Pertinence of Design Patterns in Object-Oriented Software Development

Analysis Based on an Application Designed for the Setup of an Electronic Amplifier

Jean Baltus & Nicolas Gilson

Mémoire présenté en vue de l’obtention du grade de Maître en Informatique

Année académique 2001–2002
Abstract

Design patterns are proven design solutions to recurring problems within particular contexts. They capture in a compact form a wealth of experience about the design of object-oriented software.

Power Config is an ambitious “pattern-oriented” software project designed for the setup of an electronic amplifier and based on Java technology. It makes extensive use of GoF design patterns—the core software patterns—which gave birth to the growing patterns community. This thesis documents the design patterns that are part of the subsystem responsible for the control of the amplifier. It shows how to use design patterns in the context of a “real-world” software application and emphasizes their tremendous benefits. It offers an unprecedented step-by-step method to create reusable GUI Swing components and analyzes the impact of design patterns on the quality of Swing architecture. Finally, it examines limits and flaws of design patterns in order to prevent developers from “misusing” them.

Keywords:

Design Patterns, Design Patterns Benefits, Design Patterns Drawbacks, Object-Oriented, Software Development, Software Architecture, Design Phase, Framework, Java, Swing Components.
Acknowledgements

We are very grateful for the advice and support of our advisor, Vincent Englebert. His comments and insights have greatly improved and clarified this work. We also thank him for having offered us the opportunity to live a wonderful working experience abroad.

A person we would like to thank personally is Didier Burton. He was the first to teach us the joy of learning design patterns. His vast knowledge of computer science and his enthusiasm are both great inspirations to us. Not only has he provided us with many insights but he also facilitated our integration during our internship. So, thank you Didier, for your support, time, and mentorship.

Our sincere appreciation goes to all our teachers who, since the beginning of our life, have contributed to our education.

Everyone who provided feedback to this thesis one way or another also deserves our gratitude.

Our last word is for Johan Dahl and Kristen Nygaard who regrettfully departed this world recently within two months of each other. They invented the first object-oriented programming language, Simula I. Their work has led to a fundamental change in how software systems are designed and programmed today.
# Table of Contents

**List of Figures** viii

**Introduction** 1

1 **The Power Config Software** 5
   1.1 Overview of the Power G-16 System 5
   1.1.1 What is an Amplifier 6
   1.1.2 Power G-16 System Features 7
   1.2 Overview of Power Design 9
   1.3 Overview of Power Config 10
   1.4 The Software Development Team 12
   1.5 Summary 13

2 **Object-Oriented Programming** 15
   2.1 What is an Object? 16
      2.1.1 Conceptual Level: an Object is Any Concept in the Problem Domain 16
      2.1.2 Specification Level: an Object is a Set of Responsibilities 17
      2.1.3 Implementation Level: an Object is Data and Operations 18
      2.1.4 An Object Definition 19
   2.2 Object-Oriented Concepts Explained 20
      2.2.1 Visibility = Accessibility = Access Control 20
      2.2.2 Reusing the Implementation (Composition, Aggregation, Inheritance) 21
      2.2.3 Inheritance = Is-a = Is-like-a 21
      2.2.4 Encapsulation = Hiding Anything 22
      2.2.5 Polymorphism = Dynamic Binding = Late Binding = Run-time Binding 22
   2.3 How to Think in Objects 23
      2.3.1 How to Design Objects? 23
      2.3.2 Separate Interface and Implementation 24
### 2.3.3 Separate Things That Change from Things That Stay the Same

2.3.4 Low Coupling - High Cohesion (Also Known as the Orthogonality Principle)

2.4 Summary

### 3 Object-Oriented Software Architecture

3.1 High-Level Design Vs Low-Level Design

3.2 The Need for an Architecture

3.3 Elaboration Phase

3.4 Fundamental Principles

3.4.1 Patterns

3.4.2 Enabling Techniques

3.4.3 Object-Oriented Principles

3.5 Architecture of Power Config

3.5.1 The Layers Architectural Pattern

3.5.2 The Model-View-Controller Architectural Pattern

3.6 Summary

### 4 Design Patterns Improve the Architecture

4.1 Presentation of Design Patterns

4.1.1 What is a Pattern?

4.1.2 Pattern Systems Vs Pattern Languages

4.1.3 Pattern Tools

4.1.4 Framework

4.2 Non-functional Properties of Software Architecture

4.3 Improvements of the Architecture

4.3.1 Mediator Design Pattern

4.3.2 Facade Design Pattern

4.3.3 Singleton Design Pattern

4.3.4 Proxy Design Pattern

4.3.5 Decorator Design Pattern

4.3.6 Adapter Design Pattern

4.4 Summary

### 5 Elements of Reusable GUI Components

5.1 The Model-View-Controller Architecture

5.1.1 The Observer Pattern

5.1.2 The Model-View-Controller

5.1.3 The Modified MVC Pattern from Swing

5.2 Actors and Responsibilities of a Swing Component
# TABLE OF CONTENTS

5.3 The Pluggable Look-And-Feel Architecture ...................................... 105
  5.3.1 The Abstract Factory Pattern .............................................. 105
  5.3.2 A More Flexible Factory: The UI Manager .............................. 107

5.4 The Swing Component Development Process ................................. 109
  5.4.1 General Specification of the Component ................................. 110
  5.4.2 Implementing the Model ................................................ 110
  5.4.3 Graphical Specification of the Component .............................. 115
  5.4.4 Implementing the Wrapper .............................................. 116
  5.4.5 Implementing the UI Delegate ......................................... 121

5.5 Summary ............................................................................. 128

6 Limits and Flaws of Design Patterns ........................................... 129
  6.1 Chaos .............................................................................. 130
  6.2 Implementation Overhead ..................................................... 131
  6.3 Abuse .............................................................................. 132
  6.4 Poor Traceability ............................................................... 133
  6.5 Lost Control .................................................................... 134
  6.6 Duplication .................................................................... 135
  6.7 Summary ........................................................................ 136

Conclusion .................................................................................. 137

Glossary ..................................................................................... 141

A Key Features of Design Patterns ............................................. 149

B The UML Notation .................................................................. 159

C Selectable List: Additional Code ............................................. 165

References ............................................................................... 183
List of Figures

1.1 An Amplifier Generates a New Output Signal .................. 7
1.2 Typical Deployment of the Power G-16 System ............... 9
1.3 Power Design ................................................. 10
1.4 Power Config User Interface ................................ 11
1.5 The Routing and Output Gain Control Pane ................. 12

2.1 Hairdryer Class Diagram from a Conceptual Perspective .... 17
2.2 Hairdryer Class Diagram from a Specification Perspective .. 18
2.3 Hairdryer Class Diagram from an Implementation Perspective 18

3.1 The Unified Software Development Process .................... 30
3.2 Communication Model with the Power G-16 system .......... 37
3.3 Standard Software Industry Layered Architecture ............ 39
3.4 Coarse-Grained Architecture of Power Config ............... 41
3.5 Polling System Sequence Diagram ........................... 43
3.6 Model-View-Controller Mechanism in the Coarse-Grained Architecture 47

4.1 Design Patterns Catalog ...................................... 54
4.2 Power Config Interface ...................................... 60
4.3 Coarse-Grained Architecture of Power Config ............ 61
4.4 Polling System Framework Class Diagram .................... 63
4.5 Simplified Version of Power Config Architecture ........... 71
4.6 Duplicated Version of Power Config Architecture ........... 72
4.7 Reusable Version of Power Config Architecture ............ 73
4.8 Adjust Gain Sequence Diagram .............................. 74
4.9 “Horizontal” Partitions of Power Config Architecture ...... 75
4.10 Power Config Architecture with Facades .................... 78
4.11 Amplifier Surrogate Subsystem with Proxy .................. 80
4.12 Proxy Nested Structure ..................................... 83
4.13 Proxy Class Diagram ........................................ 84
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.14</td>
<td>Surrogate Amplifier Architecture with Decorated Proxy</td>
<td>89</td>
</tr>
<tr>
<td>4.15</td>
<td>G16Proxy to Amplifier Interface Adapter Class Diagram</td>
<td>90</td>
</tr>
<tr>
<td>4.16</td>
<td>Amplifier Surrogate Subsystem with Adapter</td>
<td>93</td>
</tr>
<tr>
<td>5.1</td>
<td>The Observer Design Pattern</td>
<td>97</td>
</tr>
<tr>
<td>5.2</td>
<td>The Model-View-Controller Design Pattern</td>
<td>98</td>
</tr>
<tr>
<td>5.3</td>
<td>Modified Model-View-Controller Architecture of Swing Components</td>
<td>100</td>
</tr>
<tr>
<td>5.4</td>
<td>The <code>PropertyChangeSupport</code> Mechanism</td>
<td>101</td>
</tr>
<tr>
<td>5.5</td>
<td>Actors Relationships of the Swing Component <code>NewComp</code></td>
<td>103</td>
</tr>
<tr>
<td>5.6</td>
<td>The Abstract Factory Design Pattern</td>
<td>106</td>
</tr>
<tr>
<td>5.7</td>
<td>A Flexible Factory: The UI Manager</td>
<td>108</td>
</tr>
<tr>
<td>5.8</td>
<td>Simplified View of a Swing Component</td>
<td>109</td>
</tr>
<tr>
<td>5.9</td>
<td>Graphical Specification of a <code>SelectableList</code></td>
<td>116</td>
</tr>
<tr>
<td>5.10</td>
<td>Initialization of a UI Delegate When Calling <code>new SelectableList()</code></td>
<td>117</td>
</tr>
<tr>
<td>5.11</td>
<td>The StopperSlider and the Multislider Graphical Components</td>
<td>128</td>
</tr>
<tr>
<td>A.1</td>
<td>Standard, Simplified View of the Abstract Factory Pattern</td>
<td>150</td>
</tr>
<tr>
<td>A.2</td>
<td>Standard, Simplified View of the Adapter Pattern</td>
<td>151</td>
</tr>
<tr>
<td>A.3</td>
<td>Standard, Simplified View of the Decorator Pattern</td>
<td>152</td>
</tr>
<tr>
<td>A.4</td>
<td>Standard, Simplified View of the Facade Pattern</td>
<td>153</td>
</tr>
<tr>
<td>A.5</td>
<td>Standard, Simplified View of the Mediator Pattern</td>
<td>154</td>
</tr>
<tr>
<td>A.6</td>
<td>Standard, Simplified View of the Proxy Pattern</td>
<td>155</td>
</tr>
<tr>
<td>A.7</td>
<td>Standard, Simplified View of the Observer Pattern</td>
<td>156</td>
</tr>
<tr>
<td>A.8</td>
<td>Standard, Simplified View of the Singleton Pattern</td>
<td>157</td>
</tr>
<tr>
<td>A.9</td>
<td>Standard, Simplified View of the Template Method Pattern</td>
<td>158</td>
</tr>
<tr>
<td>B.1</td>
<td>The Amplifiers Package Includes the <code>Amplifier</code> and <code>Classification</code> Classes</td>
<td>160</td>
</tr>
<tr>
<td>B.2</td>
<td>A <code>MachineComposite</code> Object Contains either <code>Machine</code> Objects or Other Composites. The <code>Customer</code> Class Depends on the <code>LikeMyStuff</code> Class Without Instantiating it</td>
<td>161</td>
</tr>
<tr>
<td>B.3</td>
<td>You Can Indicate an Interface with either an <code>&lt;&lt;interface&gt;&gt;</code> Stereotype or a Lollipop</td>
<td>162</td>
</tr>
<tr>
<td>B.4</td>
<td>A Sequence Diagram Shows a Succession of Method Calls</td>
<td>163</td>
</tr>
</tbody>
</table>
Software architects focus on building systems in terms of the technology in vogue. It is undeniable that fashion constrains the software architect’s profession. However, a common mistake is to believe that technology plays the most important role in software development. On the contrary, a software project is designed first and foremost for the human beings who exploit the technology to achieve their ends. It is thus crucial to place the emphasis on improving the ability of developers to exploit the advantages that technology provides rather than on the technology itself.

Object-oriented technology has reached a great level of maturity and is now widely-accepted for software development. It is not really difficult to learn how to program in an object-oriented language such as Java. In a few days, anyone accustomed to procedural programming languages is up to speed with object-oriented programming. On the other hand, it takes a while to learn how to exploit the advantages that object-oriented programming provides. Once people have acquired a thorough understanding of concepts like objects, interfaces, classes, inheritance, composition, and delegation, the real challenge lies in applying them to resolve software issues such as reusability, changeability, reliability, or testability. Those non-functional properties are more than necessary for dealing with the growing complexity of software applications.

The subject of this thesis is to analyze the pertinence of design patterns in object-oriented software development. A design pattern is a proven design solution—in terms of communicating objects and classes—to a particular design problem. It records and encourages the reuse of “best object-oriented software practices” that are significant for successful software development.

We concentrate on the design patterns presented in the book “Design Patterns, Elements of Reusable Object-Oriented Software” by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (frequently referred to as the Gang of Four or just GoF). “Design Patterns” is the seminal text, its style is academic-oriented and it presents a catalog of twenty-three patterns that are referenced by basically any other design patterns. We also rely on many other existing books to deepen our understanding of those patterns and to strengthen our ability to apply them. Obviously, our study would not be complete if it was only supported by books. We considered it was also essential to analyze
concrete applications of design patterns. Hence it was for us the opportunity to share the experience we have gained with design patterns during our work experience abroad.

Indeed, last year, we spent five months within a software team. Despite the constant pressure of deadlines, the day-to-day objective of the team was to provide not only the stated functional properties but also non-functional properties. They knew that those properties have an equal impact on the quality of an application and its architecture as do the system’s functional properties. Design patterns are the ultimate tool through which they achieve their purpose. The moment we arrived, they introduced us to design patterns and emphasized their tremendous benefits.

This study allows us to present design patterns in an original way. Indeed, there is a plethora of books that describe in detail patterns and apply them in practice but, with few exceptions, none of these books introduce patterns in the large field of concrete software applications. Hence we jumped at the opportunity to present design patterns applied in a “real-world” software application.

During this internship, we worked on the Power Config software which is an application designed for the setup of an electronic amplifier. So as not to lose readers, Chapter 1 describes what an amplifier is and explains at length this “pattern-oriented” software application which controls the Power G-16 system.

Even if the solutions encompassed in design patterns do not require unusual language features or amazing programming tricks, developers need to acquire a thorough understanding of object-oriented concepts before diving into the design patterns world. In order to avoid ambiguities about the terminology
used in this thesis, Chapter 2 provides an introduction to the key concepts of
object-oriented programming as well as advice on how to think with objects.

Our contribution to the software team has included the creation of graphical
user interface (GUI) components for the specific needs of Power Config and
development of the sensitive subsystem that controls the electronic amplifier:
the amplifier surrogate subsystem.

The amplifier surrogate subsystem was designed both to reflect in real-time
the state of an electronic amplifier connected through the serial port and to
ensure the control of this amplifier. It was integrated within the whole Power
Config application and it communicates with other subsystems dedicated to the
interaction with users.

Design patterns address the sort of problems typically encountered after
the overall structure of a software system has been specified. Thus, before
studying the pertinence of design patterns in the amplifier surrogate subsystem,
we present the coarse-grained architecture of Power Config in Chapter 3. This
architecture is an adaptation of the standard layered architecture for Enterprise
Application. We analyze the benefits and drawbacks of such architecture and we
describe thoroughly the responsibilities of each layer. This vertical partitioning
lays the necessary foundation for the understanding of the amplifier surrogate
subsystem architecture.

In the first part of Chapter 4, we present the state of the art concerning
design patterns and we provide an accurate design patterns definition. We also
defend our position with regard to “hot” topics such as pattern tools and pat-
tern languages. Then, we describe the differences between design patterns and
frameworks and we present the polling system framework that we implemented
within the context of the amplifier surrogate subsystem. This framework takes
advantage of the Template Method design pattern to promote the reusability
of the code. In the second part of this chapter, we demonstrate how we im-
proved the architecture of the amplifier surrogate subsystem with the help of
six distinct design patterns. We apply the Mediator, Facade, Singleton, Proxy,
Decorator, and Adapter design patterns to transform the adapted standard layer
architecture for Enterprise Application into the final architecture.

As professional software like Power Config cannot rely on a poorly designed
user interface, it was of a major importance for the software team to create
a proprietary look for their graphical components. The application being for
the most part implemented in Java, they achieved this through the creation of
reusable Swing components that could be customized to their needs. Unfortu-
nately, this is a time-consuming approach and documents that cover the subject
are nowhere to be found. Having experienced the creation of several widgets,
we developed our own step-by-step method to create such custom-made graphi-
cal components. Since Swing architecture relies heavily on some specific design
patterns (the Observer and Abstract Factory design patterns), the creation of
reusable GUI components also requires a thorough understanding of those. In
Chapter 5, we study the design patterns related to Swing and their diverse
forms, and we present the method illustrated with the creation of a concrete widget.

Finally, we could not write a document about the pertinence of design patterns in object-oriented software development without examining their limits and flaws. Chapter 6 is dedicated to the analysis of design patterns drawbacks. We attempt to establish taxonomy of the potential problems of design patterns and give some indication as to what could be done to solve them.

All chapters conclude with a summary section that contains some reflections on the chapter. We also joined a glossary with the definitions of the important terms presented in this thesis.
Chapter 1

The Power Config Software

The Power Config software is a tool that allows you to configure hardware devices used in business music systems. The Config software is intended to be used on a PC that is connected to the system electronics by an RS-232\(^1\) serial cable or an USB\(^2\) cable.

More specifically, Power Config is used for the setup, the verification, and the commissioning of an amplifier called Power G-16 system. The hardware settings can be manually or automatically configured. The automatic mode actually configures the electronics based on a special file created by another application: Power Design.

In this chapter, we give a quick overview of the Power G-16 system and the Power Design software. We then present the Power Config software which is the application we have worked on during our internship abroad. Our contribution has included system architecture definition and module implementation which are the subjects of the next chapters.

1.1 Overview of the Power G-16 System

The Power G-16 system is an integrated four-channel 400-Watt power amplifier for 70/100V music applications. The following paragraphs explain what is an amplifier, how it works, its related terminology and then describe the particular features of an G-16 system.

\(^1\)Standard for transmitting serial data by wire. RS-232 connections are used to attach personal computers to modems, and other hardware devices.

\(^2\)Universal Serial Bus: a serial bus standard created in 1997 by Intel. USB connections are used to attach personal computers to their keyboards, printers, and other hardware devices.
1.1.1 What is an Amplifier

This section is by no means a technical description of amplifiers. In order to give you the context in which we worked during our internship, we thought it would be a good idea to give a little introduction on amplifiers. We have simplified the concepts as much as possible, and hope that it will not offend specialists.

We often use the word “amplifier” to describe an audio component that takes a weak audio signal and increases it to generate a signal that is powerful enough to drive speakers. While this statement is correct, the concept can be applied to any electrical and electronic device, not just audio. Generally speaking, an amplifier can be defined as an electronic component that accepts a low-level signal and recreates the signal with more power.

The goal of a good amplifier is to reproduce the original signal with as little distortion as possible. Distortion appears when the amplifier is not able to reproduce the exact same signal as the original one. It therefore recreates an approximation of the input signal. When we want to measure the quality of an amplifier, we often refer to the percentage of distortion.

In the audio domain, we use amplifiers to generate signals that are powerful enough to move the cone of speakers back and forth. The vibrations created by cone speakers will move air particles and reproduce the wave that we originally recorded\(^3\). A wave represents a fluctuation in air pressure. Those fluctuations are picked up by our ears and then translated into electrical signals that our brain can process. If you want to have very loud signals, you generally need several amplifiers working in a chain. Hence, every amplifier has its limitation, which we measure in watts.

Watt is the measurement unit of electrical power\(^4\). The power (P) of an electric system is equal to its voltage (V) multiplied by the current (I): \(P = VI\). Voltage (also known as the electromotive force) is a term used to designate the electrical pressure or the force that causes current to flow and is measured in volts. Current represents the flow of electrons through a conductor and is measured in amps.

Another measure that is often used for audio speakers and amplifiers is the load (R) or resistance to the current flow, which we measure in ohms (Ω). The load of an electronic system is directly related to its voltage and current. One of the most basic and well-known equation in electronics, Ohm’s Law, states that a voltage of 1 volt across a resistance of 1 ohm will cause a current of 1 amp to flow. As a result, Ohm’s equation can be written as: \(V/R = I\), or \(R = V/I\).

\(^3\)When you record music through a microphone, the sound produced by musical instruments and our voice is actually a wave in the air. That wave makes the diaphragm of the microphone move and those movements are translated into electrical signals that are recorded on tapes, cds or any storage devices. Sound is therefore represented as a varying electric current.

\(^4\)The power is actually the amount of energy (in joules) converted by a component in a unit of time, usually a second. A watt can therefore be interpreted as a joule per second.
1.1 Overview of the Power G-16 System

The maximum power that an G-16 system is able to deliver is 400 Watts. This power is shared by four output channels, thus allowing an G-16 to control four sets of amplifiers in different zones. An G-16 system can also handle four different input channels and redirect them to an appropriate output zone. In order to deliver such power, the G-16 system uses internally a chain of several power amplifiers and electronic components to insure a precise control of hi-fidelity output signals.

Each internal amplifier generates a completely new output signal based on the input signal it receives. You can understand these signals as two separate circuits as shown on Figure 1.1.

![Figure 1.1: An Amplifier Generates a New Output Signal](image)

The output circuit is generated by the amplifier’s power supply\(^5\) which smoothes out the current to generate an absolutely even, uninterrupted signal [Har00]. The power generated by this output circuit is moving the speaker cone or feeding another amplifier in the chain. The input circuit is the electrical audio signal that directly comes from an audio device (CD player, microphone...) connected to an input channel of the G-16 or from another amplifier in the chain. Its load is modifying the output circuit. It applies a varying resistance to the output circuit to recreate the voltage fluctuations of the original audio signal [Har00].

We will not go deeper into the explanations as it is not the core subject of our thesis. If you open an G-16 system, you will find a complex set of electronic components, each responsible for a particular task. Indeed the G-16 system does much more than a regular amplifier. All its features are described in the following section.

1.1.2 Power G-16 System Features

The Power G-16 system is a professional amplifier system for businesses such as restaurants, pubs, and hotels. As a single component, the chassis provides all of

\(^5\)The power supply draws energy from a battery or power outlet. If the amplifier is powered by household alternating current, where the flow of charge changes directions, the power supply will convert it into direct current, where the charge always flows in the same direction.
the processing and control features required for one-to-four zone business music applications. Among the various features that an G-16 system can handle, here are the most important ones:

- Auto volume: An G-16 system can be connected to several sense microphones in order to dynamically adjust the music level in each output zone. This allows people to always hear the music, regardless of the background noise.

- Scheduling: We can program the G-16 system for automated on/off and source selections according to time of day or day of week. Indeed, an G-16 controls everything related to the music environment, it automatically turns on, turns off, changes sources... thus allowing businesses to concentrate on their task.

- Source leveling: The Power G-16 system automatically compensates for variations in source input levels. Source leveling is often known as “automatic gain control” (AGC). This allows for the same level of music to be heard whether the source level provided is varying or not.

- Signal routing: An G-16 meets the demands of most four-zone systems, allowing for any input signal to be routed to any of the four amplifier outputs.

- Various equalization possibilities: Provides easy adjustment of tonal balance in each zone.

- Remote control connections: Four remote wall plate inputs can be connected to an G-16 for zone volume control and source select.

- RS-232 and USB connections: Provides easy interfacing to your PC. This allows the connection of an G-16 system to the Power Config software in order to configure the various parameters it offers.

- The integrated power amplifiers feature a patented power-sharing technology which allocates power to each output for a total maximum of 400 watts. For example, if you have a two-zone system that requires 5 watts in Zone 1 and 395 watts in Zone 2, the Power G-16 system distributes the power based on those needs.

- The G-16 also includes an easy-to-replace memory module, which holds the current hardware configuration and a design file created by the Power Design software described in the next section.
1.2 Overview of Power Design

The Power Design software is intended to be used by sales engineers and their customers to create an audio solution. The user interface of Power Design allows customers to describe the layout of their facility, the desired audio functionalities, and their requirements. Based on that information, Power Design creates a system solution. The user can adjust and revise the proposed system by adding, removing, moving, or adjusting components.

Once a solution is selected, the software can create several documents which can then be used to create a system proposal:
- The bill of materials lists all system components, accessories, and quantities.
- The system block diagram shows the type and quantity of system electronics and their interconnections, and settings.
- The speaker layout displays the type, quantity, and location of the loudspeakers which will be used.

All this information can be saved in a design file which will be used by the Power Config software in order to install and configure the electronic components properly.
1.3 Overview of Power Config

The Power Config software is used for the setup of the G-16 system amplifier and to verify the performance of systems designed with the Power Design software. Using the above mentioned design file, Power Config generates the necessary hardware settings for an G-16 system. The user may also perform a series of system verification measurements to ensure the quality of the equipment. The user interface of Power Config is shown on Figure 1.4.

In order to configure a hardware device, the computer must first be physically connected to the hardware device with a serial cable. When Power Config starts, it will check whether the connection is properly installed and then launch a graphical user interface as shown on Figure 1.4. The system overview pane allows the user to select the amplifier to setup. Indeed, the software is able to configure several amplifiers that are interconnected.
1.3 Overview of Power Config

The hardware pane displays a software front panel of the hardware device selected in the system overview pane. Reading from left to right, this diagram shows the functions and signal paths from input sources to output zones. All functions internal to the connected amplifier appear on a gray background. The user can choose a particular function to control by clicking on the corresponding button on the hardware pane diagram. When you select a function, all controls for that function appear in the control pane. In Figure 1.4, for instance, the InGain function of the first input is selected. This function lets us adjust the input gain of the first input channel of the amplifier. The slider which allows users to control this behavior is shown on the bottom right control pane. Next to it, note the meter displaying the current signal level of music on the input circuit.

As examples, Figure 1.5 illustrates two other G-16 functions that can be set with the Config software. Routing is done by the source assign function as shown on the left diagram. The speaker icon shows the source channel that is currently routed to the selected output channel (channel 2 in this case). The sources displayed in the right list shows the input channels which can be assigned to this output channel. The right part of Figure 1.5 shows the output gain function. The user can drag the thumb of the slider to adjust the gain of the
output circuit of the amplifier. The accompanying meter displays the signal level of the currently selected output zone. The user can also mute the signal or trigger a check line procedure. This procedure will test the hardware and return some information such as the power and load of the selected circuit.

Figure 1.5: The Routing and Output Gain Control Pane

The program also permits the automatic configuration of the G-16 system with a design file generated by the Power Design software. In order to do this, we just click on the open file button located in the top left of the interface and then select the appropriate design file.

When the configuration of the hardware is done, we have to save the new settings in the hardware. For this, we have to click on the "Flash Hardware Configuration" button (third icon on the top left in Figure 1.4) which sends hardware configuration and a design file to the G-16 hardware memory. The configuration determines the initial state of the G-16 hardware when you turn it on.

1.4 The Software Development Team

So far, we presented everything from a user point of view. We will now roughly describe how the software has been developed and see why it really needs flexibility.

In the previous sections, you have noticed that the two applications and the hardware are tightly related. Indeed, Power Config reuses design files produced by the Power Design software and stores information into the G-16 system.
1.5 Summary

In order to interact with the Config software, the G-16 system contains embedded chips driven by pieces of internal software. The internal software of the G-16 offers an interface to dialog through its RS-232 or USB port. Power Config and Design are both made in Java. Unfortunately, Java does not offer an USB API\(^6\) yet, thus forcing developers to call native C code to interact with the hardware device. That part, the C code, is developed by the hardware team since it directly relies on the features offered by the hardware.

When the project started a few months ago, all the requirements had not been defined yet. The challenge for the software team in this project was therefore to deal with changes coming from the hardware team (interface to the G-16 system) and from the marketing team who was defining the graphical user interface. This being said, we directly feel that it is not an easy task and that we need a flexible approach to software in order to cope with change.

We also need to notice that within such an organization, it is difficult to measure the cost of such changes since we are our own customer. Indeed, no contract has been subscribed between the hardware and the software team, nor between the marketing and the software team. For that reason, everybody pushes the software team further, adding and removing functionalities as the project moves on. Note that this not a rare situation in the industry. Requirements change. We have to deal with it.

The software architect of the team knew that. To make sure the project was not deemed to failure, he decided to develop the application using design patterns. A design pattern can be described as “a way to pursue an intent that uses classes and their methods in an object-oriented language”\(^7\) [Met02]. Design patterns can help to minimize consequences of shifting requirements on a system. This is the core subject of our thesis. Having designed and implemented some modules for Power Config based on design patterns, we wanted to share our experience about them and show how they helped us to come up with better designs and implementations. The rest of this paper will discuss, in depth, how design patterns can help developers in their everyday life, their benefits, and their limits. However, this assumes that you already know an object-oriented language, and preferably Java since all our examples will be in that language. As a result, we give a summary of all the concepts that belong to object-orientation in the next chapter.

1.5 Summary

Power Config is the software we worked on during our internship abroad. It is intended to be used on a PC that is connected to an electronic amplifier system by a serial cable. More precisely, this software is used for the setup

\(^6\)Application Programming Interface: the set of services that an operating system or a programming language makes available to programs that run under it.

\(^7\)This definition will be refined in chapters 3 and 4.
and the verification of amplifiers designed for businesses. It also interacts with another software that helps to automatically configure amplifier settings based on the location of speakers in a facility. In this software, our task has included object-oriented programming in Java and object-oriented analysis and design (including design patterns).
Chapter 2

Object-Oriented Programming

...It’s not difficult to learn how to program in an OO language. The problem is that it takes a while to learn to exploit the advantages that object languages provide...

Martin Fowler [Fow99b]

In this chapter, we introduce the key concepts of object-oriented programming (OOP) and their terminology. However, we assume that readers have some background in the domain and already know how to program in an object-oriented language, preferably Java.

The goal of this chapter is therefore to emphasize the fact that object-orientation is not a simple appendix to procedural programming and must be seen differently. Indeed, we will show that the thought processes of object-oriented programmers are quite different than those of procedural programmers.

In the following sections, we discuss:

• What objects are—in other words, how you can see objects.
• How object-oriented concepts make developers’ life easier.
• How programmers can think with objects (advices to program in OO).

To write this chapter, we have gathered a lot of articles and books that cover the subject. We mainly inspired from “UML Distilled” [Fow99b], “Thinking In Java” [Eck00], and “Design Pattern Explained” [ST02].
2.1 What is an Object?

Assembly languages forced programmers to think in computer structures (bits, bytes...). With imperative languages, programmers thought in algorithms and data structures instead, which is slightly better. Now OOP goes one step further: software developers think in terms of objects, objects of the domain under study. Therefore, object-oriented languages provide a higher abstraction that helps developers to solve their problems more easily.

Expressing the problem in terms of the problem itself has some advantages:

- Programmers can stay away from the computer structure and focus on a given problem.
- Analysts do not get lost too early in the details of implementation in the software development process.
- Communication between programmers and end-users is improved.

Unfortunately, we do not get this for free: we need to understand the concept of object and to use objects in a properly manner to take advantage of all those benefits.

We can see objects from three perspectives: the conceptual perspective, the specification perspective, and the implementation perspective. The first standpoint is more similar to the common representation that humans have of objects; everything can be an object, and objects of the same type have behaviors and characteristics in common. The second perspective is more software oriented: each object is taken as an entity responsible for itself and the tasks it can accomplish. All those responsibilities are defined through interfaces. The implementation view describes objects as a collection of data and operations in a programming language. At this level, we talk about method calls, fields... The following sections explain those perspectives in-depth.

2.1.1 Conceptual Level: an Object is Any Concept in the Problem Domain

From a conceptual perspective, everything can be considered as an object. An object can be a duck, an amplifier, a book, or anything. In theory, you can always represent such an object with some code at the implementation level. In this abstraction, software developers can represent any concept in the problem domain with objects. Here is a short example; if we use a class diagram (following the standardized Unified Modeling Language\(^1\) (UML)) to represent hairdryers, we can draw it as follows:

\(^1\)You will find a short tutorial of the UML in Appendix B of this thesis.
2.1 What is an Object?

Figure 2.1: Hairdryer Class Diagram from a Conceptual Perspective

To understand this diagram, we can cite Eckel: “Aristotle was probably the first to begin a careful study of the concept of type; he spoke of the class of fishes and the class of birds. The idea that all objects, while being unique, are also part of a class of objects that have characteristics and behaviors in common was used directly in the first object-oriented language, Simula-67, with its fundamental keyword “class” that introduces a new type into a program” [Eck00].

Here, Hairdryer is really the name of a type of object, of a class of object. The diagram therefore represents a class; this is the reason why it is called a class diagram. Each object is an instance of a class. In this view, the attributes (words in the middle box) mean that Hairdryers have brands and serial numbers, nothing else. The operations (in the lower box) mean that you can turn them on and off, and select cool or hot air. For instance, we can have an object called hairDryPlusOne that belongs to the class of Hairdryer. That object could have the brand Phillipe or Sonia; its serial number could be SN-2001-7898.

2.1.2 Specification Level: an Object is a Set of Responsibilities

If we had to summarize this section in one word, it would be the word “responsibility”. In OOP each object is responsible for itself and the tasks it can accomplish. If an object depends on a task that is beyond its responsibility, it must ask another object to accomplish this task. Using object-oriented parlance, we would say the first object has to send a message to the other.

We can represent hairdryers by using the same class diagram as the one used at the conceptual level. However, the meaning of the diagram would be different, as Figure 2.2 illustrates.

Here the text contained in the lower box represents the interface of the object. An interface defines the responsibilities of a type of object, the requests that you can make to that type of object. Put in other words, the interface defines what messages you can send to the object. The attributes indicate that
2.1.3 Implementation Level: an Object is Data and Operations

Here objects can be seen as a set of fields and methods (OOP jargon for functions, also known as procedures). We can now expand the diagram to represent the implementation of hairdryers:

When you have to implement the interface of an object, you have to create a method definition for each message you can send to that type of object. Those methods actually define the way objects behave when we ask them to accomplish a task. We have added four new methods to access and modify the brand and serial number. Those methods are necessary to respect the principle that an...
object is responsible for itself. An object should never directly work on internal data of another object; it is beyond the scope of its responsibilities.

Each object can have fields to store data; those fields represent the state of the object. A field is often another object; it could also be an integer, character if the language you use does not consider those as objects. We have added two fields (on and cool) in order to store the state of a hairdryer. The on field could store the power state of an hairdryer and would be used by the on() and off() methods. The cool field could store the temperature state and would be used by the cool() and hot() methods. These considerations are really details of implementation; this is the reason why these were irrelevant in the two previous perspectives.

At this level, when we talk about the “class of an object”, class means a programming class and not the general sense that Aristotle used.

2.1.4 An Object Definition

Unfortunately, the line is not always sharp between the three views that we have described. Concepts such as responsibility could also be part of the conceptual view for instance. Our intent was not to make a perfect distinction between those three views (although we tried to clarify things as much as possible) but it was to present OOP at all levels of abstraction. The most important thing to remember is that the common representation of the 80’s, which is the implementation view, is incomplete. An object is much more that a set of data and operations and if we want to reduce an object to a set of something, we can think of it as a set of responsibilities.

If we mix all perspectives, we could summarize the description of objects as follows:

Everything is an object: “Think of an object as a fancy variable; it stores data, but you can make requests to that object, asking it to perform operations on itself. In theory, you can take any concept in the problem you are trying to solve (hairdryers, cats, building, customer) and represent it as an object in your program” [Eck00].

An object has an identity, a state, and behaviors: Each object has a name, a state—which consists of data (made up of other objects)—and behaviors that are implemented as methods definitions.

Objects send messages to each other: Objects communicate through messages, i.e. method calls.

An object is responsible for itself and the tasks it can accomplish: If an object cannot perform a task, it will ask the appropriate object to execute that task.
Every object has a type: “Each object is an instance of a class, in which class is synonymous with type” [Eck00]. Objects of the same type share characteristics and behaviors in common.

2.2 Object-Oriented Concepts Explained

Now that we have examined the definition of an object, we can explore all the concepts that belong to OOP.

2.2.1 Visibility = Accessibility = Access Control

One big principle of OOP is that objects never directly manipulate data that belongs to other objects. This means that we need a mechanism to prevent objects to do so. This is what we call the accessibility mechanism. There are three main types of accessibility for a class member, whether it is a field or a method:

Public: Any class can access the member.

Protected: Only this class, its derived classes and anyone else in the same package can access it.

Private: Only this class can access it.

Since we are mainly interested in Java, Java explicitly uses those keywords (public, protected, and private) to determine the level of accessibility for each member of a class. Java has also a default access; we like to call it the package access, therefore we can refer to the four P’s. Package members can be accessed by any class of the current package.

One advantage of the visibility mechanism is that it allows developers to build classes that expose only what is necessary to the client. It really helps the client to focus on what he needs to know and not on details he should not be aware of. For instance, hiding information is very interesting in the development of libraries; it allows the creator of those libraries to change the implementation at will, provided that behaviors do not change. The concept of hiding information has been recognized as one of the big advantage of OOP: by preventing users to have access to everything, you reduce the risk of bugs. In Java, the mastery of those four P’s (public, protected, package and private) is a powerful tool for programmers.
2.2 Object-Oriented Concepts Explained

2.2.2 Reusing the Implementation (Composition, Aggregation, Inheritance)

There are several ways to reuse the implementation of existing classes. One of the simplest ways is by embedding objects references in a new container class we want to define. This is called composition or aggregation depending on which restriction you want to apply to those objects. Those two mechanisms are often referred as the “has-a” relationship between objects.

Composition means that objects which compose the container class are part of it. Without its part objects, the composite class does not exist by itself. On the other hand, the parts are usually expected to die with the composite in case of deletion. A good example would be a class `Table` that holds references to four objects `Leg`.

Aggregation means that the container class aggregates some other objects in order to have all the features needed but still exists by itself. A good example would be `CarRentalLocation` having `Cars`: `Cars` are not part of the `CarRentalLocation` but we can still say the `CarRentalLocation` has them.

Another means to reuse the implementation is by inheritance, this is the subject of the next section.

2.2.3 Inheritance = Is-a = Is-like-a

A class can inherit state and behaviors from another class called its super class or base class. For instance, we could create a class `SeniorStudent` that inherits from another class `RegularStudent`. Other ways to say this would be, the `SeniorStudent` derives from, specializes, or is a subclass of `RegularStudent`. `SeniorStudent`s could have more responsibilities than `RegularStudent`s: for instance, writing their thesis. This involves adding a `writeThesis()` method to the `SeniorStudent` class. `SeniorStudent`s could also redefine some behavior that a regular student already possess by overriding methods. Overriding a method means writing a new definition for this method. This consists of either changing the type of some parameters, adding parameters, removing parameters, or any combination of those three.

Inheritance is often described as the “is-a” relationship in textbooks: `SeniorStudent` is-a `RegularStudent`. We prefer to adopt the “is-like-a” terminology established by Eckel [Eck00]. Actually a `SeniorStudent` is like a `RegularStudent` but it has more responsibilities and it behaves differently for some operations.

The “is-a” relationship is valid when we talk about implementing an interface
without adding any new methods. Let’s say we have an interface called Vehicle with two methods declarations: drive() and stop(). We can then define classes like Car, Truck, Bus that implement the interface Vehicle, i.e. provide a definition for the methods drive() and stop(). In this case, a Car is-a Vehicle, a Bus is-a Vehicle and a Truck is-a Vehicle.

2.2.4 Encapsulation = Hiding Anything

In the eighties, too many people considered encapsulation as a way to hide data. Again, this conception is correct but incomplete. Quoting Shalloway, “In general, encapsulation means any kind of hiding: data hiding, class hiding, implementation hiding” [ST02].

Hiding data is possible through the accessibility mechanism described above. You can also hide information by programming to interfaces. An object has a public interface that other objects can use to communicate with it. The object can maintain private information and methods that can be changed at any time without affecting the other objects that depend on it. This is exactly like in real life: you do not need to understand the gear mechanism of your bike to use it; you just have to know how to shift gears. An interface can also have several implementations and therefore hide the concrete class that the client uses. This ability makes polymorphism possible.

2.2.5 Polymorphism = Dynamic Binding = Late Binding = Run-time Binding

Polymorphism comes from the Greek terms “poly” (many) and “morph” (forms). It means the ability to take many forms. In OOP, it is common to refer to objects with their interface or their abstract class. However, what we are actually referring to are specific instances of classes implementing their interface or abstract classes. Thus, when we ask the objects to do something through a method call, we get different behavior, depending upon the concrete object implemented we have. This ability allows many types (derived from the same base type) to be treated as if they were one type. Therefore, a single piece of code can work equally on different types.
2.3 How to Think in Objects

2.3.1 How to Design Objects?

When you have to write a new module\(^2\), you may not know where to start. A good way to solve this problem is to ask yourself a few questions:

- What is the intent of the module? It is important to always keep that in mind while designing your module.
- How can I organize my module into classes? In other words, what are the objects and their classes?
- What are the relationships between those classes? What messages can you send to your objects? This will help you to discover the interfaces of your objects.

Once you have roughly answered those questions, you can refine your answers by using a technique similar to the Class-Responsibility-Collaboration (CRC) cards. This technique was created by Cunningham in the late eighties. He represented classes on 3 by 5 cards. And rather than to indicate attributes and methods on the cards, he wrote responsibilities and collaborator classes.

So for each class you have discovered, you have to define three things:

The name of the class: “It is important that this name capture the essence of what the class does, so that it makes sense at a glance” [Eck00].

The responsibilities of the class, what it should do: It is really a high level description of the purpose of the class in a few sentences. If the responsibilities are straightforward, you can also directly write member functions for the class. Again, choosing appropriate names for those members is really important, especially for communication among developers.

The collaborations of the class: “Which other classes does it interact with? “Interact” is an intentionally broad term; it could mean aggregation or simply that some other object exists that will perform services for an object of the class. Collaborations should also consider the audience for this class” [Eck00].

By doing this, you will realize that:

- some classes need to be split because they were having too many unrelated responsibilities.

---

\(^2\)Here, we suppose that we have been assigned to build a few classes in order to fulfill some requirements. Therefore, module is used to refer to the set of classes we have to build.
• new classes are needed to collaborate with an existing one or to mediate
  between existing ones.

• some classes might be useless because their role is redundant with other
classes.

• some classes might be merged to form a single class because they were
tightly coupled.

After that process, you will have a first clear picture of your module and you
can start to implement your classes. Nevertheless, it is not the end of it and
the structure of your module is likely to change as you are implementing your
classes. You will start to add features and to restructure your module to support
more easily additional extensions. You will realize that some components could
be more generic, more reusable: you will change their responsibilities. However,
not all of your classes must be reusable. It is only when you will change your
objects to adapt to new situations that you will realize their real responsibilities
and the need to be more generic.

2.3.2 Separate Interface and Implementation

A good practice that has been around for a long time is to program to interfaces.
This consists of always declaring variables as their interface type, thus prevent-
ing the client to accidentally change the internal structure of objects through
methods they should not be using.

The other advantage of this principle has already been announced when we
have described the concept of encapsulation. By programming to interfaces, you
have the guarantee that changes to non-exposed (private, protected or package)
members will not affect the client code. The application of this mechanism helps
to narrow the scope of changes in the code. This leads us to the next principle.

2.3.3 Separate Things That Change from Things That
Stay the Same

One source of big frustration for programmers is that requirements change. This
is a fact we have to deal with. A common solution to that problem is to isolate
change. In order to do that, you need to consider what you want to be able to
change without redesign, then encapsulate those elements in classes. Not only
does this help developers to manage modifications but it also turns out that it
is usually simpler to understand.

“Often, the most difficult part of developing an elegant and cheap-to-maintain
design is in discovering the “vector of change. This means finding the most
important thing that changes in your system, or put another way, discovering where
2.3 How to Think in Objects

your greatest cost is. Once you discover the vector of change, you have the focal point around which to structure your design” [Eck01].

2.3.4 Low Coupling - High Cohesion (Also Known as the Orthogonality Principle)

The “low coupling-high cohesion” principle consists of decoupling objects that are unrelated and of designing objects that are self-contained. This has the huge benefit to eliminate side effects (often bugs) when adding features to a program. Components of different subsystems must be independent of each other. Programmers should be able to add features to the graphical interface without touching the code of the database or the business logic behind it.

Self-contained objects are easier to write, easier to test, easier to reuse, easier to modify and easier to design. If you have high cohesion in your objects, with well-defined responsibilities, you encourage people to reuse them through aggregation, composition, or inheritance. If such an object has a bug, it is less likely to propagate through the whole application; it will be a lot easier to find the bug and to correct it. Any changes you make will be restricted to that object.

This principle leads to higher productivity for developers. If the objects are highly cohesive, there is less chance of overlapping work. The concepts behind such objects are easier to grasp, which eases the communication between developers and avoid ambiguities. Therefore, all those benefits contribute to reduce risks for a manager.

Some people are more radical and promote the application of the “law of Demeter for methods”: this law states that any method of an object should invoke only methods belonging to: itself, any parameters that were passed in to the method, any objects it created, and, finally, any directly held component objects [HT00]. The basic idea is to avoid invoking methods of a member object that is returned by another method. When you do this, you make structural assumptions about the container object that may be likely to change.

The application of Demeter’s law will make your code more adaptable and robust, but at one cost: you end up creating a lot of wrappers methods that simply forward the request on to its components. Decoupling objects and modules in general has also a performance drawback; it is good to know if you are designing a subsystem for real-time computing or time sensitive modules. In that case, you will have to find a good trade-off between modularity and efficiency.

This chapter was neither designed to be an introductory course to Java nor to OO; its goal was to describe the context in which this thesis is written. Design patterns cannot be studied separately from the OO world. Therefore, we have presented object-oriented concepts and their terminology to avoid misun-
derstandings and ambiguities. In the next chapter, we will present software architecture principles and will see why a good architecture is important for an application.

2.4 Summary

Object-oriented concepts have been evolving since the eighties, along with the development of design patterns. Objects can be seen from three perspectives: the conceptual perspective, the specification perspective, and the implementation perspective. From a conceptual perspective, an object is a concept in the domain under study. The second perspective is more software oriented: each object is taken as an entity responsible for itself and the tasks it can accomplish. The implementation view describes objects as a collection of data and operations in a programming language.

In the second section of this chapter, we joined a reminder of all the concepts that belong to OO. More precisely, we examined the following mechanisms: visibility, aggregation, composition, inheritance, polymorphism, and encapsulation.

Finally, we gave some advices on how to program with object-oriented languages. We explained how to encapsulate change, how to separate the interface from the implementation, and how to build objects that are not tightly coupled.
Chapter 3

Object-Oriented Software Architecture

... The software architecture is the bridge between the system requirements and implementation...

Christine Hofmeister, Robert Nord, and Dilip Soni [HNS99]

During software development, the workflow goes through five major phases: requirements analysis, design, implementation, test, and deployment. Novice developers often underestimate the importance of the design part and rush—after having identified the user requirements—into the actual coding of the application without building a stable architecture. This is a serious mistake because the design phase results in an architecture that will be the skeleton of the system.

This chapter introduces the large field of the software architecture and more precisely that of the object-oriented software architecture. However, we do not plan to provide a complete survey of software architecture nor specific software development method.

The purpose of this thesis is to study the pertinence of design patterns in object-oriented software development. Design patterns address the sort of problems typically encountered after the overall structure of a software system has been specified. Hence, in this document, we concentrate especially on the improvements that design patterns bring to the architecture in the detailed design stage. Nevertheless, before examining the impact of design patterns, it is essential to present briefly the key aspects of the object-oriented software architecture and, especially, of the coarse-grained architecture. This chapter is thus dedicated to this task.
First, we clarify the distinction between the high-level and low-level design and we emphasize the fact that the analysis, the deployment, the implementation, and even the test processes have an impact on the architecture. This chapter is also the opportunity for us to stress the importance of the design phase. Afterward, we introduce patterns for software development and we present a classification of these patterns. We also study the enabling techniques that help creating successful software designs. Lastly, having regard to these enabling techniques, we describe what the characteristics of an object-oriented design are.

In the second part of this chapter, we concentrate on the architecture of Power Config software. We study thoroughly the Layers architectural pattern and its variants. Then, we describe the architecture of Power Config which is an adaptation of the standard layered architecture for Enterprise Application. We analyze the benefits and drawbacks of such architecture and we describe thoroughly the responsibilities of each layer. Finally, we analyze how the Model-View-Controller architectural pattern is applied to Power Config architecture.

### 3.1 High-Level Design Vs Low-Level Design

We have drawn our inspiration from the book “Pattern-Oriented Software Architecture” [BMR et al. 1996] in order to write this section.

Buschmann et al. [BMR et al. 1996] give a definition of the software architecture we think relevant: “A software architecture is a description of the subsystems and components of a software system and the relationships between them. Subsystems and components are typically specified in different views to show the relevant functional and non-functional properties of a software system. The software architecture of a system is an artifact. It is the result of the software design activity”.

The software design activity is commonly divided into the high-level design and the low-level design. The high-level design results in the structural subdivision of the system. It specifies the fundamental structure of the application. The low-level design results in more detailed planning like definition of interface, data structures...

However, some people [HNS et al. 1999] disagree with this decomposition and think of detailed design as implementation mechanisms, not as the software architecture. It is difficult to draw the line between design activities and implementation activities. As far as we are concerned, we consider detailed design to be part of the software design activity. Indeed, the issue of showing how the functional and non-functional requirements are fulfilled by the system is addressed in the software design activity. This proves that detailed design is part of the software design architecture because, as we demonstrate in the next chapter, a lot of low-level decisions have somehow an impact on how the non-functional requirements,
3.2 The Need for an Architecture

such as changeability or reusability, are fulfilled.

3.2 The Need for an Architecture

The design phase is a very large one—from the requirement analysis to the implementation—that deserves more and more attention from developers. Its importance grows with the emergence of increasingly complex software systems. However, the importance of the design is not clear to everybody. We try to clarify this in the next paragraphs. Most ideas of this section come from “The Unified Software Development Process” [JBR99].

Boehm’s study [Boe81], published in 1981, shows that 30% of the errors encountered during the life cycle of a software application are design errors. 18% of these errors are corrected after the release! Furthermore, the cost of correction of such errors is hugely superior to the cost of implementation errors.

According to Hofmeister et al. [HNS99], a solid, well-thought architecture helps to manage complexity, resolve trade-offs among conflicting requirements, and, in general, bring quality software to market in a more timely fashion.

The software architecture gives us an overview of the whole system. This overview helps us to control the development of the system. It guides developers through the entire conception of the software application. It describes the most important elements that compose the skeleton of the software such as subsystems, dependencies, interfaces, collaborations and it also describes how these elements work together to fulfill the systems requirements.

The architecture is also a powerful means of communication between all the members of the team and it helps the laymen understand the structure behind the system.

In addition to increasing the understandability of the system and organizing its development, the software architecture is essential to ensure the modifiability of the system. It happens that requirements change during the software development process or that we expect the software to evolve later. For instance, a software application that is intended to run on a non-distributed environment can evolve into a distributed application. Only a system with an architecture designed for change can reach the new expectations. Without such an architecture, each change that will occur to the system will be a new step towards the incomprehensibility of the structure of this system.

The cost of maintenance is a major issue for software engineering. People spend too much time trying to understand the structure of the system. It is obvious that a software application with a well-documented architecture will be easier to maintain than an application without architecture or with a poorly defined architecture.
Finally, developers have discovered the interest of the creation of domain specific components that can be reused in other systems. In order to achieve this goal, we need a design that integrates these reusable components.

Now that you are sensitive about the critical role of architecture in software development, we will try in the next sections to draw the main ideas that help designers to create a successful architecture.

3.3 Elaboration Phase

The book “The Unified Software Development Process” [JBR99] proposes an original presentation of the software development process. This approach is illustrated in Figure 3.1. According to Booch et al. [JBR99], each cycle of the development process results in a product ready for delivery and consists of four phases: inception, elaboration, construction, and transition. Each phase terminates in a milestone (models, documents...). Moreover, within each phase developers may break the work into iterations.

![Figure 3.1: The Unified Software Development Process](image)

The workflows—requirements, analysis, design, implementation, and test—are carried out in each phase to a certain extent. We observe that iterations in the elaboration phase go through all the five workflows.

Those authors also specify that the elaboration phase results in the architecture baseline. The architecture baseline is "analogous to a skeleton covered with..."
3.4 Fundamental Principles

skin but with very little muscle (the software) between the bone and the skin—just enough muscle to allow the skeleton to make basic movements” [JBR99]. In fact, in their specific approach, the architecture baseline is an aggregation of early versions of models coming from the five workflows: the use-case model, the analysis model, the design model, the deployment model, the implementation model, and the test model.

Furthermore, the architecture baseline is represented by more than models artifacts. It also includes an architecture description which is derived from versions of the architecturally significant models.

The value of this approach is that it shows us that the architecture is not only the result of the design process but also the result of the requirements, analysis, implementation, and test processes. Hence it is not surprising if, for instance, the test process has an impact on the architecture.

3.4 Fundamental Principles

The design is an activity that needs a lot of creativity. A miraculous process that can be applied to produce the best architecture does not exist. It could be interesting to take advantage of the accumulated experience of other software designers. This experience added to designers’ creativity should be a solid base to produce robust, well-thought architecture.

Designers can rely upon patterns that capture solutions that have evolved over time. Moreover, those patterns are built on enabling techniques that bring together the accumulated experience of software developers. The object-oriented principles are in accordance with those fundamental principles.

The following sections present patterns, enabling techniques, and principles of object-oriented architecture.

3.4.1 Patterns

When developers design the architecture of software applications, they are likely to encounter design problems. They need a lot of experience to address those issues but, by definition, novice designers are inexperienced in structuring system. Fortunately, recurring design problems have been tackled thousand of times and we can find solutions that help designers to effectively and elegantly solve these problems. These solutions are known as patterns.

Patterns cover various ranges of scale and abstraction. We distinguish four kinds of patterns [BMR+96]: analysis patterns, architectural patterns, design patterns, and idioms. The first category captures object modeling expertise
from different domains. The second one helps in structuring software system into subsystems. And, the third one supports the refinement of subsystems and components. According to Fowler [Fow97], “the difference between analysis patterns and architectural patterns is not always easy to understand. The difficulty lies in the fact that both help in producing models that can be deliberately very similar”. However, the purpose of the analysis is to understand the problem while the purpose of the design is to solve the problem. Analysis patterns are out of the scope of this thesis and will not be addressed. Another difficulty resides in the fact that a specific design patterns can be both relevant to low-level design and high-level design. That is, for instance, the case of the Model-View-Controller pattern which is useful to describe interactions between architecture elements and to describe the structure and interactions within architecture elements. According to the context, that kind of pattern will be considered as either an architectural pattern or a design pattern. Finally, the last category of patterns, the idioms, helps in implementing particular design aspects in a specific programming language. Buschmann et al. [BMR+96] write: “In contrast to design patterns, which address general structure principles, idioms describe how to solve implementation-specific problems in a programming language”. Once again, it is not easy to draw a clear line between design patterns and idioms because idioms are very useful to implement practically design patterns. In fact, idioms overlap the field of programming guidelines.

Although it is true that patterns are not solely dedicated to object technology, most of them and particularly design patterns use object-oriented features. For this reason, we consider that patterns are intended to be implemented in an object-oriented programming language. As architectural elements, the design patterns perfectly suited the object-oriented concepts.

The value in patterns is that they are built on fundamental principles, significant for successful software development. They complement ideally existing analysis and design methods. During years, techniques have been developed to realize these principles. Patterns are based on these techniques called “enabling techniques” in the book of Buschmann [BMR+96]. In the next section, we go over the most pertinent enabling techniques listed in this book.

### 3.4.2 Enabling Techniques

Designing a large software system is something very though that needs a lot of experience and creativity. There is no silver bullet. We do not think rules that you can blindly follow to produce the ideal architecture exist. Designers are often confronted with choices that will have deep impacts on the structure of the system. They should be very careful when they make decisions because, once the architecture has reached a certain level of maturity, restructuring the system becomes difficult.

---

1Fowler’s book “Analysis Pattern” [Fow97] tackles this kind of patterns.
A software application is always developed in a changing environment. The technology, the requirements, and the design can evolve. Change, by definition, is not defined in advance; we cannot foresee how the system will evolve. The most important part of the design is dealing with change. Therefore, the main focus of this section will be to provide some strategies to make the design process efficiently responsive to change. A good architecture is an architecture designed for change.

The enabling techniques for software architecture listed in the book “Pattern-Oriented Software Architecture” [BMR96] bring together the accumulated experience of software designers. They are built on the common sense and help create successful software architecture. These principles are carried unanimously but regrettably not always followed. You can study them as stand-alone principles but there are often closely related. Consequently, the transgression of one principle implies frequently that others principles are also transgressed. However, it happens that some principles are contradictory. Designers should deal with it. We summarize the enabling techniques we think are most significant:

**Abstraction:** Abstraction is a fundamental concept to cope with complexity. It allows designers to forget about details of implementation. Thanks to this concept, designers can create simplified overview of the whole system. The architecture is an abstract solution to a problem. It should not deal with details of implementation. For instance, during the design phase, we are not concerned about the fact that a module encapsulating the access to a database will be written using JDBC or another technology. Our only concern is that this module will encapsulate the access to a database. In fact, we are not even interested in the type of the database. It can be a brand new Oracle database or a legacy system; from a design point of view, the module will be the same.

As a general rule, we will always try to keep the highest level of abstraction in order to obtain a system intrinsically more flexible to face up to changing requirements. However, considerations such as performance of the application can influence the designers but it is strongly recommended, as much as possible, to avoid that kind of question during the design phase. Indeed, that might make things unnecessarily tricky. The system might become more difficult to understand and therefore more difficult to modify and maintain.

**Encapsulation:** A system is composed of abstractions. Designers have the responsibility to group together elements that belong to a same abstraction. The pertinence of the encapsulation has a huge impact on the changeability of the system. Indeed, when a particular aspect of the system changes, it is more than likely that the new requirement affects only elements that

---

2You might notice that these concepts are tightly related to the ones presented in Chapter 2. Indeed, object-oriented principles are based on these enabling techniques.
3Java Data Base Connectivity.
4Real-time systems need a short respond delay. In that case, the performance is part of the design.
belong to the same abstraction. Hence the change is limited to those
elements.

**Information Hiding:** This principle is without any doubt the most recognized
in software engineering. The client of a component does not need to know
about the details of implementation of this component. He just needs to
know the bare minimum in order to use it properly. This is the best way
to reduce coupling between components.

Again, considerations like performance can influence designers. They
might choose to set the details of implementation visible in order to op-
timize the use of this resource. It is extremely unadvisable because all
changes to the details of this component may affect the client.

**Modularization:** Modularization is closely related to the principle of encaps-
sulation. Modularization is concerned with the definition and documenta-
tion of boundaries within a program. These boundaries disconnect mod-
ules which encapsulate specific responsibilities or functionalities of an ap-
lication. This strict decomposition allows designers to handle system
complexity.

**Coupling and Cohesion:** We have already tackled the question of coupling.
The goal is to get weak coupling between modules. Modules are then
weakly interrelated and consequently easier to change.

Modules can be coupled by shared data. They can be coupled by direct
monitoring: a module activates another module. Ideally, modules are
coupled by parameters in order to achieve the weakest coupling. This is
the principle of object-oriented architectures.

As far as cohesion is concerned, this is the measure of the degree of con-
nectivity between the functions and elements of a module. All entities of a
module should contribute to a common goal. That fosters the reusability
of the modules.

Here are the known forms of cohesion from the worst to the most desir-
able: arbitrary cohesion, logical cohesion (functionalities externally simi-
lar), temporal cohesion, procedural cohesion (based on a procedure), com-
municational cohesion, sequential cohesion (the output of a module is the
input of another module), and functional cohesion. The functional cohe-
sion ensures that the elements of a module work together to provide some
well-bounded behavior. Obviously, object-oriented decomposition is based
on this last and most desirable form of cohesion.

**Separation of Interface and Implementation:** This principle is closely rel-
ated to “Information Hiding”. Each component consists of an interface
and an implementation. The component client should only know about
the specification of the interface that defines the functionalities provided
by the component and specifies how to use it. The client does not have
access to the internal data structure or additional functionalities not spec-
ified in the interface. The goal is to define a stable interface as soon as
possible, so that the client component can rely on it and the developers
can implement the functionalities independently of its use by other com-
ponents. They do not have to worry about potential side effects on the
other components. Furthermore, it fosters changeability because it becomes easy to change the component behavior in cases where the change does not necessitate a change to its interface.

**Single Point of Reference:** According to this principle, each item of a software system should be declared and defined only once. This principle seems obvious because it allows to avoid unnecessary work and, above all, it ensures the consistency of the whole system. Nevertheless, the “Single Point of Reference” is sometimes incompatible with other principles. Developers should reach a compromise in order to use these principles properly. However, you should always keep in mind that duplication of information can give developers big trouble.

**Divide and Conquer:** The Roman practice of divide and conquer is well-known. That strategy was used by many throughout history before and after Caesar and is still heavily used in software architecture.

### 3.4.3 Object-Oriented Principles

During the elaboration of this chapter, we have faced difficulties in establishing the exact influence of the object-oriented technology on the software architecture. When a software development process results in an implementation using the object-oriented programming, is it enough to speak about object-oriented design? It is food for thought.

One of our concerns is that what we call object-oriented design is used to describe things at the detailed-design level. Indeed, concepts such as interface, inheritance, delegation, or polymorphism are usually exploited to refine and extend the coarse-grained architecture of a software system.

However, object-oriented architectures are a lot more than that. In fact, an object-oriented architecture follows somehow the enabling techniques proposed in the previous section. Habra [Hab00] describes the object-oriented architecture: “An object-oriented architecture is an architecture where the meaningful decomposition of the system is based on weak coupling between modules, strong cohesion within modules, and notion of service”. We have already introduced the concepts of coupling and cohesion. We know the most desirable cohesion is the functional cohesion and the most desirable coupling is the coupling by parameters. The object-oriented world copes with the problem of cohesion by using the functional cohesion around data structures and it uses the coupling by parameters. In this way, modules are considered as a set of functionalities around a data structure.

Habra also introduces the problem of the relationship between modules and defines six types of relationships: procedural call, activation, importation/exportation, composition, generalization, and service. This last relationship includes all the others and consequently is the most abstract. The underlying idea of this relationship is that a module uses the service of another module and this
first module can work properly only if the second module works properly. It allows developers to factorize their work. Furthermore, it is in harmony with the fundamental principle of abstraction. Object-oriented architecture uses solely this kind of relationship.

3.5 Architecture of Power Config

This section provides an overview of the Power Config architecture. We present the structural subdivisions of this architecture. We do not want to overfeed the reader with detailed design decisions. This part will be addressed in the next chapter when we speak about improvements the design patterns bring to the design activities.

We describe the main ideas that have driven the designer through the whole design activities. We underline the benefits and flaws of that kind of architecture. The purpose is not to describe with precision each specific components of the architecture of Power Config. On the contrary, we stay as abstract as possible so that the tackled concepts can be applied in other contexts. However, we keep using the name Power Config in the whole document.

We remind you of the key aspects of Power Config that are significant for the understanding of the architecture described below. Readers are referred to Chapter 1 for a detailed presentation.

Power Config is an interface-centric application that is designed to control an external device. The device in question is an amplifier: the Power G-16 system. Figure 3.2 identifies the physical components that are part of the Power Config application and the collaborations between them. The application allows users to change the state of the amplifier through the user interface. In addition, the GUI reflects in real-time the current state of the amplifier.

As opposed to asynchronous I/O device which provokes interruptions when it is ready to handle new data or when its state changes, the amplifier is a passive I/O device. This means that we have to send requests to the amplifier at regular intervals if we want to be aware of the changes that occur to the state of the device. Otherwise, we will never be notified of changes within the device. Hence our application polls regularly the needed information in order to give a graphical feedback to the user.

As you can see above, every $x$ msec the application asks the amplifier the data that are somehow likely to change.

The state of the device can be updated in three different ways:

- Users, through the GUI of Power Config, can send requests to modify
3.5 Architecture of Power Config

Figure 3.2: Communication Model with the Power G-16 system

specific parameters (set input gain to -15dB, mute...).

- Users can “physically” change the state by using the Power G-16 wall plate (set output gain to -20dB, select a different source...).

- The amplifier is responsible for updating its “logical” state in order to reflect its actual internal situation (there is no signal on input channel 1, the signal level is clipping...).

The designer has “vertically” partitioned the architecture into layers following the standard software industry layered architecture. Then, the Model-View-Controller architectural pattern is used to ensure the automatic update of the views as new information become available in their respective models.

We present the Layers and Model-View-Controller architectural patterns in order to tackle the architecture of Power Config.

3.5.1 The Layers Architectural Pattern

The Layers architectural pattern described in the book of Buschmann et al. [BMR+96] has been used to define the architecture of Power Config. Here is the definition of this pattern given in the above mentioned book: “The Layers architectural pattern helps to structure applications that can be decomposed into
groups of subtasks in which each group of subtasks is at a particular level of abstraction”.

A layered architecture can be compared to a stack of subsystems where the top of the stack is the subsystem with the highest level of abstraction and the bottom one is the subsystem with the lowest level of abstraction.

Networking protocols are probably the best-known example of layered architectures. Their successes are undoubtedly due to the quality of the layered architecture that allows the reusability of individual layers in different contexts.

People who program in Java are very familiar with this pattern. Indeed, the JVM\(^5\) insulates code written in the Java programming language from proprietary platform by defining a platform-neutral byte-code. There are implementations of the JVM for different operating systems and processors. These specific JVMs are different implementations of the same layer. All have the same interface and can be switched depending on the platform. Code written in the Java Programming language is translated into the neutral byte-code and then interpreted by the JVM installed on the platform. This technique promotes the reusability of the source code.

Other known uses have been made of this pattern such as the layered architecture for Information System\(^6\) which, in other respects, is the skeleton of the architecture of Power Config.

Layered Architecture for Information System

Information systems from the business software domain often use the layered architecture [Fow97, Fow02]. In that case, the layers are commonly called tiers. The two-tier architecture is an old widespread division for interactive information system. The bottom layer is responsible for storing business-specific data into some form of persistence storage while the top layer includes many applications that work concurrently to fulfill different tasks. This architecture is common in client/server development. It concurrently allows the centralization of data and the meaningful presentation of this data in different places. However, the tight coupling of the user interfaces to the physical data involves a lack of evolvability and reusability. Furthermore, the storage mechanisms are often unable to give a true representation of the modeled concepts. Moreover, as the domain logic\(^7\) got more complex, it became difficult to embed the logic directly in the GUI.

In order to solve that problem, a new layer is added between the persistence layer and the application layer: the domain layer. This layer models

---

\(^5\)Java Virtual Machine.
\(^6\)The old fashion term “Information System” is now replaced with “Enterprise Application” [Fow02].
\(^7\)Business rules, validations, calculations…
the conceptual structure of the domain. “The domain logic, also referred to as business logic, involves calculations based on inputs and stored data, validation of any data that comes in from the presentation, and figuring out exactly what data source logic to dispatch depending on commands received from the presentation” [Fow02]. The three-tier architecture is more flexible than the two-tier because the physical structures and the applications are weakly coupled. It allows the modification of the physical structure without breaking the existing applications. Another advantage of this decomposition is that the domain layer can represent the semantic of the modeled domain better than the persistence layer. Come to that, the object-oriented technology is a very useful tool for implementing the domain tier. The objects that describe the domain tier are generic to the business domain and, therefore, are reusable.

The knowledge required to build the application layer increases strongly with the complexity of the domain layer. Indeed, the application tier mixes user interface and most of the application behavior. The application behavior is implemented within the user interface which causes complex interrelationships with the domain tier. A good practice to decrease the complexity of the graphical user interface is to split the application tier into the presentation tier and the application logic tier. This results in a four-tier architecture, as described in Figure 3.3. Then, the presentation tier is responsible for user interface only while the application logic tier is responsible for the accesses to the domain tier and any processing other than user interface processing. The main advantage of this division is that the presentation tier does not need to know about the domain tier. It can be developed totally independently of this one which strongly decreases its complexity and increases its reusability.

The Power Config application is intended to be deployed within a non-distributed environment. Nevertheless, by using this layered architecture, the designer takes advantage of it to promote the reusability of its components, to reduce the coupling between them, and to strengthen the cohesion within each of them. In addition, he partitions the architecture in a meaningful way. He also keeps the opportunity to deploy later Power Config in a distributed environment. However, the “horizontal” partitioning of the architecture that will be introduced in the next chapter makes the distribution of Power Config complex because it increases hugely the number of objects to wrap in order to use them with a middleware.
In the case of a client/server approach, the designer should choose where to place the domain tier. He can either place it on the client or introduce a new layer of processors, which is the domain server. A client-based domain tier has the advantage to simplify the system support but it implies a lot of data processing on the client side. On the other hand, a server-based domain tier is easier to control and update and it also allows the developer to control how data is accessed. In general, designers prefer this last choice.

Power Config

We have drawn our inspiration from both “Pattern-Oriented Software Architecture” [BMR+96] and “Analysis Patterns” [Fow97] for writing this section.

The strict use\(^8\) of the Layers pattern is very respectful of the fundamental principles described in Section §3.4.2. First of all, the notion of service is used to describe the relationship between layers in an abstract way. Each layer is responsible for invoking the services it requires from the lower layer. There are no further direct dependencies between layers. Moreover, the lower layer is unaware of the identity of its users. Consequently, the (one-way) coupling between layers is very weak which allows developers to change easily the details of a specific layer. However, this design for change is only possible if the interface and semantic of these layers remain stable. Obviously, this approach is coherent with the divide-and-conquer, encapsulation and modularization principles.

> “Each level of abstraction can be understood as a coherent whole without knowing much about the other layers. In addition to that, you can easily substitute layers with alternative implementations of the same basic services” [Fow02].

Although layering is one of the most common techniques that software designers use, the Layers pattern also imposes liabilities [BMR+96, Fow02]. We notice that we sometimes get cascading changes which is a pain for developers. It happens, for instance, that you want to add a field in the GUI that is not in the database and thus you have to add it to every layer. Moreover, a layered architecture is usually less efficient because at every layer things typically need to be transformed from one representation to another. If some services performed by lower layers perform excessive work not actually required by the higher layer, this has a negative impact on performance. Finally, it is very difficult to establish the correct granularity of layers. Indeed, a layered architecture with too few layers does not take advantage of reusability, changeability and portability resulting from this pattern. On the other hand, too many layers introduce unnecessary complexity.

The Layers architectural pattern is used when the system mixes low-level and high-level issues, where high-level operations rely on the lower-level ones. It is particularly the case of the Power Config application which handles, at the same time, low-level issues such as hardware communication, and high-

\(^8\)The Relaxed Layered System [BMR+96] is less restrictive about the relationship between layers which increases the coupling between layers.
level issues such as graphical representation of the information available in the hardware. Typically, the high-level operations consist of playing with graphical components to set new values in the amplifier. The application allows users to set up the amplifier by, for instance, modifying the volume gain or muting the volume. Users have also a graphical feedback in real-time of the amplifier state. They can, for instance, watch the signal level in order to check the quality of the sound produced by this amplifier.

For the moment, we focus on a simplified version of the Power Config application. We consider that this application is dedicated to set up one specific type of device (the Power G-16 system) and we forget features such as design file, scheduling\textsuperscript{9}... Later on, the architecture will evolve in order to handle new non-functional requirements. Moreover, we stay as abstract as possible and we avoid addressing issues such as performance at the architectural level.

By analogy with classical $x$-tiers systems, where distributed clients could execute requests through different levels of layers to database, we can build the architecture of Power Config. Indeed, the principles of this application are very similar. Actually, the architecture of Power Config is an adaptation of the standard layered architecture for Enterprise Application. The main difference lies in the fact that some parts of the system handle low-level issues such as communication with hardware rather than the classical communication with a database. Conceptually, the device can be considered like a persistence storage that handles requests. Hence the persistence layer of the standard layered architecture has been replaced with the hardware layer. The solely difference resides in the fact that the state of the device can be updated independently of any requests. Figure 3.4 represents this adapted layered architecture.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure3.4.png}
\caption{Coarse-Grained Architecture of Power Config}
\end{figure}

As you can see in this figure, a new layer between the domain and the persistence/hardware layer has been added to the standard layered architecture.

\textsuperscript{9}Chapter 1 presents in detail the Power Config application.
In fact, it is usual to add a database interface tier in layered architecture for information system in order to reduce the complexity of the domain classes. This tier is responsible for loading the domain tier with data from database and for updating the database when the domain changes. It decouples the domain layer from the persistence layer. The domain tier does not have to know if it handles relational database, legacy system, or even flat files. We have chosen to add that layer for three reasons. The first one is that the communication with the device is complex and we want to reduce the complexity of the domain tier. The second one is that we want to be able to reuse the upper layers in the case of a migration of Power Config to another platform. Finally, the third one is that we want our application to be compatible with future versions of the amplifier. In these two last cases, we just have to rewrite the code of the hardware interface layer. Higher layers will not be affected by the exchange. Once again, this design for change is only possible if the interface of this layer remains stable.

Power Config is intended to be used by one user at a time, so the connection of several clients at the same time is forbidden. Nevertheless, we need a mechanism that polls regularly information in order to update the GUI according to the amplifier state. Indeed, this passive I/O device is unable to send information to notify that its state has changed. It is also very important that this mechanism takes advantage of the domain layer—which models the conceptual structure of the domain—in order to handle meaningful information. A new type of client is thus designed. This new client is different from the first one because it does not handle graphical user interface, it just sends at regular intervals meaningful get requests through the domain layer. In fact, this client does not care about the answers. Its purpose is only to update the models in the domain tier in order to synchronize the application with the device. We have called this special client the polling system.

Figure 3.5 describes the interaction within the layers leaded by the polling system. At regular intervals, the polling system sends a set of requests to the domain layer. These requests are related to domain values that are likely to change. In this example, the requested value is computed from two domain atomic values. The domain layer owns some objects that capture the computing rules. This layer is also responsible for pulling these values from the hardware layer through the hardware interface layer. Once the domain layer has received these values, it computes the new value and it updates its model. Then, the domain layer automatically notifies the components that have previously subscribed to that kind of events. In this case, a controller is notified of this change and it acts accordingly to update the presentation layer.

With the introduction of this new client, we meet concurrency issues. It is interesting to observe that the GUI and the polling system are active ob-

---

10 A client is composed of the presentation and application logic layers.
11 The presentation layer is thus empty.
12 Readers are referred to Appendix B for a description of the UML notation.
13 Section §3.5.2, introducing the Model-View-Controller architectural pattern, describes this notification mechanism.
3.5 Architecture of Power Config

That means they are separate tasks and consequently they have their own thread. In fact, the application is solely driven by these two threads. All the other objects are passive and are executed in the thread of the calling active object. The two clients can be executed simultaneously which can imply synchronization troubles between the GUI, the models, and the device. For example, if a user sets the volume gain to -20 decibels and “simultaneously” the polling system gets the old value, the GUI might be updated with the wrong value. Fortunately, mechanisms in different programming languages address that kind of issue. However, the fundamental principles advise us to keep the highest level of abstraction in order to get a system intrinsically more flexible. So, at the moment, we forget these details of implementation.

In summary, the architecture of Power Config is divided into five layers with well-defined responsibilities:

**Hardware layer:** Beyond all the responsibilities in relation to the behavior of an amplifier, this layer is responsible for storing the meaningful values—that represent the current state of the amplifier—into some form of storage. The amplifier is also responsible for updating its “logical” state in order to reflect its actual internal situation. In order to get good performance, it is obvious that the resulting data schema inside the device can look quite different from the objects of the domain tier.

---

14 The Java programming language provides helpful utilities such as the static method `invokeLater` that allows to execute an object `Runnable` asynchronously on the AWT event-dispatching thread. We can deal with concurrency by “wrapping” all the methods call by the polling system on the domain layer in an object `Runnable` and then invoke this utility. Hence only the event-dispatching thread would deal with the domain layer. That is one solution; software people have come up with various one with different levels of complexity.

15 However, it happens often that developers should deal with concurrency at the architectural level. For example, when they want to handle transactions.
Hardware interface layer: The direct link between the domain tier and the passive I/O device can be the source of significant problems. It can complicate the domain classes excessively by giving them two independent responsibilities: modeling the conceptual structure of the domain and pulling/pushing data from/to the amplifier. The differences between the data schema of the domain layer and the data schema of the hardware layer make this problem critical. Consequently, the hardware interface layer is responsible for providing an interface to the domain tier in order to hide the complexity of the communication with the device. It handles the communication through the serial or USB port. It generates exceptions when errors occur. It provides meaningful method names to help the domain tier interacting with the amplifier. It is the ideal place to establish a policy of caching. The domain tier should be unaware of all these mechanisms.

Domain layer: It is responsible for modeling the conceptual structure of the domain. In the case of Power Config, it models all the information—coming from the amplifier—relevant to expert in audio system. In fact, this model is not directly related to a specific amplifier. It is the model of a generic amplifier that would exist in a perfect world. It is the generalization of all the amplifiers. This remark is very important to understand how our application can easily evolve to handle new kinds of amplifiers. This layer is also responsible for validating data that comes in from the presentation and for pushing it to the amplifier, and for computing values based on other values pulled from the device. Once again, we stress the fact that the resulting data schema inside the device can look quite different from the objects of the domain tier. The hardware layer is driven by performance while the domain layer should represent the true semantic of the domain. Finally, it is responsible for figuring out what data source to dispatch depending on commands received from the presentation.

Application logic layer: In order to decrease the complexity of the application tier, we have split it into the presentation tier and the application logic tier. The application tier is also partitioned “horizontally” into the GUI and the polling system. For testing purpose, it is also conceivable to add a command line to this partition. The application logic tier is responsible for the accesses to the domain tier and any processing other than user interface processing. It implements most of the application behavior. As far as the GUI partition is concerned, the application logic layer is very important for an application like Power Config that contains a significant amount of logic about what kind of screens to use. The Model-View-Controller architectural pattern is used in this application to make these decisions. Consequently, the application logic layer is populated with

---

16 That kind of policy can turn out to be an essential feature.

17 As the application gets more complex the use of the Model-View-Controller architectural pattern can lead to duplicated code, as several controller for different screens need to know what to do in a certain situation. You can remove this duplication by placing all the flow logic into an Application Controller. Input controllers then ask the Application Controller for the appropriate commands for execution against a model and the correct view to use depending on the context within the application” [Fow02]. However, this mechanism is not relevant in this case because the flow and navigation of Power Config is pretty simple; anyone can visit any screen in pretty much any order.
controllers that take user input, manipulate model and cause the view to update appropriately\textsuperscript{18}. The polling system does not handle GUI. It is thus only located in the application logic layer. It ensures the dynamic of the application by requesting—at regular intervals through the domain layer—data likely to change in the amplifier. This behavior provokes the automatic update of the views concerned by these data. It allows the user to have in real-time feedbacks of the current state of the amplifier.

**Presentation layer:** It is the only layer that is visible to the end-user. It contains all of the user interface classes of the application. It displays information to the user and it handles mouse clicks, keyboard hits... 

We have explained that one of the responsibilities of the hardware interface layer is to generate exceptions when error occurs. It is important to design an error handling strategy. An error can either be handled in the layer where it occurred or be passed to the next higher layer. A good habit is to try to handle errors at the lowest layer possible. This prevents higher layers from handling many different errors. If you choose to propagate an error, try to transform similar error types into more general error types. Otherwise, higher layers can be confronted with error messages that apply to lower-level abstractions and that the higher layer does not understand.

To conclude this discussion about the Layers architectural pattern, it is interesting to examine how to organize the domain logic of Power Config. Fowler, in his book about layered architecture for Information System [Fow02], proposes two different approaches: the Transaction Script and the Domain Model. The Transaction Script is essentially a procedure that takes input from the presentation, processes it with validations and calculations and stores data in the database. This is the simplest approach with a single procedure for each action that a user might want to do. Unfortunately, the Transaction Scripts are not really suitable for complex and/or evolving domain logic. OO technology has revolutionized the way people developed domain logic. The domain model takes advantage of this technology to handle complex domain logic. The objects model the business area that you are working in. You find objects that mimic the data in the business, and objects that capture the rules that the business uses. As far as Power Config is concerned, we follow this second approach. However, the domain logic of this application is made up of a limited number of rules. Indeed, the actions users are allowed to perform are really simple. The users can mute the system, change the music source, equalize the sound... For this reason, the domain layer is mainly composed of objects that mimic the data of a generic amplifier. We have called this set of objects the device model.

\textsuperscript{18}This mechanism is described in the next section.
3.5.2 The Model-View-Controller Architectural Pattern

Once again, the book “Pattern-Oriented Software Architecture” [BMR+96] has been valuable in order to help us to write this section\(^\text{19}\).

“The most often used mechanism for inter-layer communication is the push model. When layer \(n\) invokes a service of layer \(n-1\), any required information is passed as part of the service call. The reverse is known as the pull model and occurs when the lower layer fetches available data information from the higher layer at its own discretion” [BMR+96]. In order to reduce coupling between layers as much as possible, our architecture follows the rule according to which the lower layer should be unaware of the identity of its users. That means that the pull model is forbidden. This causes problems when the presentation needs to update on a change in the domain. Indeed, the polling system is responsible to ensure the synchronization in real-time of the application with the amplifier. It deliberately causes the domain to update by sending requests to this layer. But, once the domain is updated how can the presentation be informed of this change without breaking the aforementioned principle?

Buschmann et al. [BMR+96] propose a basic solution to avoid dependencies of lower layers on higher layers introduced by the pull model. “You can use callbacks and still preserve a top-down one-way coupling. Here the upper layer registers callback functions with the lower layer [\ldots] During start-up the higher layer tells the lower layer what functions\(^\text{20}\) to call when specific events occur” [BMR+96].

This kind of interaction is also known as publish-subscribe. The lower layer is the publisher of notifications. It notifies the upper layer that has previously subscribed to this service. The GoF has studied this interaction through the well-known Observer design pattern. According to them, this pattern is, among other things, applicable “when an object should be able to notify other objects without making assumptions about who these objects are” [GHJV95].

This pattern is the foundation of the Model-View-Controller architectural pattern which is used in our application in order to respect the underlying principles of the layered architecture. Here is the definition of this architectural pattern: “The Model-View-Controller architectural pattern (MVC) divides an interactive application into three components. The model contains the core functionality and data. Views display information to the user. Controllers handle user input. Views and controllers together comprise the user interface. A change-propagation mechanism ensures consistency between the user interface and the model” [BMR+96].

Our interpretation of this pattern is a little bit different from the one provided in the aforementioned book. Indeed, according to the original pattern, the view is supposed to define an update procedure that is activated by the

\(^{19}\)Readers are referred to Chapter 5 (Section §5.1) for a detailed presentation of this pattern.

\(^{20}\)The Command pattern shows how to encapsulate callback functions into first-class objects.
change-propagation mechanism. That means that the view should directly subscribe to the service provided by the model. However, in our architecture, the views correspond to the presentation layer and the models correspond to the domain layer. That implies that the presentation tier deals directly with the domain tier which breaks the rule specifying that a layer should only communicate with adjacent layers. Another option, depicted in Figure 3.6, is to force the controllers—which correspond to the application logic layer—to subscribe to this service. Once the controller will receive the notification of an event, it acts accordingly to update the presentation layer. Unfortunately, we are confronted to a dilemma because this alternative breaks another rule specifying that the pull model is forbidden. Nevertheless, we should remember that we have introduced the application logic layer in order to promote the reusability of the user interface. Thanks to this layer, the presentation layer can be developed totally independently of the domain layer. Moreover, in several GUI platforms, the responsibilities of the view and the controller are combined in a single component. For these reasons, it does not seem to be a serious mistake to apply the second option.

The controllers have also the responsibilities to accept user inputs as events and to update the appropriate model. The model in turn will notify the controllers that have registered their need to be informed about changes. It also happens that the behavior of a controller depends on the state of the model. In
that case, the controller will also be notified about that kind of change on the model. This mechanism allows determining the behavior of the user interface. That way, for instance, a change to a model associated with a panel that allows the user to choose the kind of information he wants to consult can be interpreted by a controller in order to display the right information.

Another advantage of this pattern lies in the fact that more than one view can observe the same core data. Moreover, new ways of presenting the core data can be integrated without major impact to the system.

We give full details of the MVC pattern in Chapter 5 which concerns the creation of reusable Swing components. Indeed, the Swing library takes advantage of this well-known pattern to produce widgets composed of reusable parts.

If you have encountered some difficulties to follow the thread of our ideas and some points remain misunderstood, do not worry! We will enlighten you on these points, in the following chapter, by studying some of these aspects thoroughly. Nevertheless, we have laid the necessary foundation for the presentation of the main subject of this thesis: the design patterns. The Layers and MVC architectural patterns are the skeleton of the architecture of Power Config. In the next chapter, we refine this architecture with the help of design patterns.

3.6 Summary

In this chapter, we explained that the object-oriented architecture is supposed to be in accordance with the fundamental principles that help designing successful software architecture. Hence an object-oriented architecture should be based on weak coupling between modules, strong cohesion within modules... We also emphasized the fact that the architecture is not only the result of the design process but also the result of the requirements, analysis, implementation, test, and deployment processes.

We presented four kinds of patterns for software development: analysis patterns, architectural patterns, design patterns, and idioms. We concentrated on the architectural patterns that help in structuring software system into subsystem. We studied the Layers and Model-View-Controller architectural patterns that are encompassed within the Power Config architecture.

Finally, we described the coarse-grained architecture of Power Config software. This architecture is an adaptation of the standard layered architecture for Enterprise Application. We studied thoroughly each layer of this architecture: the presentation layer, the application logic layer, the domain layer, the hardware interface layer, and the hardware layer.

**Hardware layer:** Beyond all the responsibilities in relation to the behavior of
3.6 Summary

an amplifier, this layer is responsible for storing the meaningful values—that represent the current state of the amplifier—into some form of storage.

**Hardware interface layer:** The hardware interface layer is responsible for providing an interface to the domain tier in order to conceal the complexity of the communication with the device. It handles the communication through the serial or USB port. It generates exceptions when errors occur. It provides meaningful method names to help the domain tier interacting with the amplifier.

**Domain layer:** It is responsible for modeling the information relevant to expert in audio system. It is the model of a generic amplifier that would exist in a perfect world. This layer is also responsible for validating data that comes in from the presentation and for pushing it to the amplifier, for computing values based on other values pulled from the device. Finally, it is responsible for figuring out what data source to dispatch depending on commands received from the presentation.

**Application logic layer:** The application logic tier is responsible for the accesses to the domain tier and any processing other than user interface processing. It implements most of the application behavior. Consequently, the application logic layer is populated with controllers that take user input, manipulate model and cause the view to update appropriately. The polling system is located in the application logic layer. It ensures the dynamic of the application by requesting—at regular intervals through the domain layer—data likely to change in the amplifier. This behavior provokes the automatic update of the views concerned by these data. It allows the user to have in real-time feedbacks of the current state of the amplifier.

**Presentation layer:** It is the only layer that is visible to the end-user. It contains all of the user interface classes of the application. It displays information to the user and it handles mouse clicks, keyboard hits…
Chapter 4

How Design Patterns Improve the Architecture

...Design patterns allow designers to create more flexible, elegant, and ultimately reusable designs without having to rediscover the design solutions themselves...

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides [GHJV95]

This chapter assumes that you are reasonably proficient in object-oriented programming and that you have some experience in object-oriented design as well. If you feel uncomfortable with concepts such as polymorphism, interface, or inheritance, it can be interesting to go deeply into Chapter 2 in order to be able to play with these concepts that are key aspects of design patterns. On the other hand, this is not a technical report and the solutions presented in this section are both simple and elegant and require neither unusual language features nor amazing programming tricks.

We are really convinced that design patterns deserve an important place in the software development process. For a long time, we have wondered what could be our contribution to the patterns community. Indeed, libraries shelves are full of books that describe patterns thoroughly and apply these patterns in practice. It is thus very difficult to bring something new about this topic. However, with few exceptions, these books do not introduce patterns in the large field of a concrete software implementation. We thought that—having actively participated to the development of Power Config which takes really advantage of design patterns—it was an opportunity for us to present design patterns from another point of view.

In the last chapter, we have already laid the foundation for the architecture of the Power Config application. The purpose of this section is to make you sen-
sitive to the importance of design patterns in the software development process by studying how design patterns strongly improve the software architecture. We also want to show you with an incremental approach how we concretely apply design patterns to refine the architecture of a real software application. Nevertheless, we know how hard it is to learn how/where/when to use design patterns and we will be fully satisfied if, at the end of this chapter, like us, you are convinced of the excellent impact of design patterns on the software architecture.

Before tackling the refinement of the Power Config architecture, we present, in the first part of this chapter, the state of the art about design patterns and we provide an accurate design pattern definition. We also defend our position with regard to “hot” topics such as pattern tools or pattern languages. Finally, we describe the differences between design patterns and frameworks and we present the polling system framework that takes advantage of the Template Method design pattern to promote the reusability of the code.

In the second part of this chapter, we concentrate on the improvements design patterns bring to the architecture of the amplifier surrogate subsystem that we developed for the needs of Power Config. We apply five distinct GoF design patterns to transform the adapted standard layered architecture for Enterprise Application into the final architecture. For each pattern, we focused on a specific non-functional property of the subsystem that we describe in detail.

4.1 Presentation of Design Patterns

4.1.1 What is a 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" [ASI+96]. This sentence, written in 1966 by Christopher Alexander, has gone round the world. Practically all the books dealing with design patterns touch on this mythical author. This definition is related to patterns in buildings but it works pretty nicely for software as well. The solely difference lies in the fact that “the solutions are expressed in terms of objects and interfaces instead of walls and doors, but at the core of both kinds of patterns is a solution to a problem in a context” [GHJV95].

In 1995, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides—the Gang of Four—wrote the book “Design Patterns, Elements of Reusable Object-Oriented Software” [GHJV95]. This book is a must and has become the bible of object designers and developers. It records experience in designing object-oriented software as design patterns. It is hard to find an object-oriented system that does not use at least a couple of the GoF patterns. Gamma et
4.1 Presentation of Design Patterns

al. give a commonly-accepted definition of a design pattern: “A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. The design pattern identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities. Each pattern focuses on a particular object-oriented design problem or issue. It describes when it applies, whether it can be applied in view of other design constraints, and the consequences and trade-offs of its use. Since we must eventually implement our design, a design pattern also provides sample [...] and code to illustrate an implementation”.

Design patterns capture solutions that have evolved over time. These solutions have matured for several years to ensure greater reuse and flexibility. They document existing, well-proven design experience. Furthermore, they capture these solutions in a succinct and easily applied form. This is the reason why patterns are so relevant. They give you insights that help to make designs more reusable, flexible, and understandable.

Design patterns also provide higher level of abstraction by looking at object composition. The abstraction is essential to tackle the growing complexity of software. The object-oriented technology, by abstracting state and behavior into objects, makes development of software much easier. Today, the emergence of design patterns allows developers to access higher level of abstraction by thinking about groups of objects that collaborate to accomplish a certain task rather than a single object.

In addition, design patterns provide a common vocabulary and understanding for design principles. They serve as a good team communications medium. The design language is extended with carefully-chosen names that convey the essence of the patterns. They facilitate effective discussion of design problems and their solutions. They are also powerful means of documenting software architecture and improve design understandability.

The GoF book organizes and presents a catalog of 23 patterns. These patterns are classified by two criteria: the purpose and the scope. Patterns can have creational, structural, or behavioral purpose. The scope specifies whether the pattern applies primarily to classes or to objects. “Creational class patterns defer some part of object creation to subclasses, while Creational object patterns defer it to another object. The Structural class patterns use inheritance to compose classes, while the Structural object patterns describe ways to assemble objects. The behavioral class patterns use inheritance to describe algorithms and flow of control, whereas the Behavioral object patterns describe how a group of objects cooperate to perform a task that no single object can carry out alone” [GHJV95]. Figure 4.1 presents the 23 patterns classified according to the above criteria.

The patterns of the GoF book are not the only ones related to software. The

---

1Patterns written in italic style are somehow tackled in this thesis.
The growing patterns community gets down to the task of defining new patterns. Since 1994, each year, the PLoP conference is held at the University of Illinois. Its focus is improving the expression of patterns. Authors have the opportunity to refine and extend their patterns with help from knowledgeable pattern enthusiasts. These conferences result in the publication of books, technical reports and journal papers that contain numerous new patterns. Another well-known conference is the OOPSLA conference that brings together practitioners, researchers, and students to share their ideas and experiences in the field of object technology.

However, the approach of those communities is slightly different from the GoF. Their patterns span several levels of abstraction—from analysis patterns through architectural and design patterns to idioms—whereas GoF book concentrates on design-level patterns. As opposed to architectural patterns, we can precise that these design patterns are intended for low-level object-oriented design. Indeed, the GoF patterns are based on practical solutions that have been implemented in object-oriented programming languages. Nevertheless, it is uneasy to draw the line between architectural and design patterns because there is some overlap between both kinds of patterns. We have already presented architectural patterns—such as the Layers and MVC architectural patterns—that can be used at the beginning of coarse-grained design, when specifying the fundamental structure of an application. This chapter concentrates on the design patterns that can be both used to refine and extend the architecture of a software system and used to specify local design aspects in the detailed design.
4.1 Presentation of Design Patterns

stage.

We really want to emphasize the fact that design patterns are very helpful to create architectures that integrate non-functional requirements such as reusability, evolvability, portability, understandability, maintainability... Our approach will be supported by “Pattern-Oriented Software Architecture” [BMR+96] that introduces some patterns that fall into the category of design patterns and the GoF book [GHJV95] that describes the key design patterns. The pattern catalog of the latter book is extensively used for the refinement of the Power Config architecture.

4.1.2 Pattern Systems Vs Pattern Languages

There are many ways in which the pattern community’s work is like Christophe Alexander’s [ASI+96]. For instance, both capture solutions that have evolved over time, and both rely on natural language to describe patterns. Nevertheless, there are also some differences. Alexander’s book is entitled “A Pattern Language”. “A pattern language implies that its constituent patterns cover every aspect of importance in a particular domain” [BMR+96]. This architect believes that his patterns will generate complete buildings. Without denying the need for creativity, “his description of how patterns generate designs implies that a pattern language can make the design process deterministic and repeatable” [GHJV95]. He also gives an order in which his patterns should be used. However, the pattern community does not claim the same affirmations.

Indeed, the GoF and the authors of “Pattern-Oriented Software Architecture” [BMR+96] share the same point of view, according to which their patterns—even when extended with all the other related patterns—only cover some but not all aspects of the construction of software architectures. There are missing patterns for certain aspects of the construction and implementation of software systems. Hence these patterns do not form a pattern language. At the moment, a pattern language that provides step-by-step instructions for designing every imaginable application does not exist.

The GoF admits that a pattern language for certain classes of applications exists. But they do not believe that there will ever be a complete pattern language for software.

We totally agree with them but we do not dare to be so categorical. According to us, at the moment, it seems too early to imagine constructing a complete pattern language for software because the computer science is not mature enough. Indeed, we have been making software systems for a short time whereas people have been making buildings for thousands of years.

Nevertheless, POSA\textsuperscript{6} authors stress the fact that, even if each pattern is

\textsuperscript{6}“Pattern-Oriented Software Architecture”.
Design Patterns Improve the Architecture

relatively independent, they are not isolated from each other. One pattern leads often to another or one pattern usually occurs if another is around. A catalog of patterns is thus not enough to reflect these interdependencies. For this reason, they have introduced a pattern system for software architecture. The pattern system can be compared with a language. “The patterns make the vocabulary of the language, and the rules for their implementation and combination make up its grammar”. However, they have preferred the word system because, as above mentioned, they have far less than a pattern language.

4.1.3 Pattern Tools

Design patterns—that capture common design problems and their solutions—are described both in a consistent and informal manner. Indeed, patterns can only be interpreted by human beings, because they are written in an informal way. In fact, the description includes informal aspects introduced in a consistent way. Each pattern is divided into sections according to a template that you can find in Section §1.3 of the GoF book [GHJV95]. This template integrates aspects such as the intent of the pattern, an example situation that introduces the problem, a partial implementation, and a discussion of the consequences and the trade-offs of applying the pattern. Each design pattern has also a descriptive name. Finally, the solution part of a pattern is typically described by the responsibilities of its participants and the collaboration between these objects.

It is not surprising that people unsatisfied by this informal description want to formalize pattern. The academic world is especially concerned about this formalization. Helm et al. [HHG90] have worked on a formal approach to specify behavioral composition. Behavioral compositions are groups of interdependent objects that cooperate to accomplish a particular task (i.e. design patterns). Their book introduces the concept of contracts as ”a construct for the explicit specification of behavioral compositions”.

This formal approach makes design solutions suitable to be interpreted and manipulated automatically. It would support the development of pattern tools much better than the informal pattern description. As illustration, we mention Meyers’ PhD thesis [Mei96] that deals with pattern tool. Meyers explains that “the powerful use of OO design patterns and the explicit representation of contracts could be combined in order to open up the path to tool support for pattern-based design”.

However, everybody does not agree with the supporters of the formalization. Buschmann et al. [BMR+96] and Fowler [Fow02] think that formal methods do not apply to pattern.

According to Buschmann et al. [BMR+96], it is harder to match a specific design problem, which is usually not formalized, to a pattern described in a formal way. It is also harder to understand the key ideas of a formalized solution.
Hence it is more difficult to create valid variants of this solution. In addition, they do not know formalism suitable for describing the benefits and liabilities of a pattern.

A paragraph of Fowler’s book has particularly drawn our attention: “/.../ once you need the pattern, you then have to figure out how to apply to your circumstances. A key thing about patterns is that you can never just apply the solution blindly, which is why patterns tools have been such miserable failures. A phrase I like to use is that patterns are “half baked”—meaning that you always have to finish them off in the oven of your own project. Every time I use a pattern I tweak it a little here and a little there. So you see the same solution many times over, but it’s never exactly the same” [Fow02].

The last authors’ ideas are akin to ours, having handled design patterns for the elaboration of the Power Config architecture, we are aware of the difficulty to match a specific design problem to a pattern. This activity needs a lot of experience and maturity in designing systems. It seems really complex to realize a tool that helps designers choosing the right pattern. Then, we experienced that the applied solution is never exactly the same as the one described in the pattern...
Design patterns are more abstract than frameworks: This is the major difference between both approaches. Schmidt et al. [SJF96] add that frameworks are generally implemented in a particular language, whereas patterns are described in language-independent manner. Hence frameworks are ready to be executed and reused whereas the design patterns have to be implemented each time they are reused.

Design patterns are smaller architectural elements than frameworks: Framework can be viewed as concrete realizations of patterns that facilitate direct reuse of design.

Design patterns are less specialized than frameworks: The GoF design patterns can be used in almost every kind of application, whereas frameworks always have a particular application domain. As far as the POSA patterns are concerned, these are often more specialized than GoF patterns. Nevertheless, even if they cannot be used in every kind of application, these kinds of patterns would not dictate an application architecture like a framework would.

As mentioned in the second point, a framework typically encompasses several design patterns. It takes advantage of design patterns to promote the reusability of the code. The Template Method design pattern\(^7\) is commonly used to allow developers to redefine certain steps executed within the framework. This is thus a way to ensure the reusability of the framework. This principle is usually called the “Hollywood Principle”\(^8\) [Vli96a]. It results in an inversion of the flow of control: developers write the code that will be called by the framework. The framework has the responsibility to organize the flow of control by invoking client’s methods at the appropriate time and place. Concretely, as mentioned in the GoF definition of framework, “you customize a framework to a particular application by creating application specific subclasses of abstract classes from the framework”. Vlissides [Vli96a] specifies that “subclasses can extend or reimplement the variable parts of the algorithm, but they cannot alter the template method’s flow of control and other invariant parts”.

Obviously, the Template Method design pattern is not the only one used within frameworks. Every pattern that can help ensuring the reusability of the code is suited for frameworks\(^9\).

For example, Fayad et al. [FSJ99] define a framework dedicated to the speech recognition that encompasses several design patterns: the Facade, the Singleton, the Adapter, and the Observer design patterns. Those patterns are used “to guide the creation of abstractions in the design phase, necessary to accommodate future changes and yet maintain architectural integrity” [FSJ99]. Those abstractions modeled by abstract classes in the framework represent major speech components. The major speech components are realized in speech

---

\(^7\)Readers are referred to p.158 (Appendix A) for a description of the Template Method design pattern.

\(^8\)“Don’t call us, we’ll call you”.

\(^9\)The Factory Method design pattern [GHJV95] is also commonly used within frameworks.
4.1 Presentation of Design Patterns

applications by the creation of concrete derived instances of the abstract classes. This mechanism allows framework developers to encapsulate the speech engine functionality in abstract terms while the concrete instances are implemented by speech application developers.

As far as we are concerned, we developed a simple framework intended for the implementation of the polling system. This is an opportunity for us to present the Template Method design pattern that is used within this framework. This pattern is not the most complicated of the GoF catalog. It should be a good introduction to get you used to design patterns.

The polling system is responsible for ensuring the synchronization in real-time of the application with the amplifier. It deliberately causes the domain to update by sending meaningful requests to this layer. Section §3.5.1 about the layered architecture of Power Config application gives you an overview of this framework. Figure 3.5 shows how the polling system interacts with the other layers. Finally, Section §3.5.2 explains how the presentation layer is updated without breaking the principle according to which the lower layer should be unaware of the identity of its users.

The application domain of our framework is limited. It does not cover a complete application as mentioned in the above definition. That means that Power Config is a lot more than this framework. It is thus customized and integrated within the whole application. Nevertheless, it fits well into GoF definition: it is a set of cooperative classes that make up a reusable design, it defines how the classes and objects collaborate, it defines the thread of control, it captures the design decisions that are common to its application domain, and it is customized to a particular application by creating application specific subclasses of abstract classes.

In few words, its application domain is limited to specific class of subsystems that are responsible for executing, at regular intervals, a certain amount of actions. These actions are determined by the subscriptions submitted to the framework. Clients of this framework are responsible for defining the subscriptions that will be handled by the framework and the actions related to each particular subscription. The framework undertakes the management/storage of the subscriptions and the threading mechanism that can be configured at will by the client. It also manages synchronization issues in order to allow simultaneous multiple accesses to the polling system. The client can (re)start, stop the activities of the framework, he can define interval between the executions of the subscribed actions... This framework is thus designed for applications that need, at predefined time, to execute a set of actions but this set is likely to change frequently. Although our framework can be reused in very different situations, most of the time, it is used to poll information. For this reason, we have called it the polling system.

It can be reused, for instance, within a virtual stock exchange that allows users to follow the evolution of the market prices of selected stocks and shares. In that kind of application, constantly, users are likely to be interested in new
kind of bonds. In this case, our framework would be useful for implementing a subsystem on the client side that registers the wished stocks and shares and, at regular intervals, asks the server the updated values of these bonds. The designer has just to customize the framework by defining subscriptions and their related actions. Subscriptions would represent specific stocks and shares while actions would define the application behavior that is necessary to update the values related to these stocks and shares. This behavior includes especially requests to the server and updates of the user interface.

Originally, we have developed this framework for the needs of Power Config application. Power Config interface is divided into different panels. One of these panels is responsible for the control of specific parameters of the amplifier. It is called the control panel. Another panel, called the hardware diagram, allows users to select which kind of information should be displayed within the control panel. These two panels are depicted in Figure 4.2 representing the Power Config interface. Users can choose, for instance, to display information about the gain of the output channel 1 (like in Figure 4.2) or the gain of the input channel 3... Whatever information is displayed, the control panel is supposed to reflect the current state of the amplifier. Hence the Power Config application is responsible for polling the relevant information from the hardware.

In order to achieve this goal, two policies can be implemented. The first one consists of polling all the “pollable” information all the time. This is the easiest solution to implement but it is also the less efficient. Indeed, we blindly request values which are most of the time useless. The other one is to poll only for the needed information according to the currently displayed control panel. This solution is more efficient than the first one but we need to find a mechanism to adapt the set of values that should be polled according to the displayed control panel. In order to increase the performance of the application,
we have decided to implement the second solution. Indeed, the communication
with the hardware through the serial cable is the bottleneck of the application
and it would have been ill-advised to request inadequate values.

Consequently, we have decided to create the above-defined framework and
to customize it according to our own requirements. Figure 4.3 reminds you of
the layered architecture of Power Config and the position of the polling system
within this architecture.

![Coarse-Grained Architecture of Power Config](image)

Figure 4.3: Coarse-Grained Architecture of Power Config

The polling system is a complete subsystem within the application logic
layer. Besides its responsibilities for accessing the domain tier, the application
logic layer implements most of the application behavior. When users select a
specific control panel with the help of the hardware diagram, the hardware dia-
gram controller is notified of this selection\(^\text{10}\) and can inform the polling system
of the values that need to be updated within the control panel. The communica-
tion of these values from the controller to the polling system is straightforward
because both components share a common vocabulary. Indeed, both take ad-
vantage of the domain layer—which models the conceptual structure of the
domain—to handle meaningful information. Hence they both handle the same
domain values.

What kind of information should be sent from the hardware diagram con-
troller to the polling system in order to define the set of values that have to be
polled? In the device model, each value is identified by a property name. This
property name is part of the notification mechanism that allows models to notify
any interested party when their data changes\(^\text{11}\). In fact, as mentioned in Section

\(^{10}\)This mechanism follows the MVC architectural patterns described in Section §3.5.2.
\(^{11}\)The architecture of Swing components exploits this mechanism to notify the view when
data changes in the component model. Section §5.1.3 studies the different variants of the
notification mechanism in detail.
§3.5.2 about the Model-View-Controller architectural pattern, controllers register to the domain layer as listeners of events. As soon as a value changes within the device model\(^\text{12}\) these controllers are notified of this event. The notification—sent by the device model to the interested controllers—contains a bunch of information related to the event\(^\text{13}\). Among this information, they find the property name that gives the semantic of the updated value. This property is both useful to identify the specific value that has change and to deduce the methods that can be used to access the variable that stores this value (and thus to access this value). These properties are usually represented with a \texttt{String}.

The property name stands thus as a good candidate for allowing the controller to indicate to the polling system the set of values that need to be polled. Furthermore, we would kill two birds with one stone because, with these property names, the polling system can deduce the methods that are used to cause the update of the domain layer\(^\text{14}\).

Once the polling system framework implemented, we just have to customize the subscriptions and their related actions. Subscriptions will contain at least the property name of the value that needs to be polled. Actions will consist of updating specific domain values. But, first of all, we study the polling system framework and the Template Method design pattern.

Figure 4.4 depicts the polling system framework class diagram\(^\text{15}\). It describes the types of objects within the framework and the static relationships that exist among them. The framework contains two interfaces (\texttt{PollingSystem} and \texttt{PollingSubscriptionHandler}), one abstract class (\texttt{AbstractPollingSubscriptionHandler}) and one class (\texttt{BasicPollingSystem}). The \texttt{PCPollingSubscriptionHandler} class is specific to Power Config application. PC stands for Power Config.

The framework can be instantiated by creating an instance of a class that implements the \texttt{PollingSystem} interface. We have developed only one class that implements this interface: the \texttt{BasicPollingSystem} class. As framework developers, we can decide to provide different implementations of the polling system. However, it is crucial for users of our framework to be able to switch from one implementation to another without any difficulty. The less code users have to modify the better it is. For this reason, all our future implementation should be compliant with the \texttt{PollingSystem} interface. Consequently, methods defined in the \texttt{PollingSystem} interface have to be chosen very carefully. These methods should be both abstract enough to meet future evolution of the framework and meaningful enough to be handled by framework users. We should also avoid providing too many methods in this interface. Otherwise, it will make framework developers’ life a burden.

\(^{12}\)The device model is located within the domain layer. It is composed of objects that mimic the data of a generic amplifier. A description of this model is given at the end of Section §3.5.1.

\(^{13}\)That kind of notification is called in Section §5.1.3 “stateful” notification.

\(^{14}\)The Java reflection package can be very useful to implement the method call based on the property name.

\(^{15}\)Readers are referred to Appendix B for a description of the UML notation.
Figure 4.4: Polling System Framework Class Diagram

Framework users that use to take advantage of the programming to interfaces\textsuperscript{16} will benefit from our efforts and will have no difficulty switching from one implementation to another. Below, we have printed a piece of code commonly written by users of our framework. We assume that the only PollingSystem interface implementation is the BasicPollingSystem class:

1 2 3 4 5 6 7 8 9 10
1 2 3 4
... public PollingSystem createPollingSystem() {
  pollingSystem = new BasicPollingSystem();
  ... pollingSystem.setPollingInterval(500); // msec
  pollingSystem.startPolling();
  return pollingSystem;
}

doIt

10 public PollingSystem getPollingSystem() {

\textsuperscript{16}Chapter 2 introduces this OO principle.
if (pollingSystem == null) {
    pollingSystem = createPollingSystem();
}
return pollingSystem;
}

private PollingSystem pollingSystem;
...

Now, imagine that we provide another implementation of the PollingSystem interface called NewPollingSystem. How will the framework user modify his code in order to use the new implementation?

All he had to do is to modify line 3:

pollingSystem = new NewPollingSystem();

That is only possible because the user has the good habit to program to interfaces. Otherwise, he would also have to change lines 2, 10, and 17!

The BasicPollingSystem class implements all the threading mechanisms. These mechanisms are included within the methods: startPolling, stopPolling, isPolling, setPollingInterval, getPollingInterval, setSleepTime, and getSleepTime.

The polling system should also manage the registering of the subscriptions through the methods subscribe and unsubscribe. However, these operations are not directly implemented within the BasicPollingSystem class. This class delegates these operations to a class that implements the PollingSubscriptionHandler interface.

"In delegation, two objects are involved in handling a request: a receiving object delegates operations to its delegate" [GHJV95]. To achieve this behavior, the BasicPollingSystem class keeps an instance variable of a class that implements the PollingSubscriptionHandler interface and delegates subscription-specific behavior to it. The arrowhead line between the BasicPollingSystem class and the PollingSubscriptionHandler interface indicates in Figure 4.4 that the BasicPollingSystem class keeps a reference to an instance of a class that implements the PollingSubscriptionHandler interface:

```java
public class BasicPollingSystem implements PollingSystem {
    ...
    public synchronized void subscribe(Object[] subscriptions) {
        if (pollingSubscriptionHandler != null) {
            for (int i = 0; i < subscriptions.length; i++) {
                if (subscriptions[i] != null) {
```
Both `subscribe` and `unsubscribe` methods receive as argument an array of instances of the `Object` class. We could have provided a `Subscription` class and passed instances of this class into these methods. However, it is the responsibility of framework users to customize the framework by defining subscriptions and their related actions. We thus have decided to make use of the root of the class hierarchy (the `Object` class) so that users are free to extend this base class according to their needs. Hence, within the framework, we cannot make any assumptions about the type of `subscribe` and `unsubscribe` methods arguments.\(^\text{17}\)

\(^\text{17}\)When framework developers need to make assumptions about the type of an application-specific object (instantiated outside the framework) handled within the framework, the Factory Method design pattern [GHJV95] is very useful.
Lines 9 and 19 of the source code of the BasicPollingSystem class illustrate the delegation mechanism. When the BasicPollingSystem class handle subscribe or unsubscribe requests, it defers those requests to an instance of a class that implements respectively add and remove methods of the PollingSubscriptionHandler interface.

Two other methods of the BasicPollingSystem class are also delegated: the poll and createSubscription methods. The poll method is a private method that is part of the threading mechanism which is implemented within the BasicPollingSystem class. When the polling system is active, this method is invoked at regular intervals according to the parameters specified by framework users. Then, this method delegates operations to the poll method of an instance of a class that implements the PollingSubscriptionHandler interface (line 36). The createSubscription method is a factory that allows creating subscriptions. This creation is also delegated to the createSubscription method of this class (line 30). The AbstractPollingSubscriptionHandler abstract class is the only class that both belongs to the framework\(^\text{18}\) and implements the PollingSubscriptionHandler interface. We examine the source code of this class:

```java
1 public abstract class AbstractPollingSubscriptionHandler
2   implements PollingSubscriptionHandler {
3     ...
4     
5     public abstract Object createSubscription();
6     
7     public abstract void pollSubscription(Object subscription);
8     
9     public void poll() {
10        Set subscriptions = getSubscriptions();
11        Iterator subscriptionsIterator = subscriptions.iterator();
12        while (subscriptionsIterator.hasNext()) {
13           pollSubscription(subscriptionsIterator.next());
14        }
15     }
16     
17     ... 
18     
19  }
20 }
```

We notice that the delegated createSubscription request is not implemented within the framework. Indeed, this method is declared abstract. As far as the poll method implementation is concerned, we observe that for each subscription the pollSubscription method is invoked\(^\text{19}\). This last method is

\(^{18}\)As a reminder, the framework contains two interfaces (PollingSystem and PollingSubscriptionHandler), one abstract class (AbstractPollingSubscriptionHandler) and one class (BasicPollingSystem). The PComparatorPollingSubscriptionHandler class is specific to Power Config application.

\(^{19}\)The poll method iterates through a set of subscriptions using the Iterator design pattern.
also declared abstract.

These two methods (createSubscription and pollSubscription) are not implemented within the framework because they are application specific. Indeed, we are supposed to customize the framework by defining subscriptions and their related actions. Hence we have to implement the createSubscription factory and to define the application behavior when the pollSubscription method is invoked. The framework is thus easily customized by creating a subclass of the AbstractPollingSubscriptionHandler abstract class. In this subclass, we will find an implementation of these two tightly coupled methods. Then, we need to notify the framework of our intention to use this specific implementation of the PollingSubscriptionHandler interface. This notification is carried out through the setSubscriptionHandler method of the PollingSystem interface (line 40 of the BasicPollingSystem source code).

The definition of abstract classes is a powerful OO instrument. It is common to define abstract primitive operations that concrete subclasses implement to specify steps of an algorithm. Frameworks abound in abstract classes that are used to implement the invariant parts of the algorithm once and leave it up to subclasses (defined by framework users) to implement the behavior that can vary. Gamma et al. [GHJV95] have captured this well-known OO practice in their catalog of 23 design patterns. This pattern that belongs to the behavioral patterns set is called the Template Method design pattern. According to these authors the intent of the Template Method design pattern is to “define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithms structure” [GHJV95]. This pattern results in an inversion of the flow of control. Indeed, surprisingly, parent classes call operations of subclasses. We have already introduced this principle called “the Hollywood principle”, that is, “Don’t call us, we’ll call you”.

In our framework, the pollSubscription method illustrates this inverted flow of control. Users of our framework redefine the behavior of the application when the framework decides to poll the information relative to a specific subscription. In Power Config application, we have created the PCPollingSubscriptionHandler class that extends the AbstractPollingSubscriptionHandler abstract class. We analyze the meaningful source code of this class:

```java
public class PCPollingSubscriptionHandler
extends AbstractPollingSubscriptionHandler {

public PCPollingSubscriptionHandler(DeviceModel deviceModel) {
    this.deviceModel = deviceModel;
}

public Object createSubscription() {
    return new PCSubscription();
}
}
```

This design pattern is fully described in the GoF book p.257 and following.
public void pollSubscription(Object subscription) {
    if (subscription instanceof PCSubscription) {
        PCSubscription PCSubscription = (PCSubscription)subscription;
    } else {
        throw new IllegalArgumentException(
                "Cannot handle subscriptions that are not created
                        with the createSubscription factory");
    }
    String propertyName = PCSubscription.getPropertyName();
    if (propertyName.equals(DeviceModel.PRE_GAIN_PROPERTY)) {
        deviceModel.getPreGain();
    } else if (propertyName.equals(DeviceModel.PRE_MUTE_PROPERTY)) {
        deviceModel.getPreMute();
    }
    ...
}

public class PCSubscription {
    private String propertyName;
    private DeviceModel deviceModel;

    public String getPropertyName() {
        return propertyName;
    }

    public void setPropertyName(String propertyName) {
        this.propertyName = propertyName;
    }
}

Our implementation is in accordance with the principle that a layer only knows about its underlying layer. Indeed, the polling system that is part of the application logic layer knows the device model that is part of the domain layer.

We observe that, unlike the framework, this implementation is specific to Power Config. The `createSubscription` factory creates instances of the `PCSubscription` class. The controller of the hardware diagram uses these subscriptions to indicate to the polling system which set of values needs to be polled. Each subscription contains a property name that identifies a device model value. Obviously, when a new control panel is selected through the hardware diagram, the hardware diagram controller first “unsubscribes” to the polling of the previously subscribed values and then subscribes to the polling of new values related to the new control panel. At regular intervals, for each subscription, the `pollSubscription` method is invoked. This method inquires the property name associated with the subscription and then causes the update of the device model.
4.2 Non-functional Properties of Software Architecture

The main objective of design patterns is to allow designers to build software with predictable non-functional properties. Designers use patterns in order to integrate these properties into the architecture. The value in design patterns is that they are explicitly built on enabling techniques. Hence they promote these non-functional properties.

But what are these non-functional properties of software architecture in relation to patterns? At this point, it is interesting to answer this question in order to realize what we can really expect about design patterns. We have chosen to present four of the non-functional properties listed in the book “Pattern-Oriented Software Architecture” [BMR+96]:

**Changeability:** Many applications tend to evolve continuously during their lifetime. To reduce maintenance burden, it is important to develop architecture designed for change. The authors of the aforementioned book distinguish four aspects of changeability: the maintainability that deals with problem fixing, the extensibility that deals with the extension and replacement of components, the restructuring that focuses on the reorganization of the components, and the portability that deals with adapting a software system to a variety of hardware platforms, operating systems...

**Reliability:** The correct behavior of applications is threatened by the occurrence of events such as errors, incorrect usage and degenerate input. A system is reliable if it has the ability to maintain its functionality in those situations. The reliability is divided into two aspects: the “fault tolerance” that deals with ensuring correct behavior in the event of errors and the robustness that focuses on the protection of applications against incorrect usage.

**Testability:** Software architecture that supports testability allows the effortless integration of temporary debugging code. Such architecture promotes fault detection and fixing. With the increasing size and complexity of software systems, this property has become essential.

---

20 The domain layer is responsible for computing this value and accessing the hardware through the hardware interface layer.

21 The enabling techniques for software architecture listed in the book “Pattern-oriented software architecture” [BMR+96] bring together the accumulated experience of software designers. They are built on the common sense and help create successful software architecture. These techniques have been introduced in Section §3.4.2.

22 This list is obviously non-exhaustive.
Reusability: Last but not least, reusability has become an object-oriented designer’s leitmotif. It allows the reduction of costs and development time for software application. Moreover, the reuse of proven components ensures better software quality. The authors distinguish two aspects of Reusability: the “software development with reuse” that focuses on the reuse of commercial libraries or existing components from previous projects and the “software development for reuse” that deals with producing components that are potentially reusable.

As far as the Power Config application is concerned, we have focused on all four of these properties. We demonstrate in the following sections how we have used the design patterns in order to consider these non-functional aspects.

4.3 Improvements of the Architecture

4.3.1 Mediator Design Pattern

In this section, we focus attention on the software reusability. Especially, we demonstrate how the Mediator design pattern\(^\text{23}\) is applied in Power Config application to allow the production of components that are reusable. Our intention is not to apply the GoF solution blindly. On the contrary, we adapt this pattern according to our circumstances. For this reason, you may notice some variation between the GoF solution and ours. We are convinced that the key aspects about patterns are to understand their intent and to figure out how to apply them according to our needs. This remark is also valid for the other design patterns introduced in the following sections.

We consider a simplified version of Power Config. In this version, we just allow users to adjust the gain and to check the gain level. Therefore, the graphical user interface is pretty simple. This interface contains a single panel composed of two widgets: a slider used to adjust the gain and a gain meter used to check the gain level.

We have chosen these two widgets for their opposite activities. Indeed, the slider allows users to modify the state of the amplifier whereas the gain meter just allows users to check the evolving state of the amplifier.

As a reminder, Figure 4.5 represents the layered architecture of this simplified application. The amplifier is located within the hardware layer. It stores the meaningful values that represent its current state. These values are either updated by the amplifier itself (e.g.: the gain level) or modified by the software application (e.g.: the gain). The application can access these values through

\(^{23}\)Readers are referred to p.154 (Appendix A) for a description of the Mediator design pattern.
the hardware interface layer. This layer provides an interface to the domain
tier in order to conceal the complexity of the communication with the hard-
ware. Within the domain tier is located the device model. This model is the
representation of a generic amplifier that would exist in a perfect world. Its
values should be synchronized with the ones from the amplifier in order to al-
low the user interface to reflect the current state of the amplifier. The polling
system ensures this dynamic by requesting, at regular intervals, to the domain
layer data likely to change. The domain layer is then responsible for computing
these values based on amplifier values. Finally, the controller mediates between
the domain layer and the presentation layer. It both receives inputs from the
domain layer and from the user interface. The values coming from the device
model (e.g.: the gain level) cause the update of the graphical user interface and
the values coming from the user interface (e.g.: the gain) are sent to the device
model.

Now, imagine that the requirements of Power Config change and that the
application should allow the simultaneous set up of two distinct amplifiers. With
the current architecture, the immediate solution is to duplicate all the objects
present in Figure 4.5. Figure 4.6 depicts this new architecture.

At first sight, this solution may seem quiet easy to implement. Nevertheless,
you should not lose sight of the fact that the simple components represented
in the above figure may actually hide complex structures. Developers in charge
of the duplication need to acquire thorough understanding of the architecture.
Also, these objects have to be hooked up together to fulfill the requirements.
Hence it will inevitably increase the complexity of the software which may be-
come messy and thus more difficult to understand. In addition, this aggregation
of objects within single subsystems does not promote the factorization of devel-
opers work.
At present, we introduce another architectural approach that allows us to meet easily this new requirement. The idea is to create two independent subsystems from the base architecture depicted in Figure 4.5.

In the base architecture, the device model is responsible for both modeling a generic amplifier and storing the values that the view handles to display significant information. This may be a good practice for small software applications. However, in the case of Power Config, the complexity of the domain does not promote the reusability of the view. In fact, if we plan to reuse the view in other contexts, the view model should be independent of amplifier-specific considerations. We have thus created a reusable subsystem that encapsulates the view. This subsystem contains a brand new view model totally independent of the device model. Figure 4.7 describes this architecture.

In fact, this new architecture produces two reusable subsystems. It produces a stand-alone graphical user interface divided into three components: the model that contains the data, the view that displays information to the user, and the controller that handles user input and implements an update procedure in order to keep the view consistent when the state of the model changes. It also produces another subsystem divided into three components: the connection that manages connections and requests to a specific amplifier, the model that represents a generic amplifier, and the polling system that causes the automatic update of the model.

At some point, these two independent subsystems need to communicate. When users adjust the gain, the updated gain value of the view model is sent to the amplifier. Nevertheless, the GUI subsystem does not have to know that this concerns the gain of an amplifier. Conversely, when the amplifier gain level is updated within the device model, the graphical user interface needs to be
Figure 4.7: Reusable Version of Power Config Architecture

notified of this modification of the amplifier state.

These subsystems communicate through their models. However, proliferating interconnections between the view and device model tend to reduce the reusability. Indeed, these interconnections make it less likely that these subsystems can work independently.

Fortunately, the Mediator design pattern addresses that kind of issue. The intent of this pattern applies perfectly to our architecture: the Mediator design pattern defines “an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently” [GHJV95].

The mediator depicted in Figure 4.7 controls and coordinates the interactions of the device model and the view model. It keeps these models from referring to each other. In fact, the mediator encapsulates the coupling between the two subsystems. It is thus not reusable at all. During start-up, the mediator knows the models and tells them that it wants to be notified when specific events occur. The Observer design pattern allows the models to “be able to notify other objects without making assumptions about who these objects are” [GHJV95]. Thus, not only the subsystems are independent of each other but also they are independent of the mediator that coordinates their activities. Both models just need to implement a change-propagation mechanism.

We have already introduced this change-propagation mechanism in Section §4.1.4. In this case, the principle is exactly the same except that the model

\footnote{Readers are referred to p.156 (Appendix A) for a description of the Observer design pattern. In addition, Section §5.1.1 presents the Observer design pattern in the context of the reusable Swing components architecture.}
registers several listeners. As a reminder, in each model, a property name identifies each value. When one of these values is updated, the model that handles this value fires an event that contains the property name of the updated value, the old value and the new value. The mediator and all the other listeners that have previously subscribed to that kind of event are then notified and act according to their tasks. As far as the mediator is concerned, it just has to update the other model.

In order to illustrate the behavior of the mediator, we have decided to describe in Figure 4.8 how the mediator and the models collaborate to achieve a single use case\textsuperscript{25}. This use case exhibits the following behavior: a user adjusts the gain to -20dB through the slider then this value is propagated through the application in order to change the state of the amplifier and to turn the volume up.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{Figure_4.8_Adjust_Gain_Sequence_Diagram.png}
\caption{Adjust Gain Sequence Diagram}
\end{figure}

In this sequence, the view controller handles user input and passes -20dB into the setGain method. Once the view model is updated, it (the model) creates an event that stores the gain property name, the old value, and the new value. After that, for each listener, the model calls the \texttt{propertyChange} method. The mediator aware of this change collects the necessary information in order to update the device model. Thanks to the property name, it knows which method to invoke to update the right value. Finally, it passes -20dB into this method. The device model is then responsible for communicating this value to the amplifier.

\footnote{Readers are referred to Appendix B for a description of the UML notation.}
This new architecture allows us to meet smoothly the new requirement. Both subsystems can be reused at will in different contexts. We just need to write mediator(s) that encapsulate(s) the coupling between subsystems.

From this simplified version of Power Config, we can deduce the actual Power Config architecture. We have already introduced all aspects of the layered architecture. Now, we introduce the “horizontal” partitions. The architecture is partitioned “horizontally” into three subsystems, which correspond especially to the hardware diagram, the control panels, and the amplifier surrogate.

Figure 4.2 (p.60) depicts the “visual” parts of the hardware diagram and control panels subsystems. The hardware diagram is the “logical” representation of the “physical” amplifier. It shows all of the electronic functions (represented by buttons) which may be configured. When an item is selected the control panel for the function is displayed in the control pane. For instance, users can adjust the amount of low/mid/hi frequency applied to the source signal, select the active system source, or adjust the amount of gain for the indicated output... We examine Figure 4.9 that represents the “horizontally” partitioned Power Config architecture.

![Diagram of Power Config Architecture](image)

Figure 4.9: “Horizontal” Partitions of Power Config Architecture

The three aforementioned subsystems are all reusable components. Both the hardware diagram and the control panels subsystems follow the MVC architectural pattern. Nevertheless, the control panels subsystem is a little bit particular in that it is composed of interchangeable panels. Moreover, the same panel can be used in different contexts\(^{26}\). Therefore the control panels subsystem provides a mechanism that allows switching a panel to another. However, in order to avoid keeping too many objects active in the application, we do not duplicate the distinct panels. For each of these panels, we keep only one

---

\(^{26}\)For instance, the output gain panel can be used to control separately different output channels.
view, one controller, and as many models as contexts in which the specific panel is involved. If we switch, for instance, the output gain panel of channel 1 to the output gain panel of channel 4, we just need to inform this panel that the model connected with channel 4 stores the information that should be currently displayed.

The amplifier surrogate subsystem handles the communication with the amplifier. Through the polling system, it mimics the “behavior” of the connected amplifier. Each time a value changes within the amplifier, this value changes also within the surrogate. Nevertheless, though it is connected to a specific amplifier, the surrogate behaves like a generic amplifier. That behavior implies that the domain layer is sometimes responsible for computing values based on other hardware values. Finally, the amplifier surrogate subsystem allows us to control the amplifier through the device model.

The mediator—between the amplifier surrogate subsystem and the control panels subsystem—promotes loose coupling by keeping these subsystems from referring to each other. This time, it mediates between the device model and all the models that belong to the control panels subsystem.

Finally, we have added within the application logic layer a new object called the application behavior controller. The application logic layer is notably responsible for controlling the application behavior. Within specific subsystems, this layer usually contains controllers responsible for managing the intra-subsystem behavior. For example, in the source assign panel, the “move between lists” button might be enabled by the source assign controller when a source is selected in the “system source” list.

As far as the application behavior controller is concerned, it is responsible for managing the inter-subsystem behavior. Like the mediator, it allows the development of stand-alone subsystems by reducing interconnections between subsystems. However, it differs from the mediator in that its protocol is unidirectional whereas mediator enables cooperative behavior with the help of a multidirectional protocol. Indeed, this application behavior controller is only interested in events coming from the hardware diagram model. When it is notified that an electronic function has been selected in the hardware diagram, it first requests the control panels subsystem to display the appropriate control panel and then it subscribes to the polling of values related to the control panel currently displayed\textsuperscript{27}. Once again, this object encapsulates the coupling between subsystems and is thus not reusable at all.

\textsuperscript{27}Before sending this subscription to the polling system, it unsubscribes to the polling of the values related to the control panel previously displayed.
4.3.2 Facade Design Pattern

Once again, we focus on the software reusability. The above mentioned Mediator design pattern helps to minimize the communication and dependencies between subsystems. Another way to achieve this goal is to introduce a facade object\textsuperscript{28} that provides a simple interface to a complex subsystem.

Gamma et al. [GHJV95] describe the difference between the Mediator and the Facade: “Mediator is similar to Facade in that it abstracts functionality of existing classes. However, Mediator’s purpose is to abstract arbitrary communication between colleague objects, often centralizing functionality that doesn’t belong in any one of them. A mediator’s colleagues are aware of and communicate with the mediator instead of communicating with each other directly. In contrast, a facade merely abstracts the interface to subsystem objects to make them easier to use; it doesn’t define new functionality, and subsystem classes don’t know about it”\textsuperscript{29}. In conclusion, a facade can be compared to an interface of a subsystem that makes the subsystem easier to use. Another difference lies in the fact that the Mediator belongs to the behavioral patterns whereas the Facade belongs to the structural patterns. Thus, the Mediator stresses the way in which objects interact while the Facade deals with the composition of objects.

The complexity of the control panels subsystem makes it difficult to handle. The different panels need to be coordinated to display the appropriate information according to the electronic function selected in the hardware diagram. We need to orchestrate the constant changes of models. Indeed, depending on whether users are interested in one specific channel or another, the control panel associated to the selected function should be supplied with the appropriate model\textsuperscript{30}.

The control panels facade depicted in Figure 4.10 makes life easier for clients of this subsystem. They communicate with the subsystem by sending requests to this facade. Then, the facade does work of its own and translates the requests into other ones that subsystem classes handle. This mechanism allows clients to think about the control panels subsystem as a whole and it prevents them from keeping too many references to objects of this subsystem. The facade is responsible for keeping references to the relevant objects of the subsystem. When these objects are only accessed through the facade, it is possible to create them within the facade as and when required at run-time rather than creating them at initialization-time.

Besides functions related to the polling system, the amplifier surrogate subsystem may provide other services. It may provide functionalities in order to

\textsuperscript{28}Readers are referred to p.153 (Appendix A) for a description of the Facade design pattern.
\textsuperscript{29}We have implemented the mediator as an observer using the Observer design pattern. Hence the colleagues are not really “aware” of the mediator. Nevertheless, they keep a reference to the mediator.
\textsuperscript{30}As a reminder, in order to avoid keeping too many objects active in the application, we do not duplicate the distinct panels. For each of these panels, we keep only one view, one controller, and as many models as contexts in which the specific panel is involved.
allow using an emulator instead of a real amplifier, selecting a caching policy within the hardware interface layer. The amplifier surrogate facade shields clients from subsystems objects by providing high-level interface to access these functionalities. In this way, it makes the subsystem easier to use. In the event of setting up several amplifiers simultaneously, this facade may also be useful to manage the different surrogates.

In Figure 4.10, we have stretched out both facades so that they cover all the layers of their subsystems. We have done that in accordance with the principle that a layer only knows about its lower adjacent layer. If we consider that the facades are part of the layered architecture, they are present within each layer. Consequently, the aforementioned principle is never broken.

4.3.3 Singleton Design Pattern

Both facade objects described in the previous section are unique in our architecture. Our aim is to ensure that only one instance of the control panels facade and one instance of the amplifier surrogate facade are created. The Singleton design pattern ensures these classes only have one instance, and provides a global point of access to it [GHJV95].

We have thus applied this pattern to keep programmers from instantiating multiple facade objects. The facade classes are responsible for keeping track of

---

31In the amplifier surrogate subsystem, the facade does not cover the hardware layer because it is a “physical” layer.
32Readers are referred to p.157 (Appendix A) for a description of the Singleton design pattern.
4.3 Improvements of the Architecture

their sole instance. Here is the control panels facade source code corresponding to the Singleton pattern. We have drawn our inspiration from Eckel [Eck00] to describe the Java source code written below:

```java
public class ControlPanelsFacade {
    static final private ControlPanelsFacade theInstance =
        new ControlPanelsFacade();
    private ControlPanelsFacade() {}

    /**
     * Returns the only shared instance of the class
     */
    static public ControlPanelsFacade theInstance() {
        return theInstance;
    }
}
```

The private constructor `ControlPanelsFacade` ensures that the direct creation of the facade by using the constructor is forbidden. Remember that if you do not explicitly create at least one constructor, the default constructor will be created automatically and it may be used to instantiate several facade objects. The object of class `ControlPanelsFacade` is created as a static final private member of `ControlPanelsFacade`. So, there is one and only one `ControlPanelsFacade` object, and you cannot get at it except through the public static method `theInstance`.

4.3.4 Proxy Design Pattern

Power Config needs to communicate with an amplifier through the serial or USB cable. Somehow, we have to control accesses to this amplifier. The intent of the Proxy design pattern\(^\text{33}\) is to “provide a surrogate or placeholder for another object to control access to it” [GHJV95]. That sounds to be exactly what we need to handle the communication with the amplifier.

The Power Config requirements specify that the application should be able to control a Power G-16 system. Hence we have developed an G-16 proxy, depicted in Figure 4.11, that acts as a stand-in for the real Power G-16 amplifier.

This proxy provides a local representative for the amplifier and it forwards requests directly to the real amplifier. It is thus responsible for encoding re-

\(^{33}\text{Readers are referred to p.155 (Appendix A) for a description of the Proxy design pattern.}\)
quests and their arguments and for sending the encoded requests to the amplifier through the serial/USB port. It allows developers to program as if they were directly handling the “physical” amplifier.

The proxy may be responsible for checking if the amplifier is in working order and for throwing exception when a request cannot be handled by the device. This exception mechanism ensures the reliability of Power Config. It ensures the correct behavior of the application in the event of amplifier errors. In order to increase the performance of the application, it may also cache the recently accessed data and forward request to the amplifier only when appropriate, depending on the time elapsed between two identical requests.

4.3.5 Decorator Design Pattern

Fowler [Fow02] has provided us with the inspiration for writing this section. Any developer who has built an application that handles an external device can speak of the frustration of being dependent upon resources completely out of his control. Feature delivery, reliability, and performance are all unpredictable. There is no getting away from it.

Such issues may slow down the development process of Power Config. It may compel developers to put some hacks into the code to compensate the incompleteness and/or the defectiveness of the amplifier. Much worse, it may break the development process because tests cannot run.

Concerned about these issues, we have emphasized the need for testability. Indeed, we want to keep the development completely within the control of the application team. Furthermore, we want to promote the effortless integration of temporary debugging code.
4.3 Improvements of the Architecture

Obviously, all these issues should be handled at the proxy level\textsuperscript{34}. The proxy is a strategic place. The errors that occur at this level may either come from the amplifier or from the application. In addition, the amplifier may be instable. Developers should often write temporary code to print out data that go through the proxy. It is thus a good practice to share this temporary code among developers. However, some developers do not want to be bothered with these printouts. Unfortunately, that is often extremely difficult to render inoperative the temporary code. We need to find a way to add/remove without difficulty this debugging code. Moreover, when a bug is detected in the amplifier, developers need to put some hacks into the code to emulate the amplifier. These developers are often interested in distinct parts of the application and write their own emulator. It would be very interesting to be able to combine these emulators easily. Ideally, all these features would be available both with and without the cache.

In summary, we should provide a proxy that forwards requests to the real amplifier. If needed, this proxy may cache information. In case of amplifier defectiveness or absence, we should provide a stand-in implementation of this device that allows its partial or complete simulation. Indeed, if certain functionalities are working properly, developers may want to test them while the others are emulated. Finally, the error-prone proxy is the ideal place to integrate temporary debugging code.

This is a real challenge to handle all these aspects without ending in a messy implementation. Fortunately, we have found a design pattern that has enabled us to take up this challenge. This is the Decorator design pattern\textsuperscript{35} that we have slightly adapted.

The intent of the Decorator pattern is to “attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality” [GHJV95].

We apply this pattern to attach additional responsibilities to the proxy. The simplest proxy, without any extra responsibilities, is the peer. It just blindly forwards requests to the real amplifier without doing additional work. We can add responsibilities to the proxy dynamically and transparently, that is, without affecting the other objects related to it. For instance, it may become responsible for caching the recently accessed data and forwarding requests to the amplifier only when appropriate. It may become responsible for writing into a log file all the proxy operations and their results in order to provide a powerful debugging tool. It may become responsible for emulating non-reliable amplifier functions. We can still imagine a lot of other decorations that are suited for the proxy\textsuperscript{36}.

\textsuperscript{34}Remember that the proxy may also be responsible for caching the recently accessed data.

\textsuperscript{35}Readers are referred to p.152 (Appendix A) for a description of the Decorator design pattern.

\textsuperscript{36}We can add a decoration that records statistic information about the accesses to the amplifier. We can implement a fault-tolerant decorator. This decorator would be responsible for detecting hardware defects and adapting transparently the proxy in run-time to ensure, at least, the emulation of the defective electronic functions...
As far as the last responsibility is concerned, purists may reproach us for going beyond the intent of this pattern and we agree with them. That is the reason why we have specified that we have slightly adapted the Decorator pattern according to our needs. Indeed, a decorator is supposed to attach additional responsibilities. When we emulate part of the amplifier, we add a new responsibility to the proxy but we forget about responsibilities of the decorated object. In spite of this remark, we consider that the emulator is a kind of decorator especially when we partially emulate the amplifier. Subsequently, in order to be consistent with the Decorator pattern notation, we keep adding to the Emulator classes the suffix `Decorator`.

We can control how and when to decorate the proxy. All we have to do is to enclose the proxy in another object that adds a specific decoration. The originality of this pattern lies in the fact that the decorator conforms to the interface of the proxy so that its presence is transparent to the proxy’s clients. This transparency “lets you nest decorators recursively, thereby allowing an unlimited number of added responsibilities” [GHJV95]. You can even add several times the same decorator.

As a general rule, the decorator forwards requests to the decorated component and may perform additional actions before or after forwarding. That is exactly the case of the proxy decorator responsible for writing into a log file the proxy operations. However, our implementation of the Decorator pattern differs slightly from that of the GoF. Under certain conditions, we do not forward requests to the underlying component. That is the case, for instance, when the requested information is supposed to be up to date in the cache or when we emulate defective electronic functions. Unfortunately, this implementation imposes some restrictions. Unlike the GoF implementation, we should be careful when we nest decorators recursively. We should always keep in mind that a cache decorator or an emulator decorator may deactivate previously added responsibilities. On the other hand, this deactivation is partial according to the available data in the cache and the defective electronics functions. Moreover, in the case of the proxy, we may benefit from these restrictions. We may decide, for instance, that the only non-emulated operations should appear in the log file or, inversely, that all the proxy related operations should be registered. All we have to do to achieve these opposite behaviors is to invert the way we decorate the proxy.

Figure 4.12 depicts a potential configuration of the proxy. This figure emphasizes the nested structure of the proxy.

All proxy configurations contain at least the peer object. It is responsible for encoding requests and their arguments and for sending the encoded requests to the amplifier through the serial/USB port. It may also generate exception when errors occur at the amplifier level. All the other responsibilities are optional. In this configuration, we have chosen to keep a log file of the operations effectively sent to the amplifier. The log decorator thus directly encloses the peer object. Then we decorated the proxy with a cache. This cache may filter requests that will not be forwarded to the log decorator, depending on the time elapsed be-
4.3 Improvements of the Architecture

Figure 4.12: Proxy Nested Structure

tween two identical requests. After that, we added successively two independent emulators. These emulators have been written independently by two developers working on different part of Power Config. The first programmer is interested in electronic functionalities related to the input channels of the amplifier while the second works on the output channels. Both have written an emulator to compensate amplifier defects. So, we have mixed these two emulators to obtain a new emulator with an enlarged scope. Finally, we have decorated the proxy with another log decorator. This time, we want to keep track of all the operations related to the proxy.

The class diagram depicted in Figure 4.13 describes the types of objects that participate in the Decorator pattern and the static relationships that exist among them. The root of this class hierarchy is the G16Proxy interface. All the objects that can have responsibilities added to them dynamically compel to this interface.

Both the G16Peer class and the Decorator abstract class implement directly this interface. That means that both G16Peer objects and Decorator objects can be decorated. However, unlike the G16Peer class, the Decorator class cannot be instantiated because it is abstract. It is thus intended to be subclassed by concrete decorator classes. Nevertheless, it is responsible for keeping a reference to an object that its concrete subclasses will decorate. In addition, it defines the default behavior of a decorator, that is, it delegates all requests to the decorated object. Here is an extract from the source code of the Decorator abstract class:

```java
1   public abstract class Decorator implements G16Proxy {
2   
3       public Decorator(G16Proxy G16Proxy){
4           this.G16Proxy = G16Proxy;
5       }
```

37 Readers are referred to Appendix B for a description of the UML notation.
This default behavior (line 9) allows Decorator concrete subclasses to override only the requests to which additional operations should be performed before and/or after forwarding the requests. We notice that the decorated object
to which requests are delegated is passed into the constructor of this abstract class. That means that, in the concrete subclasses constructor, we must explicitly write the call to the base-class constructor using the super keyword and the appropriate argument in order to initialize the base-class.

Four classes extend the Decorator abstract class. These subclasses are the G16LogDecorator, the G16CacheDecorator, the G16EmulatorIDecorator, and the G16EmulatorIVDecorator. We describe an extract from the source code of the first three. We begin with the source code of the G16LogDecorator class:

```java
1   public class G16LogDecorator extends Decorator {
2   
3       public G16LogDecorator(G16Proxy G16Proxy, String fileName) {
4           super(G16Proxy);
5           try {
6               file = new FileWriter(fileName);
7           }
8       } catch (IOException e) {} 
9       
10      
11       public DSPGain readParamDspPreGain(int channel) throws G16Exception {
12           DSPGain dspGain = null;
13           try {
14               file.write("---> " + System.currentTimeMillis() +
15                   " : readParamDspPreGain(" + channel + ") : ");
16               dspGain = super.readParamDspPreGain(channel);
17               file.write(dspGain + 
18                   );
19           } catch (G16Exception e) {
20               try {
21                   file.write(e.getMessage() + "\n");
22               }
23               catch (IOException ex) {} 
24               throw e;
25           } catch (IOException e) {} 
26           return dspGain;
27       }
28   
29   
30       private FileWriter file;
31   }
32
33
```

This subclass overrides all the methods of the Decorator abstract class. Indeed, for each request additional work should be performed. This work involves writing the operation and its result in a log file (line 15 and 18). The G16LogDecorator superclass (the Decorator abstract class) is responsible for forwarding the request to the decorated object (line 17). This decorated object is passed into the G16LogDecorator constructor. Then, the G16LogDecorator
constructor passes it into the `Decorator` constructor using the `super` keyword (line 4) and the `Decorator` class maintains a reference to it. This initialization mechanism is similar for all the concrete subclasses that extend the `Decorator` abstract class. The name of the log file is also passed into the `G16LogDecorator` constructor. It is used to create the log file (line 6). Finally, we have chosen to handle amplifier exceptions in order to print them into the log file (line 22). However, when we catch an amplifier exception we throw it again (line 25) so that proxy’s clients in their turn can handle it. Now, we examine the `G16CacheDecorator` class:

```java
public class G16CacheDecorator extends Decorator {

    public G16CacheDecorator(G16Proxy G16Proxy, long lifeTime) {
        super(G16Proxy);
        this.lifeTime = lifeTime;
        dspPreGainTable = new DSPGain[INPUT_CHANNEL_NUMBER];
        dspPreGainTimeStampTable = new long[INPUT_CHANNEL_NUMBER];

    }

    public void setLifeTime(long lifeTime) {
        this.lifeTime = lifeTime;
    }

    public long getLifeTime() {
        return lifeTime;
    }

    public DSPGain readParamDspPreGain(int channel) throws G16Exception {
        long newTime = System.currentTimeMillis();
        if (newTime > (dspPreGainTimeStampTable[channel] + lifeTime)){
            dspPreGainTable[channel] = super.readParamDspPreGain(channel);
            dspPreGainTimeStampTable[channel] = System.currentTimeMillis();
        }
        return dspPreGainTable[channel];
    }

    public void writeParamDspPreGain(int channel, DSPGain dspGain) throws ElvisException {
        super.writeParamDspPreGain(channel, dspGain);
        dspPreGainTable[channel] = dspGain;
        dspPreGainTimeStampTable[channel] = System.currentTimeMillis();
    }

    private long lifeTime;
}
```

Like the `G16LogDecorator` class, the `G16CacheDecorator` class overrides all
4.3 Improvements of the Architecture

the methods of its superclass. Besides the decorated object, the G16CacheDecorator constructor receives as argument the lifetime of the information stored within the cache. This lifetime is used to determine how long we can reasonably think that the cached data is identical to the amplifier data\textsuperscript{38}. When a read request is received, we first check if the available cached data is supposed to be up to date (line 22). If so, we reply to the read request with the cached data\textsuperscript{39}. Otherwise, we forward the request to the decorated object through the base-class and we update the data of the cache (line 23). In the case of a write request, we cannot postpone the update of the amplifier state. Hence we automatically forward the request (line 31) and we update the cache (line 32). Finally, it is possible to modify the lifetime parameter through the setLifeTime method. Unlike the other methods, this one and the getLifeTime are specific to this concrete class. Consequently, we cannot access these methods through the G16Proxy interface. We thus need to know that we handle directly a cache that adds responsibilities to the proxy when we want to invoke these methods. Here is the source code of the G16EmulatorIDecorator class:

```java
public class G16EmulatorIDecorator extends Decorator {
    public G16EmulatorIDecorator(G16Proxy G16Proxy) {
        super(G16Proxy);
        dspPreGainTable = new DSPGain[input_CHANNEL_NUMBER];
        for (int i = 0; i < input_CHANNEL_NUMBER; i++) {
            dspPreGainTable[i] = new DSPGain(0,-60,-30,-20,false);
        }
        ...
    }

    public String getDescription() {
        return description;
    }

    public DSPGain readParamDspPreGain(int channel) {
        return dspPreGainTable[channel];
    }

    public void writeParamDspPreEqTrim(int channel, DSPEQTrim dspEqTrim)
    throws G16Exception {
        ...
    }

    private String description = "Provides a complete description of this emulator."
}
```

This emulator is pretty simple and would not be very useful in practice. However, it shows you how easy it is to implement an emulator. We just need

\textsuperscript{38}The state of the amplifier is updated by the amplifier itself.

\textsuperscript{39}In this case, our implementation differs from the original pattern because we do not forward the request to the decorated object.
Design Patterns Improve the Architecture

to override the methods related to the defective electronic functionalities. This emulator class intercepts those requests and emulates the amplifier. In this example, we emulate the `readParamDspPreGain` and `writeParamDspPreEqTrim` methods. Nevertheless, if we want a more accurate emulation, we may also need to override the `writeParamDspPreGain` and `readParamDspPreEqTrim` methods. In this case, we would forward these two methods to the decorated object and we would save their arguments or result depending on whether it is a write or a read method. You can imagine all the features that could be added to the emulator...

Because they compel to the `G16Proxy` interface, all those decorators can both decorate and be decorated. The `G16Peer` class is a little bit different. Indeed, like the decorators it implements the `G16Proxy` interface. Hence it can be decorated. But it does not extend the `Decorator` abstract class. Hence it cannot decorate. That is absolutely normal because it receives requests defined in the `G16Proxy` interface and encodes them before sending it to the amplifier. It is thus always the last link in the chain and is not intended to decorate other objects. In conclusion, all proxy configurations contain at least the `G16Peer` object to which additional responsibilities can be attached. The Java code describes how to reproduce the above nested structure of the proxy and to invoke the `readParamDspPreGain` on the proxy:

```java
1  G16Proxy peer = new G16Peer();
2  G16Proxy log1 = new G16LogDecorator(peer, "log1.txt");
3  G16Proxy cache = new G16CacheDecorator(log1, 100);
4  G16Proxy emulator1 = new G16EmulatorIDecorator(cache);
5  G16Proxy emulator2 = new G16EmulatorIVDecorator(emulator1);
6  G16Proxy log2 = new G16LogDecorator(emulator2, "log2.txt");
7  G16Proxy G16Proxy = log2;
8
9  try {
10     dspGain = G16Proxy.readParamDspPreGain(1);
11  }
12  catch(G16Exception e) {
13      e.printStackTrace();
14  }
```

Thanks to the Decorator pattern, we can attach additional responsibilities to the proxy. The proxy can now be easily configured according to developers needs. This ensures the testability of the Power Config application despite the fact that the amplifier is completely out of software team’s control.

### 4.3.6 Adapter Design Pattern

In this section, we focus on the changeability non-functional property. We are especially interested in the extensibility of the application. Design for change is
a key aspect of software development. We want to be able to meet new requirements with no trouble. The difficulty lies in the fact that we must anticipate which aspects of the application may evolve in the future.

In the case of Power Config, we expect that future versions of the application should handle new kinds of amplifiers. We want to be able to adapt easily the application to a variety of amplifiers. We have already anticipated this potential requirement and we have developed in the amplifier surrogate subsystem a domain layer that models a generic amplifier. This model, called the device model, can be configured to match all kinds of amplifiers. With the exception of the hardware interface layer, the whole application is driven by this device model. All the components of the application (the polling system, the GUI, the mediators . . . ) except the hardware interface layer are tailored to this generic model. That means that, if we want the application to control another amplifier than the G-16, all we have to do is to rewrite the hardware interface layer that would contain a new proxy.

Nevertheless, we are faced with a difficulty: the domain layer communicates directly with the proxy that acts as a stand-in for a specific amplifier. But the proxy interface does not match the domain-specific interface that is required by the domain layer. Indeed, the domain layer that models a generic amplifier expects generic methods such as `getPreGain`, `setPostMid` to control the real amplifier. Unfortunately, the proxy interface defines low-level methods such as `readParamDspPreGain`, `writeParamDspPostEqTrim` and so forth. If the domain layer handles directly those low-level methods, it will not be reusable at all because each time we want to control a new kind of device we have to rewrite a great part of this layer.

Once again, the GoF book provides a pattern that solves this problem. The
Adapter design pattern helps us to ensure the changeability of Power Config. Its intent is to “convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces” [GHJV95]. Each time we want to control a new kind of amplifier, we only have to write the proxy that acts as a stand-in for the new amplifier and the adapter using the Adapter pattern that converts the proxy interface into the generic interface required by the domain layer.

In the following, we analyze our implementation of the adapter that converts the G16Proxy interface into the interface that the domain layer requires. The G16Proxy interface is the adaptee, the target interface is called the Amplifier interface, and the adapter is a concrete class called G16ProxyAmplifierAdapter class. Figure 4.15 illustrates the class diagram of those elements.

![Figure 4.15: G16Proxy to Amplifier Interface Adapter Class Diagram](image)

The Amplifier interface defines the generic methods that the domain layer

---

40 Readers are referred to p.151 (Appendix A) for a description of the Adapter design pattern.
41 Readers are referred to Appendix B for a description of the UML notation.
4.3 Improvements of the Architecture

invokes to control the amplifier. The G16ProxyAmplifierAdapter class implements all those methods to adapt the G16Proxy interface to the Amplifier interface. In order to adapt those interfaces, the G16ProxyAmplifierAdapter class keeps a reference to an instance of a class that implements the G16Proxy interface. Here is an extract of the source code of the G16ProxyAmplifierAdapter class:

```java
public class G16ProxyAmplifierAdapter implements Amplifier {
    public G16ProxyAmplifierAdapter(G16Proxy G16Proxy) {
        this.G16Proxy = G16Proxy;
    }

    public int getPreGain(int channel) throws AmplifierException {
        int gain = 0;
        try {
            gain = G16Proxy.readParamDspPreGain(channel).getGain();
        } catch (G16Exception e) {
            throw new AmplifierException("AmplifierException : getPreGain(" + channel + ")");
        }
        return gain;
    }

    public void setPreGain(int channel, int gain) throws AmplifierException {
        try {
            DSPGain dspGain = G16Proxy.readParamDspPreGain(channel);
            dspGain.setGain(gain);
            G16Proxy.writeParamDspPreGain(channel, dspGain);
        } catch (G16Exception e) {
            throw new AmplifierException("AmplifierException : setPreGain(" + channel + ", " + gain + ")");
        }
    }

    private G16Proxy G16Proxy;
}
```

When the domain layer invokes the getPreGain method, the G16ProxyAmplifierAdapter translates this method into readParamDspPreGain that it delegates to the proxy (line 11). However, the readParamDspPreGain method sends back a complex type that contains a bunch of information from the amplifier. The adapter selects the requested information and, in turn, sends it back to the domain layer (line 11). The setPreGain method is a little bit more complicated
because the G-16 amplifier does not allow the individual setting of single values. Each time we want to adjust a single amplifier parameter, we have to provide the amplifier with a record that contains the value of this parameter and other related parameters. In order to avoid erasing the current values of these related parameters, we first read the old record from the amplifier (line 23) and then we adjust the new specific parameter within the record (line 24) before forwarding the whole record to the amplifier (line 25).

Successive readings of distinct parameters that belong to the same record may be inefficient. Indeed, for each of these parameters we have to read the whole record separately. For this reason, we recommend decorating the proxy with a cache in order to avoid sending useless get requests to the hardware.

When an error occurs at amplifier level, the proxy generates an G16Exception. It is a very good practice to handle these exceptions within the adapter in order to transform these low-level exceptions into higher level exceptions (line 28). Hence the domain layer will not be confronted with errors messages that apply to the hardware interface layer.

In summary, the domain layer can be developed without any knowledge of the proxy interface. Moreover, the proxy interface does not have to compel to a generic interface which would have been impossible because the proxy is a surrogate for a real amplifier. It has thus to provide an interface identical to the amplifier interface so that it can be substituted for the amplifier. In conclusion, the adapter mechanism promotes the extensibility of the Power Config application by encapsulating the coupling between the domain layer and the hardware interface layer. This loose coupling allows substitution of the hardware interface layer without impacting other aspects of the architecture.

4.4 Summary

In this chapter, we presented design patterns that are proven design solutions to recurring problems within particular contexts.

We defended our position with regard to topics such as pattern tools and pattern languages. Like Buschmann et al. [BMR+96] and Fowler [Fow02], we think that formal methods do not apply to patterns. Hence, according to us, design patterns are not suitable to be manipulated automatically. As far as pattern languages are concerned, even if it is true that design patterns are not isolated from each other, we do not believe that, in the near future, such language might exist because there are missing patterns to cover every aspect of software architectures.

Afterward, we described the differences between design patterns and frameworks and we presented the polling system framework that we implemented within the context of the amplifier surrogate subsystem. This framework takes
advantage of the Template Method design pattern to promote the reusability of the code.

In the second part of this chapter, we examined in particular four non-functional properties: changeability, reliability, testability, and reusability. Then, we showed how design patterns enabled us to take up the challenge of providing the amplifier surrogate subsystem with those non-functional properties. These proven solutions have thus strongly improved the quality of our subsystem. Indeed, the aforementioned non-functional properties of the amplifier surrogate subsystem are indispensable for the success of the whole Power Config project. However, designing with patterns requires both experience in matching a specific design problem to a design pattern and creativity to figure out how to apply it according to specific circumstances.

We have exploited both the Mediator and Facade design patterns in order to “encapsulate” the coupling between the amplifier surrogate subsystem and other subsystems. Through an exception mechanism, the Proxy design pattern ensures the reliability of the subsystem in the event of amplifier errors. The Decorator design pattern enables developers to configure the proxy according to their needs. Lastly, the Adapter design pattern allows developers to adapt without difficulty the amplifier surrogate subsystem so that it handles new kinds of amplifiers.
Chapter 5

Elements of Reusable Graphical User Interface Components

...If you ever want to develop a custom look-and-feel, you’ll undoubtedly have to do some homework before you start writing code. It is a challenge. But, as challenges go, it is a rewarding one...

Mark Andrews [And02]

When it comes to writing Graphical User Interfaces (GUI), programmers have some alternatives. They can just create their windows using Sun’s API\(^1\) to hand-code what they want. Another approach is to use a tool that produces the code for you (this is what we call GUI builders); you just draw the interface you want, mix some graphical components together, and ask the tool to generate the code. If you think you need more, you can also buy some customizable components from companies that are specialized in creating new Swing components or, finally, you can create your own library of reusable Swing components.

The latter approach will be explored in depth in this chapter. Each approach has its benefits and drawbacks. Everything depends upon the result you want and the time and budget you have.

The most time-consuming approach is to create reusable Swing components yourself. Of course, this has huge benefits: you fully understand the code, you have graphical components tailored for your company, and with the customization you need. It is even possible to invent your own look-and-feel: Swing

---

\(^1\)Sun is a short name for Sun Microsystems, INC. It is the company that developed Sparc microprocessors, the Solaris operating system, and the Java programming language. Here, Sun’s API refers to Java.
allows you to dynamically change the appearance and behavior of the User Interface through a feature called “pluggable look-and-feel”. Therefore, a Swing component can have different appearances and behaviors just by changing the look-and-feel.

If you are lucky enough to find what you need in a third party library, you might want to save time by buying some customizable components instead of creating them. Unfortunately, these components are often too generic; you end up with a collection of components that are overly complicated, too customizable and with too many options. Even if the source code is provided, it can be difficult to change something due to the complexity of the code. On the other hand, if the component is properly coded and no changes are required, it might be worth the price.

GuI builders (visual programming environments such as JBuilder, Visual Cafe, and Forte for Java) speed the development of your graphical interface. They allow you to experiment with more things, to try out more designs so you often come up with a better solution. The simplicity and well-designed structure of Swing allows recent GUI builders to generate comprehensible code. You are therefore able to hand-change the code afterwards. The big disadvantage of those tools resides in what they cannot do: if you want to write a GUI that is very specific, with home-made components, you can be limited by the GUI builder itself. Everybody knows the adage of WYSIWYG: what you see is what you got (and not what you get). GUI builders do not allow you to do everything: Albeit that they are becoming more and more powerful, they are still too limited for some applications.

Finally, Java Swing library contains a large assortment of very flexible visual components, and in many cases a user interface can be designed using only those components. This is of course the least expensive method but you will often end-up with a very common look, nothing really proprietary.

After exploring those alternatives, we have found creating our own Swing components interesting. We have looked for books that cover the subject but did not succeed. We found lots of introductions to the subject, a lot of clues, but no tutorial explaining how to create reusable components from scratch.

Having experienced the creation of some widgets, we thought we could develop a method to create new reusable Swing components. This chapter is a first attempt to provide such a method. Unfortunately, we cannot cover everything; this chapter must therefore be seen as a collection of advice. We will first present the architecture of Swing components, the actors and responsibilities of such a component, and then examine the “pluggable look-and-feel” mechanism. Finally, we present a flexible step-by-step approach to create new Swing components. We also emphasize some pitfalls to avoid when applying this method.
5.1 The Model-View-Controller Architecture

Before we dive into the subject, a basic understanding of the underlying architecture of Swing components is necessary. Sun states that graphical components must be built around a Model-View-Controller (MVC) architecture. This is a special form of the Observer pattern defined in 1994 by the GoF.

5.1.1 The Observer Pattern

According to the Gang of Four, the purpose of the Observer 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” [GHJV95]. This pattern, also called Dependents or Publish-Subscribe, involves two key objects: the subject and the observer. There is only one subject; it holds references to its observers. When a subject changes its state, it notifies the observers who can then decide what action to accomplish. The observers could update themselves according to these changes or trigger any other actions. To clarify this, a common class diagram of the actors involved in this pattern is shown on Figure 5.1.

![Figure 5.1: The Observer Design Pattern](image)

The intent of this pattern is to decouple the subject and its observers. Only one class of observer, ConcreteObserver, is drawn on the diagram but, in most applications, you will have several concrete observers that tend to belong to
different classes. Albeit that the concrete observers have different types, all of them must implement a common interface (Observer on the diagram). This interface declares an update method that all observers will have to implement: this is where we code the actions to accomplish when the observer is notified. The subject keeps track of its observers and implements a notify method that iterates through the list of registered observers and calls the update method for each of them. The observers register or unregister themselves with the subject by respectively calling the addObserver or removeObserver method.

Observers might want to have additional information about the state of the subject or the type of change that has occurred. This allows them to trigger different actions depending on the type of event. We can accomplish this by passing additional information along with the update method: it can be either an object representing the event triggered or a reference to the calling subject. In this last case, the subject must provide some methods to allow observers to access its state. This is represented by the getState method on the diagram.

5.1.2 The Model-View-Controller

The Model-View-Controller is a special form of the Observer pattern. “The model part of the MVC holds the state of a component and serves as the subject. The view part of the MVC serves as the Observer of the Subject to display the model’s state. The view creates the controller, which defines how the user interface reacts to user inputs” [Zuk99].

As Figure 5.2 illustrates, the controller translates user inputs into changes in the model or in the view. For instance, if a user types a character in a text field, the controller must insert the new character in the model. The latter will then notify the view to reflect the change. On the other hand, if a user just moves the caret, the controller will directly update the view. The model is therefore not aware of this event.
The MVC architecture decouples models from both their graphical representation and the controller. You can therefore support multiple views of the same data model and even change the graphical representation of a model at runtime.

The implementation of a JButton in Swing is a good example of a MVC architecture. In addition to the MVC elements, each Swing component has a wrapper class that allows the user to perform the most common actions. For a JButton, the wrapper class is, as expected, JButton. The model part is implemented as an interface ButtonModel and its default implementation DefaultButtonModel. Note that this model is also used by checkboxes, radio buttons... which demonstrates the loose coupling between the model and its counterparts. The view is implemented in the BasicButtonUI class and the controller in the BasicButtonListener class. Note that those two parts are not separated in other Swing components, as we will see shortly.

While MVC is often described in the context of GUI development, it can be used for many other purposes. “The view is an interpretation of the model (perhaps a subset)—it doesn’t need to be graphical. The controller is more of a coordination mechanism, and doesn’t have to be related to any sort of input device” [HT00]. The general responsibilities of each part can therefore be described as follows:

**model:** The abstract data model representing the target model. The model has no direct knowledge of any views or controllers [HT00].

**view:** a way to interpret the model. It subscribes to changes in the model and logical events from the controller [HT00].

**controller:** a way to control the view and provide the model with new data [HT00].

### 5.1.3 The Modified MVC Pattern from Swing

We have already seen that the view and controller parts of a MVC component are tightly coupled. Swing has even collapsed these two entities into a single user-interface object called UI delegate. This new modified-MVC design is sometimes referred to “a separate model architecture” [Fow00]. Figure 5.3 shows how those parts are related to each other.

Depending on their type, Swing models use two approaches to notify any interested parties when their data changes: they can either send a “lightweight” notification or a “stateful” notification [Fow00]. A lightweight notification is represented by an event containing no more information than a reference to the

---

2If you are wondering what is the real purpose of those wrapper classes, we give more details in the next section.
calling model\(^3\). This requires the listener to respond by sending a query back to
the model to find out what has changed. A stateful notification describes more
precisely how the model has changed. For instance, it could provide information
such as the old value and new value of a model field that has changed. This
alternative requires a new event instance for each notification\(^4\).

A stateful notification mechanism for properties is already implemented in
Java which you can reuse. A property is actually a private field of a class for
which you provide methods to get and set the value following some naming
conventions. This mechanism is implemented as the \texttt{PropertyChangeSupport}
class, the \texttt{PropertyChangeListener} interface, and the \texttt{PropertyChangeEvent}
class found in the \texttt{java.beans} package. A diagram is shown on Figure 5.4

A subject can use a \texttt{PropertyChangeSupport} object to let interested ob-
servers register for notification of changes in its properties. The subject dupli-
cates the interface of the \texttt{PropertyChangeSupport} class and forwards methods
call to it. A listener object that implements the \texttt{PropertyChangeListener}
interface plays the role of the observer. The term listener is just another name for
an observer. A \texttt{PropertyChangeEvent} encapsulates a reference to the source
object, the name of the property, the new value and the old value of the property.

\(^{3}\text{A lightweight notification is actually an instance of the \texttt{ChangeEvent} class contained in the }\texttt{javax.swing.event} \text{ package. The listener class is implemented in the \texttt{ChangeListener} interface of the same package. This interface has a single generic method (public void stateChanged(ChangeEvent e))}.$

\(^{4}\text{Example of such events in Java are: ListDataEvent, ListSelectionEvent, ListDataEvent, TreeModelEvent, TreeSelectionEvent, TableModelEvent, TableColumnModelEvent, DocumentEvent and UndoableEditEvent.}$

Figure 5.3: Modified Model-View-Controller Architecture of Swing Components
5.2 Actors and Responsibilities of a Swing Component

By now, you have seen that each Swing component is made of several classes contained in different packages. We will describe the main responsibilities and naming conventions of each of those classes. Referring to our introduction on object-oriented programming, we are now at the specification level.

Let’s say that we want to create a new component called \texttt{NewComp}. Figure 5.5 illustrates the relationship between all the classes that are part of that new component.

The model part of that object is usually separated into three classes: an interface \texttt{NewCompModel}, an abstract class \texttt{AbstractNewCompModel} and a default implementation \texttt{DefaultNewCompModel}.

\texttt{NewCompModel} defines the interface of the model that will be exposed to its clients. All data models of the \texttt{NewComp} must implement this interface. Here is the list of its responsibilities:
- enumerate add and remove listeners methods (the argument type of those methods indicates the listener interface to use). If we follow Sun's coding style, the name of those methods must comply with the following pattern: \texttt{addNewCompModelListener} and \texttt{removeNewCompModelListener}, respectively.
- enumerate methods to get the state of the data (often referred as get meth-
ods).
- enumerate methods to modify the state of the data (often referred as set methods).

AbstractNewCompModel implements part of the model interface. This class handles the list of listeners and the notifying methods\(^5\). More precisely, it is responsible for the following:
- hold a list of listeners interested in knowing when the model state has changed\(^6\)
- define add and remove methods allowing listener objects to register and unregister themselves.
- define protected “fire” methods. This set of methods corresponds to the notify method of the Observer design pattern. Indeed, they iterate through the list of listeners and call the appropriate method for each one. Each fire method corresponds to one method declared in the listener interface.

DefaultNewCompModel extends the abstract model and provides a default implementation of the data model. This is the model that will be used if no other implementation is provided by the client. This class must implement the remaining model interface. Here are the responsibilities:
- encapsulate the container objects that hold the data
- provide a constructor that fills the entire model
- provide an implementation of get and set methods

The model can fire events, we therefore need to define an event class, NewCompModelEvent and the corresponding listener interface NewCompModelListener\(^7\). It is usually a good practice to have only one kind of event object for a model, so only one listener interface must be implemented to deal with all the changes that could occur.

NewCompModelEvent must always extend java.util.EventObject. The implementation of this root class is fairly simple; it is an immutable object providing access to the source object on which the event initially occurred. Therefore the implementation of a NewCompModelEvent is subject to the following constraints\(^8\):
- it must implement constructors that characterize model events using appropriate fields. This means that we need non-empty constructors which set the fields of the event.
- the first argument of any constructor must be the source object (this allows the constructor to call super(source) at first)
- it must provide get methods to fields of the event. An event being immutable,

\(^5\)If the model consists only of a list of properties, as is often the case, you can reuse the aforementioned property-change propagation mechanism implemented in java.beans.PropertyChangeSupport.
\(^6\)If you do not want to implement your own list of listeners, you may reuse the existing class javax.swing.event.EventListenerList.
\(^7\)If the abstract model uses the PropertyChangeSupport mechanism, you have to reuse the existing class java.beans.PropertyChangeEvent and its corresponding listener interface java.beans.PropertyChangeListener.
\(^8\)If you want a concrete example, you can look at the java.beans.PropertyChangeEvent source code.
it is forbidden to provide set methods.

`NewCompModelListener` must inherit the empty interface `java.util.EventListener` which is a tagging interface that all event listener interfaces must extend. The `NewCompModelListener` defines notification methods using the following pattern: `somethingHappened(NewCompModelEvent e)`\(^9\). Each method corresponds to one category of event that can be fired by the model.

```
interface NewCompModelListener
{
    void somethingHappened(NewCompModelEvent e);
}
```

Figure 5.5: Actors Relationships of the Swing Component `NewComp`

The UI delegate part is usually implemented in two classes: `NewCompUI` defines generic behaviors of the component and `BasicNewCompUI` implements the basic look-and-feel.

- `NewCompUI` defines the precise interface that a component can use to interact with the UI delegate instance. It is an abstract class\(^10\) and must extend `javax.swing.plaf.ComponentUI` which is the base class for all UI delegate instances.

\(^9\)If you want a concrete example, you can look at the `java.beans.PropertyChangeListener` source code.

\(^10\)"Interfaces were originally used here, but they were replaced with abstract classes because the API was not mature enough to withstand the concrete casting of an interface" [Fow00].
objects in the Swing modified-MVC architecture. Most of the time, you will leave this class empty because `ComponentUI` already contains all the important methods to interact with a delegate.

The design and implementation of `BasicNewCompUI` is complex due to the large scope of its responsibilities. The UI delegate object for a Swing component is responsible for implementing the aspects of the component that depend on the look-and-feel:
- paint the component on the screen. This task includes setting font, color, border, and opacity properties on the component, installing a layout manager\(^\text{11}\) and adding any appropriate child subcomponents to the component.
- listen to model events and update the view accordingly.
- listen to user events (mouse, keyboard, or any input devices) and update the model and the view accordingly.

The last actor of a Swing component is the wrapper class which is, in this case, `NewComp`. This class usually extends `javax.swing.JComponent` and is sometimes considered as part of the delegate. To avoid misunderstandings, we will use the term “component” or “wrapper”, reserving the word “delegate” for the two previous classes `NewCompUI` and `BasicNewCompUI\(^\text{12}\)`. To understand the purpose of the wrapper, we need to describe the hierarchy it comes from.

As shown on Figure 5.5, `java.awt.Component` is the parent class of all Swing components. It provides support for a number of frequently used properties such as the size of the component, background and foreground color, the font used for text, the parent container that this component has been added to... This class uses the `PropertyChangeSupport` mechanism to fire `PropertyChangeEvent`s when setting those properties. The `java.awt.Container` class extends `Component` to add the ability to contain other components. It provides a way to set a layout manager that will control the size and positions of the components that will be added in the container \cite{Spe00}. It also defines the insets\(^\text{13}\) of the component. `JComponent` is a subclass of `Container` and provides more useful properties for the client. It adds the ability to set a border to the component, a tool tip text and allows you to set its `preferredSize`, `maximumSize` and `minimumSize`\(^\text{14}\) \cite{Spe00}.

`NewComp` inherits from all those properties and “wraps” all the other parts of the component to form a whole. You might then think that the responsibilities of the wrapper and the UI delegate overlap one another. Indeed, `BasicNewCompUI`

\(^{11}\)A layout manager is responsible for deciding how child components lie based on the order that you add them in the main component. In addition, the layout managers adapt the size of your components to the dimensions of your application window.

\(^{12}\)Note that `BasicNewCompUI` has been chosen arbitrarily. You can rename this class to a more appropriate name, such as `CompanyNewCompUI`, where `Company` is the name of your company.

\(^{13}\)An inset defines a reserved space in which you cannot draw anything, it is usually used for borders and title bars.

\(^{14}\)`JComponent` also provides more sophisticated control of the UI rendering. For instance, it provides a method to decide whether double buffering must be used to draw the component on the screen. These details are beyond the scope of this chapter and will not be covered here.
is responsible for setting background and foreground color properties on the component, and the Component class, which is the parent of NewComp, allows clients to set those color properties... Luckily, this is not the case; the code to set the color remains in the Component class. So, when the delegate wants to set such properties, it invokes methods on the wrapper. This will be explained later in greater details.

The wrapper is therefore responsible for the following:
- create default models
- provide get and set methods to the model
- relay important model methods (including add and remove model listener)
- relay events from models if it seems relevant
- provide get and set methods to the UI delegate
- provide methods to install the delegate
- provide get and set methods to control the view state attributes (labels...)
- use PropertyChangeEvent when setting view attributes

You might also need to create specific events for a NewComp. In that case you will have to define a NewCompListener interface and a NewCompEvent class. Their implementation is similar to the listener and event of the model. Under those circumstances, the NewComp class is also responsible for providing fire methods for those events.

5.3 The Pluggable Look-And-Feel Architecture

Now that we have seen the actors and their collaborations, we will explain the reason why the wrapper and the UI delegate are separated.

For this, we need to understand another Swing mechanism: the pluggable look-and-feel (PLAF). The user of a Swing application can change UI delegates at runtime, so it is possible for the user of a cleverly designed Swing application to change the appearance and behavior of the applications components while the program is running [Fow00]. That is the reason we call it pluggable look-and-feel.

Sun’s pluggable look-and-feel architecture relies on a special form of the Abstract Factory, a creational design pattern. We will see what this pattern involves and how Sun’s developers have adapted it for their specific needs.

5.3.1 The Abstract Factory Pattern

According to the Gang of Four, the purpose of the Abstract Factory pattern is to “provide an interface for creating families of related or dependent objects
In our case, we need to provide an interface for creating families of UI delegates. Indeed, we need a set of UI delegates for each look-and-feel that we want to support. In Java, four families are already implemented: the Mac family which is only supported on MacOS computers, the Windows family which is only supported on Microsoft Windows operating systems, the Motif family, and the Metal Family. The last two are not locked to a particular operating system and can run on any computer. Each family actually encapsulates the implementation of look-and-feel (UI delegates) for all existing Swing Components (JList, JSlider, JScrollPane...).

A standard Abstract Factory implementation for such UI delegates is depicted on Figure 5.6. For simplicity, we have only shown the implementation of the Mac and Windows families for three Swing components. This design pattern, again, encapsulates the concept that varies: the choice of which delegate to use. Note that this is not the real implementation of Swing, as we will see shortly.

The roles of the objects involved in the Abstract Factory design pattern can be summarized as follows:
- The client object just knows who to ask for the objects it needs and how to use them [ST02].
- The Abstract Factory class specifies which objects can be instantiated by defining a method for each of these different types of objects. Typically, an Abstract
5.3 The Pluggable Look-And-Feel Architecture

Factory will have a method for each type of object that can be instantiated [ST02].
- The concrete factories specify which objects are to be instantiated [ST02].

With this architecture, when a client such as a JSlider wants to get its delegate, it will talk to the appropriate factory object, (either WindowsDelegate-Factory or MacDelegateFactory) through its interface AbstractDelegate-Factory. The client, JSlider, will be given either a MacSliderUI or a WindowsSliderUI object. JSlider does not need to worry about whether a Windows or a Mac delegate is returned since it uses both in the same manner through the interface SliderUI.

Note that this design pattern makes heavy use of the “programming to interface” principle explained in Chapter 2. This architecture was also made possible because, for each component, all concrete delegate implementations derive from the same base class. For instance, MacListUI and WindowsListUI both inherit from ListUI. However, if this was not the case, you could adapt the interface of the concrete UI delegate with the help of the Adapter design pattern, described in the previous chapter.

The Abstract Factory pattern has one major drawback: adding new kinds of objects to a family is difficult. Extending abstract factories to produce new kind of UI delegates is not easy. “That’s because the Abstract Factory interface fixes the set of products that can be created. Supporting new kinds of products requires extending the factory interface, which involves changing the Abstract Factory and all of its subclasses” [GHJV95].

Since the list of Swing components is always evolving, Sun could not rely on this architecture as it is. They needed a more flexible approach. The solution is, again, to encapsulate change: we need a mechanism allowing us to extend a family without breaking the code of existing clients.

5.3.2 A More Flexible Factory: The UI Manager

In order to increase flexibility, Sun has replaced the set of methods that were creating the delegates (createSliderUI, createListUI…) by a single generic method with a parameter indicating the kind of object to create. This parameter is actually a reference to the wrapper class for which we want to create the UI delegate. The method is called getUI and returns a ComponentUI object.

While this kind of factory is more extensible, there is a trade-off [GHJV95]: all UI delegates are returned to the client with the same interface as given by the return type. The client will not be able to differentiate or make safe assumptions about the class of a delegate. Indeed, this architecture forces the client to downcast the returned object if it wants to perform subclass-specific operations. In our case, the downcast is safe because a UI Delegate has only
one type of client: its corresponding wrapper class.

Now that the kinds of delegate are not encoded in the operation signatures, the factories need a way to know which instance to create based on the argument passed as a parameter. This information can be kept in a table that maps the wrapper type with the UI Delegate to create. This new architecture is shown on Figure 5.7.

![Figure 5.7: A Flexible Factory: The UI Manager](image)

In this diagram, LookAndFeel is the name of the abstract factory. The concrete factories are implemented as WindowsLookAndFeel and MacLookAndFeel. Note that those classes do not create the UI delegates themselves. Instead, they delegate the job to their UIDefaults table which knows the mapping between wrappers and UI look-and-feel implementations.

Sun has even gone one step further: although clients could directly talk to the LookAndFeel factory to create their UI delegates, they use the UIManager class, a facility which relays demands to the currently installed look-and-feel factory. “That way, if you want to install a new look-and-feel, or change an existing one, you don’t have to tell the Swing components directly—only the UIManager” [Zuk99].
Actually, instead of always relying on the `UIDefaults` table of the concrete factory to create UI delegates, the `UIManager` holds its own `UIDefaults` whose entries can be changed at runtime. When the client sets a new look-and-feel, the `UIDefaults` table of the UIManager will be filled with the new values of the selected look-and-feel. So, conceptually, the `UIManager` can also be considered as the Abstract Factory.

One could argue that this statement does not comply with the original Abstract Factory pattern as `UIManager` is not abstract. We feel that it is just an implementation detail. Indeed, the proposed architecture is fully compliant with the original intent: provide an interface for creating families of related or dependent objects without specifying their concrete classes. Clients can ask the `UIManager` to create a UI delegate without specifying its concrete class and it will delegate the creating process to its defaults table.

As a conclusion, if we want to represent the architecture of a Swing component, we can now represent it in a diagram like in Figure 5.8.

![Figure 5.8: Simplified View of a Swing Component](image)

5.4 The Swing Component Development Process

Now that we have listed the actors and explained the architecture of their relations, you might wonder where to start when implementing such a component. Building a reusable Swing component is an iterative process and can be viewed as a small software development project. The following paragraphs present a method to create reusable components. An example-driven approach has been chosen to explain the different stages you have to go through.
5.4.1 General Specification of the Component

The first step is to define the purpose of the component. You do not need a detailed specification of the graphical representation, however, you have to focus on what the component is designed for, i.e. its real purpose. To illustrate this, we have chosen to present the development of a “selectable list”, a component we had to build during our internship.

The intent of the “selectable list” is to allow the user to select some elements from a list. This could be displayed graphically in many different ways. For instance, you can represent it as two lists, one listing the unselected elements, the other listing the selected ones. In this case, you would need buttons allowing the user to switch elements from one list to another. Another possible representation is a single list of elements with associated checkboxes. We could select and unselect elements by clicking on the checkboxes.

Before deciding which graphical representation to choose, we had to define the goal for which the component is designed. In this case, the definition would be “a component that allows users to select elements from a list”. Based on that description, you can then choose an appropriate name for the component. We chose SelectableList.

5.4.2 Implementing the Model

Once you have defined the intent of the component, you can start to write the interface of the model. Two things must drive your thinking process while writing this interface:
- The interface must be restrained to the bare minimum. It is always easier to add functionalities later than to remove some. In other words, the interface must be as small as possible.
- The model must be designed taking into account the context of your problem, which is the intent of your component and not what the component will look like.

This part could look obvious to many but is really not trivial at all. During the first few days that we worked on the “selectable list”, we kept thinking about the look-and-feel of the component, asking for more information to the user interface requirements team. Our first thought was to build the interface of the model based on what the component would look like which is something to avoid. Another common mistake is to think about the implementation of your model. The design of the model interface cannot be driven by implementation details.

With that in mind, we wrote this simple interface\textsuperscript{15}:

\textsuperscript{15}We have stripped the javadoc comments from the source code to improve conciseness and readability. We also removed pieces of code (such as toString methods, argument checks...
5.4 The Swing Component Development Process

package source.selectablelist;

public interface SelectableListModel {
    public void addSelectableListModelListener(
        SelectableListModelListener listener);
    public void removeSelectableListModelListener(
        SelectableListModelListener listener);
    public Object[] getItems();
    public void setItemState(Object item, boolean selected);
    public boolean getItemState(Object item);
    public void addItem(Object item);
    public void removeItem(Object item);
}

The `getItems` method returns all the items that are in the list, whether they are selected or not. The `setItemState` method allows us to select or unselect an item passed as the first parameter. The `getItemState` returns true if the item passed in is selected, and false otherwise. Finally the two remaining methods `addItem` and `removeItem` allow us to add and remove items from a selectable list. The default state of newly added items will be unselected.

Note that this interface fulfills all the responsibilities defined in the previous section. If you ever wonder what to put or what not to put in a class, refer to those responsibilities. This is really the roadmap to build reusable Swing components. And, if the answer is not in there, just think about the underlying MVC architecture and its purpose. It is really less likely to make a design error when keeping the architecture principles in mind. We have applied this technique for other graphical components and it worked well: by implementing small parts at a time, we narrow the scope of the responsibilities and it allows us to create more robust classes.

After the interface is designed, you can directly create the abstract model that implements the add and remove listener methods and the methods that will fire the events. You need a fire method for each set method declared in the interface. Here is a straightforward implementation of the abstract model:

package source.selectablelist;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public abstract class AbstractSelectableListModel implements SelectableListModel {
    protected List listenerList;
    protected AbstractSelectableListModel() {
        listenerList = new ArrayList();
    }
    public void addSelectableListModelListener(
        SelectableListModelListener listener) {
    }
}

that were not relevant for the same reason.
We chose an ArrayList to hold the listeners. In each fire method, we created an event object, passing a reference to this instance of the model and some additional information as parameters. The first parameter allows listeners to know from which source the event occurred (a listener may be registered with more than one observed object) and the other parameters generally give more information about what has changed in the model. Here we have provided the item that has either been added, removed, selected or unselected. A Boolean value is also used in the last method, when we change the state of items; it will be true when we select items and false otherwise. After having created the right event, notifying methods iterate through the list of listeners and call the appropriate method for each one. Also note that notifying methods are protected, preventing clients to use them by mistake.

This class will not compile until we have defined the listener interface and its corresponding event class. In the abstract model, fire methods needs three different method calls to notify listeners that an event had occurred. Those three methods signatures define the listener interface:
package source.selectablelist;
import java.util.EventListener;

public interface SelectableListModelListener extends EventListener {
    public void itemAdded(SelectableListModelEvent event);
    public void itemRemoved(SelectableListModelEvent event);
    public void itemStateChanged(SelectableListModelEvent event);
}

To comply with Sun specification, the listener inherits the empty interface java.util.EventListener and respects the naming convention defined in the previous section. You now have one more reason to write a model interface as concise as possible; the listener being directly derived from it, you are sure that it will also be short and easy to implement for the client.

We have already described the information that we needed to encapsulate in a SelectableListModelEvent, here is the implementation:

class SelectableListModelEvent extends EventObject {
    private Object item;
    private boolean selected;
    public SelectableListModelEvent(SelectableListModel source, Object item) {
        this(source, item, false);
    }
    public SelectableListModelEvent(SelectableListModel source, 
            Object item, boolean selected) {
        super(source);
        this.item = item;
        this.selected = selected;
    }
    public Object getItem() {
        return item;
    }
    public boolean isSelected() {
        return selected;
    }
}

The event only provides get methods to the information. The reason for this immutability is fairly simple; if you look back at the abstract model, you will see that a notifying method fires the same instance of the event to all listeners. In such circumstances, we had to prevent listeners to modify them.

After having written those parts, it is a good practice to review the design of the model parts. We can check if nothing is missing for the core functionality of the model or if we did not put unneeded features in it. For instance, we could
add methods to allow the client to add, remove or select several items at once. We could provide a default behavior for those new methods in the abstract model: the new methods would iterate through the array of items provided by the client and invoke the methods that handle a single item at a time.

When the design of the model sounds stable, we can provide the default implementation that will be used by a “selectable list” when no other data model implementation has been supplied by the client. The code of our first implementation was very long, so we have decided to make a shorter version for the thesis. This, again, demonstrates the high flexibility of this approach to design GUI components; since we have separated the things that change (the implementation) from things that stay the same (the interface of the model), it was really easy to provide the new default without breaking the code of existing clients. Here is the code:

```java
package source.selectablelist;
import java.util.Collection;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
public class DefaultSelectableListModel
extends AbstractSelectableListModel {
private Map itemsMap = new LinkedHashMap();
public DefaultSelectableListModel() {
}
public DefaultSelectableListModel(Object[] items) {
    for (int i = 0; i < items.length; i++) {
        itemsMap.put(items[i], Boolean.FALSE);
    }
}
public DefaultSelectableListModel(Collection items) {
    this(items.toArray());
}
public Object[] getItems() {
    return itemsMap.keySet().toArray();
}
public boolean getItemState(Object item) {
    return ((Boolean)itemsMap.get(item)).booleanValue();
}
public void setItemState(Object item, boolean selected) {
    Boolean oldValue = ((Boolean)itemsMap.get(item));
    if (selected != oldValue.booleanValue()) {
        if (selected) {
            itemsMap.put(item, Boolean.TRUE);
        } else {
            itemsMap.put(item, Boolean.FALSE);
        }
    fireItemStateChanged(item, selected);
    }
}
public void addItem(Object item) {
```
5.4 The Swing Component Development Process

if (!itemsMap.containsKey(item)) {
    itemsMap.put(item, Boolean.FALSE);
    fireItemAdded(item);
}
}

public void removeItem(Object item) {
    if (itemsMap.containsKey(item)) {
        itemsMap.remove(item);
        fireItemRemoved(item);
    }
}

We have decided to hold the items in a LinkedHashMap, a new container in the Java 1.4 API. This container has the advantage of maintaining a linked list running through all of its entries, therefore allowing us to always retrieving the entries in the same order. The keys represent the items of the list and their value is just a Boolean object stating that the item has been selected or not. We have implemented several public constructors allowing the user to fill the list with an array or any other container implementing the interface (Collection (Vector, ArrayList, LinkedList...). We must have at least one constructor that fills all the values of a model at once; this will allow clients to initialize the model without having to call set methods that would fire unneeded events.

The getItems method returns the keys set as an array. The getItemState method returns the boolean value associated with the object passed as the argument in the map. The methods that change the state of the model (setItemState, addItem and removeItem) must all call the appropriate fire method after having proceeded to the modification. Also the setItemState method does not fire any event if we set the state of an item to the same state it was previously set.

One thing to also keep in mind while writing this model is to keep it simple. We should not try to write fancy code for efficiency’s sake; we will usually end up with overly complex holders and actually will not get the expected benefits. This is really a general programming principle: first make it work, then if we ever need it (and it is more likely that we will not), optimize. We should even say “never try to optimize things before we are sure that it is a real bottleneck for the application.

5.4.3 Graphical Specification of the Component

After having tested thoroughly the model, we can now specify what the graphical part of the “selectable list” will look like and how it should behave. The desired look of the component is shown on Figure 5.9. We have chosen to represent our component as a panel with two lists, the left list displaying all the elements that we can select and the right list displaying only the selected ones. The behavior
of this component is standard: we select an element from the left list by clicking on it and then clicking on the select button. The selected item is then grayed out in the left list and appears on the right list. Of course, the user must be able to select several items at once by holding the shift or control keys while picking the items to select. The labels on top of the two lists, the select all, unselect all, add, and remove buttons are optional.

5.4.4 Implementing the Wrapper

Now that we have a preview of what our component will look like, we can implement the often most complicated part: the UI delegate, encapsulating the look-and-feel behavior of the graphical part, and the wrapper. Since the original classes count more than 2000 lines of code, we have decided to present a simplified version. We will not implement the `selectAll`, `unselectAll`, `add` and `remove` buttons. We also kept the offered features (changing text, colors...) to a minimum.

In order to properly implement this part, we need to explain a few implementation details about the PLAF presented in the previous section. We will see how Swing components really initialize their UI delegate at construct time. Assuming that our “selectable list” is fully implemented following Sun’s conventions; what should happen when we create a new instance of the `SelectableList` is described in Figure 5.10.

As we have previously seen, Swing’s `UIManager` is the class through which components and programs access the `UIDefaults` table for the current look-and-feel. The `UIDefaults` class is actually a direct extension of `java.util.HashMap` that keeps the mapping between `UIClassID` (such as “SelectableList-UI”) and their UI delegate class names (such as “BasicSelectableListUI”).
The UIClassID is a String identifying the abstract class of the UI delegate corresponding to a particular wrapper. This String is hard-coded in the wrapper class and can be retrieved by the UIDefaults in order to know which entries it needs to lookup.

The UIDefaults class also implements a getUI method that creates an instance of the UI delegate and then returns it.

Since the code of the SelectableList and BasicSelectableListUI is very long, we will first present the skeleton and complete them as and when we implement their respective responsibilities.

```java
package source.selectablelist;

public class SelectableList extends JComponent {

    static {
        UIManager.put("SelectableListUI",
            "source.plaf.basic.BasicSelectableListUI");
    }
}```
/* +++ properties +++ */
/* +++ private fields: model, listeners +++ */
/* +++ holders for view states +++ */
public SelectableList() {
    this(new DefaultSelectableListModel());
}

public SelectableList(Object[] items) {
    this(new DefaultSelectableListModel(items));
}

public SelectableList(Collection items) {
    this(new DefaultSelectableListModel(items));
}

public SelectableList(SelectableListModel dataModel) {
    setModel(dataModel);
    updateUI();
}

/* +++ model and get methods +++ */
/* +++ relaying models methods: getItem, addItem, *
 * removeItem, setItemState, getItemState +++ */
/* +++ get and set methods to view states: *
 * labels for buttons and the two lists +++ */
public void updateUI() {
    setUI((SelectableListUI)UIManager.getUI(this));
}

public void setUI(SelectableListUI newUI) {
    super.setUI(newUI);
}

public SelectableListUI getUI() {
    return (SelectableListUI)ui;
}

public String getUIClassID() {
    return "SelectableListUI";
}

/* +++ methods to relay model events +++ */
/* +++ fire selectablelist events methods +++ */

We will see later why the SelectableList implements the model listener. First, we have put a static block that registers the UI delegate for the default look-and-feel; this will fill the map of the default look-and-feel’s with the provided key and value. We have then implemented a few constructors that use the default model and a constructor allowing the client to provide another implementation for the model. The constructor is responsible for setting the model and the UI delegate on the wrapper. In order to comply with the sequence diagram, we have implemented the getUI, setUI, updateUI and getUIClassID methods. All those methods allow a user to change the look-and-feel of a component at runtime.

The recommended practice is for the wrapper class to provide convenience for programs that do not wish to deal with a separate model. As a consequence, we must provide the ability to register listeners directly on the component: the SelectableList class must listen for changes on the model internally and then
propagate those events to any listeners registered directly on the component. We also must provide delegating methods to all methods defined in the model interface, so that the component can be manipulated ignoring the model completely. Here is an implementation for relaying models methods:

```java
public static final String MODEL_PROPERTY = "Model";
private SelectableListModel dataModel;

public void setModel(SelectableListModel dataModel) {
    this.dataModel.removeSelectableListModelListener(this);
    SelectableListModel oldDataModel = this.dataModel;
    this.dataModel = dataModel;
    dataModel.addSelectableListModelListener(this);
    firePropertyChange(MODEL_PROPERTY, oldDataModel, dataModel);
}
public SelectableListModel getModel() {
    return dataModel;
}
public Object[] getItems() {
    return dataModel.getItems();
}
public boolean getItemState(Object item) {
    return dataModel.getItemState(item);
}
public void setItemState(Object item, boolean selected) {
    dataModel.setItemState(item, selected);
}
public void addItem(Object item) {
    dataModel.addItem(item);
}
public void removeItem(Object item) {
    dataModel.removeItem(item);
}

You see that the model is a property of the SelectableList\textsuperscript{16}. The setModel method fires a PropertyChangeEvent after having set the new model by calling the firePropertyChange method. This method is actually implemented in the parent class JComponent, thus allowing clients to listen for PropertyChangeEvent on any widget inheriting from JComponent. The setModel method also adds the wrapper as a listener to the model. The other methods getItems, setItemState, getItemState, addItem and removeItem simply delegate the calls to the model. Note that those methods do not fire any event. Instead, the SelectableList listens to changes in the model and relays them:

```java
private List modelListenerList = new ArrayList();

public void addSelectableListModelListener(
    SelectableListModelListener listener) {

\textsuperscript{16}As a reminder, the property change mechanism is explained on Page 100.
modelListenerList.add(listener);
}
public void removeSelectableListModelListener
(SelectableListModelListener listener) {
    modelListenerList.remove(listener);
}
public void itemAdded(SelectableListModelEvent event) {
    Iterator listenerListIterator = modelListenerList.iterator();
    while (listenerListIterator.hasNext()) {
        ((SelectableListModelListener)
            listenerListIterator.next()).itemAdded(event);
    }
}
public void itemRemoved(SelectableListModelEvent event) {
    Iterator listenerListIterator = modelListenerList.iterator();
    while (listenerListIterator.hasNext()) {
        ((SelectableListModelListener)
            listenerListIterator.next()).itemRemoved(event);
    }
}
public void itemStateChanged(SelectableListModelEvent event) {
    Iterator listenerListIterator = modelListenerList.iterator();
    while (listenerListIterator.hasNext()) {
        ((SelectableListModelListener)
            listenerListIterator.next()).itemStateChanged(event);
    }
}

You now understand why the SelectableList implements SelectableListModelListener. It holds its own list of model listener, provides methods to register and unregister to model events and relays those. It allows client to treat a SelectableList as if no separate model existed.

By now, all the important pieces of the wrapper are in place. We can then implement the SelectableList features we want to provide to the user. As an example, we have just implemented the possibility to change the text for the buttons of the lists:

public static final String
    UNSELECT_BUTTON_TEXT_PROPERTY = "TextUnselectButton",
    SELECT_BUTTON_TEXT_PROPERTY = "TextSelectButton";
public static final int SELECT_BUTTON = 1;
public static final int UNSELECT_BUTTON = 2;
private String selectButtonText = "Select";
private String unselectButtonText = "Unselect";

public void setButtonText(int button, String text) {
    String oldText;
    switch (button) {
        case SELECT_BUTTON:
            oldText = selectButtonText;
            selectButtonText = text;
            break;
        case UNSELECT_BUTTON:
            oldText = unselectButtonText;
            unselectButtonText = text;
            break;
    }
}
5.4 The Swing Component Development Process

```java
selectButtonText = text;
firePropertyChange(SELECT_BUTTON_TEXT_PROPERTY, oldText, text);
break;
case UNSELECT_BUTTON:
    oldText = unselectButtonText;
    unselectButtonText = text;
    firePropertyChange(UNSELECT_BUTTON_TEXT_PROPERTY, oldText, text);
    break;
}
}
public String getButtonText(int button) {
    switch (button) {
        case SELECT_BUTTON: return selectButtonText;
        case UNSELECT_BUTTON: return unselectButtonText;
        default: return null;
    }
}
```

All the other features related to state views can be implemented like that, as properties of the wrapper. Note that this kind of information must absolutely be kept in the wrapper and not in the UI Delegate. Otherwise, if we change the delegate at runtime the displayed text for buttons would be reset.

5.4.5 Implementing the UI Delegate

Following sun’s convention we have to create an abstract class inheriting `javax.swing.plaf.ComponentUI`:

```java
package source.plaf;
import javax.swing.plaf.ComponentUI;

public abstract class SelectableListUI extends ComponentUI {
    //empty
}
```

The ComponentUI class contains most of the important mechanisms for making Swing’s pluggable look-and-feel work. Its methods deal with UI installation and uninstallation and painting. The `SelectableListUI` class defines the interface that all implementation of the UI delegate should implement. Here, we have left this interface empty, thus putting no restrictions on its implementation subclasses.

The skeleton of the `BasicSelectableListUI` looks as follows:

```java
package source.plaf.basic;
//import statements
```
public class BasicSelectableListUI extends SelectableListUI {
    /*+++ private fields: selectablelist, model, listeners +++*/
    public static ComponentUI createUI(JComponent c) {
        return new BasicSelectableListUI();
    }
    /*+++ setModel and getModel methods +++*/
    public void installUI(JComponent c) {
        selectableList = (SelectableList)c;
        //+ initialize the child subcomponents of a selectableList
        //+ set default font, color, border, and opacity
        // properties on the selectableList
        //+ register any required event listeners on the selectableList
        // and its child subcomponents
        //+ register appropriate model listeners to be notified when
        // to update the view
        //+ register any keyboard actions such as mnemonics for
        // the selectableList. Rmq: not needed here.
        //+ install an appropriate layout manager on the selectableList
        //+ add any appropriate child subcomponents to the selectableList
    }
    public void uninstallUI(JComponent c) {
        /*+++ uninstall everything that has been set by
           the installUI method +++*/
    }
    /*+++ paint(Graphics g, JComponent c) method +++*/
    /*+++ update(Graphics g, JComponent c) method +++*/
    /*+++ getPreferredSize, getMinimumSize and getMaximumSize methods +++*/
    /*+++ Listeners implementations +++*/
}

First notice the static createUI method necessary for the PLAF mechanism. It will be called by the UIDefaults table in order to instantiate this delegate. Other key methods of a UI delegates are installUI, uninstallUI, paint, and update. The update method just clears the background and then invokes the paint method which is ultimately responsible for rendering the contents of the component [Fow00]. All painting is therefore done through the paint method. Since we do not need any drawing for the SelectableList (our component will be made of other Swing components), we have not implemented the paint method. Also, in order to display the subcomponents properly, we will install a layout manager that will manage the size of our component and all its childs. As a consequence, we can bypass the implementation of the getPreferredSize, getMinimumSize and getMaximumSize methods.

The installUI method is responsible for a lot of things as described in the comments of the code. This method will be our starting point to implement the rest of the class. We will distribute the responsibilities among several smaller methods, as follows:

protected SelectableList selectableList;
5.4 The Swing Component Development Process

protected SelectableListModel selectableListModel;

public void installUI(JComponent c) {
    selectableList = (SelectableList)c;
    setModel(selectableList.getModel());
    initializeComponents(selectableList);
    installDefaults(selectableList);
    installListeners(selectableList);
    installComponents();

    updateView(selectableList);
}
public void uninstallUI(JComponent c) {
    SelectableList selectableList = (SelectableList)c;
    uninstallDefaults(selectableList);
    uninstallListeners(selectableList);
    uninstallComponents();

    selectableList = null;
    selectableListModel = null;
}

When we call installUI, the UI delegate has access to the wrapper passed as a parameter. In order to interact with the model, the UI delegate must keep a reference to it; this reference is asked from the wrapper and set in the installUI method. The setModel and getModel methods follow the same pattern as the one we have used for the wrapper:

protected SelectableListModel getModel() {
    return selectableListModel;
}
protected void setModel(SelectableListModel model) {
    selectableListModel.removeSelectableListModelListener
        (selectableListModelListener);
    selectableListModel = model;
    selectableListModel.addSelectableListModelListener
        (selectableListModelListener);
}

The delegate listens to model events in order to reflect the changes on the user interface. We describe this behavior later. We will first implement the methods related to build the look of our component:

protected JList selectedL, unselectedL;
protected JScrollPane selectedS, unselectedS;
protected JPanel selectUnselectJP;
protected JButton selectB, unselectB;

protected void initializeComponents(SelectableList selectableList) {

Our SelectableList is made of other Swing components: we have used two JLists decorated with a JScrollPane to represent the left and right lists of the component, and two JButton}s to select and unselect elements.

Method initializeComponents is responsible for instantiating and setting the default state of all child subcomponents used by the UI delegate. If the wrapper allows the client to set properties on those child components, as it is the case for our buttons text, the initializeComponents method must also retrieve their value in the wrapper and set them on the appropriate widgets. Here we have instantiated the JLists, JPanel}s and JButton}s that we need in order to build the SelectableList. Buttons are disabled by default and their text is set based on the wrapper component.

The installComponents method installs a layout manager on the SelectableList and adds the child subcomponents (the two lists, the select button, and the unselect button). It also sets the opaque property of the component to true. The uninstallComponents method just removes all the components that had been added to the SelectableList.

Now, if we look at the skeleton presented on Page 121, in the installUI method, we still have to implement the default properties of the SelectableList.
and listeners to implement the controller part of this UI delegate. We do not have any defaults properties besides the renderers for the cells of our JList:

```java
protected void installDefaults(SelectableList selectableList) {
    selectedL.setCellRenderer(new SelectedLCellRenderer());
    unselectedL.setCellRenderer(new UnselectedLCellRenderer());
}
protected void uninstallDefaults(SelectableList selectableList) {
}
class SelectedLCellRenderer extends JLabel implements ListCellRenderer {
    //implementation
}
class UnselectedLCellRenderer extends JLabel implements ListCellRenderer {
    //implementation
}
```

We have not displayed the code for renderers here since it involves some advance knowledge of Swing. You just have to know that it is where we define the look and behavior of the cells of our JLists. By default, an item will be displayed black on white. When we click on an item, the colors will be reversed and when we select an item via the select button, the item will be grayed out in the left list, showing that we cannot select it anymore.

Now, the last part to implement is the behavior of our component. We will use three kinds of listeners: a `SelectableListModelListener` which will update the view when the model changes, a `PropertyChangeListener` which will update the view when a property of the wrapper changes, and an `ActionListener` which will update the model when we select or unselect items\(^\text{17}\). We install and uninstall those listeners as follows:

```java
private SelectableListModelListener selectableListModelListener;
private PropertyChangeListener propertyChangeListener;
private ActionListener actionListener;

protected void installListeners(SelectableList selectableList) {
    propertyChangeListener = new PropertyChangeHandler();
    selectableList.addPropertyChangeListener(propertyChangeListener);
    selectableListModelListener = new SelectableListModelHandler();
    selectableList.addSelectableListModelListener (selectableListModelListener);
    actionListener = new ActionHandler();
    selectB.addActionListener(actionListener);
    unselectB.addActionListener(actionListener);
}
protected void uninstallListeners(SelectableList selectableList) {
    selectableList.removePropertyChangeListener(propertyChangeListener);
    selectableList.removeSelectableListModelListener (selectableListModelListener);
}
```

\(^{17}\)We actually used more listeners in order to implement behaviors such as disabling the select button when no item was selected in the left list... We prefer not to show them here for conciseness.
selectableListModelListener);
selectB.removeActionListener(actionListener);
unselectB.removeActionListener(actionListener);
propertyChangeLister = null;
selectableListModelListerner = null;
actionListener = null;
}

The **installListeners** method creates all the listeners and registers them with their respective subjects. The listeners are directly implemented as inner classes of the delegate. That is the reason why we can instantiate them with a constructor. The **uninstallListeners** method removes all the listeners that were added by **installListeners** and nullify them. Here is a possible implementation for those listeners:

class PropertyChangeHandler implements PropertyChangeListener {
    public void propertyChange(PropertyChangeEvent event) {
        String propertyName = event.getPropertyName();
        if (propertyName.equals(SelectableList.MODEL_PROPERTY)) {
            setModel((SelectableListModel)event.getNewValue());
            updateView(selectableList);
        }
        if (propertyName.equals(SelectableList.SELECT_BUTTON_TEXT_PROPERTY)) {
            selectB.setText((String)event.getNewValue());
        }
        if (propertyName.equals(SelectableList.UNSELECT_BUTTON_TEXT_PROPERTY)) {
            unselectB.setText((String)event.getNewValue());
        }
    }
}

The **PropertyChangeHandler** inner class listens to **PropertyChangeEvent** of the wrapper object. It uses the **propertyName** provided by the event to know which property has been set on the wrapper.
If the model has changed, the delegate will update its own private copy and update the view accordingly. Indeed, if a client sets a new model on the **SelectableList**, the items displayed in the two **JList**s have to be updated.
That behavior is implemented in the **updateView** method. The definition of the latter will not be shown here for the sake of brevity. If the text of a button has changed, the listener retrieves the new value from the event and set it on the appropriate **JButton**.

class SelectableListModelHandler implements SelectableListModelListener {
    public void itemAdded(SelectableListModelEvent event) {
        DefaultListModel listModel = ((DefaultListModel)unselectedL.getModel());
        listModel.addElement(event.getItem());
    }
    public void itemRemoved(SelectableListModelEvent event) {
        Object item = event.getItem();
    }
}

5.4 The Swing Component Development Process

DefaultListModel unselectedLModel =
((DefaultListModel)unselectedL.getModel());
DefaultListModel selectedLModel =
((DefaultListModel)selectedL.getModel());
unselectedLModel.removeElement(item);
selectedLModel.removeElement(item);
}
public void itemStateChanged(SelectableListModelEvent event) {
  updateView(selectableList);
}
}

The SelectableListModelHandler listens to changes in the model. If an item has been added, it will add it on the left list, using the model of the JList. If an item has been removed, we need to remove it from either the left or the right list, depending if it was selected or not. Note that we reuse the underlying MVC of the JLists; by removing or adding elements in their models, we are sure that the view will be updated correctly. If the state of an item changes, we just reuse the updateView method which will refresh the whole component based on the model state.

class ActionHandler implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    if (e.getSource() == selectB) {
      //get the items we have selected with the mouse on the left list:
      Object[] items = unselectedL.getSelectedValues();
      for (int i = 0; i < items.length; i++) {
        getModel().setItemState(items[i], true);
      }
    }
    if (e.getSource() == unselectB) {
      Object[] items = selectedL.getSelectedValues();
      for (int i = 0; i < items.length; i++) {
        getModel().setItemState(items[i], false);
      }
    }
  }
}

ActionHandler listens to events fired by the two buttons. If the user has clicked on the select button, this listener will change the state of all the selected items on the left list in the model. The behavior of the unselect button is similar.

The simplified version of the SelectableList is now fully implemented. You can find the complete source code in appendix C of this document. As you have seen, the implementation of such a component is neither easy nor straightforward. It certainly takes time and experience to master all the concepts involved in this chapter. However we hope that you have had a first overview, and that the method will be helpful if you ever want to build your own reusable graphi-
cal user interface components with Java. As examples, a snapshot of two other graphical components developed with this method is shown on Figure 5.11.

![StopperSlider and Multislider](image)

Figure 5.11: The StopperSlider and the Multislider Graphical Components

So far, we have presented design patterns as a very powerful tool for both designers and programmers. Nevertheless, there is no silver bullet and design patterns are not an exception. If we want to be complete, we cannot conclude this thesis without analyzing the limits of design patterns. We will try to identify their drawbacks in the next and last chapter.

### 5.5 Summary

“The Observer pattern appears most frequently in GUI applications and is a fundamental aspect of Java’s Swing architecture” [Met02]. Observer lets you delineate the responsibility between models and the view/controller part of a Swing component. “The Abstract Factory pattern lets you provide a client with a factory that produces objects that are related by a common theme” [Met02]. Java Swing comes with the ability to change the look-and-feel—that is the appearance and behavior—of user interface components. In order to provide such functionalities, Java implements a variant of the Abstract Factory pattern to create families of look-and-feel components.

While Swing architecture relies heavily on design patterns, the implementation of components is not simple. Indeed, each widget is made of a dozen classes having their own responsibilities. If you want to implement a reusable GUI component, it is better to follow an incremental approach in order to reduce the number of details you have to think at the same time. A flexible method that respects all the architectural principles of Swing has been presented in this chapter which you can adapt to your needs.
Chapter 6

Limits and Flaws of Design Patterns

...Patterns do nothing to remove the human from the creative process. They merely bring hope of empowerment to a possibly inexperienced, perhaps just uninitiated, but otherwise capable and creative person... 

John Vlissides [Vli98]

In the preceding chapters, we praised design patterns and demonstrated how helpful they can be for developers. However, like any type of development tool, design patterns have drawbacks. The GoF book itself warns against some pitfalls for each pattern it describes. Nevertheless, if you search for assessments about design patterns, you are more likely to be disappointed. Indeed, few articles tackle the subject, and, very often, it is only a starting point to describe one or two problems that developers have encountered with design patterns.

As a result, we found interesting to identify the most common problems that people have met with design patterns and to try to classify them. This chapter is an attempt to establish taxonomy of limits and flaws of design patterns. Our analysis is based on the benefits that design patterns can provide. Everything that prevents us for exploiting those benefits will be considered as the negative aspects of design patterns.

We have used the four main benefits pointed out by Vlissides [Vli98]. According to him, patterns claim to provide the following:

1. They capture expertise and make it accessible to non-experts.
2. They provide a common vocabulary that helps developers communicate
3. They improve the documentation of software systems.

4. They facilitate restructuring a system whether or not it was designed with patterns in mind.

Those key points clearly accentuate the principal idea behind design patterns: “namely to distribute the knowledge of good design, such that designers of software applications can benefit from work previously done within similar areas” [AC98]. We will see that a lot of things can run against this distribution of knowledge: the wrong usage of design patterns, the implementation overhead, the endless list of newly discovered designs...

6.1 Chaos

The first flaw that we would like to underline comes from the fact that there are no objective, scientific criteria to determine if a solution consists of a pattern or not. “The generally accepted definition of a design pattern is that it is a description of a well tested solution to a recurring problem within the field of software designs in object-oriented Languages. This definition leaves it up to the individual designer to decide what constitutes a design pattern since terms like “well tested” and “recurring” are not objective terms that can be evaluated “true” or “false” in an unambiguous way” [AC98]. The GoF itself says that “point of view affects one’s interpretation of what is and isn’t a pattern. One person’s pattern can be another person’s primitive building block” [GHJV95].

In such circumstances, it is not surprising that the patterns community discovers new patterns in a seemingly endless stream. Plop conferences, OOPSLA conferences, and other discussion groups come up with new patterns every year. Therefore, as time goes on, the list of patterns becomes less and less manageable. A quick look at the patterns Home Page\(^1\) should convince anyone of the jungle it has become. This, obviously, must be considered as a downside to design patterns and will, in turn, tear down the possibility of using design patterns as a common vocabulary.

This is what we have called “chaos”. If the list of design patterns continues to grow incessantly, it is more likely that designers will be overwhelmed by all the proposed generic solutions. The vocabulary will be too large to master and often too specific to a particular area. The second volume of “Pattern-Oriented Software Architecture” [DSB00] partly suffers from that harm. This book presents patterns for concurrent and distributed systems. Some of those patterns are so specific to a particular class of problem that it is difficult to imagine another example than the one provided. Moreover, as the list of design

\(^1\)http://hillside.net/patterns/patterns.html.
patterns becomes obscure, assuming that you encounter the same problem, you reduce the chances to find the solution.

Chaos could partly be solved by a new classification of design patterns. The idea would be to form families of design patterns in such a way that developers could easily narrow the scope of their research. However, “pattern taxonomy is still a largely elusive goal, as is evident from the numerous failures to achieve it” [GL98]. Researchers [AC98, GL97, Tic98, Zim95] and authors [Eck01, GHJV95, Met02, ST02] propose new taxonomy every year; each being presented with the best intents, since their goal is to preserve the benefits of design patterns. But, again, those attempts do not give objective criteria to decide whether a pattern belongs to one category or another. In some classifications, design patterns families are not mutually exclusive, thus allowing a pattern to belong to several categories.

As a result, if we want to preserve benefits of design patterns, we should try to reduce the number of patterns. We have found two papers addressing this concern; each of them has a different approach. The first one, “How to Preserve the Benefits of Design patterns” [AC98] is based on the first three benefits of Vlissides. Their belief is that the field of design patterns should be narrowed down to a minimum. Consequently, they have identified the “core of the design patterns—the fundamental design patterns—which fully provides the benefits of design patterns” [AC98]. The second article, “Design Patterns and Language Design” [GL98], classifies patterns in terms of how far they are from becoming actual language features. According to the authors, the “real patterns” are the abstractions that are not yet sufficiently mature or important enough to be language features, the rest being either clichés or idioms. This approach discards the intent of patterns and only observes their mechanics. While those articles have different standpoints, it should not discourage people to consider doing research in the same area, hoping to someday obtain a consensus.

6.2 Implementation Overhead

Since a design pattern is not implemented as language constructs, programmers have to implement a set of classes to support the pattern. Moreover, it often consists of the implementation of several methods with only trivial behavior such as forwarding a message to another object or method. This undeniably leads to an implementation overhead which decreases understandability of the resulting code [Bos98], which runs against the third and fourth benefit that design patterns could provide.

Although the design of a pattern is reused, its implementation always needs to be adapted to a particular context which forces the software developer to implement the pattern over and over again. Therefore, design patterns require more thought and more work in coding and do not make expertise more accessible to non-expert, which is the first promised benefit.
The proposed solution to that problem often consists in creating language constructs that support design patterns. One could wonder which design patterns can be included and how this should be done. We will not even go that far. Indeed, as long as the chaos problem is not solved, it seems hard to eliminate the overhead with language constructs.

6.3 Abuse

Like any good tool, design patterns can be abused. We have distinguished two ways to abuse them.

Firstly, design patterns can be overused and lead to over design. Over design has negative effects on the application by making it hard to understand and maintain. Again, this could ruin the expected benefits of patterns. Quoting Buschmann et al., “With the growing use of patterns, we have seen people overdo it. Classes are no longer simple. Every chunk of code is highly flexible and can adapt to many different contexts. Such flexibility, however, comes at a price. Flexible software often consumes more resources by using more level of indirection or increasing storage consumption” [BMR+96]. And, since we know that design patterns introduce an implementation overhead, it is not reasonable to use them everywhere, as it will also increase the cost of the whole system.

As advised in Chapter 2, people should identify the vector of change in their application—that is the parts that are more likely to change—and encapsulate those in flexible classes that can adapt to different context. This process will help developers to select a set of patterns that fit the project and use them consistently throughout the design. It is therefore perfectly fine to have some subsystems that are highly specific to an application and not reusable at all. If you realize later that some of those subsystems need additional flexibility, there are still ways to carefully restructure them.

One way to restructure a piece of software is by applying a series of “refactorings”. A refactoring is a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior. Fowler discuss the various techniques of refactoring in his book “Refactoring, Improving the Design of Existing Code” [Fow99a]. These techniques directly support the fourth benefit.

The second kind of abuse is actually a wrong usage of design patterns. By wrong usage, we mean the application of a pattern in an inappropriate context. This is sometimes referred to as an anti-pattern. This danger arises when a developer or a software development team has gained a high level of competence in design patterns. As a result every new problem is viewed as something that is best solved with a pattern. In many cases, the chosen pattern is a mismatch for the problem, but minimal effort is devoted to exploring alternative solutions [BMIM98]. The intent of a pattern is to solve a problem, however. But
each pattern has also liabilities and, sometimes, those liabilities outweigh the expected benefits of the solution.

Brown et al. have called this “The Golden Hammer AntiPattern” [BMIM98]. The golden hammer results in the misapplication of a favored tool or concept. “When your only tool is a hammer, everything else is a nail. For example, some developers learn a few of the GoF patterns and apply them to all phases of software analysis, design and implementation. Discussions about intent or purpose are insufficient to sway them from recognizing the applicability of the design pattern’s structure and force-fitting its use throughout the entire development process” [BMIM98].

Note also that a pattern can be used in an appropriate way and then evolve into an anti-pattern as the context changes over time. Although the pattern might, at first, look like it is the solution, it sometimes ends up being a very bad one. In this case, again, refactoring can be helpful.

The abuse problem demonstrates the importance of education, training, and mentoring which are required to help people become aware of other available approaches to software system construction. Especially in computer sciences, all software organizations need to develop a commitment to an exploration of new technologies and techniques. Without such a commitment, the lurking danger of over reliance on a specific technology exists [BMIM98].

6.4 Poor Traceability

The traceability problem has been identified in 1995 by Soukup. In his paper [Sou95], Soukup denotes the poor traceability of design patterns in their current implementations. This is due to the fact that programming languages do not support the concept of design pattern in their constructs.

In a large scale application, several design patterns can be mixed and can even overlap each other. Different implementations of the same pattern can also coexist as they are each adapted to a particular context. Therefore, in the final design, it is really difficult to see which patterns are involved. Two different programmers could even arrive at two different sets of patterns when trying to identify them [Sou95]. Except perhaps for the documentation or comments scattered throughout the code, the patterns are lost during implementation.

In such circumstances, the many indirections created by design patterns make the code more difficult to understand, which ruins benefit number one. For instance, if you try to read the source code of a Swing component without knowing its underlying architecture, you can bet on headaches, misunderstandings, or even sleepless night.

In Swing, several events mechanisms are implemented, each corresponding to a specific need, but all are instance of the Observer design pattern.
Many of these problems would be solved if the final code reflected the existence of the patterns in a simple and straightforward manner. This could be enforced in existing languages with the use of “attribute extension” [Hed97]. Attribute extension allows the static-semantic of a language to be extended, allowing programming conventions to be enforced, but keeps the syntax of the base language. This technique first needs to identify the different roles of a pattern and then to formulate the rules for these pattern roles [Hed97]. For instance, the Decorator presented in Chapter 4 implies that, whenever a new operation is introduced on an G-16 proxy, each decorator must implement this operation and forward it to its G-16 proxy. That rule could be enforced with attribute extension.

### 6.5 Lost Control

When we apply several patterns to the same set of classes, pattern behavior is embedded in methods associated with various application classes. “By definition, each class in each pattern calls methods of several other classes; as a result of this chain of reaction, the class depends directly or indirectly on every other class. The entire design becomes a big knot of interdependent classes and methods” [Sou95].

“Lost control” is the name we have given to this problem. As an application gets bigger and as more design patterns are added to it, we usually lose control of all the complex relationships established between all objects. Those interdependencies affect both debugging and testing [Sou95]. Classes and patterns cannot be tested individually. Indeed, it becomes more difficult to know which methods are called in which order at execution time.

This problem is even more cumbersome in a multi-thread application. For instance, the Observer design pattern implies that a subject notifies its observers when its state changes. Suppose that each observer, in turn, notifies other objects when their state changes and that those objects happen to eventually modify the first subject. This situation appears to be problematic, as a potential infinite loop could occur. That loop could still be easily located, however. Now, imagine that kind of behavior in a multi-thread application when several threads proceed to concurrent notifications and event handling. That kind of situation can become unmanageable, and risks increase with the number of indirections in the application.

Therefore, code with complex embedded relations is also more difficult to understand, which does not preserve benefit one and four of design patterns. Any method that would eliminate some of these dependency cycles would greatly improve applications which rely heavily on design patterns.
6.6 Duplication

The last flaw that we want to present is the duplication of information in the several layers that compose an application. The Power Config software is not an exception: the same data is often stored in several places in the application. For instance, the value of the input gain is stored in the model of the slider that controls it, in the model of the controller panel that contains that slider, in the device model of the domain layer, and in the amplifier itself which represents the hardware layer. And still, this is the minimum, as the value could also be stored in a cache and several other places.

This duplication of knowledge can lead to inconsistencies between the different models at runtime. For example, assuming that a model accepts values between 0 and 20 for some attribute while another only accepts values between 0 and 10, the system is likely to crash. Consequently, we also need to duplicate range of values. And, whenever changes are made to the amplifier, we need to reflect them in all the layers of the application, which introduces a new implementation overhead.

Here, we want to emphasize the fact that, although layers are loosely coupled, they are not completely independent. Vlissides underlined that problem in his article “The Trouble with Observer” [Vli96b]. He explains that the Observer pattern, due to the links between subjects and their observers, produces a lot of redundancy at run-time and in the code. “These redundancies are all included in the price for reuse... Beyond the mechanical overhead of coding and maintaining these hierarchies, there is also conceptual overhead. Programmers have to understand twice as many classes, twice as many interfaces, and twice as many subclassing issues” [Vli96b]. And that statement was made for only two layers... Again, benefit one and four of design patterns could be lost.

If too many layers introduce unnecessary complexity and overheads in an application, we still need layers for reusability and changeability. Therefore, a difficult but critical decision for any designer is to determine the good granularity of layers and the assignments of tasks to layers [BMR96].

We could not conclude this chapter without insisting on the fact that software is developed by people. “Though often ignored, this means that personal beliefs and values are driving forces for application design. This holds true in general but particularly for the design of interactive software systems” [RZ96]. Therefore, one should be aware that underlying assumptions and values have an impact on the choices made when defining architectures for software. This stresses the importance to make those beliefs explicit, so they can be made subject to discussions by the parties concerned [RZ96]. Software architects should share their vision of the future system so that the whole team understands what type of software they have in mind, and thus, their decisions when dealing with patterns.

“As time progresses, we’ll get a better handle on the benefits and pitfalls of
pattern usage. Even though initial returns are promising, we need more experience for a thorough assessment” [Vli98].

6.7 Summary

The limits and flaws associated with design patterns in traditional object-oriented languages have been identified and discussed in this chapter. These problems can be categorized into chaos, abuse, implementation overhead, poor traceability, lost control, and duplication.

Chaos refers to the endless growing number of new patterns which can overwhelm developers. Abuse can be interpreted in two ways: either an overuse of design patterns where it is not needed or the usage of patterns in wrong contexts. Implementation overhead is an obvious consequence for the software engineering when implementing design patterns, as they are not supported by current languages. Poor traceability points out that we lose track of design patterns during implementation. Lost control derives from traceability, as we also lose sight of the patterns at execution. Finally, duplication indicates that data information is scattered and duplicated in all the layers of an application, as design patterns often add several levels of indirection.

Nevertheless, all those disadvantages have potential solutions and do not appear in all circumstances. The expected benefits of design patterns are promising and, if people pursue their research to preserve those benefits, design patterns might be accepted as a must-know for every software engineer someday.
Conclusion

We have analyzed the pertinence of design patterns in the field of a “real-world” object-oriented software application. This application, called Power Config software and designed for the setup of an electronic amplifier, makes extensive use of GoF design patterns.

Chapters 1, 2, and 3, respectively called, Power Config Software, Object-Oriented Programming, and Object-Oriented Software Architecture both introduced the context of our study and provided the necessary background to allow readers to tackle the subsequent chapters.

Chapter 1 provided a thorough presentation of Power Config software. Chapter 2 gave an introduction to object-oriented programming that forms the basis of object-oriented design patterns. And, Chapter 3 described the coarse-grained architecture of Power Config which is an adaptation of the standard layered architecture for Enterprise Application. This architecture is in accordance with the enabling techniques significant for successful software development: it ensures especially weak coupling between layers and high cohesion within layers. This vertical partitioning laid the necessary foundation for the understanding of the overall Power Config architecture.

Chapters 4, 5, and 6, respectively called, How Design Patterns Improve the Architecture, Elements of Reusable GUI Components, and Limits and Flaws of Design Patterns were the core chapters of this document. They addressed the main concern of this thesis, that is, the pertinence of design patterns in object-oriented software development.

Chapter 4 demonstrated with an incremental approach that design patterns deserve an important place in object-oriented software development. We concentrated on the improvements design patterns bring to the architecture of the amplifier surrogate subsystem. This subsystem that we developed for the needs of Power Config was designed both to reflect in real-time the state of an electronic amplifier connected through the serial port and to ensure the control of this amplifier. We applied the Mediator, Facade, Singleton, Proxy, Decorator, and Adapter design patterns to transform the adapted standard layered architecture for Enterprise Application into the final architecture.
Those design patterns enabled us to take up the challenge of providing, at the same time, a reusable, testable, reliable, and extensible subsystem without having to rediscover the design solutions ourselves. These proven solutions provided in the GoF book have thus strongly improved the quality of our subsystem. Indeed, the aforementioned non-functional properties of the amplifier surrogate subsystem are at least as relevant as its functional properties and are indispensable for the success of the whole Power Config project. Nevertheless, designing with patterns requires both experience in matching a specific design problem to a design pattern and creativity to figure out how to apply it according to specific circumstances. Hence, before getting the benefits from design patterns, we had to spend several weeks studying the GoF patterns and applying them in practice in order to understand them fully.

We have exploited both the Mediator and Facade design patterns in order to “encapsulate” the coupling between the amplifier surrogate subsystem and other subsystems. Hence, not only was the amplifier surrogate subsystem reusable in other contexts, but it could also be developed totally independently of the other subsystems. Through an exception mechanism, the Proxy design pattern ensures the reliability of the subsystem in the event of amplifier errors. The Decorator design pattern enables developers to configure the proxy according to their needs. It allows them, for instance, to emulate defective electronic functions of the amplifier. Hence it ensures the testability of the subsystem and, consequently, of the whole application despite the fact that the development of the hardware device is completely out of the software team’s control. Lastly, the Adapter design pattern ensures the changeability of the subsystem. Thanks to this pattern, we are able to adapt without difficulty the amplifier surrogate subsystem so that it handles new kinds of amplifiers.

Without the help of design patterns, we would have struggled to create such a design. Indeed, design patterns capture a wealth of experience about the design of object-oriented software. They capture the essence of existing designs that have evolved over time to come close to the “ideal” solution. Furthermore, design patterns serve as a good team communications medium. They provide a common language, facilitate design documentation, and improve design understandability.

Chapter 5 presented a step-by-step method to create new reusable Swing Components. This activity provides huge benefits to the Software Company: besides the fact that the source code is fully understandable, a custom-made widget can add that extra touch of professionalism to polish up the user interface. Nevertheless, creating reusable Swing components requires a thorough understanding of both the Observer and the Abstract Factory design patterns. Indeed, the underlying architecture of Swing components encompasses these two patterns.

Having developed this method from scratch, we were in a position to appreciate the benefits of design patterns in the development of GUI components. Once again, design patterns bring to the architecture the required flexibility. The Observer design pattern lets developers delineate the responsibilities between
model, view, and controller. It ensures weak coupling between the model and the view/controller part which allows users to reuse the same view/controller part with distinct models. According to the specified look-and-feel, the Abstract Factory design pattern produces transparently families of related GUI components. Hence Java Swing comes with the ability to change at run-time the appearance and behavior of user interface components.

Chapter 6 examined the limits and flaws of design patterns. In preceding chapters, we praised design patterns and demonstrated how helpful they can be for developers. Nevertheless, despite all the promises made by the supporters of design patterns, those patterns also have drawbacks.

We establish our own taxonomy of limits and flaws of design patterns. These problems can be categorized into chaos, abuse, implementation overhead, poor traceability, lost control, and duplication. Among these drawbacks, one in particular caught our attention: the abuse of design patterns. This phenomenon can lead to over design which has negative effects on the application by making it hard to understand and maintain. Moreover, such a design consumes more resources by using more levels of indirection and it requires more thought and more work in coding. A design pattern should only be applied when the flexibility it affords is actually needed.

In conclusion, despite several drawbacks mentioned in Chapter 6, design patterns enable software developers to design well thought-out architectures and thus to provide the increasingly important non-functional properties for their software applications. Nevertheless, we agree that it is hard to match a specific design problem to a design pattern. It is also difficult to figure out how to apply design patterns to specific circumstances. We really hope our original approach to design patterns makes designers sensitive to the benefits of design patterns and will enable them, in turn, to apply the design patterns tackled and, better still the other design patterns, according to their specific needs...
Glossary

This glossary gathers definitions found in the following books: [BMR*96, Fow99a, BMIM98, GHJV95].

**Abstract class** A class whose primary purpose is to define an interface. An abstract class defers some or all of its implementation to subclasses. An abstract class cannot be instantiated.

**Abstract Factory** Creational pattern. Provide an interface for creating families of related or dependent objects without specifying their concrete classes. See key features on Page 150.

**Abuse** One type of design patterns drawbacks. Abuse can be interpreted in two ways: either an overuse of design patterns where it is not needed or the usage of patterns in wrong contexts.

**Adapter** Structural pattern. Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces. See key features on Page 151.

**Aggregate object** An object that’s composed of subobjects. The subobjects are called the aggregate’s parts, and the aggregate is responsible for them.

**Aggregation** The “has-a” relationship of an aggregate object to its parts. Aggregation means that the container class aggregates some other objects in order to have all the features needed but still exists by itself.

**Amp** Measurement unit of electrical current. (See Current)

**Amplifier** (Audio) An electronic component that takes a weak audio signal and increases it to generate a signal that is powerful enough to drive speakers. (General) An electronic component that accepts a low-level signal and recreates the signal with more power.

**Anti-pattern** A commonly occurring pattern or solution that generates decidedly negative consequences. An anti-pattern may be a pattern in the wrong context. When properly documented, an anti-pattern comprises a paired anti-pattern solution with a refactored solution.
**API** Application Programming Interface: the set of services that an operating system or a programming language makes available to programs that run under it.

**Application** A program or collection of programs that fulfills a customer’s requirements.

**Architecture** See software architecture.

**Chaos** One type of design patterns drawbacks. Chaos refers to the endless growing number of new patterns which can overwhelm developers.

**Class diagram** A diagram that depicts classes, their internal structure and operations, and the static relationships between them.

**Class** A class defines an object’s interface and implementation. It specifies the object’s internal representation and defines the operations the object can perform.

**Cohesion** The degree to which software components are self-contained, i.e. are responsible for their own tasks and independent of each other.

**Composition** The “has-a” relationship of an aggregate object to its parts. Composition means that objects which compose the container class are part of it. Without its part objects, the composite class does not exist by itself. On the other hand, the parts are usually expected to die with the composite in case of deletion.

**Concrete class** A class having no abstract operations. It can be instantiated.

**Constructor** In Java, an operation that is automatically invoked to initialize new instances.

**Coupling** The degree to which software components depend on each other.

**Current** (Electronics: I) The flow of electrons through a conductor. It is measured in amps.

**Decorator** Structural pattern. Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality. See key features on Page 152.

**Delegation** An implementation mechanism in which an object forwards or delegates a request to another object. The delegate carries out the request on behalf of the original object.

**Demeter law** States that any method of an object should invoke only methods belonging to: itself, any parameters that were passed in to the method, any objects it created, and, finally, any directly held component objects. The basic idea is to avoid invoking methods of a member object that is returned by another method.
**Design pattern** A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context.

**Design** The activity performed by a software developer that results in the software architecture of a system. Very often the term design is also used as a name for the result of this activity. The software design activity is commonly divided into the high-level design and the low-level design. The high-level design results in the structural subdivision of the system. It specifies the fundamental structure of the application. The low-level design results in more detailed planning like definition of interface, data structures...

**Domain** Denotes concepts, knowledge and other items that are related to a subject. Often used as ‘application domain’ to denote the problem area an application addresses.

**Duplication** One type of design patterns drawbacks. Duplication indicates that data information is scattered and duplicated in all the layers of an application, as design patterns often add several levels of indirection.

**Dynamic binding** See Polymorphism.

**G-16** See Power G-16.

**Encapsulation** The result of hiding a representation and implementation in an object. The representation is not visible and cannot be accessed directly from outside the object. Operations are the only way to access and modify an object’s representation.

**Facade** Structural pattern. Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use. See key features on Page 153.

**Framework** A set of cooperating classes that makes up a reusable design for a specific class of software. A framework provides architectural guidance by partitioning the design into abstract classes and defining their responsibilities and collaborations. A developer customizes the framework to a particular application by subclassing and composing instances of framework classes.

**Power Design** A professional application that creates complete audio solutions based on a facility layout.

**Power G-16** Professional electronic amplifier system for one to four zone business music applications.

**Power Config** A professional application designed for the setup and commissioning of an electronic amplifier.
**Functional Property** A particular aspect of a system’s functionality, usually related to a specified functional requirement. A functional property may be either made directly visible to users of an application by means of a particular function, or it may represent aspects of its implementation, such as the algorithm used to compute the function.

**Gang of Four** This expression refers to Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides who have written the seminal book “Design Patterns: Elements of Reusable Object-Oriented Software” [GHJV95].

**Gof** See Gang of Four.

**GUI** Graphical User Interface. The part of the program that the user sees and interacts with, as opposed to the part of the program that performs its internal processing.

**Implementation overhead** One type of design patterns drawbacks. Implementation overhead is an obvious consequence for the software engineering when implementing design patterns, as they are not supported by current languages. Moreover, it often consists of the implementation of several methods with only trivial behavior such as forwarding a message to another object or method.

**Inheritance** A relationship that defines one entity in terms of another. Class inheritance defines a new class in terms of one or more parent classes. The new class inherits its interface and implementation from its parents. The new class is called a subclass or a derived class. Class inheritance combines interface inheritance and implementation inheritance. Interface inheritance defines a new interface in terms of one or more existing interfaces. Implementation inheritance defines a new implementation in terms of one or more existing implementations.

**Instance variable** A piece of data that defines part of an object’s representation. Java uses the term data member or field.

**Instance** An object originated from a specific class. Often used as a synonym for object in an object-oriented environment. This term may also be used in other contexts.

**Interface** The set of all signatures defined by an object’s operations. The interface describes the set of requests to which an object can respond.

**Late binding** See Polymorphism.

**Layer** Layering is one of the most common techniques that software designers use to break apart a complicated software system. When thinking of a system in terms of layers, you imagine the principal subsystems in the software arranged in some form of layer cake, where each layer rests upon a lower layer. In this scheme the higher layer uses various services defined by the lower layer, but the lower layer is unaware of the higher layer. Furthermore each layer usually hides its lower layers from the layers above, so layer 4 uses the services of layer 3 which uses the services of layer 2, but layer 4 is unaware of layer 2.
Load (Electronics: R) Measurement of the resistance to the current flow in a conductor. It is measured in ohms (Ω). (See Ohm’s law)

Lost control One type of design patterns drawbacks. Lost control derives from traceability. As an application gets bigger and as more design patterns are added to it, we usually lose control of all the complex relationships established between all objects.

Mediator Behavioral pattern. Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently. See key features on Page 154.

Message Messages are used for the communication between objects or processes. In an object-oriented system the term message is used to describe the selection and activation of an operation or method of an object. This kind of message is synchronous, which means that the sender waits until the receiver finishes the activated operation.

Method Denotes an operation performed by an object. A method is specified within a class. The term is also used in “software development method”, which consists of a set of rules, guidelines and notations to be used by engineers during the development process.

Module A syntactical or conceptual entity of a software system. Often used as a synonym for component or subsystem. Sometimes, modules also denote compilation units or files. Other writers use the term as an equivalent to package when referring to a code body with its own name space. We use the term as stated in the first sentence.

Non-functional Property A feature of a system not covered by its functional description. A non-functional property typically addresses aspects related to the reliability, compatibility, efficiency, cost, ease of use, maintenance or development of a system.

Object An identifiable entity in an object-oriented system. Objects respond to messages by performing a method (operation). An object may contain data values and references to other objects, which together define the state of the object. An object therefore has state, behavior, and identity.

Observer Behavioral pattern. Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. See key features on Page 156.

Ohm’s law Basic law in electronics that states that a voltage of 1 volt across a resistance of 1 ohm will cause a current of 1 amp to flow. As a result, Ohm’s equation can be written as: \( V/R = I \).

Operation An object’s data can be manipulated only by its operations. An object performs an operation when it receives a request. In Java, operations are called methods.
Orthogonality  Principle that consists of decoupling objects that are unrelated and of designing objects that are self-contained. Also known as the “low coupling-high cohesion” principle.

Overriding  Redefining an operation (inherited from a parent class) in a subclass.

Parent class  The class from which another class inherits. Synonyms are superclass and base class.

Pattern language  A structured collection of patterns that build on each other to transform needs and constraints into an architecture. Pattern languages describe software frameworks or families of related systems.

Pattern  An abstraction that captures experience. We distinguish four kinds of patterns: analysis patterns, architectural patterns, design patterns, and idioms. The first category captures object modeling expertise from different domains. The second one helps in structuring software system into subsystems. And, the third one supports the refinement of subsystems and components.

PloP  Pattern Languages of Programs. An annual conference on the creation and documentation of patterns and pattern languages.

Polymorphism  The ability to substitute objects of matching interface for one another at run-time.

Poor Traceability  One type of design patterns drawbacks. Poor traceability points out that we lose track of design patterns during implementation.

Power  (Electronics: P) The amount of energy (in joules) converted by a component in a unit of time, usually a second. We measure electrical power in watts. The power (P) of an electric system is equal to its voltage (V) multiplied by the current (I): $P = VI$.

Proxy  Structural pattern. Provide a surrogate or placeholder for another object to control access to it. See key features on Page 155.

Refactor  To change code to improve its internal structure without changing its external behavior.

Refactoring  A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.

Relationship  A connection between components. A relationship may be static or dynamic. Static relationships show directly in source code. They deal with the placement of components within an architecture. Dynamic relationships deal with the interaction between components. They may not be easily visible from source code or diagrams.

Request  An object performs an operation when it receives a corresponding request from another object. A common synonym for request is message.

Resistance  (Electronics: R) See Load.
Responsibility  The functionality of an object or a component in a specific context. A responsibility is typically specified by a set of operations.

Role  The responsibility of component within a context of related components. An implemented component may take different roles, even within a single pattern.

RS-232  Standard for transmitting serial data by wire. RS-232 connections are used to attach personal computers to modems, and other hardware devices.

Run-time binding  See Polymorphism.

Signature  An operations signature defines its name, parameters, and return value.

Singleton  Creational pattern. Ensure a class only has one instance, and provide a global point of access to it. See key features on Page 157.

Software architecture  Description of the subsystems and components of a software system and the relationships between them. Subsystems and components are typically specified in different views to show the relevant functional and non-functional properties of a software system. The software architecture of a system is an artifact. It is the result of the software design activity.

Subclass  A class that inherits from another class. A subclass is also called a derived class.

Subsystem  A set of collaborating components performing a given task. A subsystem is considered a separate entity within a software architecture. It performs its designated task by interacting with other subsystems and components.

Template Method  Behavioral pattern. Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure. See key features on Page 158.

Traceability  See Poor Traceability.

UML  Unified Modeling Language. A language used for the visual representation of software systems. UML includes standard notation for representing classes and their attributes and associations, and it includes state transition, interaction, component, and deployment diagrams. We have joined a short tutorial of the UML on Page 159.

USB  Universal Serial Bus. A serial bus standard created in 1997 by Intel. USB connections are used to attach personal computers to their keyboards, printers, and other hardware devices.

Visibility  The visibility of a class member refers to its accessibility rights. There are four main types of accessibility for a class member, whether it is a field or a method. Public: any class can access the member. Protected:
only this class, its derived classes and anyone else in the same package can access it. Private: only this class can access it. Package: class of the current package can access it.

**Volt** Measurement unit of electrical pressure. (See Voltage)

**Voltage** (Electronics: V) The electrical pressure or the force that causes current to flow in a conductor. Also known as electromotive force. We measure voltage in volts.

**Watt** Measurement unit of electrical power. A watt actually represents a joule per second. (See Power)
Appendix A

Key Features of Design Patterns

This appendix is directly inspired from the Key Features presented in “Design Pattern Explained, a New Perspective on Object-Oriented Design” [ST02]. Additional information from “Design Patterns, Elements of Reusable Object-Oriented Software” [GHJV95] has also been added.

Creational Patterns

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Factory</td>
<td>Creates an instance of several families of classes.</td>
</tr>
<tr>
<td>Singleton</td>
<td>A class of which only a single instance can exist.</td>
</tr>
</tbody>
</table>

Structural Patterns

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adapter</td>
<td>Match interfaces of different classes.</td>
</tr>
<tr>
<td>Decorator</td>
<td>Add responsibilities to objects dynamically.</td>
</tr>
<tr>
<td>Facade</td>
<td>An interface that simplifies how to use an existing system.</td>
</tr>
<tr>
<td>Proxy</td>
<td>Surrogate for another object to control access to it.</td>
</tr>
</tbody>
</table>

Behavioral Patterns

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mediator</td>
<td>Define an object that encapsulates how a set of objects interact.</td>
</tr>
<tr>
<td>Observer</td>
<td>A way of notifying change to a number of classes.</td>
</tr>
<tr>
<td>Template Method</td>
<td>Defer the steps of an algorithm to a subclass.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Factory</td>
<td>150</td>
</tr>
<tr>
<td>Singleton</td>
<td>157</td>
</tr>
<tr>
<td>Adapter</td>
<td>151</td>
</tr>
<tr>
<td>Decorator</td>
<td>152</td>
</tr>
<tr>
<td>Facade</td>
<td>153</td>
</tr>
<tr>
<td>Proxy</td>
<td>155</td>
</tr>
<tr>
<td>Mediator</td>
<td>154</td>
</tr>
<tr>
<td>Observer</td>
<td>156</td>
</tr>
<tr>
<td>Template Method</td>
<td>158</td>
</tr>
</tbody>
</table>
The Abstract Factory Pattern: Key Features

**Intent**
You want to have families or sets of objects for particular clients (or cases).

**Problem**
Families of related objects need to be instantiated.

**Solution**
Coordinates the creation of families of objects. Gives a way to take the rules of how to perform the instantiation out of the client object that is using these created objects.

**Participants and Collaborators**
The Abstract Factory defines the interface for how to create each member of the family of objects required. Typically, each family is created by having its own unique ConcreteFactory.

**Consequences**
The pattern isolates the rules of which objects to use from the logic of how to use these objects.

**Implementation**
Define an abstract class that specifies which objects are to be made. Then implement one concrete class for each family. Tables or files can also be used to accomplish the same thing.

**GoF Reference**
Pages 87-96.

---

Figure A.1: Standard, Simplified View of the Abstract Factory Pattern.
The Adapter Pattern: Key Features

**Intent**
Match an existing object beyond your control to a particular interface.

**Problem**
A system has the right data and behavior but the wrong interface. Typically used when you have to make something a derivative of an abstract class we are defining or already have.

**Solution**
The Adapter provides a wrapper with the desired Interface.

**Participants and Collaborators**
The Adapter adapts the interface of an Adaptee to match that of the Adapter’s Target (the class it derives from). This allows the Client to use the Adaptee as if it were a type of Target.

**Consequences**
The Adapter pattern allows for preexisting objects to fit into new class structures without being limited by their interfaces.

**Implementation**
Contain the existing class in another class. Have the containing class match the required interface and call the methods of the contained class.

**GoF Reference**
Pages 139-150.

---

Figure A.2: Standard, Simplified View of the Adapter Pattern.
The Decorator Pattern: Key Features

**Intent**
Attach additional responsibilities to an object dynamically.

**Problem**
The object that you want to use does the basic functions you require. However, you may need to add some additional functionality to the object, occurring before or after the object’s base functionality.

**Solution**
Allows for extending the functionality of an object without resorting to subclassing.

**Participants and Collaborators**
The ConcreteComponent is the class having function added to it by the Decorators. Sometimes classes derived from ConcreteComponent are used to provide the core functionality, in which case ConcreteComponent is no longer concrete, but rather abstract. The Component defines the interface for all of these classes to use.

**Consequences**
Functionality that is to be added resides in small objects. The advantage is the ability to dynamically add this function before or after the functionality in the ConcreteComponent. Note: While a decorator may add its functionality before or after that which it decorates, the chain of instantiation always ends with the ConcreteComponent.

**Implementation**
Create an abstract class that represents both the original class and the new functions to be added to the class. In the decorators, place the new function calls before or after the trailing calls to get the correct order.

**GoF Reference**
Pages 175-184.

Figure A.3: Standard, Simplified View of the Decorator Pattern.
The Facade Pattern: Key Features

**Intent**
You want to simplify how to use an existing system. You need to define your own interface.

**Problem**
You need to use only a subset of a complex system. Or you need to interact with the system in a particular way.

**Solution**
The Facade presents a new interface for the client of the existing system to use.

**Participants and Collaborators**
It presents a specialized interface to the client that makes it easier to use.

**Consequences**
The Facade simplifies the use of the required subsystem. However, since the Facade is not complete, certain functionality may be unavailable to the client.

**Implementation**
Define a new class (or classes) that has the required interface. Have this new class use the existing system.

**GoF Reference**
Pages 185-193.

Figure A.4: Standard, Simplified View of the Facade Pattern.
The Mediator Pattern: Key Features

**Intent**
Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

**Problem**
Several objects are tightly coupled in the way they interact.

**Solution**
A Mediator object encapsulates collective behavior for controlling and coordinating the interactions of a group of objects.

**Participants and Collaborators**
The mediator serves as an intermediary that keeps objects in a group from referring to each other explicitly. The colleague objects only know the mediator, thereby reducing the number of interconnections.

**Consequences**
A mediator decouples colleagues. You can vary and reuse Colleague and Mediator classes independently. A mediator replaces many-to-many interactions with one-to-many interactions between the mediator and its colleagues.

**Implementation**
Define a central class that acts as a message routing service to all other classes.

**GoF Reference**
Pages 273-282.

![Diagram of the Mediator Pattern](image-url)

Figure A.5: Standard, Simplified View of the Mediator Pattern.
The Proxy Pattern: Key Features

**Intent**
Provide a surrogate or placeholder for another object to control access to it.

**Problem**
You have an existing object you want to use on another machine and you don’t want your client object to have to worry about making the connection (or even know). Or you need to add some new functionality to something that already exists.

**Solution**
You must defer the instantiation of objects, without the client objects knowing this is happening, to a Proxy. The Proxy acts just like the subject object and takes care of instantiating it when it’s required.

**Participants and Collaborators**
The Proxy controls access to the real subject. Proxy may refer to a Subject if the RealSubject and Subject interfaces are the same. Subject defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected. RealSubject defines the real object that the proxy represents.

**Consequences**
The proxy pattern introduces a level of indirection when accessing an object. The additional indirection can hide optimizations, access control, ... from the client.

**Implementation**
The Proxy pattern has a new object (the Proxy) stand in place of another, already existing object (the RealSubject). The proxy encapsulates any rules required for access to the real subject. The proxy object and the real subject object must have the same interface so that the Client does not need to know a proxy is being used. Requests made by the Client to the proxy are passed through to the RealSubject with the proxy doing any necessary processing to make the remote connection.

**GoF Reference**
Pages 207-217.

![Figure A.6: Standard, Simplified View of the Proxy Pattern.](image-url)
The Observer Pattern: Key Features

**Intent**
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

**Problem**
You need to notify a varying list of objects that an event has occurred.

**Solution**
Observers delegate the responsibility for monitoring for an event to a central object: the Subject.

**Participants and Collaborators**
The Subject knows its observers because the observers register with it. The Subject must notify the observers when the event in question occurs. The Observers are responsible both for registering with the Subject and for getting the information from the Subject when notified.

**Consequences**
Subjects may tell Observers about events they do not need to know if some Observers are interested in only a subset of events. Extra communication may be required if Subjects notify observer which then go back and request additional information.

**Implementation**
Have objects (Observers) that want to know when an event happens attach themselves to another object (Subject) that is watching for the event to occur or that triggers the event itself. When the event occurs, the Subject tells the Observers that it has occurred.

**GoF Reference**
Pages 293-303.

---

Figure A.7: Standard, Simplified View of the Observer Pattern.
The Singleton Pattern: Key Features

**Intent**
You want to have only one of an object but there is no global object that controls the instantiation of this object.

**Problem**
Several different client objects need to refer to the same thing and you want to ensure that you do not have more than one of them.

**Solution**
Guarantees one instance.

**Participants and Collaborators**
Clients create an instance of the Singleton solely through the getInstance method.

**Consequences**
Clients need not concern themselves whether an instance of the Singleton exists. This can be controlled from within the Singleton.

**Implementation**
Add a private static member of the class that refers to the desired object. Add a public static method that instantiates this class (and set this member’s value) and then returns the value of this member. Set the constructor’s status to private so that no one can directly instantiate this class and bypass the static constructor mechanism.

**GoF Reference**
Pages 127-134.

---

Figure A.8: Standard, Simplified View of the Singleton Pattern.
The Template Method Pattern: Key Features

**Intent**
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Redefine the steps in an algorithm without changing the algorithm’s structure.

**Problem**
There is a procedure or set of steps to follow that is consistent at one level of detail, but individual steps may have different implementations at a lower level of detail.

**Solution**
Allows for definition of substeps that vary while maintaining a consistent basic process.

**Participants and Collaborators**
The Template Method consists of an abstract class that defines the basic TemplateMethod (see figure below) classes that need to be overridden. Each concrete class derived from the abstract class implements a new method for the Template.

**Consequences**
Templates provide a good platform for code reuse. They also are helpful in ensuring the required steps are implemented. They bind the overridden steps together for each Concrete class, and so should only be used when these variations always and only occur together.

**Implementation**
Create an abstract class that implements a procedure using abstract methods. These abstract methods must be implemented in subclasses to perform each step of the procedure. If the steps vary independently, each step may be implemented with a Strategy pattern.

**GoF Reference**
Pages 325-330.

---

Figure A.9: Standard, Simplified View of the Template Method Pattern.
Appendix B

The UML Notation

This appendix briefly explains the features of the Unified Modeling Language (UML) that this thesis uses. It is a direct adapting of the appendix “UML AT A GLANCE” from the book “Design Patterns Java Workbook” [Met02].

UML provides conventional notation that this thesis applies to illustrate the design of object-oriented systems. Although UML is not properly complex, you can easily underestimate the richness of the features it provides. For a rapid introduction to most of the features of the UML, you can read “UML Distilled” [Fow99b]. By learning nomenclatures and notations, we learn to communicate at a design level, making us more productive.

Classes

Figure B.1 applies some of the UML features for illustrating classes. Following are notes on class diagrams.

- Indicate a package by placing the name of the package in a rectangle left-aligned with a larger box that may show classes and interfaces. Figure B.1 shows a portion of the thesis.source.amplifier package.

- UML does not require that a diagram shows everything about a portrayed element, such as the complete contents of a package or all the methods of a class.

- Draw a class by placing the name of a class centered in a rectangle. Figure B.1 shows two classes: Classification and Amplifier.

- You can show a class’s instance variables in a rectangle beneath the class name. The Amplifier class has instance variables name, price, and classification. Follow the variable’s name by a colon and the variable’s type.
You can show a class’s methods in a second rectangle beneath the class name. The Amplifier class has a constructor, a method with the same name as the class. The class also has at least three other methods: turnedOn(), getName(), and setClassification().

When a method accepts parameters, you should usually show them, as the setClassification() method does.

Variables in method signatures usually appear as the name of the variable, a colon, and the type of the variable. You may omit or abbreviate the variable name if its type implies the variable’s role.

You may indicate that an instance variable or a method is protected by preceding it with a pound sign (#). A plus sign (+) indicates that a variable or a method is public, and a minus sign (-) indicates that a variable or a method is private.

Indicate that an instance variable is static—and thus has class scope—by underlining it, as the turnedOn() method shows.

Make notes by drawing a dog-eared rectangle. The text in notes may contain comments, constraints, or code. Use a dashed line to attach notes to other diagram elements. Notes can appear in any UML diagram.
Class Relationships

Figure B.2 shows a few of UML’s features for modeling class relationships. Following are notes on class relationship notation.

- Show a class name or a method name in italics to indicate that the class or method is abstract.
- Use a closed, hollow arrowhead to point to a class’s superclass.
- Use a line between classes to indicate that instances of the classes are connected. Most commonly, a line on a class diagram means that one class has an instance variable that refers to the other class. The MachineComposite class, for example, uses a List variable that contains references to other machine components.
- Use a diamond to show that instances of a class contain a collection of instances of another class.

Figure B.2: A MachineComposite Object Contains either Machine Objects or Other Composites. The Customer Class Depends on the LikeMyStuff Class Without Instantiating it.

- An open arrowhead indicates navigability. Use it to emphasize that one
class has a reference to another and that the pointed-to-class does not have a back reference.

- A multiplicity indicator, such as 0..1, indicates how many connections may appear between objects. Use an asterisk (*) to indicate that zero or more instances of an object of a class may be connected to objects of an associated class.

- To show that a method may throw an exception, use a dashed arrow pointing from the method to the exception class. Label the arrow with a \texttt{<\texttt{throws}>} stereotype.

- Use a dashed arrow between classes to show a dependency that does not use an object reference. For example, the Customer class uses a static method from the LikeMyStuff recommendation engine.

\section*{Interfaces}

Figure B.3 shows the basic features for illustrating interfaces. Following are notes on interfaces.

- You can draw an interface by placing the text \texttt{<\texttt{interface}>} and the name of the interface in a rectangle, as Figure B.3 shows. You can use a dashed line and a close, hollow arrowhead to show that a class implements the interface.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{interface.png}
\caption{You Can Indicate an Interface with either an \texttt{<\texttt{interface}>} Stereotype or a Lollipop.}
\end{figure}

- You can also show that a class implements an interface by showing a line and circle, or “lollipop”, and the name of the interface.
• Interfaces and their methods are always abstract in Java. Oddly enough, interfaces and their methods do not necessary appear in italics, unlike abstract classes and abstract methods in classes.

Objects

A sequence diagram illustrates a sequence of objects calling methods of other objects, as Figure B.4 shows. Following are notes on sequence diagrams.

• You can show an object by giving its name and type, separated by a colon or a dash. You may optionally show just the name, or just a colon/dash and the type. In any case, underline the name and/or type of the object.

• The order of method calls is top to bottom, and the vertical dashed lines indicate the existence of the object over time.

• Arrows between the dashed lines represent method calls. The labels of those arrows represent the name of the method and the parameters passed in.

• When an object calls a method on itself, we draw a loop arrow whose begin and end are the same object.

Figure B.4: A Sequence Diagram Shows a Succession of Method Calls
Appendix C

Selectable List: Additional Code

Model Interface: SelectableListModel.java

```java
package source.selectablelist;

public interface SelectableListModel {
    public void addSelectableListModelListener (SelectableListModelListener listener);
    public void removeSelectiveListModelListener (SelectableListModelListener listener);
    public Object[] getItems();
    public void setItemState(Object item, boolean selected);
    public boolean getItemState(Object item);
    public void addItem(Object item);
    public void removeItem(Object item);
}
```

Abstract Model: AbstractSelectableListModel.java

```java
package source.selectablelist;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public abstract class AbstractSelectableListModel implements SelectableListModel {
    protected List listenerList;
    protected AbstractSelectableListModel() {
        listenerList = new ArrayList();
    }
}
```
public void addSelectableListModelListener(SelectableListModelListener listener) {
    listenerList.add(listener);
}

public void removeSelectableListModelListener(SelectableListModelListener listener) {
    listenerList.remove(listener);
}

protected void fireItemAdded(Object item) {
    SelectableListModelEvent e =
        new SelectableListModelEvent(this, item);
    Iterator listenerListIterator = listenerList.iterator();
    while (listenerListIterator.hasNext()) {
        ((SelectableListModelListener)listenerListIterator.next()).
            itemAdded(e);
    }
}

protected void fireItemRemoved(Object item) {
    SelectableListModelEvent e =
        new SelectableListModelEvent(this, item);
    Iterator listenerListIterator = listenerList.iterator();
    while (listenerListIterator.hasNext()) {
        ((SelectableListModelListener)listenerListIterator.next()).
            itemRemoved(e);
    }
}

protected void fireItemStateChanged(Object item, boolean newState) {
    SelectableListModelEvent e =
        new SelectableListModelEvent(this, item, newState);
    Iterator listenerListIterator = listenerList.iterator();
    while (listenerListIterator.hasNext()) {
        ((SelectableListModelListener)listenerListIterator.next()).
            itemStateChanged(e);
    }
}

Default Model Implementation: DefaultSelectableListModel.java

package source.selectablelist;
import java.util.Collection;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

public class DefaultSelectableListModel
    extends AbstractSelectableListModel {
    private Map itemsMap = new LinkedHashMap();
    public DefaultSelectableListModel() {
    }
    public DefaultSelectableListModel(Object[] items) {
        for (int i = 0; i < items.length; i++) {
            itemsMap.put(items[i], i);
        }
    }
}
public DefaultSelectableListModel(Collection items) {
    this(items.toArray());
}

public Object[] getItems() {
    return itemsMap.keySet().toArray();
}

public boolean getItemState(Object item) {
    return ((Boolean)itemsMap.get(item)).booleanValue();
}

public void setItemState(Object item, boolean selected) {
    Boolean oldValue = ((Boolean)itemsMap.get(item));
    if (selected != oldValue.booleanValue()) {
        if (selected) {
            itemsMap.put(item, Boolean.TRUE);
        } else {
            itemsMap.put(item, Boolean.FALSE);
        }
        fireItemStateChanged(item, selected);
    }
}

public void addItem(Object item) {
    if (!itemsMap.containsKey(item)) {
        itemsMap.put(item, Boolean.FALSE);
        fireItemAdded(item);
    }
}

public void removeItem(Object item) {
    if (itemsMap.containsKey(item)) {
        itemsMap.remove(item);
        fireItemRemoved(item);
    }
}

Model Listener Interface: SelectableListModelListener.java

package source.selectablelist;
import java.util.EventListener;

public interface SelectableListModelListener extends EventListener {
    public void itemAdded(SelectableListModelEvent event);
    public void itemRemoved(SelectableListModelEvent event);
    public void itemStateChanged(SelectableListModelEvent event);
}

Model Event: SelectableListModelEvent.java
package source.selectablelist;
import java.util.EventObject;

public class SelectableListModelEvent extends EventObject {
    private Object item;
    private boolean selected;
    public SelectableListModelEvent(SelectableListModel source,
            Object item) {
        this(source, item, false);
    }
    public SelectableListModelEvent(SelectableListModel source,
            Object item, boolean selected) {
        super(source);
        this.item = item;
        this.selected = selected;
    }
    public Object getItem() {
        return item;
    }
    public boolean isSelected() {
        return selected;
    }
}

Wrapper: SelectableList.java

package source.selectablelist;
import source.plafSelectableListUI;
import java.util.*;
import javax.swing.*;

public class SelectableList
extends JComponent
implements SelectableListModelListener{

    static {
        UIManager.put("SelectableListUI",
                "source.plaf.basic.BasicSelectableListUI");
    }

    public static final String
        MODEL_PROPERTY = "Model",
        UNSELECT_BUTTON_TEXT_PROPERTY = "TextUnselectButton",
        SELECT_BUTTON_TEXT_PROPERTY = "TextSelectButton",
        LEFT_LIST_LABEL_PROPERTY = "TextSelectLabel",
        RIGHT_LIST_LABEL_PROPERTY = "TextUnselectLabel";

    public static final int SELECT_BUTTON = 3;
    public static final int UNSELECT_BUTTON = 4;
    public static final int LEFT_LIST = 9;
    public static final int RIGHT_LIST = 10;
private SelectableListModel dataModel;
private Set modelListenerSet = new HashSet();

private String selectButtonText = new String("Select");
private String unselectButtonText = new String("Unselect");
private String leftListLabel = new String(" ");
private String rightListLabel = new String(" ");

public SelectableList() {
    this(new DefaultSelectableListModel());
}

public SelectableList(Object[] items) {
    this(new DefaultSelectableListModel(items));
}

public SelectableList(Collection items) {
    this(new DefaultSelectableListModel(items));
}

public SelectableList(SelectableListModel dataModel) {
    if (dataModel == null) {
        throw new IllegalArgumentException("dataModel must be non null");
    }
    setModel(dataModel);
    updateUI();
}

public SelectableListModel getModel() {
    return dataModel;
}

public Object[] getItems() {
    return dataModel.getItems();
}

public boolean getItemState(Object item) {
    return dataModel.getItemState(item);
}

public void setItemState(Object item, boolean selected) {
    dataModel.setItemState(item, selected);
}
public void addItem(Object item) {
    dataModel.addItem(item);
}

public void removeItem(Object item) {
    dataModel.removeItem(item);
}

public void setButtonText(int button, String text) {
    String oldText;
    switch (button) {
    case SELECT_BUTTON:
        oldText = selectButtonText;
        selectButtonText = text;
        firePropertyChange(SELECT_BUTTON_TEXT_PROPERTY, oldText, text);
        break;
    case UNSELECT_BUTTON:
        oldText = unselectButtonText;
        unselectButtonText = text;
        firePropertyChange(UNSELECT_BUTTON_TEXT_PROPERTY, oldText, text);
        break;
    }
}

public String getButtonText(int button) {
    switch (button) {
    case SELECT_BUTTON:
        return selectButtonText;
    case UNSELECT_BUTTON:
        return unselectButtonText;
    default:
        return null;
    }
}

public void setListLabel(int list, String label) {
    String oldLabel;
    switch (list) {
    case LEFT_LIST:
        oldLabel = leftListLabel;
        leftListLabel = label;
        firePropertyChange(LEFT_LIST_LABEL_PROPERTY, oldLabel, label);
        break;
    case RIGHT_LIST:
        oldLabel = rightListLabel;
        rightListLabel = label;
        firePropertyChange(RIGHT_LIST_LABEL_PROPERTY, oldLabel, label);
        break;
    }
public String getListLabel(int list) {
    switch (list) {
    case LEFT_LIST:
        return leftListLabel;
    case RIGHT_LIST:
        return rightListLabel;
    default:
        return null;
    }
}

public void updateUI() {
    setUI((SelectableListUI)UIManager.getUI(this));
}

public void setUI(SelectableListUI newUI) {
    super.setUI(newUI);
}

public SelectableListUI getUI() {
    return (SelectableListUI)ui;
}

public String getUIClassID() {
    return "SelectableListUI";
}

public void addSelectableListModelListener(
        SelectableListModelListener listener) {
    modelListenerSet.add(listener);
}

public void removeSelectableListModelListener(
        SelectableListModelListener listener) {
    modelListenerSet.remove(listener);
}

public void itemAdded(SelectableListModelEvent event){
    Iterator listenerSetIterator = modelListenerSet.iterator();
    while (listenerSetIterator.hasNext()){
        ((SelectableListModelListener)listenerSetIterator.next()).
            itemAdded(event);
    }
}

public void itemRemoved(SelectableListModelEvent event){
    Iterator listenerSetIterator = modelListenerSet.iterator();
    while (listenerSetIterator.hasNext()){
        ((SelectableListModelListener)listenerSetIterator.next()).
            itemRemoved(event);
    }
}
public void itemStateChanged(SelectableListModelEvent event){
    Iterator listenerSetIterator = modelListenerSet.iterator();
    while (listenerSetIterator.hasNext()){((SelectableListModelListener)listenerSetIterator.next()).
        itemStateChanged(event);
    }
}

Abstract UI Delegate: SelectableListUI.java

package source.plaf;
import javax.swing.plaf.ComponentUI;

public abstract class SelectableListUI extends ComponentUI {
    //empty
}

UI Delegate: BasicSelectableListUI.java

package source.plaf.basic;

import source.plafSelectableListUI;
import source.selectablelist.*;
import java.awt. *
import java.awt.event. *;
import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;
import java.util.*;
import javax.swing. *
import javax.swing.event.ListSelectionEvent;
import javax.swing.event.ListSelectionListener;
import javax.swing.plaf.ComponentUI;

public class BasicSelectableListUI extends SelectableListUI {
    protected SelectableList selectableList;
    protected SelectableListModel selectableListModel;

    private SelectableListModelListener selectableListModelListener;
    private PropertyChangeListener propertyChangeListener;
    private ActionListener actionListener;
    private ListSelectionListener listSelectionListener;

    protected JList selectedL;
    protected JLabel selectedLB;
    protected JScrollPane selectedS;

    protected JList unselectedL;
protected JLabel unselectedLB;
protected JScrollPane unselectedS;

protected JPanel selectUnselectJP;
protected JButton selectB;
protected JButton unselectB;

public static ComponentUI createUI(JComponent c) {
    return new BasicSelectableListUI();
}

protected SelectableListModel getModel() {
    return selectableListModel;
}

protected void setModel(SelectableListModel model) {
    if (selectableListModel != null) {
        if (selectableListModelListener != null) {
            selectableListModel.removeSelectableListModelListener(
                selectableListModelListener);
        }
    }
    selectableListModel = model;
    if (selectableListModel != null) {
        if (selectableListModelListener != null) {
            selectableListModel.addSelectableListModelListener(
                selectableListModelListener);
        }
    }
}

public void installUI(JComponent c) {
    if (c == null) {
        throw new NullPointerException("null component");
    }
    selectableList = (SelectableList)c;
    setModel(selectableList.getModel());
    initializeComponents(selectableList);
    installDefaults(selectableList);
    installListeners(selectableList);
    installKeyboardActions(selectableList);
    installComponents();
    updateView(selectableList);
}

protected void initializeComponents(SelectableList selectableList) {
    Icon icon;
    String text;
    selectUnselectJP = new JPanel();
    selectedL = new JList();
    selectedL = new JList();
    selectedL.setModel(new DefaultListModel());
selectedLB = new JLabel();
selectedLB.setFont(selectableList.getFont());
text = selectableList.getListLabel(selectableList.RIGHT_LIST);
if (text != null) {
    selectedLB.setText(text);
}
selectedS = new JScrollPane(selectedL);

unselectedL = new JList();
unselectedL.setModel(new DefaultListModel());

unselectedLB = new JLabel();
unselectedLB.setFont(selectableList.getFont());
text = selectableList.getListLabel(selectableList.LEFT_LIST);
if (text != null) {
    unselectedLB.setText(text);
}
unselectedS = new JScrollPane(unselectedL);

selectB = new JButton();
text = selectableList.getButtonText(selectableList.SELECT_BUTTON);
if (text != null) {
    selectB.setText(text);
}
selectB.setEnabled(false);

unselectB = new JButton();
text = selectableList.getButtonText(selectableList.UNSELECT_BUTTON);
if (text != null) {
    unselectB.setText(text);
}
unselectB.setEnabled(false);

protected void installDefaults(SelectableList selectableList) {
    selectedL.setCellRenderer(new SelectedLCellRenderer());
    unselectedL.setCellRenderer(new UnselectedLCellRenderer());
}

protected void installListeners(SelectableList selectableList) {
    propertyChangeListener = createPropertyChangeListener();
    selectableList.addPropertyChangeListener(propertyChangeListener);

    selectableListModelListener = createSelectableListModelListener();
    selectableList.addSelectableListModelListener(selectableListModelListener);

    actionListener = createActionListener();
    selectB.addActionListener(actionListener);
    unselectB.addActionListener(actionListener);

    listSelectionListener = createListSelectionListener();
}
unselectedL.addListSelectionListener(listSelectionListener);
selectedL.addListSelectionListener(listSelectionListener);
}

protected void installKeyboardActions(SelectableList selectableList) {
}

protected void installComponents() {
    GridBagLayout gridbag = new GridBagLayout();
    GridBagConstraints constraints = new GridBagConstraints();
    ((JComponent)selectableList).setLayout(gridbag);

    // install the selectUnselect panel---selectUnselectJP
    selectUnselectJP.setLayout(new GridLayout(2, 1));
    selectUnselectJP.add(selectB);
    selectUnselectJP.add(unselectB);

    // setting list position and attributes---unselectedLB
    buildConstraints(constraints, 0, 0, 1, 1, 0, 0);
    constraints.anchor = GridBagConstraints.SOUTHWEST;
    constraints.insets = new Insets(0, 0, 0, 5);
    gridbag.setConstraints(unselectedLB, constraints);
    ((JComponent)selectableList).add(unselectedLB);

    // setting list position and attributes---unselectedS
    buildConstraints(constraints, 1, 0, 1, 2, 100, 50);
    constraints.fill = GridBagConstraints.BOTH;
    constraints.anchor = GridBagConstraints.WEST;
    constraints.insets = new Insets(0, 0, 0, 5);
    gridbag.setConstraints(unselectedS, constraints);
    ((JComponent)selectableList).add(unselectedS);

    // setting list position and attributes---selectedLB
    buildConstraints(constraints, 0, 2, 1, 1, 0, 0);
    constraints.anchor = GridBagConstraints.SOUTHWEST;
    constraints.insets = new Insets(0, 5, 0, 0);
    gridbag.setConstraints(selectedLB, constraints);
    ((JComponent)selectableList).add(selectedLB);

    // setting list position and attributes---selectedS
    buildConstraints(constraints, 1, 2, 1, 2, 100, 50);
    constraints.fill = GridBagConstraints.BOTH;
    constraints.anchor = GridBagConstraints.EAST;
    constraints.insets = new Insets(0, 5, 0, 0);
    gridbag.setConstraints(selectedS, constraints);
    ((JComponent)selectableList).add(selectedS);

    // setting button position and attributes---selectUnselectJP
    buildConstraints(constraints, 1, 1, 1, 1, 50, 0);
    constraints.fill = GridBagConstraints.HORIZONTAL;
    // constraints.anchor = GridBagConstraints.SOUTH;
    constraints.insets = new Insets(0, 0, 5, 0);
    gridbag.setConstraints(selectUnselectJP, constraints);
    ((JComponent)selectableList).add(selectUnselectJP);
Selectable List

```java
((JComponent)selectableList).setOpaque(true);
}

/** updates the display of the two Jlist */
protected void updateView(SelectableList selectableList) {
    Object[] items = getModel().getItems();
    DefaultListModel unselectedLModel =
        ((DefaultListModel)unselectedL.getModel());
    unselectedLModel.clear();
    DefaultListModel selectedLModel =
        ((DefaultListModel)selectedL.getModel());
    selectedLModel.clear();
    for (int i = 0; i < items.length; i++) {
        if (!selectableList.getModel().getItemState(items[i])) {
            unselectedLModel.addElement(items[i]);
        } else {
            unselectedLModel.addElement(items[i]);
            selectedLModel.addElement(items[i]);
        }
    }
}

public void uninstallUI(JComponent c) {
    SelectableList selectableList = (SelectableList)c;
    uninstallDefaults(selectableList);
    uninstallListeners(selectableList);
    uninstallKeyboardActions(selectableList);
    uninstallComponents();
    selectableList = null;
    selectableListModel = null;
}

protected void uninstallDefaults(SelectableList selectableList) {
}

protected void uninstallListeners(SelectableList selectableList) {
    selectableList.removePropertyChangeListener(propertyChangeListener);
    selectableList.removeSelectableListModelListener(selectableListModelListener);
    selectB.removeActionListener(actionListener);
    unselectB.removeActionListener(actionListener);
    unselectedL.removeListSelectionListener(listSelectionListener);
    selectedL.removeListSelectionListener(listSelectionListener);
    propertyChangeListener = null;
    selectableListModelListener = null;
    actionListener = null;
    listSelectionListener = null;
}

protected void uninstallKeyboardActions(SelectableList selectableList) {
}
protected void uninstallComponents() {
    ((JComponent) selectableList).remove(unselectedLB);
    ((JComponent) selectableList).remove(unselectedS);
    ((JComponent) selectableList).remove(selectedLB);
    ((JComponent) selectableList).remove(selectedS);
    ((JComponent) selectableList).remove(selectUnselectJP);
}

protected void refreshButtonState() {
    unselectedL.clearSelection();
    selectedL.clearSelection();
    selectB.setEnabled(false);
    unselectB.setEnabled(false);
}

protected PropertyChangeListener createPropertyChangeListener() {
    return new PropertyChangeHandler();
}

protected SelectableListModelListener createSelectableListModelListener() {
    return new SelectableListModelHandler();
}

protected ActionListener createActionListener() {
    return new ActionHandler();
}

protected ListSelectionListener createListSelectionListener() {
    return new ListSelectionHandler();
}

private void buildConstraints(GridBagConstraints gbc, int gy, int gx, int gw, int gh, int wy, int wx) {
    gbc.gridy = gy;
    gbc.gridx = gx;
    gbc.gridwidth = gw;
    gbc.gridheight = gh;
    gbc.weighty = wy;
    gbc.weightx = wx;
    gbc.fill = GridBagConstraints.NONE;
    gbc.anchor = GridBagConstraints.CENTER;
    gbc.insets = new Insets(0, 0, 0, 0);
}

////////////////////////////////////////////////////////////////////////
/////////////// INNER CLASSES ///////////////
////////////////////////////////////////////////////////////////////////

class PropertyChangeHandler implements PropertyChangeListener {

//controller
public void propertyChange(PropertyChangeEvent event) {
    // return if wrong source object
    if (event.getSource() != selectableList) {
        return;
    }
    String propertyName = event.getPropertyName();
    if (propertyName.equals(SelectableList.MODEL_PROPERTY)) {
        setModel((SelectableListModel)event.getNewValue());
        updateView(selectableList);
    } else if (propertyName.equals(SelectableList.SELECT_BUTTON_TEXT_PROPERTY)) {
        selectB.setText((String)event.getNewValue());
    } else if (propertyName.equals(SelectableList.UNSELECT_BUTTON_TEXT_PROPERTY)) {
        unselectB.setText((String)event.getNewValue());
    }
}

class SelectableListModelHandler implements SelectableListModelListener {
    /** Invoked when items are added to the model */
    public void itemAdded(SelectableListModelEvent event) {
        Object item = event.getItem();
        DefaultListModel listModel = ((DefaultListModel)unselectedL.getModel());
        listModel.addElement(item);
        refreshButtonState();
    }

    /** Invoked when items are removed from the model */
    public void itemRemoved(SelectableListModelEvent event) {
        Object item = event.getItem();
        DefaultListModel unselectedLModel = ((DefaultListModel)unselectedL.getModel());
        DefaultListModel selectedLModel = ((DefaultListModel)selectedL.getModel());
        unselectedLModel.removeElement(item);
        selectedLModel.removeElement(item);
        refreshButtonState();
    }

    /** Invoked when items are selected */
    public void itemStateChanged(SelectableListModelEvent event) {
        updateView(selectableList);
        refreshButtonState();
    }
}

class ActionHandler implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == selectB) {
            Object[] items = unselectedL.getSelectedValues();
            for (int i = 0; i < items.length; i++) {
                getModel().setItemState(items[i], true);
            }
        } else if (e.getSource() == unselectB) {
            Object[] items = selectedL.getSelectedValues();
            for (int i = 0; i < items.length; i++) {
                getModel().setItemState(items[i], false);
            }
        }
    }
}
if (e.getSource() == unselectB) {
    Object[] items = selectedL.getSelectedValues();
    for (int i = 0; i < items.length; i++) {
        getModel().setItemState(items[i], false);
    }
    refreshButtonState();
}

class ListSelectionHandler implements ListSelectionListener {
    // tells if the event come from a selection in another list
    boolean sourceIsaList = false;

default { if (sourceIsaList) {
            sourceIsaList = false;
    } else {
        if (e.getSource() == unselectedL) {
            Object[] items = unselectedL.getSelectedValues();
            boolean enable = false;
            for (int i = 0; i < items.length; i++) {
                if (!getModel().getItemState(items[i])) {
                    enable = true;
                }
            }
            selectB.setEnabled(enable);
            unselectB.setEnabled(false);
            sourceIsaList = true;
            selectedL.clearSelection();
        }
        if (e.getSource() == selectedL) {
            unselectB.setEnabled(true);
            selectB.setEnabled(false);
            sourceIsaList = true;
            unselectedL.clearSelection();
        }
    }
}

class UnselectedLCellRenderer extends JLabel implements ListCellRenderer {
    public Component getListCellRendererComponent(final JList list,
        final Object value, final int index, final boolean isSelected,
        final boolean cellHasFocus) {
        setOpaque(true);
        setText(value.toString());
        if (isSelected) {
            if (getModel().getItemState(value)) {
                // mouse clicked on cell & item selected
                setForeground(Color.gray);
            }
        }
    }
}
setBackground(list.getBackground());
}
else {
    // mouse clicked on cell & item not selected
    setForeground(list.getSelectionForeground());
    setBackground(list.getSelectionBackground());
}
else { // reverse
    setBackground(list.getBackground());
    if (getModel().getItemState(value)) {
        // mouse not clicked on cell & item selected
        setForeground(Color.gray);
    }
    else { // mouse not clicked on cell & item not selected
        setForeground(list.getForeground());
    }
}
return this;
}

class SelectedLCellRenderer extends JLabel implements ListCellRenderer {
    public Component getListCellRendererComponent(final JList list,
            final Object value, final int index, final boolean isSelected,
            final boolean cellHasFocus) {
        setOpaque(true);
        setText(value.toString());
        setBackground(isSelected ? list.getSelectionBackground() :
            list.getBackground());
        setForeground(isSelected ? list.getSelectionForeground() :
            list.getForeground());
        return this;
    }
}
References


REFERENCES


[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.


