSCs and AppSSCs to the PbAppCs. With that being noted, the principal role of the PbAppCs is to build Pattern-based Applications (PbApps) using AppSPSs and some other application services that they develop according to the needs of their applications. In addition to combining AppSPSs and
application services to duils PbApps, PbAppCs may perform one or two other tasks. The first task is to create the required Application-specific Services (AppSSs) to provide the application-specific functionality of the subject pattern service. During the execution of this task, the PbAppCs can be referred to as Application-specific Service Creators (AppSSCs). The second task that PbAppCs need to perform is to attach the created AppSSs to the generic pattern service and apply required configuration to convert the generic pattern service into an application-specific one. When performing this task, we refer to the PbAppCs as Application-specific Pattern Service Creators (AppSPSCs).

When a PbAppC decides that a generic pattern service in a certain repository is what an application needs to perform a certain functionality, the PbAppC starts by studying the generic pattern service documentation. This enables the PbAppC to understand how it works, the type of configuration that can be applied to it, and what AppSSs need to be implemented to complement its generic implementation and turn it into a pattern service that is tailored specifically for their application. Once the PbAppC studies the documentation and understands how to use the generic pattern service, the next step is to implement the AppSSs needed. As we explained earlier in this chapter, the generic pattern service is already equipped with the proper interface and implementation for the registration and manipulation of AppSSs. The PbAppC can use this interface to register and set the AppSSs implemented. Figure 3.10 shows the first step of the methodology to convert a generic pattern service into an application-specific one. It shows how the PbAppCs register the AppSSs they have developed and set the default one to use.
The second step in the methodology for converting a generic pattern service into an application-specific one is to supply the configuration file that should contain configuration information that can be applied to the generic pattern service. An essential part of such a configuration file is the set of instructions/information that assist in the generation of the necessary interfaces to deal with the registered AppSSs and the delegation code that delegates requests and method calls to the proper AppSS. The PbAppC are the ones responsible for supplying the configuration file and executing the service or set of services that reflect the contents of the configuration file on the pattern service and generate the required implementation code.
Figure 3.11 depicts how the configuration step is done and what components it should add to the now partially implemented application-specific pattern service.

Figure 3.11: Second step of converting a generic PS into an AppSPS

It is worth noting that while the second step in the process of converting a generic pattern service into an AppSPS is mandatory for the PbAppCs, the first step may be executed by a different developer or team of developers (AppSSCs). That is why the execution of the second step is not dependent on the completion of the first. PbAppCs can perform the second step before AppSSCs finish creating the required AppSSs, as
long as the interfaces for the service methods in both the generic pattern service and the AppSSs that will be created are fixed and agreed upon. After processing the configuration file and the generation of AppSS interface and the insertion of the delegation code, the conversion process of the generic pattern service into an application-specific one is complete. At this stage the application-specific pattern service should be ready to be plugged into the target application that it is tailored for. The structure of the application-specific pattern service is shown in Figure 3.12.

![Application Specific Pattern Service Diagram](image)

Figure 3.12: General structure of an AppSPS
Steps 1 and 2 of converting a generic pattern service into an application-specific pattern service are performed by the application creator PbAppC. We refer to this conversion stage as the first binding stage or the application creation/integration stage. That is because all the processes involved in this stage take place during the creation of the software application that uses the generic pattern service or during the establishment of the integration between the application-specific components and the generic pattern service. Figure 3.13 presents the three steps involved in the application creation/integration stage, i.e. registration, configuration, and code generation.
After performing the steps involved in the first binding stage, the pattern service should by now be fully integrated with the software application. After the software application is compiled and tested, it is then released for deployment and usage.

We call the next set of users that configure and deploy the released software application the Pattern-based Application Instance Creators (PbAppICs). The PbAppICs take the released software application, configure it according to their needs, and deploy as many instances of that application as required.

Figure 3.14: PS instance and application instance binding at configuration time
What interests us in the second binding stage with regard to the pattern service is the configuration that may indirectly be applied to the pattern service. The PbAppICs may include in their application configuration files some instructions that change the default selections that are related to the pattern service and the AppSSs attached to it. The “Set” arrow in Figure 3.14 depicts the provision for an application instance creator to configure the pattern service during the application instance creation binding stage.

Once application instances are created, configured, and deployed, they become ready to be used by their end users. We refer to this set of users as the Pattern-based Application Instance End Users (PbAppIEUs). The third binding stage between the software application and its constituent pattern services takes place during the execution of the pattern-based application.

The runtime interaction between the application and a pattern service can take two forms. The first form is the application access to the functionalities provided by the pattern service. This is achieved through an application service that acts as the client to access the pattern service. The second form of application and pattern service interaction is when the flow of the application logic requires the selection or exchange of the AppSS – attached to the pattern service – to be used. This is done through the use of an application service that invokes the AppSS setting routines of the pattern service. We call this stage of application and pattern service interaction the third binding stage or the runtime binding stage. Figure 3.15 shows the two forms of interactions between an application and its constituent pattern service during runtime.
The explanation of the third binding stage concludes our step by step walk through the components and processes suggested by the general methodology idea to implement a software pattern as a pattern service.

3.4 Summary

In this chapter we introduced the proposed methodology to implement software patterns as pattern services. We first used the Stock Data Inquiry application as an example application to demonstrate how to apply the steps of the methodology on the Observer design pattern. Then we generalized the steps of the proposed methodology to show
how the methodology may be applied to implement any software pattern as a pattern service.

In Chapter 4 we describe the conceptual methodology for implementing design patterns as pattern services. We explain how a design pattern can be split into an abstract layer and concrete one to enable the implementation of the abstract layer as a generic pattern service, and the concrete layer as application-specific services. Then we classify the types of pattern services that can be implemented based on the 23 GoF design patterns. Such a classification is based on the types of interfaces the pattern service will have and the degree of application dependence of those interfaces. Examples are given to help explain each classification category.
Chapter 4

Conceptual Methodology for Design Patterns Services

In this chapter we explain the approach we take in applying the pattern as a service methodology to implement the 23 software design patterns presented in GoF as pattern services. We conceptually describe the group of steps or actions that need to be performed in order to implement design patterns as reusable pattern services. In this chapter we talk about such a conversion process from a conceptual point of view, while we dedicate the next chapter to explaining how the conceptual model – created by performing the actions described in this chapter – can be used to implement design patterns as pattern services. First, we define the conceptual model of a pattern service.

Definition 4.1: Pattern Service Conceptual Model

The conceptual model of a pattern service consists of the set of application-independent and application-dependent service interfaces that the pattern service provides to clients and registered AppSSs, classification of the pattern service based on its service interfaces, and specification of realization of provided services through service interfaces.

Building a conceptual model requires that all the components and processes involved are identified and organized based on their sequence of creation and execution. The pattern service conceptual model is no different. The right conceptual model for a pattern service will ensure that the methodology of implementing a software pattern can be organized so that the implementation stages execute smoothly and the final product – pattern services in this case – works as anticipated and planned.
Definition 4.2: Conceptual Methodology of a Pattern Service

Given a software design pattern, the conceptual methodology is the process to create the conceptual model of a pattern service that implements the given design pattern. The conceptual methodology consists of the following phases: abstract layer identification, concrete layer identification, application-independent service interface identification and creation, application-dependent service interface identification and creation, pattern service classification, and creation of conceptual service realization specification using service interfaces.

The conceptual methodology of implementing design patterns as pattern services is very similar to the general methodology idea presented in Chapter 3. The study of the subject design pattern, abstracting the generic logic and identifying the application-specific parts of the design pattern, and the creation of a generic pattern service are a few of the common aspects. The algorithm, shown in Figure 4.1, gives the sequence of actions that the methodology prescribes for creating the conceptual model of the pattern service that implements a design pattern.

Algorithm 4.1: Conceptual Methodology Process

The following list of tasks describes in more detail, the steps prescribed by the conceptual methodology algorithm.

1- Identify abstract layer

Purpose: Separate the abstract layer and all its classes and interfaces of the subject design pattern.
**User activity/task:** Study the design pattern to identify the abstract layer, which usually contains most of the generic logic of the design pattern.

**Outcome:** The identification and separation of the abstract layer of the subject design pattern with all its classes and interfaces.

2- **Identify concrete layer**

**Purpose:** Separate the concrete layer and all its classes.

**User activity/task:** Study the design pattern to separate the application-specific logic – which is usually in the concrete layer – from the generic logic of the design pattern.

**Outcome:** The identification and separation of the concrete layer of the subject design pattern with all its concrete classes.

3- **Identify generic logic**

**Purpose:** The identification and separation of all logic that is not specific to any application context and can be applied universally to any application.

**User activity/task:** Study the logic and interactions in both the abstract and concrete layers of the design pattern, and then separate the generic logic that is not specific to the application using the subject design pattern and assign it to the generic layer of the target pattern service. This logic can be divided into two types, application-independent and application-dependent logic. The application-dependent logic will need further configuration later on to make it work with the application at hand.

**Outcome:** The identification and separation of all the generic logic of the subject design pattern, which belongs to the generic layer of the target pattern service.
Figure 4.1: Algorithm showing the conceptual methodology process
4- **Identify application-specific logic**

**Purpose:** The identification of the logic that is not applicable to all applications that use the subject design pattern; rather it is specific to the application that uses the subject design pattern.

**User activities/task:** Study the subject design pattern and, using the logic in the abstract and concrete layers of the design pattern, separate the logic that is application specific and assign it to the application-specific layer of the target pattern service.

**Outcome:** The identification and specification of the application-specific logic that belongs to the application-specific layer of the target pattern service.

5- **Specify the application-independent interface set of the generic layer**

**Purpose:** The specification of the set of application-independent interfaces, which the generic layer of the pattern service provides for clients and other components of the pattern service to interact with the generic layer of that pattern service. This set of interfaces is generic and reusable regardless of the application that uses the pattern service, hence the name – application independent.

**User activities/task:** By studying the subject design pattern and based on its abstract layer interface, the generic logic involved, and any accommodations need to be made for the resulting pattern service to work, decide on the type and details of the application-independent interface that the generic pattern service offers for the interactions with the application-specific layer and any client services needed to put the pattern service to work.

**Outcome:** A set of application-independent interfaces that facilitates the integration and interactions with the logic in the generic layer of the pattern service.
6- Specify the application-dependent interface set of the generic layer

**Purpose:** The specification of the set of application-dependent interfaces, which the generic layer of the pattern service provides for clients and other components of the pattern service to interact with the generic layer of that pattern service. This set of interfaces may change according to the requirements of the application that uses the pattern service, hence the name – application dependent. This set of interfaces is the set associated with the configurable service logic added to the generic pattern service. Most of this logic is usually used to integrate the logic in the generic layer with the logic in the separate services that implement the application-specific logic.

**User activities/task:** By studying the subject design pattern and based on its abstract layer interface, the generic logic involved, and any accommodations that need to be made for the resulting pattern service to work, decide on the type and details of the interface that is application dependent, which the generic pattern service offers for the interactions with the application-specific layer and any client services needed to put the pattern service to work.

**Outcome:** A set of application-dependent interfaces that facilitates the integration and interactions with the logic in the generic layer of the pattern service.

7- Specify application-specific interface set

**Purpose:** The specification of the set of interfaces that the application-specific layer of the pattern service provides in order to establish the integration and facilitate the necessary interactions with the generic layer of the pattern service to realize the pattern service functionality.
**User activities/task:** Using the application-specific logic identified in step 4, the design pattern application-specific interface, and any accommodations that need to be added, specify the application-specific interface sets. This set of interfaces is the one associated with the application-specific logic and services that will be implemented separately from the generic pattern service.

**Outcome:** A set of interfaces that help facilitate the interactions between the generic layer of the pattern service and its application-specific one to realize the overall functionality of the pattern service according to the requirements of the application that is it.

8- **Specify runtime sequence**

**Purpose:** The specification of the sequence of events and interactions in their proper order to put the pattern service to work and make it deliver the exact functionality prescribed by the subject design pattern.

**User activities/task:** As a last step in the process, specify the sequence of events, pattern service components involved, and the client service that use the pattern service. Also specify the nature of the interactions that take place during the execution of the pattern service, their input/output data, and what purpose each interaction serves.

**Outcome:** Clear specification of the set of interactions between the client and the pattern service, and the internal interactions between the generic and application-specific layers of components that make up the pattern service.
4.1 Identifying Pattern Layers

When approaching a design pattern with the intention to study its structure and to be able to envision how it can be implemented as a pattern service, we should first bear in mind that most OO design patterns, and other software patterns for that matter, consist of three parts:

a) **Abstract layer** - This layer usually contains abstract classes that have public methods with interface signatures and logic that are mostly generic and can be applied regardless of the context and kind of application at hand.

**Definition 4.3: Application-independent Method in a Design Pattern**

Given a design pattern, the application-independent method in one of the classes of that design pattern is a method whose interface and behavior are independent from the context of the application that uses the design pattern, and whose interface and logic are generic and can be reused by any application.

**Definition 4.4: Application-independent Class in a Design Pattern**

Given a design pattern, the application-independent class in that design pattern is a class whose method interfaces and behaviors are independent from the context of the application that uses the design pattern, and whose interfaces and logic of all the methods contained in that class are generic and can be reused by any application.

b) **Concrete layer** - This layer usually contains concrete classes that mainly have private methods with interface signatures that are application dependent and are dedicated to the logic that may differ based on the context and the application that employs the pattern.
Definition 4.5: Application-dependent Method in a Design Pattern

Given a design pattern, the application-dependent method in one of the classes of that design pattern is a method whose interface and/or behavior is dependent on the context of the application that uses the design pattern, and whose interface and/or logic is non-generic and cannot be readily reused by different applications.

Definition 4.6: Application-dependent Class in a Design Pattern

Given a design pattern, the application-dependent class in that design pattern is a class in which one or more of its method interfaces and/or behaviors are dependent on the context of the application that uses the design pattern, and the interfaces and logic of one or more methods contained in that class are non-generic and cannot be reused by different applications.

c) **Client** – This is the part that is responsible for engaging the pattern and putting it to work. In most cases, the client part is independent and contains code and logic that is mainly application specific. In some cases, other parts of the pattern – such as the concrete layer – can play the role of the client of the pattern.

We begin the process of implementing a design pattern as a pattern service, shown in Figure 4.1, by studying the subject pattern and trying to identify the three different parts we listed above. First, we look for the abstract layer of the pattern and study the classes of the pattern to separate what is abstract or generic and what is application specific. The outcome of this step should be the identification of a class or a group of classes that constitute the abstract layer of the subject design pattern.
Although generic logic of a design pattern may overlap the two layers, abstract and concrete, the majority of design patterns usually have the generic logic in the abstract layer. The generic logic is what makes the generic layer of the target pattern service.

**Definition 4.7: Generic Layer of Pattern Service**

Given a pattern service that is an implementation of a design pattern, the generic layer of such a pattern service is the group of application-independent and configurable application-dependent interfaces and logic contained in the services that implement those interfaces. It also includes any data structures used by the services in this layer.

**Algorithm 4.2: Identification Process of Generic Layer of Pattern Service**

Input: Classes in abstract and concrete layers of design pattern  
Output: Interface set and behavior for PS generic layer  
Process: While (more classes to inspect)  
  If (current class is application independent)  
    Assign to PS generic layer as application independent  
  Else  
    While (more methods in current class)  
      If (method behavior is application independent and  
          method interface is application independent)  
        Assign to PS generic layer as application independent  
      Else  
        If (method behavior is application dependent)  
          Assign to PS generic layer as application dependent by  
          behavior  
        Else if (method interface is application dependent)  
          Assign to PS generic layer as application dependent  
          by interface  
        End if  
      End if  
    End while  
  End if  
End if  
End while

Once we finish with the abstract part of the design pattern, the second step is to focus on identifying the classes that we classified as concrete. These classes usually contain
methods and logic that are purely application specific. Remember that in this second step, we are not interested in client classes; rather, we are only looking to identify the classes that represent the concrete part of the design pattern. The outcome of this step is to identify what interfaces and logic should go in the application-specific layer of the pattern service.

**Definition 4.8: Application-specific Layer of Pattern Service**

Given a pattern service that is an implementation of a design pattern, the application-specific layer of such a pattern service is the group of application-specific interfaces and logic contained in the services that implement those interfaces, which is usually separate from the generic layer and is an integral part of the overall pattern service.

**Algorithm 4.3: Identification Process of Application-specific Layer of Pattern Service**

**Input:** Classes in concrete layer of design pattern  
**Output:** Interface set and behavior for application-specific layer  
**Process:** While (more classes to inspect)  
   If (current class is application specific)  
      Assign to application-specific layer of PS  
   Else  
      While (more methods in current class)  
         If (method behavior is application specific)  
            Assign to application-specific layer of PS  
         End if  
      End while  
   End if  
End while

The third and last step in the design pattern layer identification is to look for any classes that are purely client classes and whose role is to use the pattern by triggering the flow of events during its execution.
Although most of the GoF design patterns conform to the three part rule, the few exceptions can be divided into two categories:

1- **Design patterns that do not have an abstract layer** – This type of design pattern usually has one layer that may contain some generic logic but it also contains some application-specific logic at the same time. The Memento and Singleton design patterns are examples of this category of design patterns.

2- **Design patterns that do not have the client part** – Some design patterns do not employ an explicit pattern client; instead, a certain part of the pattern – usually in the concrete layer – acts as the client that triggers the sequence of actions that the pattern performs. The Mediator design pattern is an example of such a category of design patterns.

**Example:**

We use the Observer design pattern to give an example of how the layer identification procedure can be done. The reason we picked the Observer design pattern for our demonstrative example is because the Observer design pattern contains the three components we introduced above, namely, the abstract layer, concrete layer, and the client. This makes the Observer design pattern a typical case for applying the proposed methodology and a good candidate for a demonstrative example.

The Observer design pattern is one of the behavioral patterns introduced in GoF. Other names given to the Observer design pattern include Dependents and Publish-Subscribe. The intent of the Observer design pattern is to “Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically” GoF (p. 293). The structure of the Observer design pattern...
consists of a class that is usually referred to as the Subject and one or more classes called the Observers. The Observer design pattern defines the relationship between the Subject class and its Observer classes. The Observer classes register with the Subject class as its observers. Each time the state of the Subject class changes, it notifies the registered Observer classes of the change. Then the Observer classes update their states, each according to the relativity of the Subject class data to its own state. Figure 4.2 gives an overview of the structure of the Observer design pattern.

![Observer Design Pattern Diagram]

**Figure 4.2: Observer design pattern**

First of all, let us study the structure of the Observer design pattern so that we can identify the abstract and concrete layers of the design pattern and figure out where the generic and the application-specific logic are located in this design pattern.

As can be seen from Figure 4.2, the pattern has two abstract classes. The first one is the Subject class which provides the generic interface methods that all concrete subject classes must implement in order to deliver the proper functionalities of a Subject. The
second abstract class in Figure 4.2 is the Observer class. This class provides the generic interface method that every concrete observer class must implement so that they can function as observers of the subject classes and update their states accordingly. This layer of classes is clearly abstract and the logic that goes into those two classes is generic, simply because it defines the fundamental functionality that the Observer design pattern is designed to deliver. Therefore, the logic and structure of these two classes should be the same regardless of the context or application that uses the pattern.

On the other hand, the two – or potentially more – classes in the lower part of the Observer design pattern are concrete classes. This means that the structure and logic that those classes implement is mostly context and application specific. That is because different applications may need to have different implementations of the state change of concrete subjects and the update process of the concrete observers.

In OO programming, the binding between the abstract layer of classes and the concrete one is achieved using inheritance. This means that the concrete child classes automatically inherit whatever methods their parent abstract class defines. It is important to note that design patterns, and software patterns in general, identify the minimum requirements for the pattern to deliver its prescribed functionality. More logic can be added to the pattern to solve special or more specific problems as needed by the application.

By applying the inheritance rule to the Observer design pattern, we can say that ConcreteSubject classes inherit the generic implementation of the methods defined in the abstract Subject class. In addition to that, ConcreteSubject classes must implement the other two non-generic methods, namely, SetState and GetState. The SetState method
takes care of updating and manipulating the state data of the concrete subject class, while the GetState method provides a point of access to such data for the observer classes. The logic in these two methods can be generic if the concrete subject classes treat the subject state data universally; however, the logic can be non-generic or application specific in the case where concrete subject classes are different from each other or treat their state data differently.

The ConcreteObserver classes inherit the Update method that the abstract Observer class defines and each ConcreteObserver class implements its own update procedure according to the way it needs to manipulate its state data and keep it synchronized with the state data of the concrete subject class it is observing. It is also possible that the concrete observer classes may implement other methods that serve some application-specific purpose according to the nature of the context and application that uses the pattern.

Following the guidelines explained above about the common design pattern structures, and the analysis of the layers of the Observer design pattern, we can decide where to draw the line that separates the abstract layer of the Observer design pattern from its concrete one. Figure 4.3 shows such a separation step. The section of the Observer design pattern above the dashed line represents the abstract layer of the pattern, while the bottom section constitutes the concrete layer. It is important to note that although the pattern does not show a client part, the client is implicit. It is the part of code or logic responsible for changing the state data of the subject classes during runtime.
Although the separation of generic and application-specific logic into two different layers is quite common in software patterns, there may be some exceptions where application-specific logic may exist in the abstract layer or some generic logic exists in the concrete layer of the pattern. That is one of the reasons why we need to study the structure and logic of the pattern before we proceed to apply the layer separation step.

An example of a pattern that may have application-specific interface in the abstract layer is the Adapter design pattern. In the Adapter design pattern, shown in Figure 4.4, the Target class may contain an interface that is application specific. The Adapter class, in turn, adapts the client request to match the interface provided by the Adaptee. Figure 4.4 depicts the two layers of the Adapter design pattern.

In such a case where there is some overlapping between the two layers of the pattern, we mark the classes in the abstract layer that may contain application-specific logic and deal with this issue later in the implementation phase by enabling the user to add the application-specific logic to the pattern service using a configuration step.
4.2 Specifying Required Interfaces

Having identified the two layers of a design pattern and what client classes are there, the next step in the methodology is to use the OO methods and interfaces defined by both layers to design the target pattern service.

It is worth clarifying an important difference between the Object-Oriented programming paradigm and the Service-Oriented one. While OO programming uses inheritance to relate classes and to build class hierarchies that facilitate their relationships and interactions to deliver the required functionality, the Service-Oriented paradigm uses services and service calls to achieve that. This means that when we try to implement an OO design pattern as a service pattern, we need to figure out how to reap the OO inheritance benefits using services and service calls instead.

It is also as important to note that the interfaces that we talk about in this chapter are the runtime interfaces that the pattern service parts use during execution time to deliver the functionality of the pattern service. In the next chapter we talk about other types of
interfaces that the pattern service parts use for the registration and setting processes and to establish communication channels between each other.

In general, a pattern service consists of two parts, the generic part and the application-specific one. Each part is basically a service or a group of services. The generic part usually implements the abstract layer of the pattern and its generic logic. It also provides the interface that client services use to interact with the pattern services, while the application-specific part implements the concrete layer of the pattern and its application-specific logic. The application-specific part also provides the necessary interface that the generic part uses to execute the application-specific services.

Once we identify the abstract and concrete layers of the design pattern at hand and we specify what generic and application-specific logic each layer contains, the second step in the process of implementing such a design pattern as a pattern service is to decide on the interfaces needed to make the runtime interactions between the two parts of the pattern service and the client services possible. These interfaces should provide for all necessary interactions between the three components of the pattern service, namely, the generic part, application-specific part, and client to enable their cooperation to deliver the functionality originally prescribed by the design pattern. In order to achieve this, there are three types of interfaces that a pattern service and its constituent application-specific services may need to provide. These three types of interfaces are as follows:

1- **Interface to be used by the client** - This type of interface is typically – though not always – provided by the generic part of the pattern service. It is meant for the user services such as the client. This interface is the gateway for using the pattern service in an application.
2- **Interface to be used by the generic part of the pattern service** – This interface is provided by the application-specific services for the generic part of the pattern to use. It enables the execution of the services that implement application-specific logic during runtime.

3- **Interface to be used by the application-specific services** – This third type of interface is provided by the generic part of the pattern service and is meant for the application-specific services to use when they need to call the generic part back.

As briefly explained, the first type of interface is the interface that the generic part of the pattern provides for the user services of the application that incorporates the pattern service. The second type of interface is the one that the generic part of the pattern service needs in order to execute the application-specific part of the pattern service logic. This is the interface that the application-specific services provide so that the generic part can initiate the interaction with the application-specific one. The third type of interface is not as common as the first two, but some pattern services must employ it in order for them to deliver the functionality prescribed by the original design pattern. This is an interface that the generic part of the pattern service offers for application-specific services to call in case they need to request a service or data from the generic part that are only provided on request. These kinds of requests usually arise during or at the end of the execution of one of the application-specific services. A good example of such an interface is the interface for the “Notify” method that is provided by the Observer pattern service for the Subject application-specific service to notify its application-specific Observers.
The interfaces provided by the generic part of the pattern service – the first and third interface type – are further classified into two categories:

1- **Application-independent interface** – The signature of this kind of interface and the logic in the service method behind it are usually generic and fixed and can be used with any application without any alterations.

**Definition 4.9: Application-independent Interface of Pattern Service**

Given a pattern service that is an implementation of a design pattern, the application-independent interface of that pattern service includes signatures of service methods that implement application-independent methods of the design pattern. An application-independent method of a design pattern has the same signature and behavior for all applications that use the pattern.

**Algorithm 4.4: Process to Identify Application-independent Interface of Pattern Service from Design Pattern**

| Input: | Interface set and behavior of PS generic layer |
| Output: | Application-independent interface set for PS generic layer |
| Process: | While (more interfaces to inspect) |
| | If (interface is application independent) |
| | Assign to PS generic layer as application-independent interface |
| | End if |
| | End while |

2- **Application-dependent interface** – The signature of this kind of interface and the logic of the service methods that implement it in the generic layer of the pattern service are neither generic nor fixed; rather they can vary depending on the needs of the application that uses the pattern service.
**Definition 4.10: Application-dependent Interface of Pattern Service**

Given a pattern service that is an implementation of a design pattern, the application-dependent interface of that pattern includes signatures of service methods that implement application-dependent methods of the design pattern. An application-dependent method of a design pattern may have different signature or behavior for different applications that use the pattern.

**Algorithm 4.5: Process to Identify Application-dependent Interface of Pattern Service from Design Pattern**

- **Input:** Interface set and behavior of PS generic layer
- **Output:** Application-dependent interface set for PS generic layer
- **Process:**
  - While (more interfaces to inspect)
    - If (interface is application dependent)
      - Assign to configurable template set in the generic layer of PS as application-dependent interface
    - End if
  - End while

To elaborate on this, the application-independent interface is for service methods that do not contain any logic that can be classified as application dependent. This allows the signature of the interface and the logic the service method implements to be fixed. The application-dependent interface, on the other hand, is the interface for service methods that may contain some application-dependent logic; thus the signature of the interface and the logic cannot be fixed – in the next chapter we explain how such application-specific logic can be added to this kind of service method. The second type of interface is always application dependent because this type of interface is offered by the application-specific services that usually implement the concrete layer of the original design pattern.
Example:

We continue using the Observer design pattern as an illustration of the process of implementing a design pattern as a pattern service. The current step in the illustration is to explain the process of deciding on the types of interfaces that the Observer pattern service needs to provide. By studying the structure and behavior of the Observer design pattern in Figure 4.3 and the contents of the two layers – abstract and concrete – we can say that the concrete layer of the pattern consists of two types of classes; namely, the ConcreteSubject and ConcreteObserver. This means that the resulting Observer pattern service must have two types of Application-specific Services (AppSSs); the first type represents the Concrete Subject classes, while the second one represents the Concrete Observer classes.

Having specified the types of AppSSs needed for the Observer pattern service to function, the next step is to specify what interface each type of AppSSs must provide so that the generic part of the pattern service can deliver the required functionality.

For the Observer pattern service, the Subject AppSSs need to provide two important interfaces: the SetState and GetState. The SetState is an interface for the service method that changes the state data of the Subject AppSS, while the GetState interface is for the service method that returns the state data of the Subject AppSS to its requester. As far as the Observer pattern service is concerned, only SetState and GetState service methods are needed from the Subject AppSSs. This is because the other service methods, namely, Attach, Detach, and Notify, are provided by the generic part of the pattern service, since they belong to the abstract layer of the design pattern and their logic is generic and fixed.
The second group of AppSSs that the Observer pattern service employs is the Observer AppSSs. This group of Observer AppSSs is only required to provide the Update interface. The Update interface enables the invocation of a service method that updates the state of the Observer AppSS in accordance with the updated Subject AppSS state data.

Now that we specified the necessary interfaces that the application-specific services layer of the pattern service need to provide, we turn our attention to the generic layer of the Observer pattern service. The logic of the Observer design pattern requires the Subject class to notify its registered set of Observer classes using the Notify method. To reflect this functionality in our Observer pattern service, we need to equip the generic part of the pattern service with a Notify interface, which is used by the Subject AppSSs to inform the pattern service of any change to its state data that occurs. We call this interface “interface for Subject AppSS”.

When a Subject AppSS uses this interface to inform the pattern service of the state data change, it invokes a service method that in turn calls the Update interface of all of the registered Observer AppSSs. Of course, each of the registered Observer AppSSs may be interested in different chunks of Subject state data. This means that the pattern service needs to provide another interface that the Observer AppSSs can use to request the Subject state data they are interested in. We call such an interface the “GetState” interface and it is provided by the generic part of the pattern service. The GetState interface, once called, invokes a service that in turn calls the GetState interface of the Subject AppSS. The SetState service method, provided by the Subject AppSS, returns
the requested state data to the pattern service, which forwards it to the Observer AppSS that requested it in the first place.

The interfaces and services we explained so far facilitate the smooth interaction between the generic pattern service part and the application-specific one and deliver the Observer design pattern functionality. However, if the Observer pattern service is to be effectively used in an application, it must provide some interface for the client to engage it and use it as part of the application. To fulfill this requirement, the generic part of the Observer pattern service must provide three interfaces for the client to use. The first interface is Attach. This interface enables the client to register new Observer AppSSs to a Subject AppSS. The second interface is Detach. The Detach interface enables the client to deregister Observer AppSSs. Figure 4.5 shows the two layers of the Observer pattern service and the interfaces that each layer provides. To trigger the series of the Observer pattern service interactions, either the client service or other application service must invoke the SetState service method of the registered Subject AppSSs. The SetState service methods should update the state of the Subject AppSS and then invokes the Notify service method of the pattern service.

As we explained above, the interfaces and their service implementations in the generic part of the pattern service can either be application independent or application dependent. In Figure 4.5, application-independent interfaces are marked by the letter (I), while application dependent are marked by the letter (D).
Figure 4.5: Interfaces provided by the Observer PS
### 4.3 Classification of Pattern Services Based on Their Interface

#### 4.3.1 Classification Based on Interface Type

In the section above, we talked about the three types of interfaces that a pattern service may use to integrate its constituent services and enable the interaction with the client services that use it. However, it is important to note that not all pattern services provide all three types of interfaces mentioned above. A pattern service may provide all or a subset of such interfaces based on the nature and functionality requirements of the pattern service. Some pattern services may not have any interfaces for the client such as the (See Appendix A, Section A.14). Other pattern services, such as the (See Appendix A, Section A.13) do not need any application-specific service interfaces – the second type of interface, because the generic part of the pattern service does not need to interact with the application-specific services during runtime. A third set of pattern services, which includes the (See Appendix A, Section A.3), do not need the callback interface – the third type of interface, and that is because the application-specific services do not need to call back the generic part of the pattern service. Table 4.1 presents a classification of the pattern services created using the GoF 23 design patterns. The classification criterion is the types of interfaces being offered by the pattern service. The classification is concerned with the interfaces that are provided by the generic part of the pattern service and divides the pattern services into three groups. The “Pattern services that provide interface only for client” group contains the pattern services that provide interface for the client service – the first type of interface – but do not provide a callback interface for the application-specific services to call back the generic pattern service part. The “Pattern services that provide interface only for AppSS” group
consists of the pattern services that do not provide an interface for the client services to use and only provide the interface for application-specific services to call the generic part of the pattern service. Finally, the “Pattern services that provide interface for both” group consists of the pattern services that provide both types of interfaces, the one dedicated to the client services and the one used by the application-specific services.

Table 4.1: PS classification based on interface types

<table>
<thead>
<tr>
<th>Pattern services provide interface only for client</th>
<th>Pattern services provide interface only for AppSS</th>
<th>Pattern services provide interface for both</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Factory</td>
<td>Mediator</td>
<td>Command</td>
</tr>
<tr>
<td>Adapter</td>
<td></td>
<td>Decorator</td>
</tr>
<tr>
<td>Bridge</td>
<td></td>
<td>Facade</td>
</tr>
<tr>
<td>Builder</td>
<td></td>
<td>Factory Method</td>
</tr>
<tr>
<td>Chain of Responsibility</td>
<td></td>
<td>Observer</td>
</tr>
<tr>
<td>Composite</td>
<td></td>
<td>Proxy</td>
</tr>
<tr>
<td>Flyweight</td>
<td></td>
<td>State</td>
</tr>
<tr>
<td>Interpreter</td>
<td></td>
<td>Template Method</td>
</tr>
<tr>
<td>Iterator</td>
<td></td>
<td>Visitor</td>
</tr>
<tr>
<td>Memento</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prototype</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Singleton</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strategy</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example from the “Pattern services that provide interface only for client” Group:

In this section, we explain the structure and interfaces of the Adapter pattern service, as a representative pattern service of the group of pattern services that provide an interface for client use only. As can be seen from Figure 4.6 the Adapter pattern service consists of two essential parts, generic and application specific.
The generic part provides the “Request” interface which is a single interface that enables the client to invoke the service methods that provide the Adapter pattern service functionality. The generic part of the Adapter pattern service only provides an interface for the client and does not include any other interfaces that AppSSs can call, and that is because the pattern service does not need such an interface to deliver its functionality.

Note that the Request interface of the generic part of the Adapter pattern service is marked as (D) – which means that the interface and the supporting service may have
some application-dependent code that is added during the implementation phase of the pattern service.

**Example from the “Pattern services that provide interface only for AppSS” Group:**

The “Pattern services that provide interface only for AppSS” group, in Table 4.1, represents pattern services that do not provide any interfaces for the client. Instead, those pattern services usually provide an interface for the generic pattern service to interact with the application-specific part, and another interface for the application-specific part to invoke services provided by the generic part. In such a case, one or more of the application-specific services may act as the client that triggers the series of actions that the pattern service performs in order for it to deliver its functionality.

The Mediator pattern service is an example of the pattern services that do not provide an interface for the client or, rather, they do not employ client services as part of their execution. The only interface this group of pattern services provides is the one for the AppSSs to call the generic part of the pattern service. Figure 4.7 shows the interfaces provided by the Mediator pattern service. As can be seen from Figure 4.7, the generic part of the pattern service provides an interface called “Mediate”, which can be used by the application-specific concrete colleague services to direct their request to the proper colleague. The process does not involve any client services to interact with the Mediator pattern service to perform such a mediation process; instead, one of the colleague application-specific services can use the Mediate interface to make the Mediator pattern service perform the mediation process.
Figure 4.7: Interfaces provided by the Mediator PS

Example from the “Pattern services that provide interface for both” Group:

The “Pattern services that provide interface for both” group of pattern services in this classification is the group of pattern services that offer both types of interfaces, the one for the client to use, and the one for AppSSs to call the generic part of the pattern service. We use the Decorator pattern service to demonstrate how the generic part of the pattern service may provide the two types of interfaces. Figure 4.8 shows the Decorator pattern service and what interfaces it provides. As depicted in Figure 4.8, the Decorator pattern service has the generic part and the application-specific one. The application-specific part can further be divided into Concrete Decorator AppSSs and Concrete Component AppSSs.

As mentioned at the beginning of this section, in this classification, we are interested in the interfaces provided by the generic part of the pattern service. The generic part of the
Decorator pattern service provides an interface for the client to use the pattern service called “DecoratorOperation”, and another interface for the concrete decorators to call the generic part back, called “ComponentOperation”.

![Figure 4.8: Interfaces provided by the Decorator PS](image)

**Figure 4.8: Interfaces provided by the Decorator PS**
4.3.2 Classification Based on Application Dependency of the Interface

In the last interface classification section we used interface type as the classification criterion. In this section, instead of classifying pattern services based on the type of interface being offered, we use the level of application dependency of the interfaces being offered by the generic part of the pattern service as the classification criterion. This classification divides the pattern services created using the 23 GoF design patterns into three groups. The “Pattern services that provide application-dependent interface only” group consists of the pattern services that only provide application-dependent interfaces in the generic part of the pattern service diagram. We mark such interfaces with the letter (D). The “Pattern services that provide application-independent interface only” group contains the pattern services that offer only application-independent interfaces, which bear the letter (I) in the generic part of the pattern service diagram. The “Pattern services that provide both interfaces” group contains the pattern services that provide both application dependent and application-independent interfaces. In the diagram of any of the patterns services that belong to the “Pattern services that provide both interfaces” group you see interfaces that have the (D) mark and others that have the (I) mark. Table 4.2 gives the three groups of pattern services based on the application dependency classification.
Table 4.2: PS classification based on application dependency of interface

<table>
<thead>
<tr>
<th>Pattern services provide application-dependent interface only</th>
<th>Pattern services provide application-independent interface only</th>
<th>Pattern services provide both interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Factory</td>
<td>Flyweight</td>
<td>Command</td>
</tr>
<tr>
<td>Adapter</td>
<td>Iterator</td>
<td>Composite</td>
</tr>
<tr>
<td>Bridge</td>
<td>Prototype</td>
<td>Interpreter</td>
</tr>
<tr>
<td>Builder</td>
<td>Singleton</td>
<td>Observer</td>
</tr>
<tr>
<td>Chain of Responsibility</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decorator</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Facade</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Factory Method</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mediator</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memento</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proxy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>State</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strategy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Template Method</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Visitor</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example from the “Pattern services that provide application-dependent interface only” Group:

As an example of the pattern services that offer only application-dependent interface we use the Template Method pattern service. Figure 4.9 shows the structure and interfaces provided by the Template Method pattern service.
The generic part of the Template Method pattern service offers an interface for the client and another for the Template Method AppSS. Both interfaces are considered application dependent because of the following:

---

**Figure 4.9: Interfaces provided by the Template Method PS**

The generic part of the Template Method pattern service offers an interface for the client and another for the Template Method AppSS. Both interfaces are considered application dependent because of the following:
- The TemplateMethod interface is application dependent because the signature of the TemplateMethod and whatever code that goes into it are application dependent and different applications that use the Template Method pattern service would want to specify their own TemplateMethod signature, parameters, and logic.

- For the same reasons, the PrimitiveOperation1 and PrimitiveOperation2 are application dependent. The signature and logic of any primitive operation service method that will eventually call the corresponding primitive operation service method offered by the Concrete Class AppSS depend on the application that uses the Template Method pattern service and how those primitive operation service methods are to be invoked and under what conditions.

**Example from the “Pattern services that provide application-independent interface only” Group:**

To show an example of the group of pattern services that provide only application-independent interface we use the Flyweight pattern service. Figure 4.10 shows the application-independent interface – which is the only interface the Flyweight pattern service needs to expose to deliver its functionality.

The “GetFlyweight” interface enables the client to request a flyweight service from the pool of flyweight services that the Flyweight pattern service manages. It is considered application independent because the signature of the interface and the logic that the service behind this interface contains can be fixed and generic and thus reusable with any kind of application. The more application independent the generic part of the pattern service is, the easier it is to use in different software applications, since there
will be less or no application-dependent logic added to the generic part of the pattern service.

![Flyweight Pattern Service](image)

**Figure 4.10: Interfaces provided by the Flyweight PS**

There is another noteworthy aspect of the Flyweight pattern service depicted in Figure 4.10, which is the fact that the application-specific services part does not provide any interface for the generic part of the pattern service to use during runtime; the reason being that the generic part of the Flyweight pattern service does not need to interact with the application-specific services. Instead, it manages a pool of flyweight service instances with unique keys, and returns the proper service instance reference to the client on the invocation of GetFlyweight service method and the provision of the key of the required flyweight service instance.

**Example from the “Pattern services that provide both interfaces” Group:**

The “Pattern services that provide both interfaces” group in this classification houses the pattern services that have both application-dependent and application-independent
interfaces. This means that the service methods that support the exposed application-independent interfaces contain only generic logic; therefore, they can be used with any application without the need to change that logic, while other service methods must have some application-dependent logic and so they cannot be used until such application logic is applied to them. To give an example of such a group of pattern services, we describe the interfaces provided by the Command pattern service. Figure 4.11 depicts the structure and interfaces of the Command pattern service. Notice that the generic part of the pattern service contains interfaces that are marked with the letter (D) to show that they are application dependent and others that are marked with the letter (I) to indicate that they are application independent. Note that the interfaces shown in Figure 4.11 cater for both the Execute and Unexecuted functionalities by the Command pattern service.
### Command Pattern Service

<table>
<thead>
<tr>
<th>Interface for Client</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute ()</td>
</tr>
<tr>
<td>Unexecute ()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Interface for Concrete Command AppSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action ()</td>
</tr>
<tr>
<td>Unaction ()</td>
</tr>
</tbody>
</table>

### Concrete Command Application Specific Services (AppSSs)

<table>
<thead>
<tr>
<th>Interface for Command pattern service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute ()</td>
</tr>
<tr>
<td>Unexecute ()</td>
</tr>
</tbody>
</table>

### Receiver Application Specific Services (AppSSs)

<table>
<thead>
<tr>
<th>Interface for Command pattern service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action ()</td>
</tr>
<tr>
<td>Unaction ()</td>
</tr>
</tbody>
</table>

Figure 4.11: Interfaces provided by the Command PS
4.3.3 Classification based on Application-specific Service Interface Requirement

The third classification process for the pattern services resulting from implementing the 23 GoF design patterns involves the interfaces offered by the application-specific services for the generic part of the pattern service to use. Interestingly enough, not all pattern services require the services in the application-specific part to provide an interface for the runtime interactions of the pattern service. On the other hand, the application-specific services in some pattern services must provide one or more interfaces for the generic part to call in order to fulfill the pattern service’s functionality.

In this classification, we separate the 23 pattern services into two groups. The “Pattern services that need application-specific service interface” group includes the pattern services that require their application-specific services to provide interfaces for runtime interactions, while the pattern services that do not require their application-specific services to provide any runtime interfaces are placed in the “Pattern services that do not need application-specific service interface” group. The classification in Table 4.3 shows which pattern services belong to the “Pattern services that need application-specific service interface” group and which of them belong to the “Pattern services that do not need application-specific service interface” one.
Table 4.3: PS classification based on AppSS interface requirement

<table>
<thead>
<tr>
<th>Pattern services need application-specific service interface</th>
<th>Pattern services do not need application-specific service interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Factory</td>
<td>Flyweight</td>
</tr>
<tr>
<td>Adapter</td>
<td>Iterator</td>
</tr>
<tr>
<td>Bridge</td>
<td>Prototype</td>
</tr>
<tr>
<td>Builder</td>
<td>Singleton</td>
</tr>
<tr>
<td>Chain of Responsibility</td>
<td></td>
</tr>
<tr>
<td>Command</td>
<td></td>
</tr>
<tr>
<td>Composite</td>
<td></td>
</tr>
<tr>
<td>Decorator</td>
<td></td>
</tr>
<tr>
<td>Facade</td>
<td></td>
</tr>
<tr>
<td>Factory Method</td>
<td></td>
</tr>
<tr>
<td>Interpreter</td>
<td></td>
</tr>
<tr>
<td>Mediator</td>
<td></td>
</tr>
<tr>
<td>Memento</td>
<td></td>
</tr>
<tr>
<td>Observer</td>
<td></td>
</tr>
<tr>
<td>Proxy</td>
<td></td>
</tr>
<tr>
<td>State</td>
<td></td>
</tr>
<tr>
<td>Strategy</td>
<td></td>
</tr>
<tr>
<td>Template Method</td>
<td></td>
</tr>
<tr>
<td>Visitor</td>
<td></td>
</tr>
</tbody>
</table>

Example from the “Pattern services that need application-specific service interface” Group:

As can be seen from Table 4.3, the majority of the pattern services belong to the group that requires the application-specific services to provide runtime interfaces. To give an
example of this group, we explain the structure and interfaces of the Strategy pattern service. Figure 4.12 shows the interfaces provided by both parts of the Strategy pattern service.

![Diagram of Strategy Pattern Service and Concrete Strategy Application Specific Services (AppSSs)](image)

**Figure 4.12: Interfaces provided by the AppSSs of the Strategy PS**

We can see from Figure 4.12 that the generic part of the Strategy pattern service provides an application-dependent interface that is marked with the letter (D) for the client to invoke the pattern service. However, in this section we are not interested in the interface provided by the generic part of the pattern service; rather, we are interested in the interface that the application-specific service provides. The Strategy pattern service must have one or more concrete strategy services that the generic part of the pattern service calls at runtime to execute the chosen strategy algorithm. This means that each Concrete Strategy application-specific service must provide an interface that the generic part of the Strategy pattern service can call to run the service method implementing
concrete strategy algorithm. Such an interface is represented by the “AlgorithmInterface” that is shown in the Concrete Strategy application-specific service part of Figure 4.12.

Example from the “Pattern services that do not need application-specific service interface” Group:

The “Pattern services that do not need application-specific service interface” group in the classification contains the pattern services that do not involve any runtime interactions between the two parts of the pattern service; hence, there is no need for the application-specific services to provide any interface for the generic part to use. We use the Singleton pattern service as an example of pattern services that do not require their application-specific services to offer any interfaces to be used during runtime. Figure 4.13 shows that the Singleton pattern service does not employ any interfaces in its application-specific part.

![Singleton Pattern Service Diagram](image)

**Figure 4.13:** AppSSs of the Singleton PS provide no runtime interfaces
The purpose that the Singleton pattern service serves is to maintain a data structure of
the references to single instances of the singleton application-specific services. The
pattern service keeps a reference for each concrete singleton application-specific service
and returns that single reference of the designated application-specific service every
time the client requests one. This means that the generic part of the Singleton pattern
service does not need to interact directly with the application-specific service during
runtime, rather, it guarantees that only one instance of each application-specific service
is created and then those instances can be reused by the client and the application that
incorporates the Singleton pattern service.

It is important to note at the end of this section that the classification process that we
just explained is used to estimate the development effort needed to implement a given
design pattern as a pattern service. The classification results are used in Chapter 8 as
part of the input used to evaluate the development effort involved in following the
pattern service paradigm to offer design pattern implementations as pattern services.

4.4 Conceptual Pattern Service Realization through Interfaces

The third phase of the conceptual methodology for implementing design patterns as
pattern services is concerned with specifying the runtime sequence of events and actions
that, when performed, realize the prescribed functionality of the pattern service.

Definition 4.11: Conceptual Realization of Pattern Service

The conceptual realization of a pattern service is the description of the behaviors of the
pattern service through its interfaces.
Algorithm 4.6: Process for Specifying Conceptual Pattern Service Realization in Conceptual Model of Pattern Service

Input: Pattern service interface set (in both, generic and application-specific layers)

Output: Interfaces and call sequence to conceptually realize pattern service

Process:
If (PS has separate client service)
  If (client for PS generic layer)
    Client service triggers PS execution by invoking interface in generic layer
    If (PS has AppSS Interface and generic layer needs to call AppSS layer)
      While (More interactions needed)
        Generic layer invokes AppSS interface
        If (AppSS needs to callback generic layer)
          AppSS invokes callback interface of generic layer
        End if
      End while
  End if
Else if (client for AppSS layer)
  Client service triggers PS execution by invoking interface in AppSS layer
  AppSS invokes PS interface
  If (Generic layer needs to call AppSS layer)
    While (More interactions needed)
      Generic layer invokes AppSS interface
      If (AppSS layer needs to callback generic layer)
        AppSS invokes callback interface of generic layer
      End if
    End while
End if
Else
  AppSS layer triggers PS execution by invoking interface in generic layer
  If (Generic layer needs to call AppSS layer)
    While (More interactions needed)
      Generic layer invokes AppSS interface
      If (AppSS layer needs to callback generic layer)
        AppSS invokes callback interface of generic layer
      End if
    End while
End if
End if
The sequence of events and actions takes place through the invocation of the interfaces provided by the generic layer of the subject pattern service and its application-specific one. Thus, pattern service realization is basically describing the usage of the pattern at runtime in a use case scenario and identifying which component triggers the series of interactions and what are the consequences of each interaction between the three components of the pattern service, namely, the generic part, application-specific part, and the client – if one is needed.

Although the realization sequence of events can vary, a typical sequence of events in the realization of a pattern service starts with the client calling the interface that the generic layer of the pattern service provides for the client to use. The invocation of one of these interfaces, in most cases, performs some task and most likely invokes an interface that is provided by one of the application-specific services. The AppSS interface, called by the generic layer, is invoked and the task associated with that interface is performed and most likely returns results of some sort to the generic layer of the pattern service, which in turn returns the final results to the client. Some pattern services have an extra interaction between the generic part and the application-specific one, and that is when the application-specific service requires calling back a service method in the generic part for some more data or feedback that is fundamental to the execution of the pattern service. The Decorator pattern service is a good example of such a case of pattern service realization. Figure 4.14 shows the sequence of events that takes place during the runtime execution of the Decorator pattern service.
This typical sequence of events to realize a pattern service applies to the pattern services that have a client component and their generic and application-specific layers, both offering interfaces to be used during their execution. However, as we mentioned in the sections above, not all pattern services have all three components and not all of them offer a similar set of interfaces. In addition to the fact that some pattern services may not require the client component, such as the Mediator pattern service (See Appendix A, Section A.14), some application-specific services in the application-specific part of the pattern service may not need to offer any interfaces, one example is the Singleton pattern service (See Appendix A, Section A.19), and that can be attributed to the nature of the pattern service and the functionality it delivers.
Realization of Pattern Services with no Client

First, we give an example of the pattern services that do not have the client component, the Mediator pattern service. The Mediator pattern service is the only pattern service implementation originating from the 23 GoF design patterns that is classified as not having a client component. It consists of a generic layer and two or more application-specific services, in the AppSS layer, called the colleagues. The generic part mediates between those colleagues to fulfill the functionality prescribed by the Mediator pattern service. Any one of the Colleague AppSSs can trigger that sequence of mediation actions. In such a case, the Colleague AppSS which initiates the interaction acts as the client component of the pattern service. Figure 4.15 presents the runtime sequence of actions that realizes the Mediator pattern service.

![Diagram](image)

**Figure 4.15: Runtime realization of the Mediator PS**

Realization of Pattern Services with no Application-specific Service Interface

Second, we show the sequence of interactions that takes place during the execution of a pattern service that does not need to provide an application-specific service interface for its generic layer. Since we used the Singleton pattern service as an example in the classification section above, we will use the realization sequence of events for that same pattern service.
Figure 4.16 shows the sequence of events that takes place during the realization of the Singleton pattern service. There are two events that happen during the execution of the Singleton pattern service. At runtime, the client asks the Singleton pattern service for the service instance reference of the designated application-specific service, through the invocation of the SingleInstance interface, offered by the generic layer of the Singleton pattern service. Since the Singleton pattern service is keeping a list of the single instances of the all the application services, it looks up the service reference of the designated application-specific service, fetches it from the data structure, and passes it along to the client. The client then can access the returned service instance and call whatever interface that service makes available for its users.

4.5 Summary

In this chapter we discussed the conceptual methodology for implementing design patterns as pattern services. We identified the eight main steps that should be performed in order to systematically turn a software design pattern into a pattern service. These eight steps can be classified as serving two purposes. The first set of four sheds light on the task of studying the subject design pattern in the effort to identify the abstract layer and the concrete one. The identification of the two layers helps separate the application-
independent logic from the application-dependent one. The second set of four steps show how to use the results produced by performing the first set of steps to actually design the target pattern service. First they specify the set of interfaces that should represent the application-independent behavior of the pattern service. Secondly, they list the set of interfaces that represent the application-dependent behavior. Thirdly, they specify the set of interfaces that represents the application-specific behavior of the pattern service. Lastly, they specify the specification of the runtime sequence of events and interactions that realize the functionality the pattern service promises to deliver.

Then we elaborated on each step and explained the tasks involved and gave an illustrative example using the GoF Observer design. After that we classified the GoF design patterns into groups based on the characteristics of the sets of interfaces that pattern services provide. The first pattern service classification criterion is based on the type of the interfaces that the pattern services provide for the use of client services, AppSSs, and the generic part of the pattern service. The second criterion is based on the application dependency of the interfaces the pattern service exposes. The third criterion is related to classifying the pattern service based on AppSS interface requirement. We gave an example of each group of the classified pattern services. Finally, we explained how a pattern service functionality can be realized using some examples of the classified groups.

In Chapter 5, we introduce the implementation methodology for the design pattern services, which complements the conceptual one. We show how the components of a design pattern service can be developed, integrated and configured to work together.
according to the requirements of the specific application using the design pattern service.
Chapter 5

Implementation Methodology for Design Patterns Services

In the last chapter, we presented the conceptual methodology for implementing software design patterns as pattern services. In this chapter we explain how the artifacts resulting from applying the conceptual methodology can be implemented and describe the implementation stages of a pattern service. This includes the realization of the three major steps in creating and using a pattern service, namely, the building of the generic layer of the pattern service, developing the AppSSs required to complement the generic layer, and the configuration process that facilitates the integration of the two layers to deliver the pattern service functionality. These three steps are performed by three types of software developers. The generic layer of the pattern service and its services are designed and implemented by the Pattern Service Creators (PSCs), while the AppSSs are designed and implemented by the application-specific service creators (AppSSCs). The third type of software developer is assigned the task of creating and handling the configuration procedures and files that facilitate the integration between the generic and application-specific layers of the pattern service and customize the pattern service to work according to the needs and requirements of the specific software application at hand. This third type of software developer is called the application-specific pattern service creators (AppSPSCs).

Before we describe the steps prescribed by the implementation methodology for the design patterns as services, let us formally define some important terms and what they mean in the context of pattern services.
**Definition 5.1: Implementation Model of a Pattern Service**

The implementation model of a pattern service consists of the application-specific service (AppSS) binding interface, the pattern service runtime configuration interface, the reference implementation architecture of the pattern service, as well as descriptions of functionality and behavior realizations of the pattern service based on the implementation model.

This chapter explains how the interaction procedures prescribed by the conceptual methodology can be realized through the implementation of concrete service artifacts that collaborate according to those procedures.

**Definition 5.2: Implementation Methodology of a Pattern Service**

Given the conceptual model of the pattern service that implements the given design pattern, the implementation methodology is the process to create the implementation model of the given pattern service. The implementation methodology consists of the following phases: AppSS binding interface specification, PS runtime configuration interface specification, PS architecture design, and specification of PS realizations based on the PS interfaces and PS architecture components.

As explained in Chapter 4, the structure of a pattern service that implements a design pattern consists of several components: Firstly, the generic layer that implements the application-independent logic of the subject design pattern; secondly, the application-dependent layer that implements relatively generic logic, but requires some application-specific logic added to it.
Application-dependent service methods of a pattern service usually need to interact with one or more service methods from the application-specific layer. The application-specific layer of a pattern service may consist of one or more application-specific services, and these application-specific service can be classified into one or more types, depending on their functionalities and the role they play in realizing the functionality of the pattern service as a whole. The following is a formal definition of an application-specific service and what application-specific service type means:

**Definition 5.3: Application-specific Service and Application-specific Service Type**

Given a design pattern, an application-specific service type has service instances that implement the services of a concrete application-specific class in the pattern.

### 5.1 Application-specific Service and Pattern Service Binding

For a pattern service that is a result of implementing a design pattern to work and deliver the functionality prescribed by the subject design pattern, its constituent, rather separate, components must be integrated to work as one unit. The first step in the process of pattern service component integration is to bind the AppSSs that implement relevant application-specific logic of the application using the pattern service. The formal definition of the process of binding application-specific services with the generic layer of a pattern service can be stated as follows:

**Definition 5.4: Application-specific Service and Pattern Service Binding**

Given a pattern service that is an implementation of a design pattern, the application-specific service binding with the pattern service is the process of registering these application-specific services with the pattern service in such a way that facilitates the set
of interactions between the application-specific services and the pattern service to realize the functionality prescribed by the subject design pattern.

To facilitate the binding procedure, there has to be a specific service interface that a pattern service offers and implements to enable such binding. It is worth noting that the binding interface and the service methods that implement it are application-independent.

The binding interface that the pattern service offers as the gateway to the integration tool with the AppSS layer can be defined as follows:

**Definition 5.5: Application-independent AppSS Binding Service Interface of a Pattern Service**

The signature of the application-independent AppSS binding interface for all pattern services is defined as

$$\text{AppSSID} \quad \text{registerAppSS(AppSSReference, AppSSType, OptionalData)}$$

**AppSSID**

It is generated by the binding service to use as the key that the pattern service uses to access and refer to a particular registered AppSS.

**AppSSReference**

It is the URI of the AppSS to be registered.

**AppSSType**

It is the type indicator of the application-specific service to be registered and integrated with the subject pattern service. The allowable AppSSType for a particular type of
AppSS is specified in the application-specific pattern service configuration specification.

OptionalData

It may be used by some application-specific services binding procedure to supply extra, but important, information about the application-specific service being bound to the pattern service. The allowable and/or required optional data corresponding to the given AppSSType is specified in the application-specific pattern service configuration specification.

As the name suggests, the OptionalData parameter does not always carry some important piece of information to the integration process. Some AppSSs may leave it blank because they do not have any extra information to pass to the binding service other than the information provided by the AppSSReference and the AppSSType.

The application-independent binding service and its interface are designed to be application independent, which makes them reusable in the context of any application. In terms of pattern service dependability, however, the binding interface is only semi-independent. That is because different pattern services may have different AppSS type and OptionalData restrictions. The AppSS binding section, of the XML schema that validates the XML configuration file, which is used to configure generic pattern services to convert them into application-specific pattern services, defines what restrictions are applied to the values passed in the AppSSType and OptionalData parameters.
**Application-specific Service to Pattern Service Binding Process**

After the selection of the pattern service to use, and the development of AppSSs that complement the functionality of the selected PS, the turn comes for the integration procedure that enables the interaction between the PS and the AppSSs. The integration procedure actually refers to the process that binds the implemented AppSSs to the selected PS. Figure 5.1 depicts the steps involved in binding AppSSs to a PS.

![Figure 5.1: AppSS to PS binding process](image)

In the following algorithm, we describe the steps and artifacts, shown in Figure 5.1, that are involved in executing the PS to AppSS binding process:
Algorithm 5.1: Pattern Service Process to Handle AppSS Binding

Input: AppSSReference, AppSSType, and OptionalData
Output: AppSS bound to PS
Process: If (AppSSReference is a valid URI)
  If (AppSSType valid according to the configuration specification)
    Call AppSSID Generator service method
    Generate AppSSID according to generation policy
    Return AppSSID to binding service method
  If (AppSSType does not exist)
    Create AppSSType registration list
  End if
  Add AppSSID and AppSSReference to AppSSType registration list
  If (OptionalData valid for AppSSType according to the configuration specification)
    Store OptionalData and link storage location to this AppSSID's entry in AppSSType registration list
    Return AppSSID
  End if
End if

5.2 Pattern Service Runtime Configuration

After creating the application-independent and application-dependent logic in the generic layer of the PS and integrating the developed AppSSs with the generic PS, using the binding process explained in the last section, the pattern service becomes an application-specific pattern service (AppSPS). The AppSPS is integrated and configured according to the needs of a certain application and therefore is ready to be used as part of that application. When an AppSPS is used in the context of an application, some runtime configuration is usually required to enable the application to make the AppSPS behave according to the specific choices either by the application clients or the logic of the application itself. This runtime configuration requirement usually deals with the selection or preparation for the process that the AppSPS is to perform and which
components – e.g. the AppSSs – of the AppSPS that will participate in such a process.

We give a formal definition of the pattern service runtime configuration below:

**Definition 5.6: Pattern Service Runtime Configuration**

At runtime, a running pattern service can be configured by client in two ways:

**Selecting current AppSS for an AppSS type**

For each AppSS type, a client can select a designated AppSS in the AppSS type's registration list as the current AppS, so the PS will interact with the selected current AppSS only for this client after this configuration.

**Chaining AppSSs**

A client can set up a sequence of AppSSs, so the PS can interact with the chained AppSSs in sequence for this client after this configuration.

The definition refers to the selection of one or more AppSSs by either the client of the pattern service or the entity responsible for performing such a selection. Depending on the nature of the design pattern and the way it solves the design problem, the number of AppSSs to be selected in the pattern service that implements this design pattern may range from just one AppSS to a chain of AppSSs. That is why the pattern service runtime configuration service should define two service methods, one for single AppSS selection, and the other for the selection of a chain of AppSSs.

Although there are two types of runtime configuration routines, each for a specific type of configuration, the two routines, their interfaces, and logic should be application and pattern independent. What we mean is that the two runtime configuration routines
should have fixed reusable interfaces and implementations that become the standard for configuring any pattern service during runtime.

**Definition 5.7: Application-independent Pattern Service Configuration Service Interface**

There are two application-independent pattern service configuration interface signatures. The signature of the AppSS selection interface for all pattern services is defined as:

```c
void selectAppSS(AppSSID, AppSSType, OptionalData)
```

**AppSSID**

It is the AppSSID of the AppSS which will be selected as the current AppSS for the given AppSS type. A valid AppSSID must be the AppSSID of a registered AppSS for the given AppSS type of the PS.

**AppSSType**

It is the type indicator of the application-specific service to be selected to interact with the subject pattern service.

**OptionalData**

It may be used by an application-specific service selection procedure to supply extra, but important, information about the application-specific service being selected to interact with the pattern service.

An example of when the optional data is required is when selecting subsystem AppSS in the Façade pattern service. The OptionalData parameter should contain the signature
of the service method of the subsystem that the façade operation should call (See Appendix A, Section A.9).

The signature of the chained AppSSs selection interface for all pattern services is defined as

```c
void chainAppSS (AppSSID [*], OptionalData)
```

**AppSSID [*]**

It is an array of AppSSID of AppSSs which will be selected as the current chain of AppSSs. Each of the AppSSID in the array must be the AppSSID of a registered AppSS for one of the AppSS types of the PS.

**OptionalData**

It may be used by some chained AppSS selection procedure to supply extra, but important, information about the application-specific service being selected to be part of the chain, or information that shows the PS how to interact with a particular AppSS in the chain.

An example of a pattern service that implements the chain selection runtime configuration procedure and does not require the optional data is the Chain of Responsibility pattern service. The only information needed is the AppSSID for the concrete handlers to be chained. The Decorator pattern service, on the other hand, requires that the Concrete Component’s ID be provided when chaining the decorators. The OptionalData parameter can be used to hold such an id, which in this case is an important piece of data for the chaining process.
Figure 5.2 shows the two processes that the runtime configuration service provides. The first process enables the selection of one AppSS and stores the AppSSID of the selected AppSS in the data structure that the pattern service provides for this purpose. The second process facilitates the selection of a chain of AppSSs that can be put in a certain sequence – according to the application requirements – to perform a certain functionality. The pattern service provides a different data structure to store the selected chain of AppSSIDs.

The following two algorithms explain the steps involved in the runtime configuration process of a pattern service.
Algorithm 5.2: Process of Pattern Service to Handle Runtime Select Configuration

Input: AppSSID, AppSSType, and OptionalData
Output: The current AppSS register for AppSSType is updated
Process: Validate AppSSID by checking AppSSType registration list
        if (valid)
            Save AppSSID into the current AppSSType register
            if (OptionalData not empty)
                Unlink optional data storage from the current AppSSType register
                Save optional data to storage
                Link the storage to the current AppSSType register
            End if
        End if

Algorithm 5.3: Process of Pattern Service to Handle Runtime Chain Configuration

Input: AppSSID[*] and OptionalData
Output: The current ChainedAppSS register is updated
Process: Validate each AppSSID in the AppSSID array by checking AppSSType registration list
        if (All AppSSIDs in AppSSID[*] are valid)
            Save AppSSID[*] into the current ChainedAppSS register
            if (OptionalData not empty)
                Unlink optional data storage from the current ChainedAppSS register
                Save optional data to storage
                Link the storage to the current ChainedAppSS register
            End if
        End if

5.3 Design Pattern Service Implementation Reference Architecture

To show how the implementation methodology of design patterns as pattern services may be systematically applied to implement different design patterns as pattern services, we present a pattern and application-independent implementation reference architecture that developers applying the proposed pattern service methodology can follow to perform the implementation steps that produce fully functioning application-specific
pattern services ready to be incorporated in the software application at hand. The implementation reference architecture, described in this section, covers the three aspects of the implementation process, namely, the AppSS binding, application-dependent program logic generation, and runtime configuration. The architecture identifies the services and controllers that manage all three components of the implementation process. It also identifies the data structures that each component needs in order to store relevant data that is used by the same component and – in some case – by other components to realize the overall functionality of the design pattern service created. Figure 5.3 shows the design of the implementation reference architecture. The main three components of the implementation reference architecture are evident. The topmost three services are the gateways to the functionalities provided by each component. The AppSSBindingService and its subsequent services controller and data structures take care of the AppSS binding process. The FunctionalityService and its subsequent service, controllers, and data structures implement the pattern service logic both application-independent and application-dependent. Finally, the AppSSConfiguration service and its subsequent services, controllers, and data structures facilitate the runtime configuration process.

Next, we describe each component of the three components shown in Figure 5.3 and highlight the interactions that take place within each component and those that go beyond component boundaries.
Figure 5.3: Reference Implementation Architecture of Design Pattern Services
1- **AppSS Binding Component**: This component takes care of registering application-specific services with the pattern service. It offers a service interface that accepts the URI of the AppSS as a reference, the type of the AppSS, and any optional data that can assist in the registration of the AppSS and its integration with the pattern service. The AppSS binding component dedicates a data structure – referred to in the diagram as the AppSSRegistrationListByType – to store the registration information of the AppSS for the other components of the implementation reference architecture to use.

The AppSS registration list maintains lists of AppSS by their AppSSType. Some of the important pieces of information that are stored about the registered AppSSs include the unique AppSSID that is generated during the registration process; the AppSSRef, which is the URI of the actual AppSS; the AppSSType, which signifies the type of the AppSS in cases where there is more than one AppSS type; and the optional data that might be a requirement to register some types of AppSSs.

2- **Pattern Service Functionality Component**: This component implements all the services, service methods, and controllers suggested by the conceptual model. This group of services and their constituent service methods implement the pattern service generic logic, both application-independent and application-dependent. As can be seen from Figure 5.3, the functionality component groups application-independent service methods together and dedicates the data structure “ApplicationIndependentDS” to their use. Similarly, the application-dependent service methods are grouped by the functionality component in a separate group and are allocated their own data structure, “ApplicationDependentDS”. An integral part of the functionality component is the application-specific services block. This block
contains all the services and service methods that implement pure application-
specific logic, which the application-dependent service and service methods need to
interact with to deliver the overall functionality of the pattern service. The
functionality communicator is the layer that facilities communication between the
blocks inside of the PS functionality component and the services in the AppSS
block.

3- **AppSS Configuration Component**: The service methods in this component
facilitate the runtime configuration of the pattern service in terms of which AppSSs
to use in certain interactions between the generic layer of the PS – mainly
application-dependent service methods – and the designated AppSSs. This
configuration component provides two main service methods. The first enables the
selection of a single AppSS that is designated to be the target of service method
calls that originate from the pattern service side towards the AppSS side. Also,
service methods of such selected AppSS may have some service method callbacks
targeting service methods on the PS side. The configuration component dedicates a
data structure, “SelectedAppSSRegister”, for storing the AppSSID of the selected
AppSS for PS use during the actual interactions that realize the PS functionality.
The second service method provided by the runtime configuration component is
similar to the first one in the sense that it enables the selection of AppSSs to be the
designated participants in the set of interactions between the generic part of the PS
and the AppSS one. However, in the case of this second service method, the number
of the AppSSs to be selected is not limited to only one. In fact, this service method
enables the selection of a group or chain of AppSSs to represent the AppSS side of
interactions during the realization of part or all of the functionality of the pattern
service, hence the name “chainAppSS”. Contrary to the “selectAppSS” service method which accepts a single AppSSID as one of its parameters, the chainAppSS accepts a chain of AppSSIDs. This chain of AppSSIDs are stored in the “ChainedAppSSRegister” data structure and is used by the functionality component to identify the selected chain of AppSSs that should be involved in the pattern service interactions.

The implementation reference architecture also suggests the use of an independent component that facilities the management of the “OptionalData” supplied during the binding and/or configuration processes. This centralized optional data management component is accessed by all other three components of the architecture and should assist them in storing, retrieving, and interpreting the contents of the optional data.

5.4 Implementation Level Pattern Service Realization

In Chapter 4 we discussed the conceptual realization of a pattern service through its set of interfaces. In this section we explain how functionality of a pattern service can be realized using the components and following the interaction set prescribed by the implementation reference architecture. To do that, we first define what the realization of a pattern service at the implementation level means.

**Definition 5.8: Implementation Realization of Pattern Service**

The implementation level realization of a pattern service is the description of the binding, configuration, and functionality behaviors of the pattern service by interactions of the pattern service's architectural components.
Algorithm 5.4: Realization of Application-specific Service Binding

Input: Pattern service’s AppSS Binding component
Output: The binding of an AppSS with the generic PS
Process: AppSSBindingController receives AppSS binding message with following parameter (AppSSReference, AppSSType, OptionalData)
AppSSBindingController validates parameters
  Validate AppSSReference
  Validate AppSSType against XML document
  Validate OptionalData against XML document
AppSSBindingController checks if AppSSRegistrationList for this AppSSType exists
If (does not exist)
  Create AppSSRegistrationList for this AppSSType
End if
Generate AppSSID as key for the AppSS
Add AppSS registration to AppSSRegistrationListByType data structure
AppSSBindingController forwards OptionalData to OptionalDataHandler
OptionalDataHandler links OptionalData to AppSSID
AppSSBindingController returns AppSSID

Algorithm 5.5: Realization of Single AppSS Selection Runtime Configuration

Input: Pattern service’s AppSS Runtime Configuration component
Output: The runtime selection of an AppSS
Process: AppSSConfigurationController receives Single AppSS selection message with following parameter (AppSSID, AppSSType, OptionalData)
AppSSConfigurationController validates parameters
  Validate AppSSType against AppSSRegistrationDS
  Validate AppSSID against AppSSRegistrationDS
  Validate OptionalData against XML document
AppSSConfigurationController stores selected AppSSID in SelectedAppSSRegister data structure
AppSSConfigurationController forwards OptionalData to OptionalDataHandler
OptionalDataHandler links OptionalData to selected AppSSID
Algorithm 5.6: Realization of Chaining AppSSs Runtime Configuration

Input: Pattern service’s AppSS Runtime Configuration component
Output: The runtime selection of a chain of AppSSs
Process:
1. AppSSConfigurationController receives Chain AppSS message with following parameter (AppSSID[()], OptionalData)
2. AppSSConfigurationController validates parameters
   a. For each AppSSID in the chain
      i. Validate AppSSID against AppSSRegistrationDS
   b. Validate OptionalData against XML document
3. AppSSConfigurationController stores chained AppSSIDs in ChainedAppSSRegister data structure
4. AppSSConfigurationController forwards OptionalData to OptionalDataHandler
5. OptionalDataHandler links OptionalData to chained AppSSIDs

Algorithm 5.7: Pattern Service Implementation Level Realization

Input: Pattern service’s Program Logic component
Output: The delivery of the functionality of the PS
Process:
1. ProgramLogicController receives a message from the client
2. ProgramLogicController checks the type of target service method
   a. If (application-independent)
      i. Forward message to ApplicationIndependentProgramLogicController
      ii. ApplicationIndependentProgramLogicController invokes the proper service method
      iii. Service method uses ApplicationIndependentDS data structure
   b. Else
      i. Forward message to ApplicationDependentProgramLogicController
      ii. ApplicationDependentProgramLogicController invokes the proper service method
      iii. Service method uses ApplicationDependentDS data structure
3. End if
4. While (Interaction with AppSS layer is needed)
   a. Get AppSSID of target AppSS from SelectedAppSSRegister/ChainedAppSSRegister
   b. Call target service method in selected AppSS
   c. If (Callback to generic layer is needed)
      i. Service method in selected AppSS layer calls a service method in generic layer
   d. End if
5. End while
As an example, we apply the pattern service implementation realization procedure to the Observer pattern service. The Observer pattern service is an example of the pattern services whose client interacts directly with the AppSS part of the pattern service to trigger the sequence of actions that realizes the pattern service’s implemented functionality.

**Algorithm 5.8: Observer Pattern Service Implementation Level Realization**

**Input:** Observer pattern service component set (generic and application specific)

**Output:** The state of all Observer AppSS observing the Subject AppSS affected by the state change is updated

**Process:**
- Client calls SetState service method of the designated Subject AppSS
- SetState service method of the designated Subject AppSS service calls Notify service method of the PS generic layer providing SubjectAppSSID
  
  **Notify Procedure:**
  - Using the SubjectAppSSID, PS fetches the list ObserverAppSSID of all Observer AppSSs observing this Subject AppSS
  - Notify service method calls the Update service method of every Observer AppSS in the list of ObserverAppSSID
  
  **End Notify Procedure**
  
  **Update Procedure:**
  - Update service method of every Observer AppSS notified of the state change of the Subject AppSS calls the PS back by calling PS’s GetState service method
    
    **GetState Procedure:**
    - GetState service method of the PS, in turn, calls GetState service method of the Subject AppSS with the state change
      
      **GetState service method of the Subject AppSS:**
      - Returns state data to the GetState service method of the PS
        - GetState of the PS returns the state data it receives from the GetState service method of the Subject AppSS to the Update service method of the Observer AppSS
          
          **End GetState Procedure**
          
          **End Update Procedure**
5.5 Application-specific Pattern Service Generation

The next step in the process of developing a pattern service that implements a specific design pattern and can work seamlessly with the specific application at hand is to configure the PS to work according the rules and specifications of the application that will use the PS. The binding procedure integrates the AppSSs with the generic PS to make the two components work together. The application-dependent services in the generic pattern service, however, still need to be configured to add all application-dependent logic to them. This application-dependent logic is what enables the application-dependent service methods to know what application-dependent behavior they should execute and which service methods, in the AppSS section, they ought to interact with. Part of the application-dependent configuration to the application-dependent methods of the PS supplies the proper interfaces of the service methods in the AppSS part. The following is a formal definition of the process of configuring the application-dependent service methods and generating the application-dependent layer of the pattern service:

**Definition 5.9: Application-dependent Pattern Service Generation**

Given a generic layer of a design pattern service and given application-dependent configuration information, the generation of the application-dependent interface of that pattern service includes the generation of the signatures of services that implement application-dependent methods of the design pattern. The given application-dependent configuration information include proper signature for this service method, implementation information used to generate service method implementation from the
subject service template, and the signatures of any AppSS service methods this service method needs to interact with.

**Algorithm 5.9: Process for Generating Application-dependent Interface of a Pattern Service**

**Input:** The generic layer of PS, which includes application-dependent service signature templates and implementation template, and application-specific configuration information

**Output:** Complete application-specific pattern service

**Process:**
- Identify the application-dependent interface in the generic layer of the given PS
- For each service template in the application-dependent Interface
  - Find the application-specific signature of the service template in the given application-specific configuration information
  - Replace the service signature template by the application-specific signature of the service
  - Generate the application-specific implementation of the service from the template implementation of the service
- End for

To facilitate the easy use of the generic layer of a pattern service, there has to be good documentation on what AppSSs need to be developed, their types, and the functionalities of each type; this makes the job of AppSSCs and AppSPSCs much easier. In addition to that, the generic pattern service implementers must supply the rules and guidelines that those who use the generic patterns service in their applications need to abide by. To that end, we recommend that the PSCs provide an XML schema that AppSPSCs can use to create an XML file that carries all the binding and configuration information that can be used to register the AppSSs, created by the AppSSCs, with the generic layer of the PS, and apply all required configuration to the PS, in order to turn it into an application-specific pattern service (AppSPS).
The pattern service XML schema for a certain pattern service should have three sections: a generic section that is used with all pattern services, a pattern specific one that provides all the pattern specific configuration information, and a third section that applies any application-specific configuration to the pattern service. Every generic pattern service should have its custom schema that consists of the three parts explained above. AppSPSCs can use the schema to create their XML configuration files that configure and adapt the pattern service to the needs of their applications.

5.5.1 Generation Methodology

The integration of the generic layer of the pattern service with the AppSSs requires the provision of a binding interface and services on the part of the generic layer. Such an interface and its realization services and their service methods should enable the AppSSs – in the application-specific services layer – to register with the generic layer so that the interaction between the two layers becomes possible. We call this process the Binding process. PSCs that develop the generic part of the pattern service should provide services that are designed to enable such a binding process.

The steps of generation an AppSPS using a generic PS are identical to the reference implementation architecture shown in Figure 5.3. We have already explained the three main components of the architecture presented in the figure. Here we will just explain briefly the sequence of actions that take place to convert a generic pattern service into an application-specific one, then in the next subsection, we describe the structure of the configuration document that should be prepared and applied to the pattern service to perform the actual conversion of a PS to an AppSPS. So the steps of the methodology to generate an AppSPS using a generic PS can be described in the following:
1- **Developing the necessary application-specific services (AppSSs):** Although it is the AppSSCs responsibility and it is not the task that AppSPSCs usually perform, the generation of an AppSPS requires that any AppSS that implements part of the application-specific logic of the pattern service be implemented and its interface and URI are known.

2- **Configuring the generic pattern service (PS):** This step is at the heart of the duty of the AppSPSCs. Configuration of a generic pattern service to convert it into an application-specific one can, in turn, be divided into the following steps:

   a) **Studying the pattern service documentation:** Studying and understanding the documentation provided by the PSCs, who developed the generic PS, is a key element in the process of generating an AppSPS. Such documentation should explain the structure of the PS and highlight the application-independent and application-dependent components and interfaces of the PS. If well-written, it can guide the AppSPSCs through the process of putting together a complete configuration file that correctly converts the PS into an AppSPS.

   b) **Building the configuration file:** Good understanding of the PS documentation will enable AppSPSCs to put together a configuration file that contains all the information necessary for binding, configuring, and generating application-dependent code that turns the generic PS into an AppSPS. The file should contain configuration data to help bind AppSSs to the PS, generate the application-dependent set of interfaces, and configure application-dependent service method templates (AppDSTs) to generate the actual code for application-dependent service methods (AppDSMs). The following is a list of the required pieces of data that the configuration file should contain:
i. Application-independent interface set of the pattern service.

ii. Application-dependent interface set of the pattern service.

iii. Definition of application-specific service types required by the pattern service.


v. Names of pattern service methods that call service methods of application-specific services.

vi. Callback interface of the pattern service needed by the application-specific services.

vii. Binding information, which includes the URI and type of the application-specific service to be registered, plus any optional data required for the registration and usage of that application-specific service.

viii. Single AppSS selection runtime configuration information, if applicable.

   This includes, AppSSID and any optional data needed to facilitate the single AppSS selection process.

ix. Chain AppSS selection runtime configuration information, if applicable.

   This includes, an array of AppSSID and any optional data needed to facilitate the chain AppSS selection process.

c) Running the configuration routines: The final step in the AppSPS generation process is the execution of the configuration routines that will use the information supplied by the configuration file to bind the AppSSs to the generic PS and to adapt it to the specific requirements of the subject application, using the rest of the configuration data in the configuration file.
5.6 Application-specific Pattern Service Configuration File

We mentioned at the beginning of the last section that the AppSPSCs, who will be configuring the generic pattern services and turning them in application-specific ones, have to build an XML configuration file for each pattern service they intend to convert. Such an XML configuration file should contain all the necessary information to bind the AppSSs to the generic pattern service, configure the pattern service according to the application needs, and configure any templates that will generate the application-dependent program logic that the pattern service needs to be complete and ready for use.

For the AppSPSCs to be able to provide an XML file that conforms to the requirements and rules of the pattern service, they need to have some guidelines to follow in building that XML file. These guidelines are provided through the publication of a pattern service specific XML schema by the PCs, as part of the generic PS documentation. Although most of the configuration process is generic and is applicable to any design pattern, some aspects can be pattern-specific. For example, the types of AppSSs, the amount and nature of the optional data, and the pattern-specific set of interface can be different from one pattern service to another. So the AppSPSCs who choose to use a particular pattern service in their applications can use the XML schema of that pattern service to construct and validate their XML pattern service configuration file that is suitable for the subject pattern service.

5.6.1 XML Configuration Schema

For the AppSPSCs to create the proper XML configuration file to configure a particular generic design pattern service and turn it into an application-specific one, they need to have an XML schema that is tailored to the design pattern service at hand. The XML
schema for a pattern service serves two purposes. The first is to have validation controls that can be used to check the correctness of the XML configuration document that AppSPSCs create and run on the target pattern service to make it application specific. The second purpose is for the schema to act like a document that outlines the interfaces for all the components of the subject pattern service. This includes application-independent, application-dependent, and AppSS interfaces.

Generally, the schema for a pattern service should be divided into four main sections. The first section defines the simple types and reusable elements of the XML schema. These reusable types and elements are used throughout the schema when there is need for them. The second section is the one which validates XML code that applies configuration to the program logic of the pattern service. This includes specifying its interfaces and their relationships with the interfaces of the AppSS service methods. This section, in turn, is divided into two subsections. The first subsection establishes the rules for configuring the set of interfaces that the pattern service provides for the client’s use. This set of interfaces includes both application-independent and application-dependent service method interfaces. The second subsection, of the program logic part of the XML schema, takes care of validating the information about the interfaces of all the AppSS service methods and the interfaces of the callback service methods, the ones in the generic pattern service part, which the AppSS service methods may need to callback as part of the delivery of their functionalities. Remember that the configuration is only applied to the generic part of the pattern service; therefore, the information in this subsection can be used as documentation of what the interfaces of the AppSS service methods look like and what parameters they accept. Also, it is used in the
configuration of the AppISMs and AppDSMs of the generic part to let them know which AppSS service method they are to call when a call to an AppSS is needed.

The third main section of the general XML schema takes care of controlling the binding process of the AppSSs with the generic pattern service. The binding step comes before the steps in the configuration process, because the operations of the configuration steps may depend on the AppSSIDs that this step assigns to the registered AppSSs. The fourth and last section of the general XML schema is the one that controls the runtime configuration data that is used to select one or more AppSSs to participate in certain interactions of the pattern service. This section is also divided into two smaller subsections. The first subsection deals with the rules to control the runtime selection of a single AppSS, while the second subsection takes care of controlling the chaining of two or more AppSSs to use in chained AppSSs type of operations that some pattern services perform.

5.6.2 XML Schema for the Observer Pattern Service

As an example of what a pattern service specific XML schema may look like, we picked the Observer pattern service to show how to build an XML schema specific for the Observer pattern service. We use this Observer pattern service XML schema to create and validate an example XML configuration file that can be used to apply the necessary generic PS configuration, the binding of AppSSs, and the runtime configuration to the generic Observer pattern service. In the following few figures we show the contents of the XML schema, then we show the contents of the example XML configuration file in another sequence of figures.
Figure 5.4 to Figure 5.12 show the contents of the Observer pattern service specific configuration XML schema. Note that because of the size of the file, we split it into several figures.

```xml
<x:schema version="1.0"
  xmlns:x="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">

<!-- definition of AppSS Binding specific simple element -->
<x:simpleType name="AppSSRef" type="xs:string"/>

<!-- definition of Binding & Configuration simple elements -->
<!-- definition of AppSS Type simple element -->
<x:simpleType name="OptionalDataChoices">
  <xs:restriction base="xs:NMTOKEN">
    <xs:enumeration value="NONE"/>
  </xs:restriction>
</x:simpleType>
<x:element name="OptionalData" type="OptionalDataChoices"/>

<!-- definition of AppSS Type simple element -->
<x:simpleType name="AppSSTypeChoices">
  <xs:restriction base="xs:NMTOKEN">
    <xs:enumeration value="SUBJECT"/>
    <xs:enumeration value="OBSERVER"/>
  </xs:restriction>
</x:simpleType>
<x:element name="AppSSType" type="AppSSTypeChoices"/>

<!-- definition of whether am method returns a value simple element -->
<x:simpleType name="ReturnChoices">
  <xs:restriction base="xs:NMTOKEN">
    <xs:enumeration value="YES"/>
    <xs:enumeration value="NO"/>
  </xs:restriction>
</x:simpleType>
<x:element name="Return" type="ReturnChoices"/>

<!-- definition of Pattern Service simple element -->
<x:element name="AppSSID" type="xs:string"/>
<x:element name="MethodName" type="xs:string"/>
<x:element name="ParameterType" type="xs:string"/>
<x:element name="ParameterName" type="xs:string"/>
<x:element name="CallerPSMethod" type="xs:string"/>
</x:schema>
```

Figure 5.4: XML schema specific for the Observer PS
Figure 5.5 (Cont.): XML schema specific for the Observer PS
Figure 5.6 (Cont.): XML schema specific for the Observer PS
Figure 5.7 (Cont.): XML schema specific for the Observer PS
Figure 5.8 (Cont.): XML schema specific for the Observer PS
Figure 5.9 (Cont.): XML schema specific for the Observer PS
Figure 5.10 (Cont.): XML schema specific for the Observer PS
Figure 5.11 (Cont.): XML schema specific for the Observer PS
5.6.3 Example Configuration XML Document for the Observer Pattern Service

In this section we show the XML code and tags for an example XML document to configure an example Observer pattern service and turn it into an application-specific one. The Observer pattern service XML schema is used to validate this example
Observer pattern service configuration XML file. The following group of figures – Figure 5.13 until Figure 5.17 – show the contents of such an XML file. Note that because of the size of the file, we split it into several figures.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ObserverPatternService>
  <ClientInterface>
    <ApplicationIndependentInterfaces>
      <AppInterface>
        <Return>YES</Return>
        <MethodName>Attach</MethodName>
        <ParameterList>
          <Parameter>
            <ParameterType>String</ParameterType>
            <ParameterName>SubjectAppSSID</ParameterName>
          </Parameter>
          <Parameter>
            <ParameterType>String</ParameterType>
            <ParameterName>ObserverAppSSID</ParameterName>
          </Parameter>
        </ParameterList>
      </AppInterface>
    </ApplicationIndependentInterfaces>
  </ClientInterface>
  <AppInterface>
    <Return>YES</Return>
    <MethodName>Detach</MethodName>
    <ParameterList>
      <Parameter>
        <ParameterType>String</ParameterType>
        <ParameterName>SubjectAppSSID</ParameterName>
      </Parameter>
      <Parameter>
        <ParameterType>String</ParameterType>
        <ParameterName>ObserverAppSSID</ParameterName>
      </Parameter>
    </ParameterList>
  </AppInterface>
</ObserverPatternService>
```

Figure 5.13: Example XML document for configuring Observer PS
Figure 5.14 (Cont.): Example XML document for configuring Observer PS

```xml
<AppIIInterface>
  <Return>NO</Return>
  <MethodName>Notify</MethodName>
  <ParameterList>
    <Parameter>
      <ParameterType>String</ParameterType>
      <ParameterName>SubjectAppSSRef</ParameterName>
    </Parameter>
  </ParameterList>
</AppIIInterface>
</ApplicationIndependentInterfaces>

<ApplicationDependentInterfaces>
  <AppDInterface>
    <Return>YES</Return>
    <MethodName>GetState</MethodName>
    <ParameterList>
      <Parameter>
        <ParameterType>String</ParameterType>
        <ParameterName>SubjectAppSSID</ParameterName>
      </Parameter>
      <Parameter>
        <ParameterType>String</ParameterType>
        <ParameterName>ObserverAppSSRef</ParameterName>
      </Parameter>
    </ParameterList>
  </AppDInterface>
</ApplicationDependentInterfaces>
</ClientInterface>
```
Figure 5.15 (Cont.): Example XML document for configuring Observer PS
Figure 5.16 (Cont.): Example XML document for configuring Observer PS
5.7 Summary

In this chapter, we explained how the proposed methodology can be implemented and showed the overall framework of the artifacts that constitute a pattern service and how
they can be configured and integrated to work together in a specific application and deliver the required functionality. We proposed an implementation reference architecture diagram that describes the overall structure of the components that realize the pattern service methodology at the implementation level. Based on the proposed implementation reference architecture, we first explained the binding process between the generic layer of the pattern service and the AppSS one. Then we described the runtime configuration procedures that can be executed on the pattern service. We gave some algorithms that explain how these processes can be realized. In the AppSPS generation section, we demonstrated how the generic pattern service can be configured and what stages of configuration a generic pattern service needs go through to be converted into an application-specific pattern service. To further specify and explain the configuration requirements for the proposed pattern services, we proposed building an XML schema document for each pattern service that can be used as the governing body for the XML configuration file that AppSPSCs create to configure that generic pattern service and make it application specific. Finally, we gave an example XML schema for the Observer pattern service and an example XML configuration file that is built and validated according to the example Observer pattern service XML schema specifications.

In the coming chapter we describe some of the use cases where the proposed methodology can be applied and pattern services can be put to work. First, we show an example of how pattern services can be composed to fulfill certain functionalities. Then, we show how the proposed methodology can be used to implement SOA design patterns as pattern services.
Chapter 6

Case Studies

In this chapter, we show how pattern services can be used in the design and implementation of some case study software systems. We first give an example of pattern service composition. This example combines Command and Memento design pattern services (See Appendix A, Sections A.6 and A.15) to make them deliver a specific complex functionality. The result of the two pattern service composition is an Undoable Command pattern service. This composition can provide the do/Undo functionality to any application that requires it. Then we show how this pattern composition can be used to provide the do/undo functionality to a chess game example application.

In the second part of this chapter, we discuss how pattern services can be used in Service-Oriented Architecture field and the development of Service-Oriented systems, and give some examples of how pattern services can be used in the design and implementation of some SOA patterns. We also list some of the SOA design patterns that either use one or more of the 23 GoF design pattern or share some similarities, in the structure or functionality, with those design patterns.

6.1 Chess Game Application using Command and Memento Pattern Service Composition

The Undoable Command pattern service composition is an example of using more than one pattern service in the effort to provide a combined functionality. The purpose of the Undoable Command pattern service composition is to design a pattern service that is
actually a composition of two pattern services, which can be used to do and undo actions or operations. Undoing an action should reverse all of the changes that a do action applied to an application. In a pattern service context, this requires a pattern service that enables the execution of commands which apply the required changes to the target software artifacts, and another pattern service that helps in keeping track of the changes and supporting the undo action by supplying all the necessary data to restore those artifacts to their previous state.

6.1.1 Overview of the Pattern Service Composition

To design the Undoable Command pattern service composition that fulfills the stated requirements above, we need to make use of two pattern services, namely, the Command and Memento pattern services. Depending on the application creator’s choice, and the requirements of the software application, there can be at least three variations of the Undoable Command pattern service composition implementation. In other words, the Command and Memento pattern services can be combined in at least three different ways to perform the do/undo action. To help explain the structure of the Undoable Command pattern service composition, we describe the three variations of such a pattern service composition in the context of them being used in a chess game software application.

Before we delve into the details of how the Undoable Command pattern service composition can be used in the chess game to take care of facilitating the “move/undo move” a chess piece in the game – which is equivalent to the do/undo action – we describe the general functionality of a chess game software application, focusing mostly
on the areas where the Undoable Command pattern service is to be used to support the “move piece /undo move piece” actions of the game.

6.1.2 Chess Game Application using Undoable Commands

In general, a chess game is composed of a chess board that has two sets of chess piece of two different colors – usually a black set and a white one. Two players compete in the chess game by moving pieces of their sets to attack the opponent’s. In a real chess game, once the players move one of their chess pieces, they are not permitted to undo that move. However, in training or in a rather friendly game, the undo action is usually permitted. To demonstrate how the Undoable Command pattern service composition can be used to implement the move and undo move actions in a chess game, we assume that the chess game permits the undo action.

Let us first start this discussion by presenting an example design of a chess game that shows the software components that cooperate to deliver the chess game functionality. The system design shown in Figure 6.1 suffices to give an idea of what a chess game software application may consist of. As far as the Undoable Command pattern service composition is concerned, what interests us in the diagram in Figure 6.1 are the Player, Game, and Move classes, because they are the components that are either initiate the do/undo action, or they perform the actual actions and reflect their effect on the game.

Assuming that the chess game in Figure 6.1 needs to support the functionality that enables players to undo their chess piece moves, using the Undoable Command pattern service composition provides just that type of functionality.
Figure 6.1: Chess Game Application Design (Adapted from http://massivetechinterview.blogspot.com/2015/07/design-chess-game-using-oo-principles.html)
In the context of the chess game and as far as the do/undo actions are concerned, the actions of moving chess pieces and undoing those moves can be controlled by the Undoable Command pattern service composition. Such a control means that the every player’s action has to go through the Undoable Command pattern service composition. This helps the Undoable Command pattern service composition keep track of the moves so that undoing those moves and applying the necessary adjustments to the game is possible.

Now that we have laid the ground for what the Undoable Command pattern service composition will be responsible for, it is time to discuss the possible ways that such a composition can be implemented to perform its task. We mentioned earlier in this section that there can be at least two ways in which the Undoable Command pattern service composition can be designed and implemented to carry out the do/undo actions. In the rest of this section we describe the structure of the Undoable Command pattern service composition in each of the three proposed ways, in the context of a chess game application.

1- The first design that can be adopted to implement the Undoable Command pattern service composition uses the Command and Memento pattern services as two separate and independent services; the chess game AppSS takes care of coordination between these two pattern services. Figure 6.2 gives the overall structure of the way the Undoable Command pattern service composition can be used by the chess game to support the do/undo of player actions.
To establish interaction between the two pattern services and the Chess Game AppSS, the Chess Game AppSS must register itself with the Command pattern service as the receiver of the Execute/Unexecute commands, and with the Memento pattern service as the originator of the memento data. In the figure, the player actions, namely, Move Piece and Undo Move, invoke the Execute and Unexecute service of the Command pattern service respectively. Figure 6.2 shows how the Undoable Command pattern service composition can be used with the chess game to provide the do/undo action according to the first design of the Undoable Command pattern service composition.

Figure 6.3 presents the interaction diagram that shows the sequence of actions and operations that take place in a chess game application that employs the first proposed Undoable Command pattern service composition.
When the player moves a chess piece, the Execute service of the Command pattern service is executed, which in turn invokes the Action service, i.e., movePiece, in the Chess Game AppSS, which starts the process of applying the changes related to moving a chess piece to the chess board. Before moving that chess piece and applying the changes to the chess board, however, the movePiece service calls the CreateMemento service of the Memento pattern service to enable it to store the current state of the chess board before the changes are applied. The CreateMemento service invokes the CreateMemento service implemented by the Chess Game AppSS to get the chess board state data. Once received, the Memento pattern service stores the chess board state data in its memento repository.

Upon completion of the memento process, the chess piece move action is applied to the chess board and by doing that, the player’s Piece Move action is complete.
The second player’s possible action is to request the undoing of the last chess piece movement action. This is started by the user invoking the Unexecute service of the Command pattern service through the Undo Move action. To perform the action of undoing a chess piece move, the Unexecute service invokes the Unaction service, i.e. the undoMove service in the Chess Game AppSS, which starts the process of restoring the chess board to its previous state. The undoMove service calls the SetMemento service of the Memento pattern service. The SetMemento service of the Memento pattern service retrieves the relevant chess board state date from the memento repository and forwards it to the SetMemento service of the Chess Game AppSS. The SetMemento service, implemented by the Chess Game AppSS, applies the chess board data it receives from the Memento pattern service to the chess board to restore it to its state before the last chess piece move action.

2- The second design of the Undoable Command pattern service composition is to make the Command pattern service support the undo action by internally using the Memento pattern service. In this design, the Command pattern service hides or encapsulates the Memento pattern service from the application, chess game in this case. The application only deals with the Command pattern service. The Command pattern service undertakes the responsibility of storing and managing memento data by using the Memento pattern service internally.

Figure 6.4 gives the design of the second proposed Undoable Command pattern service composition, in which only the Command pattern service directly interacts with the Chess Game AppSS component. Notice from the diagram that the Chess Game AppSS must register with the Command pattern service as the receiver of the Execute/Unexecute commands, but does not have to register with the Memento
pattern service. Instead, the Command pattern service registers with the Memento pattern service as the originator of the memento data and the Memento pattern service registers with the Command pattern service as an AppSS so that Command can forward chess game state data to the Memento pattern service and can retrieve memento data back from it.

![Chess Game using Undoable Command PS – second design](image)

**Figure 6.4: Chess game using Undoable Command PS – second design**

According to this design, the implementation of the Execute service of the Command pattern service should first call the getState service of the Chess Game AppSS to obtain the memento/state data and pass it to the Memento pattern service before calling the Action service of the AppSS. Once the Memento pattern service stores the memento data, the Execute service of the Command pattern service finally calls the Action service of the Chess Game AppSS to perform the chess piece moving action. Figure 6.5 shows the sequence of actions that are performed to provide the do/undo functionality for the chess game application using the second design of the Undoable Command pattern service composition.
To provide for the undo functionality in this design, the implementation of the Unexecute service of the Command pattern service first gets the saved memento/state data from the Memento pattern service, and then it invokes the Chess Game AppSS setState service to change the AppSS state accordingly.

Finally, the Unaction service of the Chess Game AppSS is invoked to apply the required changes to the chess game board and put the game back to the state before the last player’s chess piece move operation.

6.2 Using Design Pattern Services to Implement SOA Patterns

6.2.1 Introduction to SOA Patterns

The purpose of the SOA patterns is to solve problems that arise when building distributed systems that adopt the Service-Oriented paradigm. While SOA has its own set of patterns that are solely devised to solve service-oriented system design and
implementation problems, a considerable number of the SOA patterns are derived from, influenced by, or associated with patterns that trace back to established design concepts, approaches, and previously published design pattern catalogs (Erl, 2009).

Figure 6.6: The primary influences of SOA design patterns (after Erl (2009, p. 90))

Figure 6.6 shows the different design pattern areas that influenced the establishment of the SOA design patterns, beginning with Alexander’s pattern idea and how a group of patterns can form a pattern language. In the software development field, the first set of design patterns was the GoF OO design patterns. Later on, a few groups of patterns stemmed from OO design patterns to solve several design and architectural problems that software application development teams ran into. Other design patterns that came from areas such as software architecture design patterns, enterprise application architecture design patterns, enterprise application integration design patterns, and other fields and areas of software application development.
It is worth noting that according to Erl (2009), SOA design patterns are further divided into three main categories: Service Inventory Design Patterns, Service Design Patterns, and Service Composition Design Patterns. Each of these three categories includes a group of SOA design patterns that are further arranged under smaller subcategories. The Service Inventory Design Pattern category includes design patterns that solve problems about building, managing, and securing SOA inventory of services. The Service Design Patterns category houses the SOA design patterns concerned with designing, implementing, interacting with, and securing the services themselves. Lastly, the Service Composition Design Patterns category contains SOA design patterns that facilitate the composition of services to make services interact with each other in certain contexts, using certain technologies in order to achieve more complex and coarse-grained functionalities.

6.2.2 Applying General Pattern Service Methodology to SOA Patterns

The principal rule in the proposed methodology to implement a software pattern as a pattern service is to first separate the generic components of the subject pattern from the non-generic or application-specific portions. Then the generic components are implemented as a generic pattern service, while the application-specific parts are left for the PbAppCs to develop according to the needs of their applications. The implemented generic pattern service must provide PbAppCs with the means to attach their AppSSs and facilitate the proper interaction means with those AppSSs.

Applying the proposed pattern service methodology to SOA patterns should follow these broad guidelines. This means that implementing an SOA pattern as a pattern service starts with the extraction of the abstract generic parts of the SOA pattern. Once
all the generic functionality of the SOA pattern is defined, the pattern service design for this functionality is laid out. The design of the generic pattern service must always include all needed additional data structures, relationships, interfaces, and services that are necessary for the attachment and integration of any needed AppSSs. To further explain how the proposed methodology can be used to implement an SOA pattern as a pattern service, we use the Request/Reaction pattern (Arnon, 2012) as an example.

6.2.2.1 Request/Reaction SOA Pattern

Synchronous communication is often important, but it is not always desirable. In Service-Orientation, synchronous interactions mean that the service consumer needs to sit and wait for the service to finish processing the request before the consumer can continue with whatever it was doing.

![Request/Reaction SOA Pattern](image)

**Figure 6.7: Request/Reaction SOA pattern** after Arnon (2012, p. 116)

There are situations where the service consumer does not want or cannot afford to wait but is still interested in getting a reply when it is available (Arnon, 2012). That is why a
pattern such as Request/Reaction that enables asynchronous communication between services and their consumers is required. Figure 6.7 shows the structure of the Request/Reaction SOA pattern. The intent of the Request/Reaction pattern as presented in Arnon (2012) is to “Temporally decouple the request from a service consumer and the reply from the service” (p. 115).

6.2.2.2 Request/Reaction Pattern Service

Since the Request/Reaction SOA pattern mainly regulates the asynchronous interaction between a service and its consumers, we can place the generic pattern service as a middle man that regulates such asynchronous interaction. In accordance with this proposition, the generic part of the Request/Reaction pattern service should implement the services that the AppSSs can use to register themselves and be able to interact with the generic part of the pattern service. Figure 6.8 depicts the design and structure of the Request/Reaction pattern service. The Target Service AppSS Registration List in Figure 6.8 is responsible for storing references to the services that are the target of the asynchronous communication. The Reaction Setting Service is used to associate the proper acknowledgement message with the client before the actual result of the operation is returned. These acknowledgements are stored in the Reaction Map with association links to the proper AppSSs.

The AppSS Dependent Interface defines the Request Service that the client calls to perform the asynchronous service call. In the process of making the asynchronous call, the client provides the proper address to which the pattern service can send the response that carries the actual results of the requested operation.
The application service as Reaction Setter is used to supply the generic pattern service with acknowledgement messages that are then associated with the registered AppSSs so that the pattern service returns the right acknowledgement message to the client while the actual request is being processed.

The sequence of interactions between the Request/Reaction pattern service and its client begins by the client selecting the Target Service AppSS to use. This enables the pattern...
service to identify the target service to interact with and what acknowledgement message to return to the client.

Having identified the Target Service AppSS to interact with, the client can call the Request Service of the pattern service. Upon the reception of the request from the client, the pattern service fetches the proper acknowledgement message associated with the selected Target Service AppSS and dispatches it to the client. At the same time the pattern service gets the reference of the selected Target Service AppSS from the list and calls the Request service method defined by that AppSS, passing it all the required parameters, and waits for the results to be returned. Once the pattern service receives the results that are returned by the called Request service method of the selected Target Service, it forwards them to the client at the address that the client application service provided as a parameter during its call to the Request Service of the pattern service.

6.2.3 Design Pattern-based SOA Patterns

Due to the fact that SOA patterns are in fact software patterns, they eventually have some similarities with the other types of software patterns such as analysis, architectural, and design patterns and share some of their characteristics. In this section we classify a group of SOA patterns that we studied in Arnon (2012) and Erl (2009) according to their relationship to the software design patterns, which are the main focus of this thesis and are discussed in the previous chapters. The two books contain about a hundred SOA patterns that are categorized according to their purpose or the service-oriented system building stage at which they are applied. Examples of these categories of SOA patterns include SOA Inventory patterns, Service Design patterns, Service

While some of these categories are common to most software patterns such as design, implementation, and security, others are more SOA specific categories and they include patterns that have fewer relationships to other software patterns. In the following sections of this chapter we create a table that lists the SOA patterns that carry some similarity or can be implemented using one or more software design patterns.

As mentioned in section 6.2.1 some of the SOA patterns are either derived from the GoF OO design pattern or have some resemblance in their structure or functionality.

As Erl (2009) puts it, “The set of 23 patterns produced by GoF expanded and helped further establish Object-Orientation as a design approach for distributed solutions. Some of these patterns have persisted within the Service-Orientation, albeit within an augmented context and new names” (p. 91). This means that most SOA patterns employ some other patterns from other fields to solve service-oriented system problems. In this research we are only interested in the GoF design patterns aspect. Table 6.1 lists some of the SOA patterns that derive from, are linked to, or may use one of the GoF OO design patterns.

To give an example of how design pattern services can be used to implement SOA patterns, in the following subsection we will describe the steps of using the Façade design pattern service in implementing the Current Contracts SOA pattern.
<table>
<thead>
<tr>
<th>S. N.</th>
<th>SOA Pattern</th>
<th>OO Design Pattern</th>
<th>S. N.</th>
<th>SOA Pattern</th>
<th>OO Design Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Active Service</td>
<td>Observer</td>
<td>19</td>
<td>Non-Agnostic Context</td>
<td>Mediator</td>
</tr>
<tr>
<td>2</td>
<td>Capability Composition</td>
<td>Composite</td>
<td>20</td>
<td>Protocol Bridging</td>
<td>Adapter and Bridge</td>
</tr>
<tr>
<td>3</td>
<td>Capability Recomposition</td>
<td>Composite</td>
<td>21</td>
<td>Proxy Capability</td>
<td>Proxy</td>
</tr>
<tr>
<td>4</td>
<td>Command Invoker</td>
<td>Command</td>
<td>22</td>
<td>Request Mapper</td>
<td>Factory Method and Command</td>
</tr>
<tr>
<td>5</td>
<td>Concurrent Contracts</td>
<td>Façade</td>
<td>23</td>
<td>Response Mapper</td>
<td>Builder and Factory Method</td>
</tr>
<tr>
<td>6</td>
<td>Data Format Transformation</td>
<td>Adapter</td>
<td>24</td>
<td>Service Connector</td>
<td>Façade and Proxy</td>
</tr>
<tr>
<td>7</td>
<td>Data Model Transformation</td>
<td>Adapter</td>
<td>25</td>
<td>Service Controller</td>
<td>Command</td>
</tr>
<tr>
<td>8</td>
<td>Decoupled Contract</td>
<td>Bridge</td>
<td>26</td>
<td>Service Decomposition</td>
<td>Façade</td>
</tr>
<tr>
<td>9</td>
<td>Distributed Capability</td>
<td>Façade</td>
<td>27</td>
<td>Service Façade</td>
<td>Façade</td>
</tr>
<tr>
<td>10</td>
<td>Edge Component</td>
<td>Façade and Proxy</td>
<td>28</td>
<td>Service Firewall</td>
<td>Façade and Proxy</td>
</tr>
<tr>
<td>11</td>
<td>Event-Driven Messaging</td>
<td>Observer</td>
<td>29</td>
<td>Service Perimeter Guard</td>
<td>Proxy</td>
</tr>
<tr>
<td>12</td>
<td>File Gateway</td>
<td>Mediator</td>
<td>30</td>
<td>Service Refactoring</td>
<td>Façade</td>
</tr>
<tr>
<td>13</td>
<td>Intermediate Routing</td>
<td>Mediator</td>
<td>31</td>
<td>Stateful Services</td>
<td>Memento</td>
</tr>
<tr>
<td>14</td>
<td>Inventory Endpoint</td>
<td>Proxy</td>
<td>32</td>
<td>State Messaging</td>
<td>Flyweight</td>
</tr>
<tr>
<td>15</td>
<td>Inversion of Communication</td>
<td>Observer</td>
<td>33</td>
<td>State Repository</td>
<td>Memento</td>
</tr>
<tr>
<td>16</td>
<td>Legacy Wrapper</td>
<td>Adapter and Proxy</td>
<td>34</td>
<td>UI Mediator</td>
<td>Mediator</td>
</tr>
<tr>
<td>17</td>
<td>Message Screening</td>
<td>Proxy</td>
<td>35</td>
<td>Virtual Endpoint</td>
<td>Façade and Proxy</td>
</tr>
<tr>
<td>18</td>
<td>Multi-Channel Endpoint</td>
<td>Adapter</td>
<td>36</td>
<td>#</td>
<td>#</td>
</tr>
</tbody>
</table>

Table 6.1: List of SOA patterns that apply design pattern ideas
1- Concurrent Contracts SOA Pattern

The purpose that the Concurrent Contracts SOA pattern serves as stated by Erl (2009) is to make a service facilitate multi-consumer coupling requirements and abstraction concerns at the same time. This can be achieved by creating multiple contracts for a single service; each is aimed at a specific type of consumer. Figure 6.9 shows the structure of the Concurrent Contracts SOA pattern.

As can be seen from Figure 6.9, using the Concurrent Contracts SOA pattern we can create more than one contract for a particular service. The contracts are the means by which consumers know how to interact with the service. The service guarantees that it fulfills all the service methods defined in the Service Contract. The need for more than one contract per service may arise from the need to offer different functionalities to different types of consumers. The service must implement all the service methods promised by all the contracts of that service, but each contract may contain only a subset of those service methods that the service implements. Based on the type of consumer, they are given different Service Contracts to use when they interact with the service.

The Façade design pattern is used in Figure 6.9 to facilitate and coordinate the interaction between the service and its consumers, each according to the Service Contract that guides their interaction with the service.
Concurrent Contracts Implementation Using Façade Design Pattern Service

The Façade layer of the Concurrent Contracts SOA pattern can very easily be replaced by the Façade design pattern service (See Appendix A, Section A.9). Figure 6.10 shows the creation of the mapping between the Façade design pattern service and the Core Service Logic AppSSs.

After the registration of the Dispatcher and Core Service Logic AppSSs with the Façade design pattern service, this mapping is done using configuration files that contain details of the different contracts that each Core Service Logic supports.
Figure 6.10: Façade to Core Service Logic mapping step

This mapping is very important for the Façade design pattern service. It helps it identify which Façade to use with the different consumers that would be requesting to interact with the registered Core Service Logic AppSSs.

Once the registration of the AppSSs and the configuration step, to map contracts to different consumers and Core Service Logic AppSSs, are completed, the generation of the AppSS dependent interface and its implementation services is performed and the Concurrent Contracts SOA pattern that uses the Façade design pattern service is ready for use. Figure 6.11 depicts the Concurrent Contracts SOA pattern after using the Façade design pattern service to implement it. Note that in Figure 6.11, the Core Logic
Setter application service is now used to select the current Core Service Logic to deal with.

Figure 6.11: Concurrent Contracts SOA pattern using Façade PS

The interaction with the design pattern service begins by a consumer requesting to call one of the service methods permitted by the consumer’s service contract and supported by the currently selected Core Service Logic AppSS. The client application service
receives the request and forwards it to the design pattern service by invoking the Façade Operation Service of the design pattern service.

The Façade Operation Service calls the Operation service of the Dispatcher AppSS. The Operation service of the Dispatcher AppSS contains a callback to the Core Logic Mapping Callback Service of the design pattern service. This service accesses the Façade to Core Logic Mapping to determine if there is a mapping between the consumer’s contract and the specific service method of the Core Service Logic. If a mapping is found, the requested operation is performed and the target service method of the Core Service Logic AppSS is called and executed.

6.3 Discussion

The relationship, shown in Figure 6.6, between design patterns and SOA patterns depicts the fact that design patterns can be used as building blocks for SOA patterns. In fact, just as shown in Table 6.1, an SOA pattern may employ one or more design patterns to deliver its functionality. An example of SOA patterns that use more than one design pattern is the Service Firewall SOA pattern. It uses the Façade design pattern for the part responsible for receiving requests and deploying them to the proper service and uses the Proxy design pattern to filter those requests and may execute certain logic before allowing the requests to be dispatched to the underlying services.

Also, SOA patterns can be used together with design patterns to implement different SO systems’ functionalities. Regardless of whether the SOA patterns used in the design and implementation of a SO system are built on design patterns or not, independent design patterns can be used to provide solutions to part of the design problems that the subject
SO system faces. An example that supports this claim is the use of the Observer design pattern in the service-orientated design of the example Stock Data Inquiry application described in Chapter 3. Regardless of whether other SOA patterns are used to help implement the example application or not, the fact that the application can be designed and implemented as services based on the proposed structure of the Observer design pattern is a good example of the possibility of coexistence of SOA patterns and design patterns in the same application without being closely tied or one being part of the other.

Although SOA patterns are mainly architectural, while design patterns have a slightly finer granularity than SOA patterns, which means that they are used to design the components and the details of the sub-systems, an SO system can be designed and implemented entirely using either SOA patterns or design patterns. SOA patterns are designed specifically for SO systems; however, design patterns are generic and flexible enough to be used to design and implement almost any software system. Apart from the need to implement them in the form of services, the inner structure and algorithm implementations of SO systems are just like any other software development paradigm. This means that design patterns can be implemented in the form of interacting and cooperating services that constitute the necessary components of an SO system.

It probably takes a homogeneous mixture of SOA patterns and design patterns to design and develop the most robust and scalable SO system. However, either of the two types of patterns can be used solely to design and implement an SO system. Take for example the Stock Data Inquiry application that we introduced in Chapter 3. The application can be implemented using SOA patterns as an SO system without using the Observer design pattern and the system should work. Conversely, the same application can be
implemented as a group of service using the Observer design pattern without applying any of the SOA patterns, and the solution still works. In this particular situation SOA patterns and design patterns can appear as competing patterns, however, limiting the design and implementation of SOA systems to only one of the two types can have some advantages and some disadvantages at the same time. The pros of using SOA patterns to develop an SO system would be that the system components are architecturally built according to the recommendations and best practices of SO systems which take into consideration that the interacting artifacts of the system are actually services, which may dictate certain behaviors and/or limitations to be observed. On the other hand, using only design patterns to design and build an SO system means that the SO system will benefit from the strength and good solution quality that design patterns are known for, but it may lack the agile architectural design of the whole system due to the lack of SOA patterns in building the architectural structure of that SO system.

To conclude this discussion, it is important to note that SOA patterns and design patterns – for the most part – are not competing patterns; rather, they are cooperating ones in such a way that one type can use some of the functionalities or services that the other type provides, and vice versa.

### 6.4 Summary

In this chapter we discusses some use cases that demonstrate how the pattern service methodology can be applied to different software pattern fields. Firstly, we gave an example of composing the Command and Memento design pattern services into an Undoable Command design pattern service. Secondly, we demonstrated how such a design pattern service composition can be useful in some applications. We used the
example of a chess game application to show how the Undoable Command design pattern service composition can be used to support the do/undo moves of the game. Thirdly, we elaborated on how the proposed pattern service methodology can be used to implement SOA patterns. We also classified some of the SOA patterns based on their relationship to the GoF design patterns. Then, as an example of how the proposed methodology can be used with SOA patterns, we showed how the Concurrent Contracts SOA pattern can be implemented using the Façade design pattern service. Finally, we discussed the use of SOA patterns and design patterns in SO systems and how the two types of patterns can be used either separately or together to build SO systems. We also explained the pros and cons of using one type of patterns without the other in building SO systems.

In the first section of Chapter 7 we introduce the design of the Pattern as a Service (PaaS) system. Then we talk about the prototypical implementation of the system. After that we talk about the case study applications – the Online Discussion Group (ODG) and Online Stock Market Ticker (OSMT) - and how we implemented the Observer design pattern as a design pattern service and used it to implement the two case study applications.
Chapter 7

Pattern as a Service (PaaS) System Design

7.1 Introduction

In this chapter, we present the design of the Pattern as a Service (PaaS) system that we designed and implemented to control and manage the pattern services and how they can be used in software applications. The PaaS system also manages the storage and instance deployment of those pattern services and the application components that use them.

In our presentation of the PaaS system we start by describing the functionality the system delivers then we explain the architecture and design of the system and show the sequence of actions and interactions that take place to deliver the different functionalities of the system. After that we talk briefly about the prototypical implementation of the system. Finally, we conclude this chapter by presenting two case study systems that put the PaaS system to work and use the Observer pattern service in their work. The first case study software application is an online discussion group application that uses the Observer pattern service to facilitate the discussion between groups of people belonging to different discussion rooms. The second case study software application is an online stock market ticker. The stock market ticker also uses the Observer pattern service to provide the stock market information to the users by having the users create a stock market portfolio of stock symbols and subscribing to the ticker service using the Observer pattern service. Of course, both case study software
applications and the Observer pattern service are stored, deployed, and managed using the PaaS system.

### 7.2 Functionality and Use Cases

To enable the usage of pattern services in building Pattern-based service-oriented systems, we need to design and implement a system that governs and facilitates such usage. To fulfill this requirement, we designed a system we call Software Pattern as a Service (SPaaS) or PaaS for short.

#### Table 7.1: Functionalities supported by PaaS system

<table>
<thead>
<tr>
<th>S.N.</th>
<th>Functionality</th>
<th>Actors</th>
<th>Use Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>PS Management</td>
<td>PSC</td>
<td>Create, store, document, and manage generic pattern services.</td>
</tr>
<tr>
<td>2</td>
<td>PS Lookup</td>
<td>AppSSC, AppSPSC</td>
<td>Search for and select generic pattern services.</td>
</tr>
<tr>
<td>3</td>
<td>AppSS Management</td>
<td>AppSSC</td>
<td>Create, store, document, and manage application-specific services.</td>
</tr>
<tr>
<td>4</td>
<td>AppSS Lookup</td>
<td>AppSPSC</td>
<td>Search for and select application-specific services.</td>
</tr>
<tr>
<td>5</td>
<td>AppSPS Conversion</td>
<td>AppSPSC</td>
<td>Convert generic pattern services into application-specific services.</td>
</tr>
<tr>
<td>6</td>
<td>AppSPS Management</td>
<td>AppSPSC</td>
<td>Store, document, and manage application-specific pattern services.</td>
</tr>
<tr>
<td>7</td>
<td>AppSPS Lookup</td>
<td>PbAppC</td>
<td>Search for and select application-specific pattern services.</td>
</tr>
<tr>
<td>8</td>
<td>AppSPS Composition</td>
<td>PbAppC</td>
<td>Composition of one or more application-specific pattern services to achieve a complex functionality.</td>
</tr>
<tr>
<td>9</td>
<td>PbApp Management</td>
<td>PbAppC</td>
<td>Compose, Store, document, and manage pattern-based applications.</td>
</tr>
<tr>
<td>10</td>
<td>PbApp Lookup</td>
<td>PbAppIC</td>
<td>Search for and select ready to deploy pattern-based applications.</td>
</tr>
<tr>
<td>11</td>
<td>PbApp Deployment</td>
<td>PbAppIC</td>
<td>Configure and deploy instances of the chosen pattern-based applications.</td>
</tr>
</tbody>
</table>

190
The main functionalities that the PaaS system should provide are as follows:

1- **PS Management**: Facilitates the creation, composition, storage, and documentation of generic Pattern Services (PSs; See Glossary for definition).

2- **PS Lookup**: Facilitates the search for and selection of pattern services by two types of users. The first type of user is the AppSPSCs. They use this functionality to search for generic PSs to configure and convert into AppSPSs. The second type of user is the AppSSCs, who need to search for generic PSs when they need to create the AppSSs required to complement the generic PSs’ functionalities.

3- **AppSS Management**: Facilitates the creation, storage, and documentation of Application-specific Services (AppSSs). AppSSCs use this functionality to help create, store, and document the AppSSs that they need to use with the selected generic pattern service.

4- **AppSS Lookup**: Facilitates the search for and selection of application-specific services by the AppSPSCs. AppSPSCs use this functionality to search for AppSSs to use them in the application-specific service layer of their AppSPS.

5- **AppSPS Conversion**: Facilitates the process of making generic pattern service application specific. AppSPSCs use this functionality to apply all their application-specific configuration to the generic pattern service – including AppSS binding and application-dependent code generation – to convert the generic pattern service into an application-specific pattern service.

6- **AppSPS Management**: Facilitates the storage and documentation of AppSPSs. AppSPSCs use this functionality to store, document, and manage the AppSPSs that are created as a result of performing the AppSPS conversion functionality in (4).
7- **AppSPS Lookup**: Facilitates the search for and selection of application-specific pattern services by the PbAppCs. PbAppCs use this functionality to search for AppSPSs to use them in their pattern-base applications.

8- **AppSPS Composition**: Facilitates the composition of application-specific pattern services and additional application services to form Pattern-based Applications (PbApps). PbAppCs use this functionality to use two or more of the AppSPSs, produced by the AppSPSCs, in their PbApps.

9- **PbApp Management**: Facilitates the storage and documentation of PbApps. PbAppCs use this functionality to store, document, and manage the PbApps that they created as a result of combining some AppSPS together with other service.

10- **PbApp Lookup**: Facilitates the search for and selection of PbApps in order to create, configure, and deploy as many Pattern-based Application Instances (PbAppIs; See Glossary for definition), as required.

11- **PbAppI Deployment**: Facilitates the configuration and deployment of PbAppIs. PbAppICs use this functionality to apply instance specific configuration to PbApps and deploy as many instances as needed.

These functionalities are used by five types of users:

1- **Pattern Service Creators (PSCs)**: This type of user represents the pattern experts that will be responsible for creating generic pattern services and managing them using the PS Management functionality of the PaaS system.

2- **Application-specific Pattern Service Creators (AppSPSCs)**: This second type of user represents those that select generic pattern services and configure them to turn them into application-specific pattern services.
3- **Application-specific Service Creators (AppSSCs):** They are the users that develop application-specific services that provide the components that complement the functionality of a generic pattern service.

4- **Pattern-based Application Creators (PbAppCs):** This type of user includes those that build pattern-based application using application-specific pattern services, application-specific pattern service compositions, and other application services necessary for developing Pattern-based applications.

5- **Pattern-based Application Instance Creators (PbAppICs):** This type of user includes the immediate users of the PbApps, who select ready PbApps, create instances of these PbApps, configure the created instances, and deploy them to their end users.

In the following, we list all the use cases the PaaS system supports. We arrange them by the user type or actor. For each use case, we give a brief description of the purpose, the actions performed, and the achieved results.

1- **Pattern Service Creator (PSC):**

   a) **Pattern service creation and management:**

      **Purpose:** The purpose of this use case is to create and manage generic pattern services to be reused in different applications after being configured according to the subject application requirements.

      **Outcome:** The result of this use case is the implementation of the generic layer of a pattern service based on some software pattern. Also, documentation of the implemented generic pattern service may be produced.
**Actions:** The process of performing this use case involves studying the subject software pattern, identifying the pattern abstract and concrete layers, identifying the generic logic that goes in the generic layer of the target pattern service, and implementing that logic in the generic layer of the pattern service with all proper service methods and configurable service method templates that support the generic pattern service functionality.

2- **Application-specific Service Creator (AppSSC):**

a) **Search for and select generic pattern services:**

**Purpose:** AppSSC searches for and selects generic pattern services to study their documentation and create the appropriate AppSSs to implement the concrete application-specific layer that will be integrated with the generic layer of the pattern service to deliver the full functionality of the pattern service.

**Outcome:** The location and selection of the required generic pattern service.

**Actions:** Use of search tools that the pattern service repository provides to search for generic pattern services that fit the supplied search criteria.

b) **Application-specific service creation and management:**

**Purpose:** Implement and manage all AppSSs that are required to complement the functionality promised by the generic layer of the selected pattern service.

**Outcome:** The creation and safe keeping of one or more AppSSs that implement the application-specific logic for the given pattern service.

**Actions:** Study the subject generic pattern service, and then decide on the types and number of AppSSs that need to be implemented and what service methods each AppSS should implement. Then, implement the specified AppSSs and all
their service methods. Finally, store the resulting AppSSs in the application-specific pattern service repository.

3- **Application-specific Pattern Service Creator (AppSPSC):**

a) **Search for and select generic pattern services:**

**Purpose:** AppSPSC searches for and selects generic pattern services to study their documentation and create the appropriate configuration file to use it to configure the generic pattern service and convert it into application-specific service.

**Outcome:** The location and selection of the required generic pattern service.

**Actions:** Use of search tools that the pattern service repository provides to search for generic pattern services that fit the supplied search criteria.

b) **Search for and select application-specific services:**

**Purpose:** AppSPSC searches for and selects application-specific services that implement the application-specific logic which the selected generic pattern service needs to interact with to deliver the overall functionality of the pattern service. **Outcome:** The location and selection of the required application-specific services.

**Actions:** Use of search tools that the application-specific service repository provides to search for application-specific services that fit the supplied search criteria.

c) **Application-specific pattern service conversion:**

**Purpose:** Convert the selected generic pattern service into an application-specific one.
**Outcome:** An application-specific pattern service that behaves according to the specific subject application is created.

**Actions:** The AppSPSC composes a configuration file that contains all necessary configuration operations that binds the selected AppSSs with the generic pattern service, configures the application-dependent service method templates so that they generate the proper service method logic for the corresponding AppDSM, and sets any default configuration values in the AppSPS data structures.

d) **Application-specific pattern service management:**

**Purpose:** Store and manage AppSPSs that resulted from converting generic pattern services into application-specific ones.

**Outcome:** Management and safe keeping of the AppSPSs in the application-specific pattern services repository.

**Actions:** Store the created AppSPSs after applying the configuration step to the generic pattern services in the application-specific pattern service repository.

4- **Pattern-based Application Creator (PbAppC):**

a) **Search for and select application-specific pattern services:**

**Purpose:** The selection of the application-specific pattern services that AppSPSCs configured and made ready specifically for the application that the PbAppC is going to compose.

**Outcome:** The location and selection of all the required one or more application-specific pattern services that are tailored to be used with the subject PbApp.
**Actions**: Use of search tools that the application-specific pattern service repository provides to search for the application-specific pattern services that fit the supplied search criteria.

b) **Application-specific pattern service composition**:  
   **Purpose**: Compose two or more application-specific pattern service to achieve a complex functionality delivered by the pattern service composition.  
   **Outcome**: The creation and safe keeping of pattern service compositions that can be used to deliver special purpose functionalities.  
   **Actions**: The PbAppC uses the application-specific pattern service repository to search for and select the AppSPSs to compose. The PbAppC applies the composition routines and implement the necessary logic to realize the pattern service composition according to the specific needs of the application that uses the pattern service composition. After the pattern service composition is finished and it is used to help deliver the required functionality of the subject PbApp, the whole PbApp, including the AppSPS composition component is stored in the pattern-based applications repository.

c) **Pattern-based application creation and management**:  
   **Purpose**: Compose and manage pattern-based applications.  
   **Outcome**: The creation and storage of the PbApps that thePbAppCs compose using AppSPSs that are configured to the exact needs of the subject PbApp, and other application services that PbAppCs also create as part of the PbApps being developed.  
   **Actions**: The PbAppC develops the necessary application service that will interact with the selected AppSPSs. The PbAppC also develops any other
application services that the application needs, together with the AppSPSs and the application service that interact with and use those AppSPSs, to deliver the functionalities that the PbApp promises. PbAppCs store their complete PbApp in the pattern-based applications repository.

5- Pattern-based Application Instance Creator (PbAppIC):

a) Search for and select pattern-based applications:

Purpose: Search for and select the pattern-based application that needs to be configured and deploy instances to the end users.

Outcome: The location and selection of the required pattern-based application.

Actions: Use of search tools that the pattern-based applications repository provides to search for the pattern-based applications that fit the supplied search criteria.

b) Pattern-based application instance deployment:

Purpose: Configure and deploy as many instances of the selected PbApp as needed.

Outcome: The configuration and deployment of a certain number of instances of the selected PbApp.

Actions: The PbAppIC prepares a configuration file to apply the last set of configuration procedures to the PbApp to make it work according to its usage requirements and rules. Once the instance has been configured, the PbAppIC starts the deployment process of the PbApp to make it available for its end users.
Overall Usage Process of the PaaS System:

The overall process of using the PaaS system and the usage sequence to facilitate the creation of pattern-based applications starting with the creation of the set of generic pattern services all the way up until the deployment of pattern-based application instances to their prospective users can be summarized in the following steps:

1- PSCs use the PaaS system to create generic pattern services and store them in the pattern service repository.

2- AppSSCs use the PaaS system to look up and select generic pattern services from the pattern service repository, then develop the AppSSs that the selected generic pattern services need to complement their functionalities.

3- AppSPSCs use the PaaS system to search for and select generic pattern services from the pattern service repository. Then search for and select AppSSs that complement the generic pattern service functionality. Finally, the system facilitates the configuration of the generic pattern service and binding of the AppSSs so that an AppSPS is shaped and stored in the application-specific pattern services repository.

4- PbAppCs use the PaaS system to search for and select application-specific pattern services from the application-specific pattern services repository to build PbApps. This is done by allowing the use of one or more AppSPSs, and possibly AppSPS compositions, along with other application services. PaaS helps integrate all these components to enable them to work as one PbApp. Finally, it enables the storage of the created PbApps in the pattern-based applications repository.

5- PbAppICs use the PaaS system to search for and select PbApps from the pattern-based applications repository. Once a PbApp has been selected, the PaaS system
enables the PbAppICs to create instances of that PbApp and apply the required configurations to each instance. Finally, it facilitates the process of deploying these configured PbAppIs to their respective end users.

**Overall usage process of PaaS system**

The resources and sequence of events involved in using the PaaS system are shown in the activity diagram presented in Figure 7.1. The activities start by the PaaS system enabling the PSCs to create generic pattern services and store them in a repository dedicated to PSs (PSR; See Glossary for definition). Then the PaaS system facilitates the easy use of the generic pattern services by enabling AppSPSCs and AppSSCs to search for and select generic PSs. For AppSPSCs, the PaaS system enables them to look up generic pattern services and any related documentation in the PSR. The AppSPSC creates the proper configuration file; which when applied to a generic pattern service converts it into an AppSPS. The AppSSC, on the other hand, looks up generic PSs and documentation to understand what AppSSs need to be created, their types, and the service methods that go in each AppSS. After selecting and studying the generic PS, the AppSSCs create needed AppSSs and store them in the Application-specific Service Repository (AppSSR; See Glossary for definition).

The other 2 types of PaaS users and the sequence of activities they perform are shown in the continuation diagram shown Figure 7.2. The fourth user type of the PaaS is the PbAppC. The PaaS allows them to search the AppSPSR to select the AppSPSs they need to incorporate in their PbApps. They integrate the selected, already configured, AppSPSs with the application services they create to collaborate with each other to
deliver the functionality of the target PbApp. Once built, the PbAppCs store their PbApps in the PbAppR.

Figure 7.1: PaaS system activities to manage pattern services
The fifth and last type of user of the PaaS system is the PbAppIC. They select PbApps from the PbAppR, apply their specific configuration to the instances they create from the selected PbApp, and finally deploy as many instances as they need.

Figure 7.2 (Cont.): PaaS system activities to manage pattern services
### 7.3 System Architecture Design

The design of the PaaS system can be divided into four main components: Pattern Service (PS) management, Application-specific Service (AppSS) management, Application-specific Pattern Service (AppSPS) management, and Pattern-based Application (PbApp) and Pattern-based Application Instance (PbAppI) management. Each component has one or more types of users. The users of the PS component are the PSCs, AppSPSCs, and AppSSCs. Although each type of user has its specific access rights and roles, the three types need to interact with and use the pattern service management component and its associated repository (PSR) to perform their tasks. The users of the AppSS are the AppSSCs. The users of the AppSPS are the AppSPSCs and PbAppCs. The users of PbApps are the PbAppCs and PbAppICs. Figure 7.3 shows the high-level architecture of the PaaS systems and highlights its five components with the repository allocated to each component.

Before we talk about the function each component of the four PaaS system components provides for its set of users, we will introduce the basic common structure between the components and the functionalities that each component provides. First, the PaaS system should provide a repository to store each type of artifact involved in implementing and using pattern services. It should dedicate separate sections of its repository for storing implementation, configuration, and documentation files that belong to each of the three main types of artifacts, i.e. PSs, AppSSs, AppSPSs, PbApps, and PbAppIs. Second, there should be some conformity in the common functionalities that all three components provide for their respective users which include the ability to upload, download, search for, select, and delete files to and from their respective
repository section in the PaaS system. The possibility to test-deploy PSs, AppSSs, AppSPSs, PbApps, and PbAppIs should also be provided by the PaaS system to help PaaS system users test their products before releasing them for use by the actual users of those products. Finally, there should be several other functionalities that differ based on the subject PaaS system component.

Figure 7.3: High-level architecture of the PaaS system

The diagram in Figure 7.3 depicts the four management components of the PaaS system. The following are some details about each component and the function it serves:
1- Pattern Service Management Component and Repository: This management component provides all the services needed to develop, store, test, and look up generic pattern services. Its immediate type of users are PSCs; however, AppSSCs and AppSPSCs need to use it to search for and select required generic pattern services. The reason that any component in the diagram needs to access the other component’s repository has to go through the owner component of the repository is for consistency and data protection and encapsulation purposes. Both AppSSCs and AppSPSCs must use the PS management component to access the PSR to search for and select generic PSs, hence the one directional connecting arrows from the AppSS and AppSPS components to the PS component. The arrows are one directional because the PS component does not need to access any of the other managing components or their repositories. In fact, all the arrows connecting the different management components are one directional because in the architecture there are no two management components that need two directional access.

2- Application-specific Services Management Component and Repository: This management component is dedicated for the development, storage, testing, and looking up of application-specific services (ApSSs). The immediate user type of this component is the AppSSCs; they are the ones who develop and maintain the AppSSs. The AppSPSCs are the second user type of this management component and they use it to look up and select AppSSs to use with their AppSPSs. As the connecting arrow from the AppSPS component to the AppSS component suggests, when AppSPSCs need to access the AppSSR, they must go through the AppSS management component for consistency and the application of proper protection and security measures.
3- **Application-specific Pattern Service Management Component and Repository:**

This component takes care of managing all aspects of AppSPSs. The main task this component performs is supporting the configuration of the generic pattern services and converting them to AppSPSs. It also provides tools that enable PbAppCs to search for and select already configured AppSPS to use with their PbApps. The arrows connecting AppSPS management component with the AppSS and PS components show that AppSPSCs need to access both repositories (PSR and AppSSR) to properly select and configure generic pattern services and establish the integration between generic PSs and their complementing AppSSs. To access the PSR, the AppSPSCs must use the PS management component, and to access the AppSSR, they must use the AppSS management component.

4- **Pattern-based Applications and application Instances management component and Repository:** The is the last component in the architecture, and it takes care of managing PbApps and enables PbAppCs to combine several AppSPSs and other application services to build PbApps. It also supports the configuration and deployment of the pbAppIs to their end users. The arrow from this component to the AppSPS one implies that when PbAppCs need to compose AppSPSs and import them to be used in their PbApps, they need to go through the AppSPS management components to perform that task. The AppSPS management component manages the AppSPSR, so all access to this repository has to go through this managing component.
After introducing the general architecture and functionalities of the PaaS system components, we explain the details of the design of the PaaS system and elaborate on the functionalities it should provide for each type of it user.

In accordance with the proposed general methodology idea for implementing patterns as services, the PaaS system must provide for its different user types, namely, PSCs, AppSSCs, AppSPSCs, PbAppCs, and PbAppICs, to use and operate on pattern services, application-specific services, and other additional application services in such a way that enables them to effectively perform their tasks and eventually benefit from the PaaS system according to their needs.

As mentioned above, the first set of the methodology users are the PSCs. The PaaS system should help this type of user to create, configure, store, compose, test, and document PSs. This means that the PaaS system must include a repository for storing developed pattern services as well as functionalities that the PSCs can use to manage their pattern services within this repository. Such repository related functionalities include uploading, downloading, searching, updating, and removing pattern services. This repository – which we call Pattern Service Repository (PSR; See Glossary for definition) – should contain sections for storing and managing pattern service files and other files such as documentation files and files that contain directions on how to configure the generic pattern service and make it application specific.

Once PSCs develop some pattern service or compose two or more pattern services, they will certainly need to test and debug them before they can release them for official use by the next set of users – the PbAppCs. The PaaS should include functionalities to apply any configuration files to the pattern services and create test stubs to test-deploy pattern
services in order to check and monitor their behaviors and fix any problems that may arise during the testing.

When the testing phase is completed and the PSCs are confident that the generic pattern service or pattern service composition is reliable enough to be released for official use, they must first provide sufficient documentation about the pattern service at hand. Such a documentation may include how the pattern service can be used, what AppSSs need to be developed and attached to it, what aspects of the pattern service are configurable, what configuration files need to be supplied, etc. The documentation files must be uploaded to a dedicated section in the repository, (PSR), so that immediate users of the generic PSs, namely, AppSSCs and AppSPSCs, can search for and download them to understand how they can effectively add the missing components and configurations to the subject generic PSs.

For the second type of users – the AppSSCs – the PaaS system needs to provide what we call an Application-specific Service Repository (AppSSR; See Glossary for definition). The AppSSR is identical in its structure and functionalities to the PSR; however, it is used for storing and managing AppSSs as opposed to PSs. Just like the PSR, the AppSSR provides the same functionalities of storing and managing its contents of AppSSs. Using the AppSS management component of the PaaS, AppSSCs can upload, download, search, update, and remove AppSSs from the AppSSR. The PaaS system should also enable AppSSCs to test-deploy their AppSSs to fix any bugs before endorsing them for use with the generic PS and AppSPSs after the configuration procedure, performed by the third type of user, which converts generic PSs into AppSPSs.
For the third type of user – the AppSPSCs – the PaaS system needs to provide the means to develop and store pattern service configuration files to use them to convert generic PSs into application-specific ones. The PaaS system should provide what we call an Application-specific Pattern Service Repository (AppSPSR; See Glossary for definition) to enable AppSPSCs to store and manage configured AppSPSs. The AppSPSR is identical in its structure and functionalities to the PSR and AppSSR. Just like the PSR and the AppSSR, the AppSSR provides the same functionalities of storing and managing its contents of AppSPSs. Using the PaaS, AppSPSCs can upload, download, search, update, configure, and remove AppSPSs from the AppSPSR. The PaaS system should also enable AppSPSCs to test-deploy their AppSPSs to fix any bugs before endorsing them for use with PbApps.

For the fourth type of user – the PbAppCs – the PaaS system needs to provide a Pattern-based Application Repository (PbAppR; See Glossary for definition). The PbAppR is identical in its structure and functionalities to the other repositories; however, it is used for storing and managing completed PbApps that are ready to be deployed. The PbAppR provides the same functionalities of storing and managing its contents of PbApps. Using the PaaS, PbAppCs can upload, download, search, update, configure and remove PbApps from the AppSSR. The PaaS system should also enable PbAppCs to test-deploy their AppSSs to fix any bugs before endorsing them for instance creation and deployment for the end users.

The management component and repository that the PaaS system provides for the PbAppCs also provides support for the PbAppICs. So ready to use PbApps can be located, selected from the PbAppR, and configured for deployment, and the PbAppICs
can deploy as many instances to their end users as they need. Before any PbApp can be completed and ready for deployment, however, all the pattern services used by that PbApp must be configured and adapted to work according to the application needs. This configuration and code generation needs to be applied to the selected pattern services by the AppSPSCs in order to convert them to AppSPSs. Then the next step is to perform the registration processes that will attach AppSSs to the selected generic pattern services. The PaaS system should provide the necessary means to facilitate such integration between PSs and AppSSs and make their interactions run smoothly. Figure 7.4 gives an overall view of the processes and functionalities of the PaaS system explained thus far. The upper part of Figure 7.4 shows the PSCs’ role and part of the PbAppCs’ role when using the PaaS system. It shows how a PSC develops and stores PSs and PS compositions in the PSR. This also includes the provision of any needed configuration and documentation files that help users of the stored PSs.

The lower part of Figure 7.4 depicts how the PbAppCs perform the steps of accessing the PSR to search for and select PSs. It also shows how the PbAppCs create needed AppSSs and store them in the AppSSR. The last section of the lower part of Figure 7.4 shows how the PbAppCs perform the steps of converting generic PSs to application-specific ones by registering – attaching – their AppSSs to the subject PS, applying their configuration, and generating all needed PS to AppSS interaction code. The PaaS system must cater for such registration and integration capabilities so that PbAppCs can put their PbApps together.
Once the selected generic pattern services are made application specific by the steps stated above, the PbAppCs should develop any other application services that may be needed to glue the application-specific pattern services together and help put the several pieces of the target PbApp together.

After the proper configuration and integration of the different components of a resulting PbApp, it can be put to test. Testing PbApps is another essential functionality that the PaaS needs to provide for its users. Being composed of, possibly, several PSs, a battery of AppSSs, and a few additional application services such as the client and Setter
services. PbApps are trickier to test than other smaller entities such as PSs and AppSSs. This testing phase is particularly important and more complex than previous tests because it tests the complete PbApp, how it functions as a whole, and how each of its components behaves under different circumstances. The results of this test phase may uncover problems and bugs either in the constituent PSs or AppSSs and may consequently require more bug fixing on the PS and/or AppSS levels.

After performing the testing on their PbApps and fixing any discovered bugs, the PbAppCs will naturally release their PbApps for use by the next set of PaaS users, the PbAppICs. Released PbApps are stored in the Pattern-based Application Repository (PbAppR; See Glossary for definition) – a repository dedicated to storing and managing released PbApps.

It is important to note that the PaaS system should separate released artifacts whether they be PSs, AppSSs, or complete PbApps from those still under development or testing. That means that PbAppCs can only see, search, and use released PSs.

The first set of functionalities that the PaaS needs to provide for the third set of its users – the PbAppICs – enables them to access the PbAppR and search for and select released PbApps and their usage documentation. The second set of functionalities that the PaaS system should provide to the PbAppICs comes into play after the PbAppICs read the documentation associated with the selected PbApp. This set of functionalities includes enabling the PbAppICs to carry out the configuration step – which further customizes the selected PbApp to the exact needs of the PbAppICs. Such a configuration step may use one or more configuration files. The need for more than one configuration file may arise if the PbAppICs need more than one deployment of the PbApp and each
deployment acts differently according to its configuration. So it is important to arm the PaaS system with a capability to allow the PbAppICs to apply different configurations to different Pattern-based Application Instances (PbAppIs). This makes PbApps more flexible and enables their usage in potentially different circumstances.

Once the PbAppIs of the selected PbApp have been configured, another functionality of the second set comes into play. It is the one that enables the PbAppICs to deploy as many of the PbAppIs as they need. The upper part of Figure 7.5 depicts the processes of putting PbApps together by the PbAppCs and storing the ready to use PbApps in the PbAppR. As can be seen from the figure, the PbAppCs take whatever application-specific pattern services they created using the processes executed in Figure 7.4 and use them to form the PbApps. To form a complete PbApp, however, the PbAppCs need to create and include additional application service in the mix. The result of such a mix is what we call a PbApp. The PbAppCs store the resulting PbApps in the PbAppR, release them for use, and upload any needed configuration and documentation files so that the next set of PaaS users can use them.

The lower part of Figure 7.5 shows how the third set of the PaaS users – the PbAppICs – perform their task. They basically select the PbApp they need from the PbAppR, apply their own configuration to it, then they deploy as many instances of it as they require. The lower part of the figure also shows the fourth set of users who are not direct users of the PaaS system; rather, they are the end users of the PbAppIs deployed by PbAppICs.
Figure 7.5: PaaS support for PbApps creation and PbAppIs deployment

As can be noticed, implemented generic pattern services may be used directly or they may be combined to build PbApps. The possible number of PbApps that can be created using the PaaS system and pattern services is potentially large. Furthermore, PbApps can be configured in different ways and a considerable number of instances of these PbApps may be deployed. This is where cloud computing and the abundance of computing resources come into play. Although it can be done without resorting to cloud computing, cloud computing technology, its offering models, and the plentiful resources in data centers makes it a very appealing choice to run and manage the PaaS system and
to give the required flexibility to PbAppICs to scale their applications up and down according to their needs.

To show the feasibility and usability of the proposed design for the PaaS system, we developed a prototypical system that follows the same design guidelines explained in the previous section. We also created some case studies of a PS and two PbApps to test the three main components of the PaaS system. Testing the developed prototypical PaaS system revealed that it works correctly and according to the stated design guidelines.

The screenshots in Figure 7.6 and Figure 7.7 show sample screens from the implemented PaaS system. Figure 7.6 shows a screenshot of the main page and the menu, while Figure 7.7 shows a screenshot of the application instance deployment process.

![Figure 7.6: Screenshot of the PaaS System’s main page.](image-url)
7.4 PaaS System Case Studies

In order to verify the feasibility of the idea and design of the PaaS system, we picked the Observer design pattern GoF as a case study for our feasibility test. We implemented the Observer design pattern as a generic set of Web services that can be used by software applications that need the Observer design pattern to deliver part of their functionality. As shown in Figure 7.8, the implemented web services follow the guidelines of the proposed general methodology idea for implementing patterns as services and are identical to those followed by the Observer PS (See Appendix A, Section A.16). As can be seen in Figure 7.8, the implementation of the Observer pattern service contains three AppSS independent services, namely, Subject Registration Service, Observer Registration Service, and Subject-Observer association setting service. These three services take care of the registration process of Subject and Observer AppSSs with the Observer pattern service, and establish the associations.
between those Subject and Observer AppSSs according to the rules of the Observer design pattern. The case study implementation of the Observer pattern service contains another set of services that come into play after the registration and setting processes of the AppSSs and it performs the AppSS dependent interactions. This set of services includes the Attach Service, Detach Service, Notify Service, and Get State Service. This set of services interacts with the registered AppSSs to facilitate the attachment and detachment of observers and perform the process of keeping the Observers’ state updated every time the Subjects’ states change.

![Observer Pattern Service](image)

**Figure 7.8: Case study implementation of the Observer PS**

To use this Observer pattern service implementation in a PbApp, we need to implement the Subject and Observer AppSSs, register them with the pattern service, and associate each registered Observer with at least one registered Subject. We also need to implement a service method in every Subject AppSS that will execute the Notify service of the pattern service every time the state of that Subject AppSS changes. Furthermore, we need to implement a service method called Update in every Observer AppSS that the
pattern service can execute to cause the Observers to act on the Subject state change notification and update their states accordingly. Somewhere within the Update service method of the Observers, a call to the Get State Service of the pattern service has to be placed so that the Observers can get the data they need to update their states.

To test the implemented pattern services, we created two applications that use the implemented Observer pattern service. The first application is an Online Discussion Group (ODG) application. The ODG enables its members to hold discussions online. The second application is an Online Stock Market Ticker (OSMT) application that users can use to monitor their stock market portfolio online.

7.4.1 Online Discussion Group

The ODG is an online discussion group software application that uses the implemented case study Observer pattern service. The ODG allows users to login and choose the specific discussion they would like to join based on the available Fields of Discussion (FoD; See Glossary for definition), e.g. Art, Sport, Science, Politics, etc. As depicted in Figure 7.9, the first user request to join a particular discussion group causes the ODG to open that discussion group and register it as the Subject for all its prospective Observer members and registers the user as the first Observer – Member – of that discussion group. Every time a new user joins a discussion group, the ODG application authenticates that user and after that it invokes the Observer Registration Service and Subject-Observer Association Setting Service of the Observer pattern service to register the user as a member of that discussion group. Whenever one of a discussion group posts a comment, the subject discussion group notifies the Observer pattern service of the new update of state.
The next step uses the Observer pattern service to take care of streamlining the process of registering, and notifying the Observers of each discussion group when an event occurs, in the discussion group they are registered with, that requires them to update their state. Figure 7.10 shows the components of the ODG and their interactions with the Observer pattern service during runtime.
Upon the invocation of the Notify Service of the pattern service, the pattern service calls the Update service methods of all the Observer AppSSs that are observing the Subject AppSS with the state change. The Update method then calls back the Get State service method of the pattern service, which in turn fetches the latest state data of the Subject AppSS – in this case the Discussion component – by executing the Get State service.
method of the Subject AppSS that returns the Subject’s state data to the pattern service. As soon as the pattern service receives the Subject’s state data, it forwards the data to the proper Observer AppSSs that are associated with that Discussion to make them update their states accordingly.

In the process of developing the ODG application, we create a few application services, such as the client, and application-specific services. The topmost component that the application end-user directly interacts with is the Graphical User Interface (GUI) component of the ODG application. The GUI component is responsible for accepting ODG member interactions and presenting the latest state of the ODG application to them. Underneath the GUI component sits the Event Router component. The Event Router works like a controller in the Model-View-Controller software pattern. It is responsible for all the coordination between the GUI component and the underlying discussion group components. It receives interactions coming from different ODG members and dispatches them to the proper Discussion instance. The Event Router is also responsible for redirecting all member registration, login, logout, and other member profile information updates to the Member Account Management component. After signing up, users become members of the ODG and they can login to the ODG of their choice based on the FoD they choose. Once verified by the Member Account Management service, they are directed to the proper ODG.

A Discussion instance is dedicated to each available FoD. Members join one of these Discussion instances when they log in based on their FoD choice. Each Discussion instance has its discussion topic, a group of discussion participants – discussion
members who post discussion comments, and a number of discussion comments (discussion content) that are posted by the members of that Discussion instance.

The discussion topic and all of the discussion comments posted by the discussion group members are saved in the discussion group database. This makes discussion content available for future reference or in case a discussion was not completed and the members would like to resume the discussion on a later time.

To keep track of its subscribed members and to ensure that all discussion posts and other events are delivered to them, each Discussion instance makes use of the Observer pattern service we implemented. Every time a new member joins a Discussion instance, a request to register that member with that specific instance is dispatched to the pattern service through the attach service and the registration and association services.

Once registered, the new member’s GUI is updated to reflect all the events, discussion posts, and a list of the other subscribers to that Discussion instance. The new member can now participate in the discussion and post comments.

We carried out some testing on the use case ODG application by creating several Discussion instances and registering a number of members to each of those Discussion instances. The result was very good; the application worked smoothly and behaved exactly as expected. There was proper separation between different instances of the Discussion, their sets of registered members, and the events triggered in each instance.

The screenshots below show the pattern-based case study ODG application. Figure 7.11 shows a screenshot of the group administrator screen.
Figure 7.11: GUI screenshot of the administrator of the ODG

Figure 7.12 shows a screenshot of an ordinary member of the same discussion group.
The second application that we implemented to test the Observer pattern service is an Online Stock Market Ticker. The OSMT enables its users to select a group of stock market symbols that form their stock market portfolio. Stock market data of the users’ portfolios are displayed on their screens and updated every few seconds to allow them to monitor the market and how their portfolios of stock are performing in the market. Figure 7.13 shows the registration and setting processes of the Subject – Stock Ticker – and the stock Observers – Users – with the Observer pattern service.
Once registered, users build a portfolio of stock symbols to monitor. Every time there is some new stock data, the Stock Ticker component notifies the pattern service. The pattern service invokes the Update service methods of the Observer AppSSs, in this case users of the stock market data. The Update service methods of the AppSSs callback the pattern service’s Get State Service to ask for the updated stock data. The pattern service
retrieves the stock data from the Stock Ticker via the Get State Service of the Stock Ticker AppSS. After the retrieval of the latest stock data, the pattern service forwards them to the Observer users to let them update their lists of stock market data. Figure 7.14 depicts the runtime interactions between the different components of the OSMT including the Observer pattern service.

Figure 7.14: OSMT runtime interactions with the Observer PS
The OSMT allows users to either sign up to create an account or just use the OSMT as anonymous users. The difference is that when a user signs up and creates an account, the OSMT can store the user’s portfolio for future use and that user will not have to recreate a portfolio for each login.

Figure 7.15 shows a screenshot of a user who uses the OSMT application as an anonymous user without creating an account or saving a portfolio.

![Stock Market Ticker](image)

**Figure 7.15: GUI screenshot of an anonymous user of the OSMT**

On the other hand, if the users log in anonymously, then they will have to create their portfolio of stock each time they use the OSMT application.

Figure 7.16 shows a screenshot of an example of a registered user for whom the OSMT stores a portfolio and loads it as soon as that user logs in. The inner workings of the GUI, Event Router, and User Account Management are almost identical to those of the ODG, so we are not going to talk about that part here since it is already explained in the
The Stock Ticker (ST; See Glossary for definition) component, however, is different from the Discussion component in the ODG. Its job, basically, is to coordinate and time stock tick retrieval and to engage the Observer pattern service accordingly.

To get stock market ticks for a set of stock market symbols, the ST component provides the Stock Ticker Feed (STF; See Glossary for definition) with the symbols and waits for the response with stock market data related to those symbols. The STF component relies on Google Stock Ticker Service and Yahoo Stock Ticker Service to get stock market tick data. The STF queries both Google’s and Yahoo’s Stock Ticker Services and then returns the data back to the ST component which informs the Observer pattern service of the new data. At this point the interactions between the pattern service and the

---

**Figure 7.16: GUI screenshot of a registered user of the OSMT**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Index</th>
<th>Previous Close Price</th>
<th>Last Trade Price</th>
<th>Last Trade Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMZN</td>
<td>NASDAQ</td>
<td>757.77</td>
<td>765.65</td>
<td>Dec 19, 3:52PM EST</td>
</tr>
<tr>
<td>BABA</td>
<td>NYSE</td>
<td>88.67</td>
<td>88.94</td>
<td>Dec 19, 3:52PM EST</td>
</tr>
<tr>
<td>CC</td>
<td>NYSE</td>
<td>24.14</td>
<td>24.74</td>
<td>Dec 19, 3:52PM EST</td>
</tr>
<tr>
<td>EBAY</td>
<td>NASDAQ</td>
<td>29.61</td>
<td>29.41</td>
<td>Dec 19, 3:52PM EST</td>
</tr>
<tr>
<td>MSFT</td>
<td>NASDAQ</td>
<td>62.3</td>
<td>61.55</td>
<td>Dec 19, 3:52PM EST</td>
</tr>
<tr>
<td>PPSI</td>
<td>NASDAQ</td>
<td>5.55</td>
<td>5.70</td>
<td>Dec 19, 3:41PM EST</td>
</tr>
<tr>
<td>T</td>
<td>TSE</td>
<td>42.65</td>
<td>42.81</td>
<td>Dec 19, 3:38PM EST</td>
</tr>
<tr>
<td>YELP</td>
<td>NYSE</td>
<td>35.83</td>
<td>35.59</td>
<td>Dec 19, 3:32PM EST</td>
</tr>
<tr>
<td>YHOO</td>
<td>NASDAQ</td>
<td>38.61</td>
<td>38.38</td>
<td>Dec 19, 3:52PM EST</td>
</tr>
</tbody>
</table>
AppSSs kick off and the processing continues until each Observer updates its portfolio of stock market symbols with the latest data.

## 7.5 Summary

In this chapter we described the requirements and design of the Pattern as a Service (PaaS) system. We explained the rationale behind developing the design of the PaaS system and what purpose it serves. We then described in detail the architecture of the system which consists of four main components. The first component facilitates the creation, management, and storage of generic pattern services. The second component enables the creation, management, and storage of application-specific services to use with the generic pattern services. The third component supports the configuration of the generic pattern services and the integration of the AppSSs with it. The fourth component provides for the composition of pattern services, the creation of application services, and the construction of pattern-based applications using all the types of services and pattern services mentioned above. Moreover, the fourth component supports the generation, configuration, and deployment of pattern-based application instances. During the description of the architecture of the PaaS system and its functionalities, we described the five types of the system users, namely, the pattern service creators (PSCs), the application-specific service creators (AppSSs), the application-specific pattern service creators (AppSPSs), the pattern-based application creators (PbAppCs), and the pattern-based application instance creators (PbAppICs).

We also described the four repositories used by the system, which are the pattern service repository, the application-specific services repository, the application-specific pattern services repository, and the pattern-based applications repository. Then we
described the flow of events during the usage of the PaaS system by its five types of users. After we had described the PaaS design, we explained how we implemented a prototypical PaaS system to verify the design.

We dedicated a section of this chapter to the two case study applications in which we described how we implemented the Observer design pattern as a generic pattern service, and then how we used it to implement the two case study applications, the Online Discussion Group (ODG) and Online Stock Market Ticker (OSMT). We showed the design of the two applications and how each one uses the Observer pattern service.

In the next chapter, we run some evaluation procedures and estimate the effort needed to develop pattern services based on the proposed methodology. We also highlight the pattern knowledge requirements for the participants in developing pattern services. We compare the figures with those of the same design pattern implementation using the SOA paradigm. We proceed to measure the anticipated performance overhead of using the proposed PS methodology. Finally, we run some performance and scalability tests on the Observer PS and compare the results with its implementation using the SOA paradigm.
Chapter 8

Evaluation Methodology and Experiments

In this chapter we present the evaluation methodology for pattern service development. This evaluation methodology is for the use of pattern service developers and users to systematically evaluate their work and the benefits of developing pattern services. We start by introducing the analysis-based evaluation part, in which we develop the mathematical equations to evaluate and measure the amount development effort involved in developing pattern services. The second subsection of the analysis-based evaluation takes care of evaluating the knowledge skillset required for the design and development teams to develop pattern services. This subsection describes how the skillsets for the Pattern service Creators (PScs), Application-specific Service Creators (AppSSCs), and Application-specific Pattern Service Creators (AppSPSCs) can be measured using some mathematical formulae. The third subsection explains how to evaluate the performance overhead involved with developing and using pattern services in software applications. In all three of the analysis-based evaluations, we compare the direct SOA implementation of a design pattern to the implementation of the same design pattern as a pattern service. Then we apply the mathematical formulae we developed for each evaluation type on the 23 GoF design patterns (See Appendix A), and give the comparison results between the direct SOA and the pattern service implementations for each design pattern. In the section that follows, we discuss how to test pattern services for functionality, conformability, and performance and scalability. We describe how to build test cases for each type of test with the test case artifacts and the testing procedure involved. Finally, we give the results of some experiments that we
ran to test the performance and scalability of the Observer pattern service that we implemented as a case study pattern service. We compare the results of the pattern service implementation of the Observer design pattern to those of the direct SOA implementation of the same design pattern.

8.1 Analysis-based Evaluation

This section is dedicated to evaluating the proposed design pattern services by analyzing the design of pattern services. The analysis process is performed through comparing some of the usage of the PS methodology to develop and use design patterns as PSs against the direct design pattern implementation using the (SOA) paradigm. This chapter begins by analyzing the effort it takes software developers to implement PSs using design patterns, adding application-specific service components (AppSSs), and finally configuring and converting the generic PSs into application-specific pattern services (AppSPS). Then we assess the amount of pattern knowledge required by all the participants in the three stages, namely, the pattern service creators (PSCs), application-specific service creators (AppSSCs), and application-specific pattern service creators (AppSPSCs), to perform their jobs and produce the final version of the PS that is tailored to work with a specific software application. After that we measure the anticipated performance overhead of the PS paradigm compared to the direct (SOA) one.

8.1.1 Development Effort Evaluation

The development of any software involves some effort being contribution by the team of developers. The amount of such effort can be minor or major, based on the software system to be developed and how many of the system components are ready and can be
reused in the system at hand. The amount of effort required to develop a software system is one of the important stakes in the decisions taken by the system owners, designers, and developers. The development effort and the time to get the software system ready for marketing can decide which software development paradigm to choose.

In this section of the analysis-based evaluation, we develop the mathematical formula to use to assess the development effort needed to use design patterns in the development of software applications. We first calculate the effort needed to implement the design pattern using the direct Service-Oriented Architecture paradigm, then we do the same when the same design pattern is implemented using the proposed Pattern Service one. Then we determine the relative development effort needed when using either paradigm. Finally, we apply the formula on the 23 design patterns presented in GoF and present a table that shows the percentage of relative effort involved when the proposed PS methodology is adopted in implementing each design pattern as a pattern service.

To develop such a formula, we first need to determine the bare minimum number of service methods to be implemented using the two paradigms to complete the implementation of a given design pattern. Thus, we can find out the number of methods required to implement a design pattern using the direct SOA paradigm by counting/estimating the number of service methods need to be developed in the set of services that deliver the functionality of the design pattern. We use “MinDirect” to denote the minimum number of methods to be developed using the SOA paradigm.

\[
\text{MinDirect} = \text{Minimum Number of Methods in SOA Implementation}
\]
For the SOA paradigm, the development effort for implementing a design pattern to be used in a SOA software application equals the minimum number of service methods to be developed. For example, the “MinDirect” for the SOA implementation of the Observer design pattern shown in Figure 8.1 equals 6, which is the minimum number of service methods, Attach, Detach, SetState, Notify, GetState, and Update.

*Figure 8.1: Direct SOA implementation of the Observer PS after GoF (p. 293)*

For a Pattern Service implementation of a design pattern, however, there are two types of service sets that contain the service methods that deliver the functionality of the design pattern, the Pattern Service (PS) set – which includes both application-independent methods and application-dependent configurable templates – and the Application-specific Services (AppSSs). This means that the minimum number of methods required to implement a design pattern as a complete and functional pattern service can be calculated by adding the minimum number of methods in both sets of services – PS and AppSS. We use “MinPS” to denote the minimum number of methods...
and method templates needed in the PS, and “MinAppSS” to denote the minimum number of methods needed in the Application-specific Service part.

The weight given to the application-independent methods – which have generic code that works with any application – is different from the weight given to the application-specific methods – the ones to be configured to work with specific applications. For the application-dependent methods, Pattern Service Creators (PSCs) create configurable templates that can be used to configure and create the logic for those application-dependent methods to work according to the requirements of the specific application that is using the pattern service. Some of the application-dependent methods do not contain much behavior logic in them; they just forward requests to other methods. In such a case, the weight that is given to this type of application-dependent method is 0.5. In other cases where application-specific methods do have some behavior and some fairly complex logic is involved – such as multiple interactions with AppSS service methods (AppSSMs), we give the weight of 1. We denote the weight factor as W. We refer to the application-independent service methods in the PS as “MinAppISM” and the application-dependent service templates as “MinAppDST”.

As the diagram in Figure 8.2 shows, the implementation of the Observer design pattern as a pattern service consists of 4 service methods: Attach, Detach, Notify, and GetState. Attach, Detach, and Notify are application-independent because they consist of generic code that is independent of the application that uses the pattern service; thus for the Observer pattern service, MinAppISM equals 3. On the other hand, GetState service method is application dependent, which means that parts of their implementation code
depend on the application that uses the pattern service. As a result, the value assigned to MinAppDST, in the case of Observer pattern service, is 1.

So to calculate the total weight of the minimum number of methods in the PS we sum up the result of multiplying each type of template by its assigned weight and add the result to the minimum number of application-independent methods, “MinAppIM”, using Equation 8.1.

**Equation 8.1:** \[ \text{MinPS} = \text{MinAppIM} + \sum_{k=1}^{\text{MinAppDST}} W_k \]

![Diagram of Observer Pattern Service](image)

**Figure 8.2:** PS implementation of the Observer design pattern after GoF (p. 293)
The weight calculation for the development effort required by the AppSSCs who develop the application-specific services, AppSSs, depends on the number of methods needed to provide the PS functionality. We refer to the minimum number of AppSS service methods (AppSSMs) needed to complement the functionality of the PS as “MinAppSS”. To calculate the amount of effort required developing the minimum number of AppSSMs for a PS, we just count the number of methods and give each method the weight 1.

Using the Observer pattern service shown in Figure 8.2 as an example, the minimum number of AppSSMs needed is 3. They are the SetState and GetState service methods, implemented by the Subject AppSS, and the Update service method, implemented by the Observer AppSS. This means that in the case of the Observe pattern service, MinAppSS is assigned the value 3.

Finally, in order to calculate the actual development effort needed to build an Application-specific Pattern Service (AppSPS) – the Pattern Service that is adapted and configured to be used with a specific software application – first, we need to establish a variable, N representing the probable number of times a pattern service is used in different software applications. Second, we should reiterate the fact that the conversion of a PS into an AppSPS involves the implementation of the needed AppSSs, and the preparation of a configuration file that performs the necessary steps of integrating the AppSSs with the PS and adapts the PS to work according to the specificities of the software application that is using it, and lastly, the configuration of the application-dependent method templates. To perform this task, a configuration file needs to be created and run, introducing an extra step to the overall development effort of the
proposed pattern service paradigm. We use $C$ to denote the constant value – for the effort needed to build the configuration file – to be added to the formula to calculate the development effort of applying the PS paradigm to implement and use design patterns as pattern services in software applications.

We use “PSDEffort” to denote the overall development effort when using the proposed PS paradigm. Such a development effort can be calculated by dividing the minimum number of methods needed in the PS part by the probable number of pattern service reuse $N$, adding to that, the minimum number of methods in the AppSS section and the constant weight value of creating the configuration file. Equation 8.2 calculates the development effort involved in building a pattern service and its AppSS part, and the effort needed to configure it to be used with a specific software application.

**Equation 8.2:**  
\[ PSDEffort = \frac{\text{MinPS}}{N} + \text{MinAppSS} + C \]

Now that we have established the formulae to calculate the development effort needed to implement a design pattern using the two paradigms, we proceed to discuss the formula that we can use to calculate the relative added/reduced development effort when using the proposed PS paradigm to implement design patterns as reusable pattern services in relation to the SOA paradigm.

So far, we have established the formula to calculate the development effort for the direct implementation represented by the minimum number of methods needed to implement a given design pattern using the SOA paradigm. We have also established the formula to calculate the development effort of using the PS paradigm, represented by the minimum number of PS service methods divided by the probable number of PS reuse, plus the
minimum number of AppSSMs to be developed and the constant value of creating and running the configuration file.

Using the results obtained by applying the formulae explained above, we can calculate the relative development effort – we refer to it as Pattern Service Development Effort Percentage “PSDEPercent” – of adopting the pattern service paradigm in relation to the direct paradigm using Equation 8.3:

\[
\text{Equation 8.3: } \text{PSDEPercent} = \frac{(\text{MinPS} / N) + \text{MinAppSS} + C)}{\text{MinDirect}} \times 100\%
\]

Shorthand of the above formula to calculate the percentage of relative development effort involved when adopting the PS paradigm is shown by Equation 8.4 below:

\[
\text{Equation 8.4: } \text{PSDEPercent} = \frac{\text{PSDEffort}}{\text{MinDirect}} \times 100\%
\]

The next step is to apply this formula to the 23 design patterns in the GoF book to see how much development effort will be added by the proposed pattern service methodology when implementing each of those 23 patterns.

Before we do that, however, we need to decide on the value that we give to the constant C. As explained earlier – constant C represents the extra effort added to construct and run the configuration file to configure and adapt the whole PS to make it ready for use with the target software application. Considering the lower complexity of building XML configuration files compared to the development of the pattern methods in one of the high-level programming languages and the estimated average number of XML elements involved in constructing the XML configuration document, we can say that the development of the configuration document can be considered to be the equivalent of
0.5 of a pattern service method implemented in a high-level programming language such as the ones used to develop SOA and PS components. Based on this assumption, we can assign the value 0.5 to the constant C.

\[ C = 0.5 \]

Table 8.1: Comparison of the PS and SOA development effort

<table>
<thead>
<tr>
<th>S.N.</th>
<th>Pattern Name</th>
<th>Min Direct</th>
<th>Min PS</th>
<th>Min AppSS</th>
<th>PsDEPercent where N is</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>100</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Abstract Factory</td>
<td>2</td>
<td>0.5</td>
<td>1</td>
<td>77.5% 75.3% 75%</td>
</tr>
<tr>
<td>2</td>
<td>Adapter</td>
<td>3</td>
<td>1.5</td>
<td>2</td>
<td>88.3% 83.8% 83.4%</td>
</tr>
<tr>
<td>3</td>
<td>Bridge</td>
<td>2</td>
<td>0.5</td>
<td>1</td>
<td>77.5% 75.3% 75%</td>
</tr>
<tr>
<td>4</td>
<td>Builder</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>86.7% 83.7% 83.4%</td>
</tr>
<tr>
<td>5</td>
<td>Chain of Responsibility</td>
<td>2</td>
<td>0.5</td>
<td>1</td>
<td>77.5% 75.3% 75%</td>
</tr>
<tr>
<td>6</td>
<td>Command</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>90% 84% 83.4%</td>
</tr>
<tr>
<td>7</td>
<td>Composite</td>
<td>8</td>
<td>4</td>
<td>5</td>
<td>73.8% 69.3% 68.8%</td>
</tr>
<tr>
<td>8</td>
<td>Decorator</td>
<td>3</td>
<td>1.5</td>
<td>2</td>
<td>88.3% 83.8% 83.4%</td>
</tr>
<tr>
<td>9</td>
<td>Façade</td>
<td>2</td>
<td>1.5</td>
<td>2</td>
<td>132.5% 125.8% 125.1%</td>
</tr>
<tr>
<td>10</td>
<td>Factory Method</td>
<td>2</td>
<td>1.5</td>
<td>2</td>
<td>132.5% 125.8% 125.1%</td>
</tr>
<tr>
<td>11</td>
<td>Flyweight</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>80% 75.5% 75.1%</td>
</tr>
<tr>
<td>12</td>
<td>Interpreter</td>
<td>8</td>
<td>4</td>
<td>5</td>
<td>73.8% 69.3% 68.8%</td>
</tr>
<tr>
<td>13</td>
<td>Iterator</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>47.5% 38.5% 37.6%</td>
</tr>
<tr>
<td>14</td>
<td>Mediator</td>
<td>2</td>
<td>0.5</td>
<td>1</td>
<td>77.5% 75.3% 75%</td>
</tr>
<tr>
<td>15</td>
<td>Memento</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>65% 63.5% 62.6%</td>
</tr>
<tr>
<td>16</td>
<td>Observer</td>
<td>6</td>
<td>3.5</td>
<td>3</td>
<td>64.2% 58.9% 58.4%</td>
</tr>
<tr>
<td>17</td>
<td>Prototype</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>80% 75.5% 75.1%</td>
</tr>
<tr>
<td>18</td>
<td>Proxy</td>
<td>3</td>
<td>1.5</td>
<td>2</td>
<td>88.3% 83.8% 83.4%</td>
</tr>
<tr>
<td>19</td>
<td>Singleton</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>160% 151% 150.1%</td>
</tr>
<tr>
<td>20</td>
<td>State</td>
<td>2</td>
<td>1.5</td>
<td>1</td>
<td>82.5% 75.8% 75.1%</td>
</tr>
<tr>
<td>21</td>
<td>Strategy</td>
<td>2</td>
<td>0.5</td>
<td>1</td>
<td>77.5% 75.3% 75%</td>
</tr>
<tr>
<td>22</td>
<td>Template Method</td>
<td>2</td>
<td>1.5</td>
<td>2</td>
<td>132.5% 125.8% 125.1%</td>
</tr>
<tr>
<td>23</td>
<td>Visitor</td>
<td>3</td>
<td>1.5</td>
<td>2</td>
<td>88.3% 83.8% 83.4%</td>
</tr>
</tbody>
</table>
We calculate the relative development effort percentage based on the assumed reuse values of 10, 100, 1000.

\[ N = 10, \quad N = 100, \quad \text{and} \quad N = 1000 \]

Table 8.1 lays out the details of relative development effort involved in implementing the 23 GoF design patterns using both direct and PS paradigms. Equation 8.3 is used to calculate the percentage of the relative development effort involved when PS paradigm is used, with the three proposed PS reuse values.

The results of applying the relative development effort of implementing the 23 GoF design patterns as pattern services compared to the direct SOA implementation of the same patterns show that the development effort when using the pattern service paradigm is less than that when the direct SOA paradigm is used. Table 8.1 shows that 19 out of 23, which constitutes 82.6%, implementations of the design patterns as pattern services have less development effort, while only 4 design patterns, which constitutes only 17.4%, will require more development effort when implemented as pattern services.

Also, we can notice from Table 8.1 that as pattern service reuse increases, the development effort drops, and that is due to the fact that the generic logic of the pattern service is being reused and no effort is needed to re-implement it with every reuse of the pattern service. Another point that is worthy of noticing is that higher the number of generic reusable service methods with generic logic the lower the development effort is. This is because the generic logic will only be implemented once; after that only the application-specific logic needs to be implemented, and a bit of application-dependent logic configuration is needed to have a fully functional pattern service ready for use with the new application.
8.1.2 Knowledge Skillset Requirements Evaluation

Granted, software developers who intend to use some of the software design patterns in the implementation of the software applications need to have a certain degree of knowledge of them. If the software developers are adopting the direct implementation paradigm, the SOA paradigm in this context, they must have a comprehensive knowledge of the subject design pattern, its structure, use cases, and any compromises that come with its use in the software application.

Software developers that participate in the development of a software application using the proposed Pattern Service paradigm can be divided into three groups. The first group consists of the developers whose task is to implement the service methods and templates in the Pattern Service (PS). We call this type of developer, the Pattern Service Creator (PSC). The second type consists of the developers who implement the required AppSSs to interact with the PS, the Application-specific Service Creators (AppSSCs). The third class of developers are the ones who build the configuration file and run it to integrate the AppSSs with the PS and configure any application-specific template methods in the PS to make them work with the specific application at hand. This configuration process converts a PS into an Application-specific Pattern Service (AppSPS). We call the third type of developer the Application-specific Pattern Service Creator (AppSPSC).

The following three subsections explore the tasks performed by each type of developer and construct the formulae to calculate the knowledge percentage of the pattern service each type needs to possess in order to fulfill their task in implementing, and configuring a pattern service and, finally, using it in a software application.
8.1.2.1 Design Pattern Knowledge Requirement for PSCs

Since this type of developer is the principal team to design and develop the generic reusable Pattern Service (PS), members of this team must have full knowledge and understanding of the subject design pattern for the following three reasons:

1- They need to have a complete knowledge of the subject design pattern so that they can extract the abstract layer of that design pattern and implement it as the generic layer of the PS.

2- Full knowledge of the design pattern enables PSCs to clearly describe the requirements that the AppSSCs need to meet and understand in order to design and develop the right AppSSs.

3- The PSCs need to be able to build code templates for application-dependent methods declared in the generic PS.

Therefore, to fulfill these three requirements, PSCs must have 100% knowledge of the subject design pattern. Using the Observer design pattern as an example, shown in Figure 8.2, the task of the PSCs is to implement all the application-independent service methods, Attach, Detach, and Notify, plus the application-dependent configurable template, GetState.

8.1.2.2 Pattern Service Knowledge Requirement for AppSSCs

The percentage of pattern service knowledge the AppSSCs developers need to have – to develop the necessary AppSSs for a PS – depends on the number, content, and complexity of the service methods they need to develop. In some cases of pattern services such as the Singleton and Prototype, AppSSCs require very little knowledge of the PS. In fact, all they need to know is how their AppSSMs should behave and what
functionality they deliver, with little or no need to know exactly how the PS uses them to fulfill its task. Also, in most cases where an AppSS does not need to call back any of the service methods in the PS, AppSSCs do not actually need to know much about the inner workings of the PS, because service calls, which are not considered callbacks, are usually initiated from the PS to the AppSS not the other way round. So, in both cases, explained above, the degree of PS knowledge required for the AppSSCs to be able to develop the AppSSs is negligible. That is why we assign the weight of 0 to all the AppSSMs that do not require to callback any of the PS service methods. In other cases, however, AppSSCs need to have more knowledge of the PS, in order for them to design and implement their AppSSs correctly. This is usually the case when there are some service calls that need to be made from the AppSSs back to the PS. In such a case, the AppSSCs need to know which service method to call, what data items to pass, and what the result of that service call will be. As a result of the discussion above, we come to the conclusion that we need to count the number of application-dependent service methods (AppDSMs) that get called by one or more AppSSMs. We refer to such a number of service methods with callbacks as “MinSrvCallbackAppDSM”. Note that for MinSrvCallbackAppDSM, we only count each AppDSM with at least one callback once, no matter how many times it is called back by AppSSMs.

Building on the analysis of the types of AppDSS service methods and the need for callbacks between them and the service methods in the PS, we can calculate the Pattern Service Knowledge Percentage, “PSKPercent”, for AppSSCs, by dividing the minimum number of the AppDSMs that are called back MinSrvCallbackAppDSM, by the total number of service methods in the PS part. If we refer to the total number of service
methods in the PS, as “TotalPSSM”, then the formula should look like Equation 8.5 below:

\[
\text{Equation 8.5: } \quad \text{PSKPercent} = \frac{\text{MinSrvCallbackAppDSM}}{\text{TotalPSSM}} \times 100\%
\]

Table 8.2 – on the next page – gives the estimated percentage of pattern service knowledge required for AppSSCs to be able to design and develop the needed AppSSs.

If we analyze the results shown in Table 8.2, which are the results of applying Equation 8.5 to the 23 GoF design patterns when they are to be implemented as pattern services, we come to the conclusion that knowledge skillset that AppSSCs need to have ranges from 0% to 50%. If we were to use the direct SOA implementation of these design patterns, the knowledge skillset will be a fixed 100% of the design pattern at hand. So in this case the worst case for developing the AppSSs required for a specific pattern service is only half of the direct SOA counterpart.

In many cases shown in Table 8.2, the knowledge skillset of the subject pattern service for AppSSCs to build the required AppSSs is 0%. This is due to the fact that AppSSCs only need to know about the service methods in the generic layer of the pattern service, if the service methods in the AppSS layer that they are to develop need to interact with some service methods in the generic layer. If not, then AppSSCs do not need to know how the service methods work in the generic layer, hence the 0% knowledge skillset requirement. Of course, the more service methods in the AppSS layer that need to callback service methods in the generic layer, the higher the knowledge skillset for AppSSCs.
### Table 8.2: Estimated percentage of PS knowledge required for AppSSCs

<table>
<thead>
<tr>
<th>S.N.</th>
<th>Pattern Name</th>
<th>Min SrvCallback</th>
<th>Total PSSM</th>
<th>PSK Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Abstract Factory</td>
<td>0</td>
<td>1</td>
<td>0%</td>
</tr>
<tr>
<td>2</td>
<td>Adapter</td>
<td>1</td>
<td>2</td>
<td>50%</td>
</tr>
<tr>
<td>3</td>
<td>Bridge</td>
<td>0</td>
<td>1</td>
<td>0%</td>
</tr>
<tr>
<td>4</td>
<td>Builder</td>
<td>0</td>
<td>2</td>
<td>0%</td>
</tr>
<tr>
<td>5</td>
<td>Chain of Responsibility</td>
<td>0</td>
<td>1</td>
<td>0%</td>
</tr>
<tr>
<td>6</td>
<td>Command</td>
<td>1</td>
<td>2</td>
<td>50%</td>
</tr>
<tr>
<td>7</td>
<td>Composite</td>
<td>0</td>
<td>4</td>
<td>0%</td>
</tr>
<tr>
<td>8</td>
<td>Decorator</td>
<td>1</td>
<td>2</td>
<td>50%</td>
</tr>
<tr>
<td>9</td>
<td>Façade</td>
<td>1</td>
<td>2</td>
<td>50%</td>
</tr>
<tr>
<td>10</td>
<td>Factory Method</td>
<td>1</td>
<td>2</td>
<td>50%</td>
</tr>
<tr>
<td>11</td>
<td>Flyweight</td>
<td>0</td>
<td>1</td>
<td>0%</td>
</tr>
<tr>
<td>12</td>
<td>Interpreter</td>
<td>0</td>
<td>4</td>
<td>0%</td>
</tr>
<tr>
<td>13</td>
<td>Iterator</td>
<td>0</td>
<td>4</td>
<td>0%</td>
</tr>
<tr>
<td>14</td>
<td>Mediator</td>
<td>0</td>
<td>1</td>
<td>0%</td>
</tr>
<tr>
<td>15</td>
<td>Memento</td>
<td>0</td>
<td>2</td>
<td>0%</td>
</tr>
<tr>
<td>16</td>
<td>Observer</td>
<td>1</td>
<td>4</td>
<td>25%</td>
</tr>
<tr>
<td>17</td>
<td>Prototype</td>
<td>0</td>
<td>1</td>
<td>0%</td>
</tr>
<tr>
<td>18</td>
<td>Proxy</td>
<td>1</td>
<td>2</td>
<td>50%</td>
</tr>
<tr>
<td>19</td>
<td>Singleton</td>
<td>0</td>
<td>1</td>
<td>0%</td>
</tr>
<tr>
<td>20</td>
<td>State</td>
<td>1</td>
<td>2</td>
<td>50%</td>
</tr>
<tr>
<td>21</td>
<td>Strategy</td>
<td>0</td>
<td>1</td>
<td>0%</td>
</tr>
<tr>
<td>22</td>
<td>Template Method</td>
<td>1</td>
<td>2</td>
<td>50%</td>
</tr>
<tr>
<td>23</td>
<td>Visitor</td>
<td>1</td>
<td>2</td>
<td>50%</td>
</tr>
</tbody>
</table>

#### 8.1.2.3 Pattern Service Knowledge Requirement for AppSPSCs

For the Application-specific Pattern Service Creators (AppSPSCs), the percentage of knowledge of the subject PS depends on the size and complexity of the configuration.
file they need to assemble in order to perform the final steps of, first, integrating the generic PS with the AppSSs, then, configuring the application-dependent service method templates – provided by PSCs – that create the actual implementation code for the Application-dependent Service Methods (AppDSM). The degree of PS knowledge needed by AppSPSCs depends mainly on the number of AppDSM templates (AppDSTs) to be configured. The reason is that the configuration of such templates involves one or more AppSSMs that will interact with that AppDSM. As a result, for the AppSPSCs to be able to correctly configure the templates, they need to have some knowledge of the proper signatures of both the AppDSM and AppSSMs. Moreover, some AppSSMs need to engage in a more sophisticated interaction with the PS through service callbacks initiating from the AppSSM towards a service method in the PS. This scenario further increases the required percentage of the AppSPSCs’ knowledge of the subject PS.

Based on the discussion above, we can list the main three factors that affect the percentage of knowledge required for AppSPSCs to build the right configuration file as follows:

1- Knowledge of the proper AppDSM signatures, so the AppDSTs can be configured correctly to generate proper logic for the AppDSMs.

2- The proper AppSSM signatures for the AppSSMs that will be called and used by those AppDSMs.

3- Knowledge of the necessary service method signatures and data items involved in every AppSSM callback to service methods in the subject PS.
Now that we have explained the major factors that affect the percentage of knowledge needed by AppSPSCs, it is time to think about the proper weight that should be given to each of the three factors. Given the fact that AppSPCs only create a configuration file that adds the application-specific content to the Application-dependent Service Templates (AppDSTs), we think that the weight to be given to the percentage of knowledge that AppSPSCs need to be able to perform such a configuration task should not exceed 50% of the overall knowledge of the subject PS. That is why we assign the maximum weight of 25% for the collective weight of all the AppDSM configuration requirements mentioned in the first factor in the list above. We calculate such a weight by dividing the minimum number of AppDSM we refer to as “MinAppDSM” by the total number of service methods in the PS, which we refer to as “TotalPSSM”. Then, to keep the weight below or equal to the 25% maximum percentage, we multiply the division result by 0.25.

To add the extra knowledge weight incurred by the need for AppSSM callbacks to the service methods in the PS, we also assign the fixed weight of 25% for that. The reason we assign this fixed weight is because in most cases callbacks are made to the application-dependent methods rather than to the application-independent ones. Having already given a weight for configuring AppDSMs, the weight to be given to the extra knowledge needed to take care of the callback routines should not be large. We refer to this extra knowledge weight as “CallbackWeight”. So when calculating the percentage of knowledge required for AppSPSCs for a given PS to perform the task indicated by the first factor in the list above, we divide the minimum number of the AppDSMs,
MinAppDSM, by the total number of service methods in the PS. Next we add the weight CallbackWeight, incurred by the callback extra knowledge requirement.

We refer to the knowledge percentage that AppSPSCs need to configure a PS as “PSKPercent. So to calculate “PSKPercent” we use Equation 8.6:

\[
PSKPercent = \left[ \frac{\text{MinAppDSM}}{\text{TotalPSSM}} \times 0.25 \right] + \text{CallbackWeight} \times 100\%
\]

Table 8.3 – on the next page – gives the knowledge percentage required for the AppSPSCs to be able to build the configuration file(s) to integrate the pattern service components and configure it to work with a specific application. PSKPercent, shown in Table 8.3 is calculated using Equation 8.6.

The results shown in Table 8.3 reflect the degree of knowledge skillset that AppSPSCs require in order to be able to convert a generic pattern service (GPS) into an application-specific pattern service (AppSPS). In this regard, AppSPSCs need to study and understand all service methods/templates that have configurable application-dependent logic. Also, they need to study and understand the interfaces and logic of the service methods in the AppSS layer that will be involved in some interactions with the service methods in the generic layer. This information is essential for the creation of the configuration file and to apply the configuration to the generic pattern service and turn it into an AppSPS. As can be seen in Table 8.3, the knowledge skillset ranges from 0% to 50%. The smaller the number of application-dependent service methods in the generic layer, the lower the knowledge skillset required. Moreover, the fewer the interactions
between the generic and application-specific layer, the less knowledge of the pattern service’s service methods AppSPSCs need.

**Table 8.3: knowledge percentage required for the AppSPSCs**

<table>
<thead>
<tr>
<th>S.N.</th>
<th>Pattern Name</th>
<th>Min AppDSM</th>
<th>Total PSSM</th>
<th>Callback Weight</th>
<th>PSK Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Abstract Factory</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>25%</td>
</tr>
<tr>
<td>2</td>
<td>Adapter</td>
<td>2</td>
<td>2</td>
<td>0.25</td>
<td>50%</td>
</tr>
<tr>
<td>3</td>
<td>Bridge</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>25%</td>
</tr>
<tr>
<td>4</td>
<td>Builder</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>25%</td>
</tr>
<tr>
<td>5</td>
<td>Chain of Responsibility</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>25%</td>
</tr>
<tr>
<td>6</td>
<td>Command</td>
<td>2</td>
<td>4</td>
<td>0.25</td>
<td>37.5%</td>
</tr>
<tr>
<td>7</td>
<td>Composite</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>6.25%</td>
</tr>
<tr>
<td>8</td>
<td>Decorator</td>
<td>2</td>
<td>2</td>
<td>0.25</td>
<td>50%</td>
</tr>
<tr>
<td>9</td>
<td>Façade</td>
<td>2</td>
<td>2</td>
<td>0.25</td>
<td>50%</td>
</tr>
<tr>
<td>10</td>
<td>Factory Method</td>
<td>2</td>
<td>2</td>
<td>0.25</td>
<td>50%</td>
</tr>
<tr>
<td>11</td>
<td>Flyweight</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>12</td>
<td>Interpreter</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>6.25%</td>
</tr>
<tr>
<td>13</td>
<td>Iterator</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>14</td>
<td>Mediator</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>25%</td>
</tr>
<tr>
<td>15</td>
<td>Memento</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>25%</td>
</tr>
<tr>
<td>16</td>
<td>Observer</td>
<td>1</td>
<td>4</td>
<td>0.25</td>
<td>31%</td>
</tr>
<tr>
<td>17</td>
<td>Prototype</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>18</td>
<td>Proxy</td>
<td>2</td>
<td>2</td>
<td>0.25</td>
<td>50%</td>
</tr>
<tr>
<td>19</td>
<td>Singleton</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>20</td>
<td>State</td>
<td>2</td>
<td>2</td>
<td>0.25</td>
<td>50%</td>
</tr>
<tr>
<td>21</td>
<td>Strategy</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>25%</td>
</tr>
<tr>
<td>22</td>
<td>Template Method</td>
<td>2</td>
<td>2</td>
<td>0.25</td>
<td>50%</td>
</tr>
<tr>
<td>23</td>
<td>Visitor</td>
<td>2</td>
<td>2</td>
<td>0.25</td>
<td>50%</td>
</tr>
</tbody>
</table>
8.1.3 Performance Overhead Evaluation

Although adopting the proposed design pattern as a service methodology to implement, use and reuse design patterns as services provides many benefits in terms of pattern service reuse and the degree of design pattern knowledge set required for every actor involved in developing and customizing the pattern services, the methodology does come with some performance overhead. Although the overhead is considered small compared to the benefits promised by the methodology, it is worth noting that it may slightly affect the performance and response time of the applications that use pattern services.

Due to the fact that complete pattern services are divided into two separate sets of services, namely, the PS and the AppSSs; this adds the overhead of some additional service calls, when compared to the same design patterns implemented directly using the SOA paradigm. As a matter of fact, both paradigms, SOA and PS, introduce some overhead involved in service calling; however, the overhead introduced by the PS paradigm can be a bit more than the one brought by the SOA implementation. The reason is that in the PS paradigm, there might be the need for more back and forth service calls between the two sets of services in order to fulfill the required functionality and to conform to the subject design pattern.

To establish criteria that we can apply to estimate the amount of performance overhead that PS implementation of a certain design pattern incurs, we developed two test cases using the Observer and the Decorator design patterns. We chose these as our test case because both patterns entail a number of service calls above the average of two service calls per pattern. We implemented each of the two design patterns, first, directly, using
the SOA paradigm, and then we implemented the same design patterns using the PS paradigm. Then we compared the number of service calls that each implementation needed to fulfill the task according to the rules laid out by the subject design pattern.

Using the results from the two test cases, we proceeded to develop the formula to measure the performance overhead – “PerformOvhead”; we started by giving the minimum number of service calls needed by the PS paradigm to carry out the subject design pattern’s rules and functionality the term “MinPSSrvCall”. The calculation of MinPSSrvCall includes two arguments. This first is the minimum number of service calls that the PS part makes to the AppSSs one. The second argument consists of any service callbacks that one or more AppSSMs need to make to a service method in the PS part. We refer to the former as MinPSAppSSCall, and the latter as “MinAppSSPSCallback”. This gives us Equation 8.7, shown below, to calculate the minimum service calls required by the PS paradigm.

Equation 8.7: \( \text{MinPSSrvCall} = \text{MinPSAppSSCall} + \text{MinAppSSPSCallback} \)

Finally, we refer to the total number of service calls incurred by the direct SOA implementation as “MinSOASrvCall”. Once we had all the terms we needed, we proceeded to build Equation 8.8 to calculate the percentage of performance overhead incurred by the PS paradigm compared to its SOA counterpart:

Equation 8.8: \( \text{PerformOvhead} = \frac{(\text{MinPSSrvCall} - \text{MinSOASrvCall})}{\text{MinSOASrvCall}} \times 100\% \)

Next, we applied this formula to the rest of the 23 GoF design patterns. Table 8.4 shows the percentage of performance overhead incurred when the proposed PS paradigm is
used, instead of the direct implementation using the SOA paradigm. The percentage of
performance overhead – PerformOvhead – values, shown in Table 8.4, are calculated
using Equation 8.8.

Table 8.4: Estimated percentage of performance overhead incurred

<table>
<thead>
<tr>
<th>S.N.</th>
<th>Pattern Name</th>
<th>MinPS SrvCall</th>
<th>MinSOA SrvCall</th>
<th>Performance Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Abstract Factory</td>
<td>2</td>
<td>2</td>
<td>0%</td>
</tr>
<tr>
<td>2</td>
<td>Adapter</td>
<td>4</td>
<td>3</td>
<td>33%</td>
</tr>
<tr>
<td>3</td>
<td>Bridge</td>
<td>2</td>
<td>2</td>
<td>0%</td>
</tr>
<tr>
<td>4</td>
<td>Builder</td>
<td>3</td>
<td>3</td>
<td>0%</td>
</tr>
<tr>
<td>5</td>
<td>Chain of Responsibility</td>
<td>2</td>
<td>2</td>
<td>0%</td>
</tr>
<tr>
<td>6</td>
<td>Command</td>
<td>4</td>
<td>3</td>
<td>33%</td>
</tr>
<tr>
<td>7</td>
<td>Composite</td>
<td>6</td>
<td>5</td>
<td>20%</td>
</tr>
<tr>
<td>8</td>
<td>Decorator</td>
<td>4</td>
<td>3</td>
<td>33%</td>
</tr>
<tr>
<td>9</td>
<td>Façade</td>
<td>4</td>
<td>2</td>
<td>100%</td>
</tr>
<tr>
<td>10</td>
<td>Factory Method</td>
<td>4</td>
<td>2</td>
<td>100%</td>
</tr>
<tr>
<td>11</td>
<td>Flyweight</td>
<td>1</td>
<td>1</td>
<td>0%</td>
</tr>
<tr>
<td>12</td>
<td>Interpreter</td>
<td>6</td>
<td>5</td>
<td>20%</td>
</tr>
<tr>
<td>13</td>
<td>Iterator</td>
<td>4</td>
<td>4</td>
<td>0%</td>
</tr>
<tr>
<td>14</td>
<td>Mediator</td>
<td>2</td>
<td>2</td>
<td>0%</td>
</tr>
<tr>
<td>15</td>
<td>Memento</td>
<td>4</td>
<td>4</td>
<td>0%</td>
</tr>
<tr>
<td>16</td>
<td>Observer</td>
<td>5</td>
<td>3</td>
<td>67%</td>
</tr>
<tr>
<td>17</td>
<td>Prototype</td>
<td>1</td>
<td>1</td>
<td>0%</td>
</tr>
<tr>
<td>18</td>
<td>Proxy</td>
<td>4</td>
<td>3</td>
<td>33%</td>
</tr>
<tr>
<td>19</td>
<td>Singleton</td>
<td>1</td>
<td>1</td>
<td>0%</td>
</tr>
<tr>
<td>20</td>
<td>State</td>
<td>3</td>
<td>2</td>
<td>50%</td>
</tr>
<tr>
<td>21</td>
<td>Strategy</td>
<td>2</td>
<td>2</td>
<td>0%</td>
</tr>
<tr>
<td>22</td>
<td>Template Method</td>
<td>4</td>
<td>2</td>
<td>100%</td>
</tr>
<tr>
<td>23</td>
<td>Visitor</td>
<td>4</td>
<td>3</td>
<td>33%</td>
</tr>
</tbody>
</table>
The results of applying Equation 8.8 to the 23 GoF pool of design patterns implemented as pattern services are that the percentage of performance overhead of the pattern service paradigm, compared to the direct SOA implementation, can range from 0% to 100%, where 0% means that both paradigms have the same performance efficiency, while 100% means that the pattern service paradigm incurs double the overhead of the direct SOA paradigm. The main factor the controls the increase in the performance overhead in the pattern service paradigm is the fact that pattern services are divided into two layers of services. Most times, these two layers of services need to integrate and interact to fulfill their tasks. Having the two layers physically separated means that their integration has to occur over a network connection. That is the source of performance overhead in the case of the pattern service paradigm. So, the more back and forth interactions required between the two layers of a pattern service the higher the performance overhead.

8.2 Test-based Evaluation

Testing is the act of running a completed program with various inputs to discover problems. It also includes any evaluation that is performed by a human or a machine to assess the quality of the evolving system. The goal of testing is to achieve complete code coverage, meaning all code is executed at least once. Minimally for a service, that means every service method is called at least once. Furthermore, tests should include at least some typical values and some border cases.

Robust good testing covers all possible scenarios and tests for all possible service interactions that may occur when the software – pattern service in this case – is being used. One of the qualities of a good test is to focus only on the erroneous results or
unexpected behavior, rather than showing the correct results and proper behavior, on the part of the software artifact (Lewis and Loftus, 2017).

To complement the analysis-based evaluation of the proposed PS methodology and the PS that applying such a methodology produces, we need to add another section to this chapter covering the test-based evaluation of the PSs built according to the proposed methodology. This section and its subsections are dedicated to explaining the testing procedures that a pattern service and its complementing components, such as the AppSSs, should be put through to confirm that the pattern service is ready to be used to build software applications. The testing includes testing the pattern service, the application-specific services, and the binding and configuration procedures that convert the subject pattern service into an application-specific pattern service.

Before we delve into the details of each type of test and its components and artifacts, let us first define what a test case is and what it involves:

“A test case consists of a set of inputs, user actions, or other initial conditions, along with the expected output” (Lewis and Loftus, 2017, p.334).

In the subsections below we describe the test cases that should be built and run to test certain aspects of the pattern service. Such aspects include testing the functionality of the pattern service and making sure that the pattern service components function properly in difference circumstances using different ranges of input data and execution environments and scenarios. The Functionality test in turn is divided into three phases: the first involves testing the application-independent service methods, the second takes care of testing the PS after it has been converted into an AppSPS by configuring it to the
requirements of a certain application. So the second set of tests test the AppSPS to make sure it functions properly and according to the application needs, in this phase all the AppSSs required by the PS are substituted with simple test stubs. The third phase of functionality test adds the actual AppSSs to the mix and tests complete AppSPS, with the focus on testing AppSS functionality to make certain that the AppSPS is ready to be plugged into the application.

In addition to the functionality test, a PS must be tested for pattern conformability. This test is concerned with how the PS implementation of a design pattern actually follows the rules, purpose, and guidelines of the subject design pattern. So, a test case that tests for conformability needs to be executed in a way that follows the exact behavior and collaborations recommended by the design pattern. The third, and last, test case that needs to be performed on a PS is to test the implementation for performance and response time. This is particularly important if the PS is to be used in an application where performance efficiency and response time is of high importance to the application. Another important software quality that a PS needs to be tested for is the scalability of the PS. In applications where the PS may be used heavily, and in some cases simultaneously, by a large number of users, both client and other components within the application itself, the scalability of the PS and how gracefully, and efficiently it delivers its functionality is of utmost importance. The third part of this test-based evaluation section is dedicated to the test case design for performance and scalability. At the end of this section, we show some performance and scalability test results of the Observer design pattern, comparing its implementation as an SOA service and as a PS.
8.2.1 Test Case Design for Functionality

The purpose of functionality testing is to carry out a battery of unit tests to ensure the soundness of the implementation and correctness of the results produced by the different services that make up the pattern service. The correctness of the overall results of the pattern service as a whole depends, of course, on the correctness of the results produced by each interaction between the service methods that make up the pattern service, including the AppSS part. That is why, in order to perform a complete functionality test for a given pattern service, we need to carry out three sets of functionality tests. The first test case focuses on the generic part of the pattern service. The focus of this test is the application-independent service methods (AppISMs) of the PS. The second test case is concerned with testing the application-dependent service methods (AppDSMs). These are the service methods whose code is generated by the applications dependent service templates (AppDSTs) after running the configuration file. The third test case takes care of the test procedures meant to test the AppSSs and their service methods that need to be integrated with the PS to enable it to deliver the functionality prescribed by the subject design pattern.

a) Test Case Design for Generic Pattern Services (PSs)

As mentioned in the last section, the purpose of this test case is to test the AppISMs, implemented by the PSCs as part of the generic PS. AppISMs of a PS usually implement the generic logic of the design pattern; therefore, they should work the same way regardless of the application that uses that PS without the need for any application-dependent logic or configuration.
• Test Case Artifacts

To build test cases for the generic part of the PS, application-independent service methods (AppISM) must first be identified. The focus of this test case is to test the generic service methods of the PS – not any application-dependent ones. Therefore, the artifacts available for the testing team that builds and runs the test cases for this part of the PS are

1- All the AppISMs that represent the generic logic of the subject design pattern that the PS implements;

2- Any service method stubs that the AppISMs need to interact with to deliver their functionalities;

3- The test drivers/client services that initiate the test interactions involved in testing every AppISM;

4- Any test data needed to run the tests, including all data required when invoking AppISMs and other data that AppISMs may need to pass to the test service method stubs.

• Test Case Procedure

The steps this type of test case performs may vary slightly, depending on the nature of the AppISM to be tested. For AppISMs that fulfill their purpose without the need to interact with services from other parts of the pattern service, such as the AppSS, test cases only need to test the functionalities of those services, with no need for any AppSS components/stubs. Even though this might be the case with some AppISMs, the possibility of AppISMs needing to call one or more AppSSMs to fulfill their functionality is always there. Testing AppISMs that need to call one or more AppSSMs
requires the creation of the suitable AppSSM stubs that can be used instead of the actual — fully fledged — AppSSMs. The reason is, at this stage, the PS has not been released for use yet. Consequently, no actual AppSSs or AppSSMs have been developed. This means that to test an AppISM that requires to perform a call to one or more AppSSMs, service method stubs substituting for the actual AppSSMs need to be developed as part of the development of the overall test cases. It is worth mentioning that the AppSSM stubs must mimic the correct and exact functionality of the full-fledged counterparts. This includes the provision for any requirements for service callbacks that an actual AppSSM needs to make to any of the service methods in the PS.

Testing an AppISM that does not require a call to an AppSSM is done by the test driver service calling that AppISM, passing it the test data required to invoke the AppISM.

Testing an AppISM that performs one or more service calls to other service methods that are not already implemented, such as AppSSMs, requires that the test data that the test driver service that starts the test procedure passes to that AppISM include the proper data to be used in the extra service calls involved in performing the test. Once invoked, the AppISM should be executed and the respective service method stubs are called.

b) Test Case Design for Application-specific Pattern Services (AppSPSs)

Once the functionality test for the generic PS and its AppISM is done, the PS should be put through the second phase of testing. This second testing phase is concerned with testing the functionalities and behavior of the PS that is application dependent. As explained in general methodology idea for implementing design patterns as pattern services, the application-dependent logic of a design pattern is implemented as configurable application-dependent service templates (AppDSTs). AppSPSCs run a
configuration file that is designed and built to convert the PS into an AppSPS that is tailored to work with a specific software application. Part of the task that the configuration file does is the configuration of the AppDSTs to generate the application-dependent service methods (AppDSMs) with the application-specific logic that makes the PS work according to the needs of the software application.

- **Test Case Artifacts**

This test case focuses on testing the process of converting the PS into an AppSPS and how the generated application-dependent logic behaves. The artifacts that are involved in performing this kind of test include

1- The implemented and tested AppISMs of the PS,

2- All the AppDSTs that are used to generate the application-dependent logic that turns a generic PS into an AppSPS,

3- The configuration file that is used to configure the AppDSTs,

4- The service method stubs that all AppISMs and AppDSMs need to interact with during the execution of the test case,

5- Test drivers/client services needed to initiate all the test cases for every possible interaction the newly added/configured components of the PS may make,

6- All test data necessary to run the test case with all its stages and the interactions between the different service methods involved.

- **Test Case Procedure**

This test case starts with applying the necessary configurations to the PS. This is done by running the configuration file against the PS. The process of running the configuration file should result in the generation of AppDSMs. Once the application-
specific logic has been added to the PS, the next step to be made is to run some tests on every one of the newly generated AppDSMs to make sure that their application-specific logic is correct and that all the changes that the configuration step made to the PS are proper and do not break the PS. To do this, of course, the rest of the artifacts, listed above, need to be put in action. Firstly, for every test case, the test drivers/client services that invoke the parts of the AppSPS being tested should be started. In addition to the AppDSMs, these test drivers may need to use some of the AppISMs, tested in the previous step. The service methods that the test drivers interact with may in turn need to call some AppSSMs. The proper service method stubs are used as a substitute to the AppSSs and their service methods to enable the completion of the testing.

c) Test Case Design for Application-specific Services (AppSSs)
Testing AppSSs and their service methods is the act of putting the actual, full-fledged, AppSS component of the PS into action and checking the integrity and correctness of its results and interactions with the PS. In order for this type of test to work, all services and service methods of the PS must be implemented, configured, and ready to be used. Such a requirement can, in most cases, make this test a comprehensive one that tests the whole AppSPS, making the completion of this test results in an AppSPS that is configured, tested, and ready to be used with the specific application at hand.

- Test Case Artifacts
As noted above, this test and its test cases focus on testing the AppSSs and their service methods. In doing so, it uses almost all of the other services and service methods in the AppSPS. This means that the artifacts that need to be there to carry out such a test include the following:
1- The implemented and tested AppISMs of the PS,

2- All the AppDSMs with the proper application-specific logic,

3- The configuration file that is used to integrate the AppSSs and their service methods with the PS,

4- Test drivers/client services needed to initiate all the test cases for every possible interaction that is directed from the PS to one or more of the AppSSMs and vice versa,

5- All test data necessary to run the test case with all its stages and the interactions between the different service methods involved.

- Test Case Procedure

Once the PS and AppSPS tests have been performed and the AppSPS is ready for the AppSS component to be integrated, the proper configuration file is run to bind the AppSSs with the PS. When the proper communication and integration between the PS and AppSS components have been established, this test can be started by executing the proper test driver that initiates the series of interactions that take place in the AppSPS being tested. Depending on the nature of the pattern, the test driver may invoke a service method in either the PS or the AppSS component. If the test driver invokes a service method in the PS component, the first interaction between the PS and AppSS components is initiated from the PS component to the AppSS. If the test driver first calls a service method in the AppSS component, however, the initial service call between the two components originates from one of the services in the AppSS component towards a service method in the PS one. Either way, the minimum
requirement for this test is to cover all possible interactions prescribed by the PS that involve service methods on the AppSS side.

8.2.2 Test Case Design for Conformability

Conformability testing of a PS is meant to check whether that PS follows the rules and behavior laid out by the subject design pattern or not. Since a PS is basically a service-based reusable implementation of a specific design pattern, it is very important for the PS to deliver the same functionality and reflect the exact behavior described by the subject design pattern. That is why a conformability test is as important as the functionality ones. It should be noted, though, that a PS conformability test cannot be executed until the subject PS passes all three functionality tests. The reason is that the PS conformability test can only be effectively run on a complete and functional PS.

The conformability test of a PS can be done by testing all of that pattern’s prescribed collaborations and paths of execution that lead to the pattern’s anticipated results. Due to the fact that a PS is a service-based implementation of a design pattern, the collaboration of the PS components may not exactly match those depicted by the subject design pattern; therefore, a successful PS conformability test proves that the overall collaboration between the PS components, services, and service methods eventually delivers the same exact functionality and results that the collaboration prescribed by the subject design pattern does.

If we use the Observer design pattern as an example, the collaboration prescribed by this design pattern is that one or more Subjects have one or more Observers. Every time the state of a Subject changes, it notifies all its observers of the recent state change. When an Observer receives a notification from a Subject it is observing, it requests the
specific pieces of data of interest from that Subject and updates its state accordingly. An Observer PS should follow the guidelines laid out by the Observer design pattern and implement the sequence of interactions between its Subject and Observer services that eventually deliver the same collaboration. A conformability test case for an Observer PS examines the interactions of the Observer PS and, based on the sequence of interactions and their results, it concludes whether the PS implements the correct collaboration prescribed by the Observer design pattern or not.

8.2.3 Test Case Design for Performance and Scalability

The adoption of a PS as a part of a software application depends on its performance and how the performance of the PS affects the overall performance of the software application. After running the functionality and conformability battery of tests, the time comes for testing the performance of the PS and how scalable its implementation is. In the performance test of a PS, we are looking to measure how good the PS implementation is in terms of efficiency and response time. It is important for a PS to be efficient in order to assist in improving the overall efficiency of the software application that uses it.

A PS performance test case tests the PS in the normal usage environment with typical use cases. By typical use cases we mean that the PS should be tested with a number of client services/stubs that the PS deals with during execution of the software application in normal circumstances. Tests for extreme use cases, where multiple client services/stubs are involved, are run during the scalability tests.

Scalability testing is a special type of performance testing. It is also an important factor in the adoptability of the PS and endorsing it to be an integral part of a software
application. The scalability test of a PS measures the efficiency and response time of the PS when it is being used – simultaneously, most times – by several client services and/or processing large volumes of data. The scalability test case usually starts by running the PS with a certain number of client services/stubs to determine the response time of the PS. Then, the number of the client services/stubs and/or volume of data to process is gradually increased, the response time is measured, and the degree of efficiency of the PS is monitored. The monitoring of the efficiency of the PS enables the testers to decide when to use multiple instances of the PS and apply some load balancing strategy to distribute the workload between the PS instances to maintain the efficiency of the PS.

8.3 Experiments and Results

As a live example of PS performance and scalability testing, we implemented the Observer design pattern first according to the SOA paradigm, and then we implemented it using the proposed PS methodology. The goal is to compare the performance and scalability of the two implementation paradigms. We expect the efficiency and response times for the SOA implementation to be a little better than the PS one. This is because of the few extra service calls between the generic PS and its AppSSs, involved in the PS paradigm. However, one should keep in mind that this performance and scalability testing constitutes only a part of the overall picture painted by this chapter of evaluating the PS paradigm. Reusability of the PSs created using the PS paradigm, and the distribution of pattern knowledge among the software developer involved in creating the complete AppSpS are also important factors in painting such a picture.
During the battery of performance and scalability tests, described below, we compare the response times of the Observer design pattern in the two implementations, SOA and PS. For all test cases, we run each test case 10 times on both implementations, with a certain number of Subject state change operations in each test case. Of course, each Subject state change operation triggers the sequence of interactions between the Subject and Observer services, as prescribed by the Observer design pattern and implemented according to the SOA and PS paradigms.

To calculate the number of Observer state update operations in each test case we multiply the following three values (total number of Subject state changes messages, the total number of Subjects, and the total number of Observers of each Subject) as shown in the following equation:

**Equation 8.9:** \( \text{TotObsUpd} = \text{TotSubChgMsg} \times \text{TotSub} \times \text{TotSubObs} \)

The test stubs used in testing the two implementations are designed to measure and display the fastest and slowest response times for those 100 Subject state change operations. Note that the response time is measured in milliseconds.

We start these test cases by measuring the response times when there is only one Subject and one Observer in the application. Table 8.5 gives the results of the performance test for the Observer design pattern in both implementations, the (SOA) and (PS). Table 8.5 shows the comparison in response time between the two implementations of the Observer design pattern, with one Subject and one Observer. According to Equation 8.9, \( \text{TotObsUpd} = 100 \times 1 \times 1 = 100. \)
Table 8.5: SOA and PS impl. using 1 Subject and 1 Observer

<table>
<thead>
<tr>
<th>S.N.</th>
<th>(SOA) response time in (ms) for 100 Subject state change operations</th>
<th>(PS) response time in (ms) for 100 Subject state change operations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fastest</td>
<td>Slowest</td>
</tr>
<tr>
<td>1</td>
<td>0.213</td>
<td>0.876</td>
</tr>
<tr>
<td>2</td>
<td>0.233</td>
<td>0.667</td>
</tr>
<tr>
<td>3</td>
<td>0.234</td>
<td>0.974</td>
</tr>
<tr>
<td>4</td>
<td>0.208</td>
<td>0.549</td>
</tr>
<tr>
<td>5</td>
<td>0.207</td>
<td>0.736</td>
</tr>
<tr>
<td>6</td>
<td>0.215</td>
<td>0.608</td>
</tr>
<tr>
<td>7</td>
<td>0.240</td>
<td>0.574</td>
</tr>
<tr>
<td>8</td>
<td>0.211</td>
<td>0.542</td>
</tr>
<tr>
<td>9</td>
<td>0.227</td>
<td>0.941</td>
</tr>
<tr>
<td>10</td>
<td>0.229</td>
<td>0.517</td>
</tr>
</tbody>
</table>

As can be seen from the results shown in Table 8.5, the average difference in response time between the two paradigms is about 0.24 milliseconds. This delay or – performance overhead – is essentially a result of the layer separation of the pattern service paradigm. Of course, this latency is subject to the speed and quality of the network being used.

After measuring the average performance ratio between the two implementations of the Observer design pattern, we proceed to measure how these two implementations scale when the number of Subjects and /or Observers is gradually increased.

Table 8.6 gives the response time for the (SOA) implementation of the Observer design pattern when there is one Subject and many Observers. This test case involves 1 Subject and 10 Observers. According to Equation 8.9, \( \text{TotObsUpd} = 100 \times 1 \times 10 = 1000 \).
Table 8.7 gives the response time for the (PS) implementation of the Observer design pattern when there is one Subject and many Observers. This test case involves 1 Subject and 10 Observers. According to Equation 8.9, \( \text{TotObsUpd} = 100 \times 1 \times 10 = 1000 \).

Comparing the average response of the SOA paradigm, shown in Table 8.6, against the average response time of the pattern service paradigm, shown in Table 8.7, we can conclude that as the number of subjects and observers, in both implementations of the observer design pattern, increases, the delay in the response time also increases. Moreover, because the overhead incurred by network latency is relative to the number of messages that need to be exchanged between the two layers of the pattern service implementation of the Observer design pattern, the difference between the average response times of the two paradigms increases. However, the increase is not high.
considering the 10-fold increase of the number of interactions involving network usage. The difference in the average response times between the two paradigms is about 1.5 milliseconds.

Next we show the scalability test results for the two implementations of the Observer design pattern, when several Subjects and Observers are involved simultaneously.

Table 8.7: PS implementation using 1 Subject and many Observers

<table>
<thead>
<tr>
<th>S.N.</th>
<th>(PS) response time in (ms) per Observer for 100 Subject state change operations</th>
<th>(PS) response time in (ms) for all Observers for 100 Subject state change operations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fastest</td>
<td>Slowest</td>
</tr>
<tr>
<td>1</td>
<td>0.250</td>
<td>1.594</td>
</tr>
<tr>
<td>2</td>
<td>0.249</td>
<td>0.842</td>
</tr>
<tr>
<td>3</td>
<td>0.251</td>
<td>0.985</td>
</tr>
<tr>
<td>4</td>
<td>0.248</td>
<td>1.384</td>
</tr>
<tr>
<td>5</td>
<td>0.255</td>
<td>1.242</td>
</tr>
<tr>
<td>6</td>
<td>0.245</td>
<td>1.255</td>
</tr>
<tr>
<td>7</td>
<td>0.252</td>
<td>1.563</td>
</tr>
<tr>
<td>8</td>
<td>0.257</td>
<td>0.926</td>
</tr>
<tr>
<td>9</td>
<td>0.254</td>
<td>1.150</td>
</tr>
<tr>
<td>10</td>
<td>0.248</td>
<td>0.983</td>
</tr>
<tr>
<td></td>
<td>Average per Observer</td>
<td>0.722</td>
</tr>
</tbody>
</table>

Table 8.8 gives the response times for the (SOA) implementation of the Observer design pattern when there are many Subjects having many Observers. The states of the Subjects are simultaneously updated which triggers simultaneous notification and update procedures of their Observers. In this test case there were 10 Subjects; each has 10 Observers. Table 8.8 reflects the response times when 1 state change operation is applied to all 10 Subjects, which, in turn, triggers the notification and update of the state
of the 10 Observers that are registered with each Subject. According to Equation 8.9,
\[ \text{TotObsUpd} = 1 \times 10 \times 10 = 100. \]

<table>
<thead>
<tr>
<th>S.N.</th>
<th>(SOA) response time in (ms) per Observer for 10 Subject state change operations</th>
<th>(SOA) response time in (ms) for all Observers for 10 Subject state change operations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fastest</td>
<td>Slowest</td>
</tr>
<tr>
<td>1</td>
<td>0.083</td>
<td>2.434</td>
</tr>
<tr>
<td>2</td>
<td>0.082</td>
<td>2.253</td>
</tr>
<tr>
<td>3</td>
<td>0.086</td>
<td>1.902</td>
</tr>
<tr>
<td>4</td>
<td>0.085</td>
<td>1.989</td>
</tr>
<tr>
<td>5</td>
<td>0.083</td>
<td>1.913</td>
</tr>
<tr>
<td>6</td>
<td>0.090</td>
<td>3.306</td>
</tr>
<tr>
<td>7</td>
<td>0.088</td>
<td>2.785</td>
</tr>
<tr>
<td>8</td>
<td>0.083</td>
<td>2.498</td>
</tr>
<tr>
<td>9</td>
<td>0.079</td>
<td>2.264</td>
</tr>
<tr>
<td>10</td>
<td>0.077</td>
<td>3.078</td>
</tr>
<tr>
<td></td>
<td><strong>Average per Observer</strong></td>
<td><strong>1.263</strong></td>
</tr>
</tbody>
</table>

Table 8.9 shows the response times for the (PS) implementation of the Observer design pattern when there are many Subjects having many Observers. The states of the Subjects are simultaneously updated which triggers simultaneous notification and update procedures of their Observers. According to Equation 8.9,
\[ \text{TotObsUpd} = 1 \times 10 \times 10 = 100. \]
Table 8.9: PS impl. using many Subjects and many Observers – test1

<table>
<thead>
<tr>
<th>S.N.</th>
<th>(PS) response time in (ms) per Observer for 10 Subject state change operations</th>
<th>(PS) response time in (ms) for all Observers for 10 Subject state change operations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fastest</td>
<td>Slowest</td>
</tr>
<tr>
<td>1</td>
<td>0.111</td>
<td>3.323</td>
</tr>
<tr>
<td>2</td>
<td>0.108</td>
<td>3.016</td>
</tr>
<tr>
<td>3</td>
<td>0.107</td>
<td>4.094</td>
</tr>
<tr>
<td>4</td>
<td>0.111</td>
<td>4.397</td>
</tr>
<tr>
<td>5</td>
<td>0.111</td>
<td>3.707</td>
</tr>
<tr>
<td>6</td>
<td>0.111</td>
<td>2.146</td>
</tr>
<tr>
<td>7</td>
<td>0.111</td>
<td>2.645</td>
</tr>
<tr>
<td>8</td>
<td>0.110</td>
<td>2.544</td>
</tr>
<tr>
<td>9</td>
<td>0.111</td>
<td>3.237</td>
</tr>
<tr>
<td>10</td>
<td>0.109</td>
<td>2.997</td>
</tr>
</tbody>
</table>

| Average per Observer | 1.660 | Average for all Observers | 8.416 |

Tables 8.8 and 8.9 give the change of the response time delay when the numbers of Subjects/Observers are increased. Each Subject sends 1 state update notification to its Observers. The average response time delay when the two paradigms are compared is about 1.79 milliseconds. The slight increase in response time delay when using the pattern service here is due to the fact that now there are 10 Subjects that need to send the notifications to 10 different Observers, instead of just one subject in the previous test.

Table 8.10 gives the response time when the (SOA) implementation of the Observer design pattern is tested with many Subjects having many Observers. The Observers’ states are updated simultaneously using a total of 10000 state update operations. The figures in Table 8.10 reflect the amount of time it took the (SOA) implementation to
process 100 Subject state change operations for 10 Subjects and the notification and update procedures for 10 Observers for each Subject. According to Equation 8.9, \( \text{TotObsUpd} = 100 \times 10 \times 10 = 10000 \).

Table 8.10: SOA impl. using many Subjects and many Observers - test2

<table>
<thead>
<tr>
<th>S.N.</th>
<th>(SOA) response time in (ms) per Observer for 1000 Subject state change operations</th>
<th>(SOA) response time in (ms) for all Observers for 1000 Subject state change operations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fastest</td>
<td>Slowest</td>
</tr>
<tr>
<td>1</td>
<td>0.096</td>
<td>2.556</td>
</tr>
<tr>
<td>2</td>
<td>0.097</td>
<td>2.741</td>
</tr>
<tr>
<td>3</td>
<td>0.094</td>
<td>2.737</td>
</tr>
<tr>
<td>4</td>
<td>0.095</td>
<td>2.609</td>
</tr>
<tr>
<td>5</td>
<td>0.096</td>
<td>2.942</td>
</tr>
<tr>
<td>6</td>
<td>0.095</td>
<td>3.027</td>
</tr>
<tr>
<td>7</td>
<td>0.097</td>
<td>2.912</td>
</tr>
<tr>
<td>8</td>
<td>0.093</td>
<td>2.700</td>
</tr>
<tr>
<td>9</td>
<td>0.095</td>
<td>2.742</td>
</tr>
<tr>
<td>10</td>
<td>0.094</td>
<td>3.114</td>
</tr>
</tbody>
</table>

| Average per Observer | 1.452 | Average for all Observers | 59.441 |

Table 8.11 gives the response time when the (PS) implementation of the Observer design pattern is tested with many Subjects having many Observers. The Observers’ states are updated simultaneously using a total of 10000 state update operations. The figures in Table 8.11 reflect the amount of time it took the (PS) implementation to process 100 Subject state change operations for 10 Subjects and the notification and update procedures for 10 Observers for each Subject. According to Equation 8.9, \( \text{TotObsUpd} = 100 \times 10 \times 10 = 10000 \).
Table 8.11: PS impl. using many Subjects and many Observers – test2

<table>
<thead>
<tr>
<th>S.N.</th>
<th>(PS) response time in (ms) per Observer for 1000 Subject state change operations</th>
<th>(PS) response time in (ms) for all Observers for 1000 Subject state change operations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fastest</td>
<td>Slowest</td>
</tr>
<tr>
<td>1</td>
<td>0.107</td>
<td>4.003</td>
</tr>
<tr>
<td>2</td>
<td>0.109</td>
<td>3.290</td>
</tr>
<tr>
<td>3</td>
<td>0.107</td>
<td>3.098</td>
</tr>
<tr>
<td>4</td>
<td>0.106</td>
<td>3.553</td>
</tr>
<tr>
<td>5</td>
<td>0.108</td>
<td>2.888</td>
</tr>
<tr>
<td>6</td>
<td>0.109</td>
<td>3.097</td>
</tr>
<tr>
<td>7</td>
<td>0.106</td>
<td>3.093</td>
</tr>
<tr>
<td>8</td>
<td>0.107</td>
<td>2.948</td>
</tr>
<tr>
<td>9</td>
<td>0.108</td>
<td>3.325</td>
</tr>
<tr>
<td>10</td>
<td>0.105</td>
<td>3.051</td>
</tr>
</tbody>
</table>

| Average per Observer | 1.671 | Average for all Observers | 67.125 |

By the same token, the delay in response time incurred by the pattern service paradigm, when comparing results in Table 8.10 and Table 8.11 is due to the increase of the number of Subjects, Observers, and the number of messages exchanged. The average response time delay shown in Table 8.10 and Table 8.11 amounts to 7.68 milliseconds. In conclusion, and taking into account the amount of increased Subjects, Observers, and exchanged messages, the ration of response time increase is consistent and reasonable; After all, the increase is a fraction of a millisecond or, in the worst case a couple milliseconds, when the numbers of Subjects, Observers, and/or messages exchanged over the network is increased 10-fold. A major factor in the response time delay is, of course, the quality of the underlying network connection. For the proposed
pattern service paradigm, better and faster network connection results in faster response time.

8.4 Summary

In this chapter we described the procedures we used to evaluate the proposed design pattern service methodology. The first section of this chapter took care of the analytical evaluation of the proposed methodology. In the analysis-based evaluation, we developed the mathematical equation to measure the effort involved in developing pattern services, the pattern knowledge each participant in the development needs to have, and the performance of the resulting pattern services. We applied those mathematical equations on both implementation paradigms, the PS and SOA. Then we compared their results to show the pros and cons of using the proposed methodology.

In the second section of this chapter, we discussed the test-based evaluation of the proposed methodology. We laid out the procedures that should be followed in building the test cases to test the resulting pattern services in terms of their functionality, conformability to the subject design pattern rules, and the performance and scalability of the pattern service at hand. Finally, we gave some test results for the Observer pattern service and compared its performance and scalability to those of the same pattern in its (direct) SOA implementation. In the next chapter, we conclude this research thesis by first giving a summary of the contents of the thesis. Then we highlight the contributions of this research. Finally, we identify the areas of research that can be pursued for future work.
Chapter 9

Conclusion and Future Work

9.1 Summary

In this PhD thesis we propose a methodology for implementing software design patterns as services that can be used to build service-oriented software systems. Although the proposed methodology is generic enough to potentially be adopted to implement most types of software patterns as pattern services, our work focuses particularly on applying the proposed methodology to software design patterns. Before going deep into the application of the proposed methodology to design patterns, however, we gave a step by step example of how the methodology can be applied from the extraction of the abstract parts of the subject pattern and how to implement them as a generic Pattern Service (PS) all the way to the details of classifying the concrete portions of the pattern, implementing them as Application-specific Services (AppSSs), and attaching them to the generic pattern service, which becomes an application-specific pattern service (AppSPS) after configuring, and adding the AppSS part to it. We also explained how the AppSS attachment to the generic pattern service can be done and what interfaces, service implementations, and other configuration requirements need to be put in place for the generic pattern service to be customized and become application specific and function as required by the Pattern-based application. Further, we made it clear that a generic pattern service must be converted into an application-specific one before it can be used in a software application. The generic part of a software pattern is rarely enough to carry out the complete functionality of the pattern, which is why a complementing
application-specific part has to be integrated with the generic part to produce a useful pattern service that can readily be used by the PbAppCs to build pattern-based software applications.

We used as an example a Stock Market Data Inquiry application to show the steps of using the Observer design pattern as the backbone of the application. Then we showed the steps of implementing the Observer design pattern as a pattern service and used it to manage and coordinate the provision of the latest stock data to the application’s customers. Then we generalized the major steps involved in implementing the Observer design pattern as a generic pattern service and the other steps that transformed the generic Observer pattern service into an application-specific one. The result of such a generalization is a process of four stages. The first stage involves studying the software pattern at hand, extracting the abstract and generic parts of it, and implementing them as a generic pattern service that is equipped with all the integration procedures that are needed to integrate application-specific services that complement the generic pattern service’s functionality. The other three stages of the generalized process deal with the creation of the services that implement the application-specific logic of the pattern at hand and the integration of these services with the generic pattern service, the configuration of the now semi-application-specific pattern service, and finally the usage of the completed pattern service in applications. We call the first stage the Application Creation/Integration stage. This stage takes care of the attachment (registration) of the AppSSs to the generic pattern service and generates the necessary interface and service method implementation to enable communication and interaction between the generic pattern service and the newly attached AppSSs.
We call the second stage the Application Instance Creation stage. This stage comes after
the complete integration of the generic pattern service with the AppSSs, and its purpose
is to further configure the now application-specific pattern service, set some of the
default values and AppSSs, and get the pattern service ready for service instance
creation that happens during the execution of the application that employs the pattern
service.

The third stage shows what happens during runtime; not surprisingly, we call this stage
the Runtime stage. This stage starts when instances of the components of the pattern
service get created. During this stage, the direct users of those instances, the client and
Setter application services, interact with the pattern service instances and manipulate
them according to the application needs.

After explaining in detail how the general methodology idea can be applied to an
example software pattern, in the following chapter we started the step by step process of
explaining how the proposed methodology can be applied specifically to software
design patterns. Starting from Chapter 4, we focused our work on the 23 GoF design
patterns. First we discussed the methodology conceptually and classified the subject 23
GoF design patterns into groups based on certain criteria and gave an example of each
group. Then in Chapter 5, we described in detail how the implementation of the
methodology can be realized through the binding and configuration steps that a pattern
service must be put through to be converted into an application-specific pattern service.

In Chapter 6 we presented some use cases that briefly showed how the pattern service
methodology can be used. First we gave a pattern service composition example where
the Command and Memento pattern services are combined to help provide the do/undo
functionalities for a chess game application. Then we gave some examples of how the methodology can be applied to implement some of the SOA patterns and classified them based on their relationship to the GoF design patterns.

In Chapter 7 we introduced the design and architecture of the Software Pattern as a Service (SPaaS) system, or (PaaS) for short. This system works as the platform for managing, integrating and deploying pattern services and Pattern-based Applications (PbApps). In the first section of Chapter 7 we gave the design of the system and described the system’s functionalities and highlighted the users of each stage of the process of creating and using pattern services and PbApps using the system. Then we introduced the prototypical implementation of the system and how it is tested using two case study applications, namely, an Online Discussion Group and Online Stock Market Ticker. We explained how the Observer design pattern is implemented as a pattern service, then how the two case study applications use that Observer pattern service to facilitate their interactions to deliver their functionalities. The storage, management, and deployment of the two use case applications and the Observer pattern service are taken care of by the implemented prototype PaaS system.

The testing of the PaaS system and the case study applications that use the implemented Observer pattern service proved robust, and the applications functioned properly under different execution circumstances and with different numbers of concurrent users.

In Chapter 8 we presented some evaluation procedures that we ran on the proposed methodology and compared its development effort, pattern knowledge requirements, functionality and performance to the SOA paradigm. Finally, we ran several
performance and scalability tests over the implementations of the Observer design pattern, as a PS and as a SOA, and listed the test results.

9.2 Discussion

This thesis proposes a few methodologies and sub-methodologies to guide the process of dissecting software design patterns, extracting their layers and logic, both generic and application-specific, and using those layers and logic to design a reusable pattern service that delivers the same functionality prescribed by the subject software design pattern. Similar to the nature and purpose of software design patterns, the resulting reusable pattern services are considered building blocks for robust and scalable service-oriented systems.

The process of implementing a software design pattern as a reusable pattern service involves using all the methodologies and artifacts proposed by this thesis. Firstly, the conceptual methodology steps to build the conceptual model of a pattern service using the given design pattern must be applied. Secondly, an implementation model must be created by applying the implementation methodology on the conceptual model, provided by the first step. Thirdly, the PaaS system is used to support the steps prescribed by the implementation methodology. Fourthly, the resulting application-specific pattern service is evaluated and tested using the proposed evaluation methodology. Finally, the application-specific pattern service becomes ready for use with the target service-oriented system.

Design pattern services that are built using the proposed group of methodologies are the service-based counterparts of the subject design patterns. This means that their
granularity is comparable to that of the subject design patterns; thus, they are classified as design building blocks for service-oriented systems. Since the granularity of design pattern services is equivalent to that of the subject design patterns, and design patterns are classified as design building blocks for software applications, design pattern services can be used exactly the same way and at the same levels that software design patterns are used, except that design pattern service are used to build service-oriented systems instead of other types of software applications. This includes the size and magnitude of the software application being built. It is proven that software design patterns provide solutions to design problems that can be found in small scale applications as well as enterprise-scale systems. By the same token, design pattern services should provide solutions to service-oriented system problems regardless of size. The fact that design pattern services are reusable service-based implementations of the design patterns brings an extra factor to the table, which is what is being saved by reusing those design pattern services. This is a valid question to ask before deciding whether to reuse the implemented design pattern service, or maybe implement your own from scratch. After all, reusing design pattern services built using the proposed methodology does incur a little bit of overhead represented in the layer separation of the generic part of the pattern service and its application-specific one. This overhead is largely dependent on the quality of the network. However, the other qualities that the proposed reusable pattern service paradigm brings, such as the standardized and correct implementations of the patterns, the quality control that comes with tested implementations, and the potential reduced amount of development effort and the fact that design pattern services do not require their users to have deep technical knowledge of the subject design patterns can make reusing design pattern services more appealing.
As part of this research, after designing the design pattern services for the 23 GoF design patterns, we applied the evaluation methodology to those designs in Chapter 8. The results in Tables 8.1, 8.2, 8.3, and 8.4 showed that the majority of the designed pattern services built using the proposed design pattern service paradigm save either some development effort, or they can be used by people with minimal knowledge of the pattern service itself and the subject design pattern. A few of the 23 design pattern services, however, proved to have the same development effort as if the design pattern was implemented using the direct SOA paradigm. But in terms of the required knowledge skillset, the design pattern service surpasses the direct SOA implementation paradigm, because in the latter, developers must have 100% of the subject design pattern they intend to implement. When it comes to performance overhead, however, the direct SOA implementation paradigm may perform better than the pattern service one. This is due to the fact that the pattern service paradigm implements the reusable pattern service as two separate groups of services that need to integrate and interact at runtime using the underlying network. This indirection step is what hurts the pattern service performance. But the amount of performance overhead is subject to the quality of the underlying network. This means that the better and faster the underlying network, the less the overhead will be.

Finally, taking into consideration all the advantages and reusability of the proposed pattern service paradigm, we think that even with the small margin of incurred performance overhead, the proposed reusable pattern service paradigm is still significantly better than the direct SOA implementation approach.
9.3 Conclusion

In conclusion, we can state that this thesis provided the important components that make up a solid methodology to guide software developers in a step by step manner to implement software design patterns as reusable pattern services. Adopting the proposed methodology should reduce the development effort involved in using design patterns in software applications. This is achieved through having reusable service implementations of the subject design patterns in an accessible repository for potential reuse.

As described in Chapter 1, the major contribution of this thesis is the development of the design pattern as a pattern service methodology. Also, as noted in Chapter 1, this proposed methodology has the potential of becoming a general methodology for implementing software patterns from areas other than design as pattern services. We stress that the conjecture of the proposed methodology becoming a general methodology does not have enough evidence in this thesis, and more research and experiments need to be done to confidently claim that the proposed methodology is undoubtedly applicable to any type of software patterns.

Among the principal contributions of this thesis is the development of systematic group of methodologies to guide the process of implementing software design patterns as pattern services. These systematic methodologies include a conceptual methodology to build a conceptual pattern service model of a given design pattern, and an implementation methodology that can be followed to put together an implementation pattern service model based on the conceptual pattern service model, provided by the conceptual methodology. The third methodology is an evaluation methodology that can be used to evaluate the whole pattern service paradigm. It is in turn divided into two
sub-methodologies: an analysis-based evaluation sub-methodology, and a test-based evaluation one. The analysis-based evaluation sub-methodology is developed and tested using the GoF pool of design patterns. The test-based evaluation sub-methodology, on the other hand, described the testing procedures for the resulting pattern services in terms of functionality, conformability, and performance and scalability. This thesis also includes some testing results that were performed on the implementation of the Observer design pattern as a pattern service. Among the contributions of this thesis is the design of the Pattern as a Service (PaaS) system, which functions as a platform for managing artifacts created and used by the pattern service paradigm. The last contribution of this thesis is the creation of a catalog that contains the complete design of all 23 GoF design patterns as pattern services using the proposed methodologies.

9.4 Future Work

The following areas are identified for probable future improvement to the overall offering of patterns as services:

1- The integration process between the two layers of the proposed pattern service methodology could be automated. Due to the requirement for two phases of a pattern service development, namely, the generic phase, and the application-specific one, the layering of the pattern service methodology is inevitable. This layering comes with a performance penalty that is incurred by the two layer indirection. This indirection and the performance penalty it causes are things worth further studying in order to improve the performance and responsiveness of pattern services, especially those that will potentially be used in applications and systems where response time is a paramount requirement.
2- Although the general methodology idea this research proposes, described in Chapter 3, may potentially work with software patterns other than design patterns, the investigation of specific pattern service development methodologies and designs for other types of software patterns, such as architectural and analysis patterns, can be useful. More research might produce a pattern service development methodology that is specifically tailored for a certain type of software patterns, e.g. architectural or analysis software patterns.

3- In addition to the configuration XML schema document that we are proposing in this research, some more standardization of the documentation of the developed generic pattern services is something worth looking into. Such a standardized documentation helps different types of pattern service users understand how to use pattern services, how to develop the complementing AppSS, and how to effectively configure and use them in their applications and systems:

Good and standardized documentation is an important asset in the success and ease of use of any artifact including software artifacts. Therefore, researching a well-designed documentation methodology for documenting the developed pattern services, especially the generic part, is effort and time well worth spending because the benefits of a standardized documentation structure and content save a lot of effort and time for the users of the pattern service and help reduce the overall development time and cost of Pattern-based Applications and systems.
Bibliography


http://www.Businessdictionary.com/definition/pattern.html


http://www.roughtype.com/archives/2008/01/a_little_too_mu.php


Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. USA: Adison-Wesley.


http://www.oxforddictionaries.com/definition/english/pattern


programming, systems, languages, and applications - OOPSLA '98 (pp. 117 - 133). Vancouver, BC, Canada: ACM.


295


### Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Creator:</td>
<td>A person or group of people who develop applications.</td>
</tr>
<tr>
<td>Application-dependent Service Method:</td>
<td>A service method that is part of a pattern service but its logic is dependent on the application that uses the pattern service.</td>
</tr>
<tr>
<td>Application-dependent Service Template:</td>
<td>A configurable service method template that generates actual application-dependent service method when it is properly configured.</td>
</tr>
<tr>
<td>Application-independent Service Method:</td>
<td>A service method of a pattern service that contains generic logic that is independent of the application that uses the pattern service.</td>
</tr>
<tr>
<td>Application-specific Pattern Service:</td>
<td>A pattern service that is configured and adapted to work with a certain application.</td>
</tr>
<tr>
<td>Application-specific Pattern Service Creator:</td>
<td>A person or group of people who configure and adapt a generic pattern service to convert it into an application-specific pattern service.</td>
</tr>
<tr>
<td>Application-specific Pattern Service Repository:</td>
<td>A repository that holds application-specific pattern services.</td>
</tr>
<tr>
<td>Application-specific Service Method:</td>
<td>A service method that is part of an application-specific service.</td>
</tr>
<tr>
<td>Application-specific Service:</td>
<td>A service that contains logic specific to a certain application.</td>
</tr>
<tr>
<td>Application-specific Service Repository:</td>
<td>A repository that holds application-specific services.</td>
</tr>
<tr>
<td><strong>Application-specific Service Reference:</strong></td>
<td>The service reference used to register the application-specific service with the generic pattern service.</td>
</tr>
<tr>
<td><strong>Field of Discussion:</strong></td>
<td>The area of discussion such as sports, economics, and politics … etc.</td>
</tr>
<tr>
<td><strong>Generic Pattern Service:</strong></td>
<td>A software service or group of services that implement just the generic (abstract) logic of a software pattern as a service.</td>
</tr>
<tr>
<td><strong>Online Discussion Group:</strong></td>
<td>A discussion group application implemented as a case study.</td>
</tr>
<tr>
<td><strong>Online Stock Market Ticker:</strong></td>
<td>A stock market ticker application implemented as a case study.</td>
</tr>
<tr>
<td><strong>Pattern as a Service System:</strong></td>
<td>A software system that functions as a platform for creating, storing, configuring, integrating, deploying, and managing pattern services, application-specific services, and pattern-based applications.</td>
</tr>
<tr>
<td><strong>Pattern-based Application:</strong></td>
<td>An application in which one or more pattern services are used in its development.</td>
</tr>
<tr>
<td><strong>Pattern-based Application Creator:</strong></td>
<td>A person or group of people who develop pattern-based applications. This person or group of people may also assume the roles of the application-specific service creators or the application-specific pattern service creators.</td>
</tr>
<tr>
<td><strong>Pattern-based Application Instance:</strong></td>
<td>A deployed instance or image of a Pattern-based Application.</td>
</tr>
<tr>
<td><strong>Pattern-based Application Instance Creator:</strong></td>
<td>A person or group of people who configure and deploy instances or images of ready to use pattern-based applications.</td>
</tr>
<tr>
<td><strong>Pattern-based Application Instance End User:</strong></td>
<td>A person or group of people who use deployed pattern-based application instances.</td>
</tr>
<tr>
<td><strong>Pattern-based Application Repository:</strong></td>
<td>A repository that holds fully developed and ready to deploy pattern-based applications.</td>
</tr>
<tr>
<td><strong>Pattern Service:</strong></td>
<td>A software service or group of services that implement a software pattern as a service.</td>
</tr>
<tr>
<td><strong>Pattern Service Creator:</strong></td>
<td>A person or group of people who implement the generic pattern services and create pattern service compositions.</td>
</tr>
<tr>
<td><strong>Pattern Service Repository:</strong></td>
<td>A repository that holds implemented generic pattern services and pattern service compositions.</td>
</tr>
<tr>
<td><strong>Stock Ticker:</strong></td>
<td>A report of the price for certain securities, updated continuously throughout the trading session by the various stock exchanges.</td>
</tr>
<tr>
<td><strong>Stock Ticker Feed:</strong></td>
<td>A software program (part of an application) that pulls stock data information from the stock ticker.</td>
</tr>
</tbody>
</table>
Appendix A

Catalog of Design Pattern Services

A.1 Abstract Factory Design Pattern Service

Abstract Factory Design Pattern:

![Abstract Factory Design Pattern](image)

Figure A.1: Abstract Factory design pattern after GoF (p. 88)
Design of Abstract Factory Pattern Service:

Figure A.2: Abstract Factory PS
Table A.1: Interface of the generic layer of the Abstract Factory PS

<table>
<thead>
<tr>
<th>Application-dependent Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Template&gt; CreateProduct</td>
</tr>
</tbody>
</table>

![Diagram of Runtime Interactions of the Abstract Factory PS](image)

Figure A.3: Runtime interactions of the Abstract Factory PS
A.2 Adapter Design Pattern Service

Adapter Design Pattern:

![Diagram of Adapter Design Pattern]

Figure A.4: Adapter design pattern after GoF (p. 141)
Design of the Adapter Pattern Service:

Figure A.5: Adapter PS
Table A.2: Interface of the generic layer of the Adapter PS

<table>
<thead>
<tr>
<th>Application-dependent Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Template&gt; Request</td>
</tr>
<tr>
<td>&lt;Template&gt; SpecificRequest</td>
</tr>
</tbody>
</table>

![Diagram]

Figure A.6: Runtime interactions of the Adapter PS
A.3 Bridge Pattern Service

Bridge Design Pattern:

![Bridge Design Pattern Diagram]

Figure A.7: Bridge design pattern after GoF (p. 153)
Design of the Bridge Pattern Service:

Figure A.8: Bridge PS
Table A.3: Interface of the generic layer of the Bridge PS

<table>
<thead>
<tr>
<th>Application-dependent Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Template&gt; OperationImplementation</td>
</tr>
</tbody>
</table>

Figure A.9: Runtime interactions of the Bridge PS
A.4 Builder Pattern Service

Builder Design Pattern:

Figure A.10: Builder design pattern after GoF (p. 98)
Design of the Builder Pattern Service:

Figure A.11: Bridge PS
Table A.4: Interface of the generic layer of the Builder PS
Application-dependent Interface
<Template> BuildPart
<Template> GetResult

Figure A.12: Runtime interactions of the Builder PS

312


A.5 Chain of Responsibility Pattern Service

Chain of Responsibility Design Pattern:

Figure A.13: Chain of Responsibility design pattern after GoF (p. 225)
Design of the Chain of Responsibility Pattern Service:

![Diagram of the Chain of Responsibility Pattern Service]

Figure A.14: Chain of Responsibility PS
Table A.5: Interface of the generic layer of the Chain of Responsibility PS

<table>
<thead>
<tr>
<th>Application-dependent Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Template&gt; HandleRequest</td>
</tr>
</tbody>
</table>

Figure A.15: Runtime interactions of the Chain of Responsibility PS
A.6 Command Pattern Service

Command Design Pattern:

Figure A.16: Command design pattern after GoF (p. 236)
Design of the Command Pattern Service:

Figure A.17: Command PS
Table A.6: Interface of the generic layer of the Command PS

<table>
<thead>
<tr>
<th>Application-independent Interface</th>
<th>Application-dependent Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>&lt;Template&gt; Action</td>
</tr>
</tbody>
</table>

Figure A.18: Runtime interactions of the Command PS
A.7 Composite Pattern Service

Composite Design Pattern:

Figure A.19: Composite design pattern after GoF (p. 164)
Design of the Composite Pattern Service:

Figure A.20: Composite PS
Table A.7: Interface of the generic layer of the Composite PS

<table>
<thead>
<tr>
<th>Application-independent Interface</th>
<th>Application-dependent Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddChild</td>
<td>&lt;Template&gt; Operation</td>
</tr>
<tr>
<td>RemoveChild</td>
<td>#..................................</td>
</tr>
<tr>
<td>GetChild</td>
<td>#..................................</td>
</tr>
</tbody>
</table>

Figure A.21: Runtime interactions of the Composite PS - Selected Component is a Composite
Figure A.22: Runtime interactions of the Composite PS - Selected Component is a Leaf
A.8 Decorator Pattern Service

Decorator Design Pattern:

![Decorator Design Pattern](image)

*Figure A.23: Decorator design pattern after GoF (p. 177)*
Design of the Decorator Pattern Service:

![Diagram of Decorator Pattern Service]

Figure A.24: Decorator PS
Table A.8: Interface of the generic layer of the Decorator PS

<table>
<thead>
<tr>
<th>Application-dependent Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Template&gt; DecoratorOperation</td>
</tr>
<tr>
<td>&lt;Template&gt; ComponentOperation</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Client App Service</th>
<th>Selected Concrete PS</th>
<th>Decorator Concrete Component Decorator Sequence AppSS AppSS List</th>
</tr>
</thead>
</table>

Figure A.25: Runtime interactions of the Decorator PS
A.9 Façade Pattern Service

Façade Design Pattern:

![Façade Design Pattern](image)

Figure A.26: Façade design pattern after GoF (p. 187)
Design of the Façade Pattern Service:

Figure A.27: Façade PS
Table A.9: Interface of the generic layer of the Façade PS

<table>
<thead>
<tr>
<th>Application-dependent Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Template&gt; FaçadeOperation</td>
</tr>
<tr>
<td>&lt;Template&gt; SubsystemOperation</td>
</tr>
</tbody>
</table>

Figure A.28: Runtime interactions of the Façade PS
A.10 Factory Method Pattern Service

Factory Method Design Pattern:

Figure A.29: Factory Method design pattern after GoF (p. 108)
Design of the Factory Method Pattern Service:

Figure A.30: Factory Method PS
Table A.10: Interface of the generic layer of the Factory Method PS

<table>
<thead>
<tr>
<th>Application-dependent Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Template&gt; AnOperation</td>
</tr>
<tr>
<td>&lt;Template&gt; FactoryMethod</td>
</tr>
</tbody>
</table>

Figure A.31: Runtime interactions of the Factory Method PS
A.11 Flyweight Pattern Service

Flyweight Design Pattern:

Figure A.32: Flyweight design pattern after GoF (p. 198)
Design of the Flyweight Pattern Service:

Figure A.33: Flyweight PS
Table A.11: Interface of the generic layer of the Flyweight PS

<table>
<thead>
<tr>
<th>Application-independent Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetFlyweight</td>
</tr>
</tbody>
</table>

Figure A.34: Runtime interactions of the Flyweight PS
A.12 Interpreter Pattern Service

Interpreter Design Pattern:

Figure A.35: Interpreter design pattern after GoF (p. 245)
Design of the Interpreter Pattern Service:

![Diagram of Interpreter Pattern Service]

Figure A.36: Interpreter PS
Table A.12: Interface of the generic layer of the Interpreter PS

<table>
<thead>
<tr>
<th>Application-independent Interface</th>
<th>Application-dependent Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddChild</td>
<td>&lt;Template&gt; Interpret</td>
</tr>
<tr>
<td>RemoveChild</td>
<td>################################</td>
</tr>
<tr>
<td>GetChild</td>
<td>################################</td>
</tr>
</tbody>
</table>

Figure A.37: Runtime interactions of the Interpreter PS - Selected Expression is NonTerminal
Figure A.38: Runtime interactions of the Interpreter PS - Selected Expression is Terminal
A.13  Iterator Pattern Service

Iterator Design Pattern:

![Diagram of Iterator Design Pattern]

Figure A.39: Iterator design pattern after GoF (p. 259)
Design of the Iterator Pattern Service:

Figure A.40: Iterator PS
Table A.13: Interface of the generic layer of the Iterator PS

<table>
<thead>
<tr>
<th>Application-independent Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
</tr>
<tr>
<td>Next</td>
</tr>
<tr>
<td>IsDone</td>
</tr>
<tr>
<td>CurrentItem</td>
</tr>
</tbody>
</table>

Figure A.41: Runtime interactions of the Iterator PS
A.14 Mediator Pattern Service

Mediator Design Pattern:

Figure A.42: Mediator design pattern after GoF (p. 276)
Design of the Mediator Pattern Service:

Figure A.43: Mediator PS
Table A.14: Interface of the generic layer of the Mediator PS

<table>
<thead>
<tr>
<th>Application-dependent Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Template&gt; Mediate</td>
</tr>
</tbody>
</table>

Figure A.44: Runtime interactions of the Mediator PS
A.15 Memento Pattern Service

Memento Design Pattern:

Figure A.45: Memento design pattern after GoF (p. 285)
Design of the Memento Pattern Service:

Figure A.46: Memento PS
Table A.15: Interface of the generic layer of the Memento PS

<table>
<thead>
<tr>
<th>Application-dependent Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Template&gt; CreateMemento</td>
</tr>
<tr>
<td>&lt;Template&gt; SetMemento</td>
</tr>
</tbody>
</table>

Figure A.47: Runtime interactions of the Memento PS - CreateMemento
Figure A.48: Runtime interactions of the Memento PS - SetMemento
A.16 Observer Pattern Service

Observer Design Pattern:

Figure A.49: Observer design pattern after GoF (p. 294)
Design of the Observer Pattern Service:

Figure A.50: Observer PS
Table A.16: Interface of the generic layer of the Observer PS

<table>
<thead>
<tr>
<th>Application-independent Interface</th>
<th>Application-dependent Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attach</td>
<td>&lt;Template&gt; GetState</td>
</tr>
<tr>
<td>Detach</td>
<td>################################</td>
</tr>
<tr>
<td>Notify</td>
<td>################################</td>
</tr>
</tbody>
</table>

Figure A.51: Runtime interactions of the Observer PS
A.17 Prototype Pattern Service

Prototype Design Pattern:

![Prototype Design Pattern Diagram]

Figure A.52: Prototype design pattern after GoF (p. 119)
Design of the Prototype Pattern Service:

Figure A.53: Prototype PS
Table A.17: Interface of the generic layer of the Prototype PS

<table>
<thead>
<tr>
<th>Application-independent Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clone</td>
</tr>
</tbody>
</table>

Figure A.54: Runtime interactions of the Prototype PS
A.18 Proxy Pattern Service

Proxy Design Pattern:

Figure A.55: Proxy design pattern after GoF (p. 209)
Design of the Proxy Pattern Service:

Figure A.56: Proxy PS
Table A.18: Interface of the generic layer of the Proxy PS

<table>
<thead>
<tr>
<th>Application-dependent Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Template&gt; Request</td>
</tr>
<tr>
<td>&lt;Template&gt; RealSubjectRequest</td>
</tr>
</tbody>
</table>

Figure A.57: Runtime interactions of the Proxy PS
A.19 Singleton Pattern Service

Singleton Design Pattern:

Figure A.58: Singleton design pattern after GoF (p. 127)
Design of the Singleton Pattern Service:

Figure A.59: Singleton PS
Table A.19: Interface of the generic layer of the Singleton PS

<table>
<thead>
<tr>
<th>Application-independent Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>SingleInstance</td>
</tr>
</tbody>
</table>

Figure A.60: Runtime interactions of the Singleton PS
A.20 State Pattern Service

State Design Pattern:

![State Design Pattern Diagram]

Figure A.61: State design pattern after GoF (p. 306)
Design of the State Pattern Service:

Figure A.62: State PS
Table A.20: Interface of the generic layer of the State PS

<table>
<thead>
<tr>
<th>Application-dependent Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Template&gt; Handle</td>
</tr>
<tr>
<td>&lt;Template&gt; ChangeState</td>
</tr>
</tbody>
</table>

Figure A.63: Runtime interactions of the State PS
A.21 Strategy Pattern Service

Strategy Design Pattern:

![Strategy Design Pattern Diagram]

Figure A.64: Strategy design pattern after GoF (p. 316)
Design of the Strategy Pattern Service:

Figure A.65: Strategy PS
Table A.21: Interface of the generic layer of the Strategy PS

<table>
<thead>
<tr>
<th>Application-dependent Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Template&gt; AlgorithmInterface</td>
</tr>
</tbody>
</table>

Figure A.66: Runtime interactions of the Strategy PS
A.22 Template Method Pattern Service

Template Method Design Pattern:

Figure A.67: Template Method design pattern after GoF (p. 327)
Design of the Template Method Pattern Service:

Figure A.68: Template Method PS
Table A.22: Interface of the generic layer of the Template Method PS

<table>
<thead>
<tr>
<th>Application-dependent Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>⟨Template⟩ TemplateMethod</td>
</tr>
<tr>
<td>⟨Template⟩ PrimitiveOperation</td>
</tr>
</tbody>
</table>

Figure A.69: Runtime interactions of the Template Method PS
A.23 Visitor Pattern Service

Visitor Design Pattern:

![Visitor Design Pattern Diagram]

Figure A.70: Visitor design pattern after GoF (p. 334)
Design of the Visitor Pattern Service:

Figure A.71: Visitor PS
Table A.23: Interface of the generic layer of the Visitor PS

<table>
<thead>
<tr>
<th>Application-dependent Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Template&gt; Accept</td>
</tr>
<tr>
<td>&lt;Template&gt; VisitConcreteElement</td>
</tr>
</tbody>
</table>

Figure A.72: Runtime interactions of the Visitor PS
Curriculum Vitae

Candidate’s Full Name: Eltaher Mohamed El-Shanta

Universities Attended:

1. Sebha University, Sebha, Libya, 1990, Bachelor of Computer Science.

Publications:


Conference Publications: