Extensibility Design Patterns From The Initial Stage of Application Life-Cycle

An Empirical Study Using GoF Patterns and Swift Programming language

Theepan Karthigesan

Thesis submitted for the degree of
Master in Programming and Networks
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2018

Extensibility Design Patterns From The Initial Stage of Application Life-Cycle

An Empirical Study Using GoF Patterns and Swift Programming language

Theepan Karthigesan

Abstract

We consider extensibility in software development as an essential characteristic due to the fast growing technologies that demand a quick response from the developers. This study will therefore mainly focus on achieving extensibility from the beginning of an application life-cycle, the design phase.

During this study, we explored and discussed the GoF (Gang of Four) patterns and pointed out those we considered as extensibility patterns. Afterwards, we picked six patterns from the pointed out list and implemented using a case-study application to demonstrate how they provide support for extensibility in a real-world application. We used iOS development platform and Swift programming language in the case-study. To limit the scope of this study, we decided to only look into GoF design patterns.

Evaluation of the case-study is done by putting the patterns up against some extensibility characteristics based on our subjective view, data from the self-designed extension scenarios tables and the lessons from implementation of the six patterns using the case-study. Decoupling and encapsulation are also taken into consideration as evaluation criteria, but advanced measurement of these criteria are out of the scope; thus, we rely on our subjective view on decoupling and encapsulation.

We divided the found extensibility pattern into two groups, one with the less complex patterns where extension generates less work than the other group in which contains more challenging pattern when it comes to extending. The results also show that in some cases the complexity grows by increased decoupling, which in turn will lead to more amount of work when extending, that is the case in the second group.

Acknowledgements

I am using this opportunity to express my gratitude to Professor Eric Bratley Jull who supported me through the learning process of this master's thesis. I am thankful for his guidance, amazing motivational speech, and useful critiques.

Contents

List of Figures

Listings

List of Tables

Part I

Introduction

1

Part I: Introduction

Introduction

1.1 Designs Patterns

Designs patterns is a kind of pension fund for software development. A pension fund is a saving account to ensure that you have enough money to spend when you retire from work. The small amount of money you invest in the fund every month is insurance for your future. In software development, design patterns are insurance against future changes. The payout can be dramatic benefits, such as avoiding painful and time-consuming rework of existing code when change or extension is needed.

Let us illustrate the term "Design pattern" with another real-world example. If you are building a new house from scratch, it is not necessary to find a new solution for every problem; instead, you can follow a given pattern. Rooms, doors, steps, and roof, etc. can be built and connected by following patterns that have already been adopted by other house builders. Even though this example describes the patterns in houses or buildings; the same principle applies to software development. Instead of talking about rooms and doors, we are dealing with classes and objects, but the core idea for both sets of entities is pinpointing old problems and reusing existing solutions.

In this empirical study, we explore and discuss design patterns from the GoF [8] list to find those that can be used in design phase of an application life-cycle to achieve extensibility.

1.2 Motivation

The motivation for this study is to find extensible patterns that may be taken into consideration at the design phase of application development. A robust designed application with extensibility in mind may be less challenging to expand with new features. We are therefore interested in knowing if the design patterns can solve such problem. To limit the scope of this study, we decided to only look into GoF design patterns. It is also important for us to point out those that require the least rework and those that are more challenging when expanding.

1.3 Goals

The primary goal is to find design patterns from the GoF list that are suitable for extensibility from the initial phase of an application lifecycle. It is important that these patterns contribute to design a robust application that makes the future extensions easy as possible.

1.4 Approach

To achieve the goal of finding the proper design patterns for extensibility, we discussed the GoF design patterns and pointed out those we considered as extensibility patterns based on their purpose. Furthermore, we picked six patterns from the pointed out list; two from each design pattern category that we found to fit into our case-study application. Purpose of the case-study is to examine the extensibility patterns from a practical perspective.

1.5 Scope

This study focuses on achieving extensibility from the initial phase of application development using design patterns. We look only into the 23 GoF design pattern. The patterns designed to increase the performance, minimize the memory allocation, extend a tightly coupled software and other purposes that do not relate to achieving extensibility are not in this scope. Quantitative measurement of these design patterns is also not in this scope. Instead, we use a self-designed scenarios table for some measurements.

1.6 Research/Work Done

We looked into the GoF design patterns and pointed out those we considered to support extensibility from the design phase of application development based on their purpose and our subjective view. We have also uniquely done measurements of extensibility, utilizing some extension scenarios table designed for this study by ourselves, where we look at the pattern's design differently. We see the supporting classes in which the data or client component uses to connect the building blocks in a design pattern, as a pattern component.

Furthermore, we discussed the pointed out patterns more in detail and implemented six of them using the case-study to evaluate their extensibility characteristic.

We also described the patterns that were excluded and the reason for not including them.

1.7 Evaluation

We looked into extensibility characteristics based on our subjective view, measured data from extension scenarios tables and the lessons from implementation of the six patterns using the case-study. Decoupling and encapsulation are also taken into consideration as evaluation criteria. However, quantitative measurements of these criteria are out of the scope. Thus, we rely on our subjective view on decoupling and encapsulation.

1.8 Results

The results show that some extensibility patterns are more challenging than the majority of them, regarding extensibility characteristic. We support this statement based on the data from the extension scenarios, our subjective view, and experience from the implementation. Based on the results, we divided these patterns into two groups, *"Less complex patterns"* and *"Challenging patterns"*. We also picked one pattern from each category that we believe has the advantage of implementing or should at least taken into consideration at the design phase of application life-cycle.

1.9 Contributions

Our contributions are the design patterns that we pointed out as extensible patterns from the GoF list and the implementation of six patterns using the case-study. We believe that the patterns we recommend at the end of this study will contribute to design a robust application from the beginning of applications development life-cycle.

1.10 Limitations

To limit the scope of this study, we do not perform quantitative measurement of decoupling or encapsulation. However, we analyze the patterns based on our subjective view and the analysis from self-designed extension scenarios tables.

1.11 Conclusion

In some extensibility patterns, an extension generates work in only one component, which makes it less complex to implement, understand and extend. We found that in some cases complexity grows by increased decoupling, which in turn will lead more amount of work when extending, especially when the creation of new classes in the data component need to have a one-to-one relationship with the pattern component classes. Increased decoupling may be beneficial for code maintainability and understandability, but it requires quite a lot of work when extending. This finding is the opposite of the exportation we had at the beginning of this study. We believed that decoupling had only positive intentions on extensibility.

As the result, we have divided the extensibility patterns into two groups, one with the less complex patterns where extension generates less work than the other group in which contains more challenging patterns when it comes to extending.

1.11.1 Less complex patterns

In this group, the average number of needed actions when extending is less then 2,5.

- **Creational:** Builder
- **Structural:** Bridge and Facade

- **Behavioral:** Template method, Mediator, Observer, Strategy and Visitor

1.11.2 Challenging patterns

In this challenging group average number of needed actions exceeds 2,5.

- **Creational:** Factory Method and Abstract Factory
- **Behavioral:** Command and Iterator

During this study, we made up one favorite from each category that we believe has the advantage of implementing or should at least taken into consideration at the design phase of application life-cycle. Builder, Facade, and Visitor are patterns that not only require the least actions when extending, but they also solve some conmen problems every application may have, such as, encapsulating configuration of an object, hide complexity behind wrapped interface and change behavior of an object at runtime. These may not sound like extensible features, but in fact, these flexibilities make it easier to expand the system, which is described in the scenarios tables in chapter 7 and 8.

1.12 Further work

We excluded some patterns and considered these as not extensibility patterns due to their purpose. This decision was done based on our assumptions. However, it would be interesting to explore some of the excluded patterns that had in our point view another purpose then solving extensibility problems. The future research may then focus on the excluded patterns to see if they also can be used as extensibility patterns.

We are divided the excluded patterns into three groups:

- **Patterns for extension after design phase:**
  - Adatper and Decorator
- **Performance related patterns:**
  - Singleton, Prototype and Flyweight
- **Other patterns:**
  - Composite, Interpreter, Chain of Responsibility, Memento and State

The interesting patterns to look into when it comes to extensibility is the group of other patterns that we struggled to decide its characteristics during this study.

Based on the scope of this study, we looked only into the Go4 list and implemented six of the twelve pointed out patterns. Further work may also look into other design patterns or implement the remaining the six patterns using the same case-study application.

Part II

Background

2

Part II: Background

Introduction

In this part, we do a review of related literature and works. Furthermore, we describe what extensibility means in software development and concept of design patterns.

2.1 Outline

We have divided this part into three chapter as the following bullet points explain.

- **Chapter 3: Literature Review and Related Works**
  We describe some the books, articles and research papers related to extensibility and design patterns.
- **Chapter 4: Extensibility in Software Development**
  In this chapter, we look into what achieving extensibility in software development means.
- **Chapter 5: Design Patterns in Software Development**
  We describe the design patterns in more detail and look into the Go4 list to prepare for our research part.

3

Part II: Background

Literature Review and Related Works

3.1 Introduction

There are a lot of books, articles and research papers regarding design patterns out there. One can even find works related to extensibility, but as of writing none of these are focusing on achieving extensibility from the design phase of an application life-cycle, especially using GoF design patterns and the way we do our empirical study. We also do measurements of extensibility uniquely, utilizing some extension scenarios table designed for this study by ourselves, where we look the pattern design differently. We see the supporting classes in which the data or client component uses to connect the building blocks in a design pattern, as a pattern component.

3.2 Primary Studies

The book *Design Patterns: Elements of Reusable Object-Oriented Software* [8] from 1995 is a must-read for anyone who is interested in object-oriented programming and design patterns. The authors Erich Gamma, Ralph Johnson, John Vlissides and Richard Helm, also called Gang of four (Go4) [10, p. 135]. This book raised the popularity of design patterns in software development when it was out and still cited by a lot of writings. The programming languages in which the patterns were demonstrated in this book are Smalltalk and C++. After the release of this book, lots of programming languages and developers have taken advantage of these patterns. We are using these patterns to experiment extensibility, but with a different approach because we are focusing on achieving extensibility from the initial stage of application development to make it successful in the future.

Pro Design Patterns in Swift [7] and *Pro Objective-C Design Patterns for iOS* [4] are two books that show how to put design patterns into action in iOS application development. Both of these are describing design patterns in iOS, but differs from the language in which the patterns are explained and implemented. We used these books to acquire knowledge of programming in the iOS platform so that we could implement some of the design patterns in our case-study using Swift language. We were also interested in to see how other authors interpret GoF design patterns. From this book, we can see that patters can be customized based on the needs or even to fit into specific programming language's platform.

Analyzing Design Patterns for Extensibility [2]. In this paper, the author is focusing on achieving extensibility from the initial phase of an application life-cycle. It emphasizes the importance of thinking all the possible changes that may need for the application in the future. Developers need to choose the proper design patterns that are suitable for the application at the beginning and analyze them extensibility before moving forward. This paper is not analyzing the GoF list, but describe some ideas of how to think forward and use design patterns to achieving extensibility. It also describes the difference between extensibility and scalability that is often confused with each other.

Testing extensible design patterns in object-oriented frameworks through scenario templates [22]. This paper addresses testing extensible design patterns in an object-oriented framework. Testing techniques that are used in this paper is not same way we are doing our evaluation. However this paper's work relates the term extensibility to the software development and design patterns.

Encapsulation, reusability and extensibility in object-oriented programming languages [16]. This paper is a comparative study of object-oriented programming and its support for encapsulation, reusability and extensibility. It describe the importance of the encapsulation that we find in the object oriented programming language, and the inheritance mechanism that allows extensibility.

Decoupling metrics for services composition [18]. In this paper, the authors describe a way for evaluating decoupling between service-oriented components in the service composition. They are focusing on web services and the importance of decoupling the components which have impacts on the understandability, maintainability, reliability, testability, and reusability of software components.

A Possible Composite Design Pattern for Immature REST APIs [13]. This study focus on implementing combined design patterns into a multithread task based application and analysing the output to measure its maintainability, reusability and testability.

Design Patterns In A Smartphone Environment [15] This study looks into implementing design patterns in an Android environment and how they fit that development platform. It is also addressing complexity of patterns and the related maintenance needed, such as when extending some part.

Assessing the Impact of Using Design Patterns of Enterprise Application Architecture [3]. This study uses several well-established methods to measure complexity and maintainability of enterprise application architecture. The author i is focusing on four design patterns, Model View Controller, Page Controller, Front Controller and Template View. He emphasizes a lot of complexity and maintainability impacts when using design patterns, which is an interesting topic and can relate to our focus on extensibility characteristics.

3.3 Summary

Summary of literature and related works:

- **Books**
  - Design Patterns - Elements of Reusable Object-Oriented Software [8].
  - Pro Design Patterns in Swift [7].
  - Pro Objective-C Design Patterns for iOS [4].
  - Swift 2 Design Patterns [14]
- **Articles**
  - Analyzing Design Patterns for Extensibility [2].
  - Testing extensible design patterns in object-oriented frameworks through scenario templates [22].
  - Encapsulation, reusability and extensibility in object-oriented programming languages [16].
  - Decoupling metrics for services composition [18].

- **Thesis**
  - A Possible Composite Design Pattern for Immature REST APIs [13].
  - Design Patterns In A Smartphone Environment [15].
  - Assessing the Impact of Using Design Patterns of Enterprise Application Architecture [3].

4

Part II: Background

Extensibility in Software Development

4.1 Introduction

Technologies are continuously growing, and the demand for extensible software is increasing, which means that developers are forced to consider the extensibility when designing applications. Nowadays and modern applications need to be extended by other development teams as well as the original team. The new team, who are not familiar with the existing code do not want to make changes to the core part of the application to avoid error to the current functions.

Extensibility in software engineering is the capacity to extend functionality without or minimal rework of existing code or design. Making changes in existing code of an application might not be easy in all situations. For a small and straightforward application, the changes in existing code might not affect the development phase negatively since the recompiling and redeploying can be done quickly and efficiently. A complex application that is not loosely coupled is very hard to extend because every part is interdependent. The words loosely coupling and decoupling have almost the same meaning and are used for each other in this study. To be specific, decoupling means there is no coupling at all between two components, but in another hand loosely coupling is trying to keep the coupling between many components as loose as possible [2].
.

Extensibility must not be confused with flexibility, which are also a part of software development strategy. That being said, it does not mean that flexibility is out of the focus in this study; in most cases, a system must also be flexible in order to extend its functionality.

Extensibility depends on how the different modules depend on each other, for example, up to what extent they are coupled to each other. An extensible design should be loosely coupled which means low interdependency. As the coupling increases, the dependence between the modules also increases which means any change made to a module will result in changes in the other modules also. The main aim of extensibility is to minimize the impact when any change has been made to the existing system.

A simple example of extensibility design is the traditional object orientated programming and inheritance, as shown in fig. 4.1. We can see that data model can be extended using inheritance without affecting the calling component. This is due to the decoupling of the different components and encapsulating objects, which are briefly explained in the following sections.

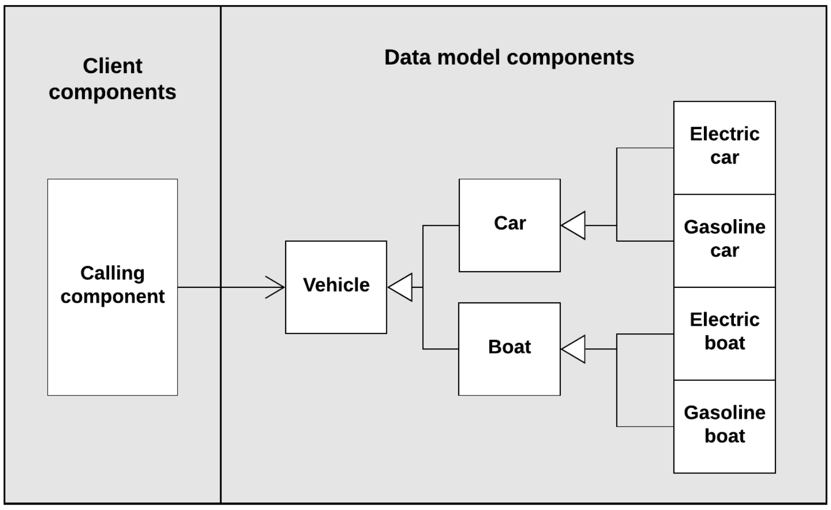


Figure 4.1
Relational between calling and data model components
UML Diagram - Left side arrow: association, Right side arrows: Inheritance

Use of design patterns can help the designer to evaluate the design for extensibility even before starting the implementation. When some commonly used patterns such as factory pattern, strategy pattern or visitor pattern are used and by analysing the structure itself the designer will get an idea whether any change can be made in the code without affecting the other parts. Similarly, each and every design pattern in the system need to be checked if it allows changes without affecting the other components of the application. It can be easily checked by how loosely or tightly the components are coupled. As the coupling increases, extensibility decreases. So the designer has to choose a design pattern for that particular purpose and which has the least coupling. If one wants to fulfill extensibility needs for their application, it must be appropriately planned from the starting of the application life-cycle [2]. Extensible design patterns must also be able to allow developers to add new classes and methods to the system either at compile time or runtime.

An example of extensible application could be a banking system that provides services to the different type of customer. The services might be accounts, loans, credit card and fund. It is said to be extensible when it is possible to add more services, such as savings account and insurance without making many changes in the existing system. After the extension, the system should keep existing services.

4.2 Design Phase of Software

The critical phase for achieving extensibility for software is the design phase. In this phase, the designers and developers must be proactive and think about extensibility and the future needs that might come.

Developing an extensible software needs proper planning and understanding of the future changes in which the application has to face. A best practice is to avoid tight coupling when designing application, fig. 4.2 on the next page illustrates an example of a tightly coupled software. This kind of designed software is extremely hard to extend because all the objects depend on each other. As shown in the same figure, if one of the objects changes it impacts three others. For example, if B changes, it will affect A, C, and D.

Loosely coupling is far more practical; the objects do not have many dependencies as shown in fig. 4.3 on the following page. To achieve extensibility in software development one should adopt more or less this policy. If A changes, it does not impact any other objects. However, it can be challenging to avoid all dependencies, but one should always try to minimize the total amount of dependencies [12].

It takes amount of energy and time to build a large system. The future development cost can be dramatically reduced by design it properly from the beginning. Reuse of components is also a huge beneficial.

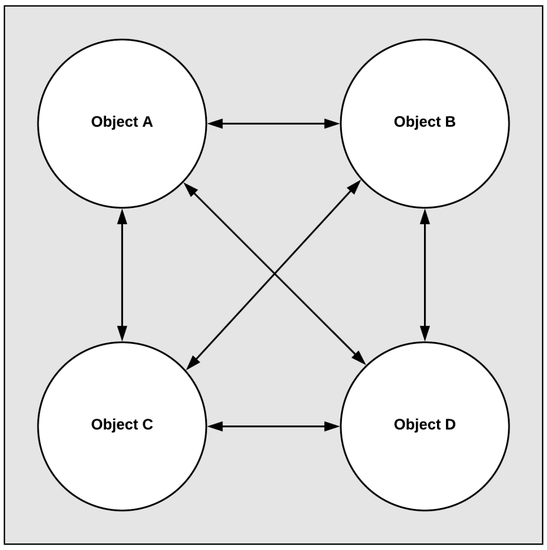


Figure 4.2
Tightly coupled software
O : Object, <->: Two-way relationship, ->: One-way relationship

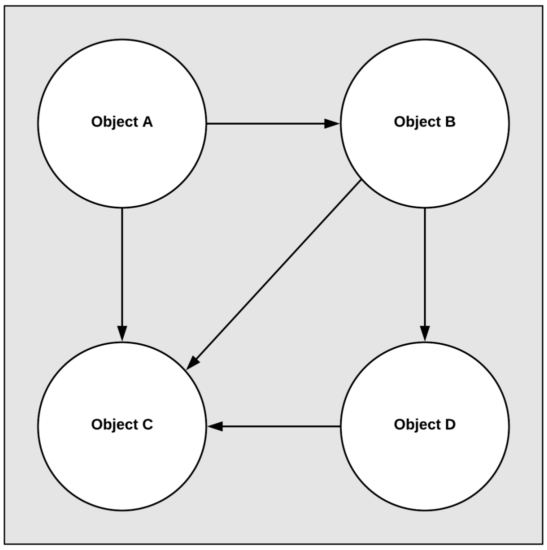


Figure 4.3
Loosely coupled software
O : Object, <->: Two-way relationship, ->: One-way relationship

4.2.1 Decoupling

Coupling is all about the relationship between two classes in a software system. Decoupling is ensuring that two different software parts are not tightly dependent on each other [19]. When a class have reference to another class, or communicates with it, it is said to depend on that other class, which also means these classes are coupled. When a Class X is heavily coupled to Class Y, any changes on Class Y will affect the Class X. This is called tight coupling. However, if the coupling between these classes is light, any changes on Class Y may not affect the Class X, which is a loose coupling. Decoupling in other hand means there is no dependency between class X and Y

Decoupling of components in very important in a large development project where there are many developer involved. Avoiding decoupling in a system in which contains many components that are changed or extended over time will increase development cost. One developer might not have information about the components that are build or modified by other people, but how the system interact between his responsible area and others.

4.2.2 Encapsulating

Encapsulation minimize the number of components that need to be changed as it undergoes new extensions over a period of time to meet demands from new specifications. Encapsulation is hiding the inner workings of a software component behind a defined interface [19]. Decoupling is ensuring that two different software components are not tightly dependent on one another. Encapsulated components can be accessed by clients via an interface. All the details and tasks are hidden behind this wrapped interface. The primary goal here is to let the implementation part extend or change without having to update the clients that are using it.

4.3 Summary

We summarize this chapter by saying that decoupling, also knows as loosely coupling and encapsulating are the essential characteristics of an extensible design. To build an application with such traits, we need to follow some rules or patterns. It is in this context that design patterns could be a problem solver. In the following chapter, we describe what design patterns are and how it could be used in the software development to solve problems.

5

Part II: Background

Design Patterns in Software Development

5.1 Introduction

In software development, the term “Design pattern” describes a reusable solution to a problem. It’s not a class or a library as it is known from the programming languages, but much more than that. It is more like a template that must be used in a correct situation. Patterns must also be language-independent, which means that, in most cases, they should be implemented in all programming languages. By adapting design patterns, one can be more efficient and avoid unexpected issues because the patterns have been tested and successfully implemented by other developers in the past.

However, a design pattern may work against its will if implemented in a wrong place or way. It can be disastrous and create unnecessary problems. In this study, we search for patterns that can solve extensibility challenges one may meet in the future. Let us imagine that we design an application without having the future needs in mind and also do the programming with lack of structure and best practice design. Such behavior will most likely cause us problems; this is where design patterns may help us.

Software design patterns have been existing for several decades. The term "pattern" comes from architecture design as in building and towns. Christopher Alexander's first pattern related writing named *The Timeless Way of Building* coined the term "pattern" in the late 1970s [1]. Software design pattens we are discussing about has basically adapted Christopher Alexander's ideas.

In the early 1990s, design patterns raised popularity in the software engineering industry after the release of book *Design Patterns: Elements of Reusable Object-Oriented Software* [8], by the Gang of Four (GoF). This book documented the well thought through design patterns in software development. It was used as pattern lexicon by many developers [5]. It contains 23 fundamental software design patterns that are named, explained and reviewed.

Basically, a pattern includes following fundamental elements: The pattern name usually describes the problem and its solution in few words, the problem specifies when to apply the pattern, the solution illustrates the elements that make up the design, the consequences addresses flexibility, extensibility or portability [8, p. 12-13]. Our focus in only on achieving extensibility.

In this study, our focus is on one consequence that the patterns addresses, and that is extensibility. The scope of this study will not cover other consequences, such us flexibility, portability, maintainability, understandability, and testability. However, our understanding is that, to be able to make an application expendable some of these other consequences must be the foundation of the pattern building block. For example, a system needs to have the flexibility to be expandable. An example of extensibility characteristics in an application is that one can easily extend the inventory in a web-shop without needing to inform the other components in the system. An internet API (Application programming interface) should be able to extend the parts that are behind the exposed interface without changing the request mechanism that is provided to calling components. To achieve such goals, design patterns that decouple components and encapsulate the requests can be utilized.

During this chapter, we explain more about the GoF design patterns, its history, categorization, and intent of using it. In the next part, we start our research using these patterns.

5.2 Categorization

The patterns in software development are organized into three categories: creational, structural, and behavioral. This kind of categorization is called purpose, for the reason that reflects what a pattern does. The second classification is called scope, as it specifies whether the pattern applies primarily to classes or objects, as shown in table 5.4 on page 28 [8, p. 21-22]. The table also shows the 23 design patterns that were described by the authors Gang of Four in the book: "Design Patterns: Elements of Reusable Object-Oriented Software". To limit the scope of this study, we only discuss these 23 patterns. In the research part we explore, implement and evaluate patterns categorize by categorize.

Purpose	**Design Pattern**	**Scope**
Creational	Abstract Factory	Object
	Builder	Object
	Factory Method	Class
	Prototype	Object
	Singleton	Object
Structural	Adapter	Class, Object
	Bridge	Object
	Composite	Object
	Decorator	Object
	Facade	Object
	Flyweight	Object
	Proxy	Object
Behavioral	Interpreter	Class
	Template Method	Class
	Chain of Responsibility	Object
	Command	Object
	Iterator	Object
	Mediator	Object
	Memento	Object
	Observer	Object
	State	Object
	Strategy	Object
	Visitor	Object

Table 5.1
GoF design patterns and categorization
GoF book [8].

5.2.1 Creational

Creational category patterns provide ways to deal with object creation mechanisms, which means that, trying to create objects while hiding the creation logic and gives the program much more flexibility to choose the right object to the current situation.

Such behavior decouples the client from the data component by putting encapsulation in between, which means that, the responsibility of creating these objects are removed from the client and system becomes more independent [14].

For example, one can configure an object with default values without specifying details from the client. It gives us also the flexibility to change the values without modifying the client or object itself.

Design Pattern	**Intent**
Abstract Factory	Abstract Factory is similar to the Factory method, but this pattern creates related objects which do not share the same abstract class, which also means that it is not used to create a single object.
Builder	Encapsulate default configuration values of a complex object. It is typically used when default values are many and must not often be changed
Factory Method	The factory method pattern makes it easy to create objects from a set of potential concrete classes that share a common interface, without specifying the exact class of object that must be created
Prototype	Prototype design pattern is used to create a new object by copying an existing object. It is also beneficial to avoid expensive initialization.
Singleton	Ensures that only one instance of a class exist at any time and there is a global pointer to that instance.

Table 5.2
Creational patterns and their intent.[8].
GoF book [8]

5.2.2 Structural

Structural design patterns lack difficulty by identifying a simple way to realize relationships between entities; it is all about class and object composition. Structural class patterns use inheritance to compose interface, while structural object patterns provide a mechanism to compose objects to obtain new functionality.

These patterns help also to encapsulate the composition of object and put a interface in front that can be used by clients. Clients will not interact with the objects directly but though use of the interface, which give the flexibility to dynamically modify the composition [14].

Design Pattern	Intent
Adapter	Adapter pattern is used to integrate two incompatible components when the components itself cannot be modified
Bridge	A bridge pattern is generally used to the exploding class hierarchy problem. The advantage is when adding a new feature, it only requires a single class.
Facade	The facade design pattern provides one straightforward interface to a complex underlying system. It may wrap a single or multiple classes, and the wrapped classes are isolated from the client
Composite	Composite design pattern creates a tree structure of a group of object and treats them as a single object.
Decorator	Decorator design pattern adds new additional functionality to an object without rewrite or alters existing code, at the same time keeps the new functionality separate
Flyweight	Uses sharing to provide large numbers of fine-grained objects efficiently.
Proxy	Proxy design pattern acts as a placeholder for another object to control access to it. Proxy also means "on behalf of." It is usually used to when we want to shield a class from the client.

Table 5.3
Structural patterns and their intent
GoF book [8]

5.2.3 Behavioral

Behavioral patterns make it easy to define communication between classes or objects. It is a mechanism that describes how different classes and objects pass messages to each other to collaborate and to make things happen. These patterns can for example, change part of ab object's operations at runtime without modifying the object or client [14].

Design Pattern	Intent
Interpreter	The interpreter is a mechanism to convert an input to a specific output using expressions.
Template Method	The template method pattern is used when we want the implementation classes to either replace or extend functionality.
Chain of Responsibility	This pattern creates a chain of receiver objects for a request.
Command	Command pattern is used to decouple a request for an action from the object which initially has the action method.
Iterator	Using Iterator design pattern, we can assess different collection of Objects in an uniformal way.
Mediator	Mediator design pattern provides loose coupling by taking responsibility to handle communications between different objects.
Memento	Memento design pattern takes a snapshot of an object's state that can be used to do a roll-back to the previous state.
Observer	Observer pattern gives the responsibility of monitoring changes to an independent object..
State	The state design pattern allows an object to change its behavior when its internal state changes.
Strategy	Strategy pattern allows us to change a method's behavior at the runtime.
Visitor	The visitor pattern is used when we want to change the executing algorithm of an element object at runtime.

Table 5.4
Behavioral patterns and their intent
GoF book [8]

5.3 Summary

In this chapter, we explained design patterns in more detail. Vi looked into the history, categorization, purpose, intent and consequences of using these. In chapter 4, we described the extensibility in software development. In the next part and the following chapters, we discuss these patterns by looking into the motivation and point out those we mean support extensibility from the initial stage of an application life-cycle followed by implementation of some them using a case-study.

Part III

Research

6

Part III: Research

Introduction

In the background part, we explained overall about extensibility and design patterns in software development. In this part, we explore and discuss all of the GoF design patterns to point out those we mean support extensibility from the design phase of application development. Afterwards, we implement some selected patterns using a case-study to demonstrate how they provide support for extensibility in a real-world application. We used iOS platform (operating system) and swift programming language to build the case-study application.

6.1 Outline

We have divided this part into two chapters as the following bullet points explain.

- **Chapter 7: Pointing out Extensibility Design Patterns**
  In this chapter, we walk-through all the GoF design patterns and point out those we consider as extensibility patterns.
- **Chapter 8: Implementation of Extensibility Design Patterns**
  We implemented two designs pattern from each category that we pointed out in chapter 6 to demonstrate how they provide support for extensibility in a real-world application.

7

Part III: Research

Pointing out Extensibility Design Patterns

7.1 Introduction

Our primary goal throughout this chapter is to point out design patterns from the GoF book that support extensibility from the beginning of an application life-cycle, the design phase. The term components here refers mostly to the caller (client), pattern (supporting classes) and data (business logic) components. The caller component is known as *ViewController* in iOS development (case-study platform). Thus, we use this term in the figures and explanations. We believe that decoupling components, between different components, will contribute to support extensibility. Therefore, we discuss and measure the needed amount of work when extending an application with some parts. Due to that, we investigate the pointed out patterns by putting them up against some extensibility characteristics based on our subjective view and data from the extension scenarios table.

We are not interested in patterns that fix a problem when one wants to extend a tightly coupled application design, due to this, we exclude some extensible patterns that are meant to be used for extension of an application that is not decoupled from the beginning. We also exclude patterns where the intent was to increase the performance, minimize the memory allocation, extend a tightly coupled software and other purposes that we meant did not relate to extensibility. However, we discuss every pattern before we make a decision. Due to the simpler understanding of the concept, we try to relate all of the patterns to a graphical representation using class diagram and concrete real-world examples. Patterns that we consider to be used for extensibility are discussed more in detail and tested for possible extension scenarios.

7.2 Extensibility Design Patterns

In the previous part, we explained how Go4 design patterns are categorized and the intent of using it. In the following sections, we discuss and describe the motivation of using the patterns and point out those we consider as extensibility patterns based on their purpose. Patters that we believe can be used for extensibility are discussed, and measured using extensible scenarios. We start with the creational patterns followed by structural and end up the behavioural patterns. We summarize the finding in last sections in this chapter.

Implementation of some of the extensibility design patterns are demonstrated in the next chapter; we pick two patterns from each category that fit into our case-study.

As mentioned in the introduction, we are use UML (Unified Modeling Language) class diagrams to demonstrate the building blocks from each design pattern. The fig. 7.1 shows a sample UML class diagram.

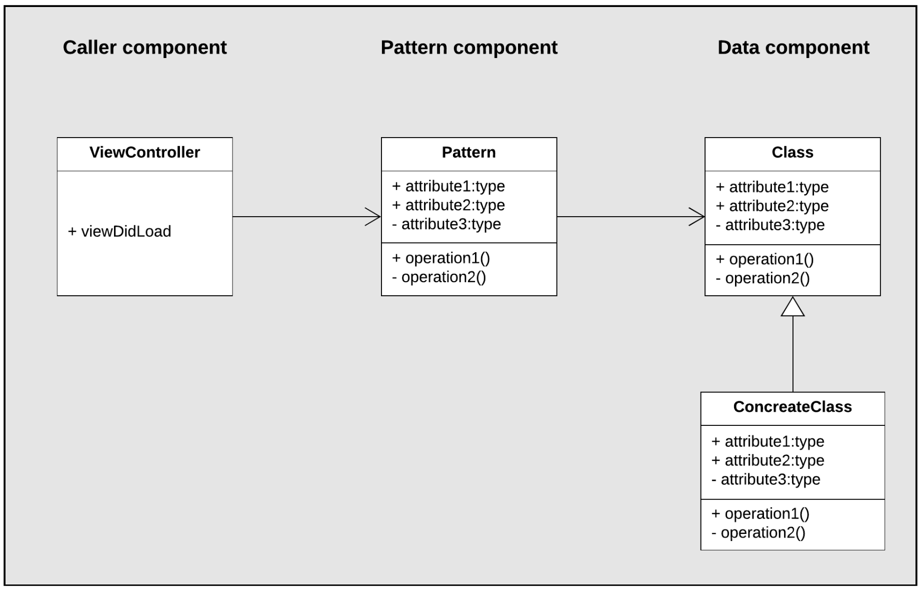


Figure 7.1
Sample UML class diagram used for explanation
Client: ViewController, Pattern: Pattern, Data: Class

ViewController, *Pattern*, *Class* and *ConcreateClass* are normal classes. *ViewController* is the calling component or client that has a method named *viewDidLoad* that is called first when application starts, this is just a simplified representation of the calling component. *Pattern* is the supporting class in which the data and client component uses to connect the building blocks in a design pattern. *Class* and *ConcreateClass* are in the data component.

The arrow between *ViewController* and *Pattern* shows that there is a relationship, which means that, you can navigate from *VivewController* to the *Pattern*. *Class* and *Concreteclass* are also connected together using an arrow in which demonstrate that the *Concreteclass* inherits from *Class*. Both classes have some attributes and operations, the plus symbol indicates a public attribute and minus means private, same principle applies for the operations.

When we discuss the Go4 patterns, we describe the motivation and demonstrate some concrete examples. The motivation text is how we interpret the patterns from the Go4 book [8].

The patterns that are considered to support extensibility is marked with: *Yes*, as shown below.

Extensibility pattern: Yes

For *Yes* pattern, we also go through potential extension scenarios.

Extension scenarios: Described on the facing page.

For every *Yes* pattern, we give our subjective view on it to support the decision, as shown below.

Justification: "Our subject view on this sample pattern shows that..."

7.3 Scenarios tables

The patterns that we mean support extensibility is also illustrated using some possible extension scenarios. Each scenario is put in a table row and counted for any addition of new classes or modification of existing classes in all components (client, pattern, and data), as shown in table 7.1. We have also included the extending part in the count. The *R* indicates required action, *O* stands for optional action and *N* for no work needed. The average number of required and optional of all scenarios is shown in the last two rows, one for required and one for optional action.

Required action, either addition or modification means that something needs to be done in order to make an extension complete. Optional action is when components have a choice to make an action when another part is extending. A good example is a choice a client component has to take advantage of extended functionality by adding or modifying lines of code in the class file. Thus, this action is not necessary to fulfill the extension itself.

Type	Addition of classes			Modification of classes		
	Client	Pattern	Data	Client	Pattern	Data
New class type	N	R	R	O	N	N
New concrete class	N	N	R	O	N	N
Average						
Required	0	0,5	1	0	0	0
Optional	0	0	0	1	0	0

Table 7.1
Sample scenario table
Client: ViewController, Pattern: Pattern, Data: Class
R: Required, O: Optional

7.3.1 Creational

7.3.1.1 Factory method

Motivation:

The factory method pattern makes it easy to create objects from a set of potential concrete classes that share a common interface, without specifying the exact class of the object that must be created. Which means, the pattern selects right subclass to satisfy a calling components request at the runtime. The caller component does not need to know the logic of choosing the proper concrete class, which mean that, the logic can change without affecting the caller component, this is due to encapsulation of object creation. We mean such characteristics form the basis for extensibility. The fig. 7.2 shows how a view controller is connected to the factory and the concrete classes that are hidden behind.

Extensibility pattern: Yes

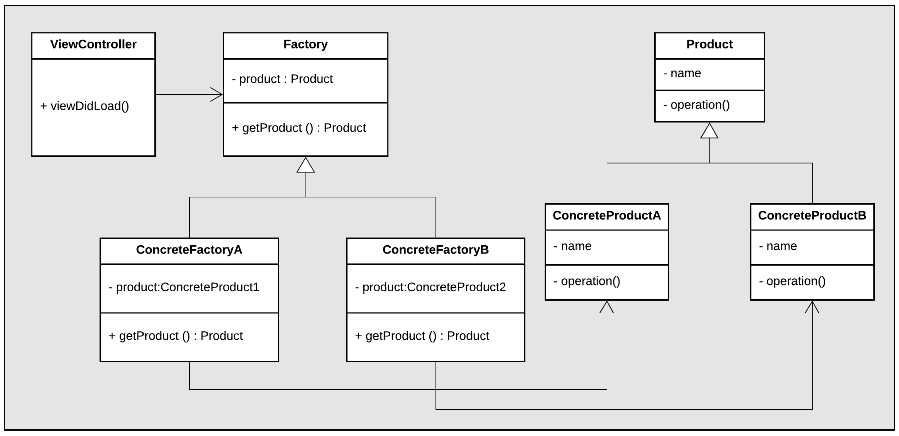


Figure 7.2
Factory Method - Class diagram
Client: ViewController, Pattern: Factory, Data: Product

Concrete example:

Let us assume that we need to create a color object from a set of potential color collection that share a same base class. Instead of creating a color object directly on the client, we delegate the work to a factory that will take responsibility for creating the proper subclass based on the client's requirement. What is happing behind the factory is completely unknown for the client. We can add more color objects that can be chosen by the factory without informing the client.

Extension scenarios:

The table 7.2 illustrates the consequences of introducing a new concrete product to the collection. Adding a new product needs both addition and modification of classes in the different components.

Type	Addition of classes			Modification of classes		
	Client	Pattern	Data	Client	Pattern	Data
New product	N	R	R	O	N	N
Average						
Required	0	1	1	0	0	0
Optional	0	0	0	1	0	0

Table 7.2
Factory method - Scenarios
Client: ViewController, Pattern: Factory, Data: Product
R: Required, O: Optional

We must create a new class in data component, the new product class. If we want to use the new product, we must also create a new concrete factory for the new product. For that reason, we have put required in the addition of data and pattern classes. The same principle applies to the client; creating a new factory may cause changes on the client code if that new product needs to be created from the client. We end up with two required and one optional action.

Justification:

Our subjective view of this pattern is that it decouples data and the client component by putting factory as a middle-ware; it is also encapsulating the object creation logic. However, creating a new factory every-time a new concrete product is introduced seems to be excessive even we avoid one-to-many relationship between factory and the concrete product. We see that this patten's logic can be customized to only use one concrete factory to chose between different concrete products, which will minimize the creation of classes when extending. If we do that, the decoupling will decrease due to the one-to-many relationship between the concrete factory and concrete products.

7.3.1.2 Abstract Factory

Motivation:

Abstract Factory is similar to the Factory method, but this pattern creates related objects which do not share the same abstract class, which also means that it is not used to create a single object. Like the Factory method, it hides the creation logic that is used to build them and the underlying concrete products, as shown in fig. 7.3. One can modify the classes used to create objects without changing the caller components, which means that an extension of products is possible without rewriting much code on the client.

Extensibility pattern: Yes

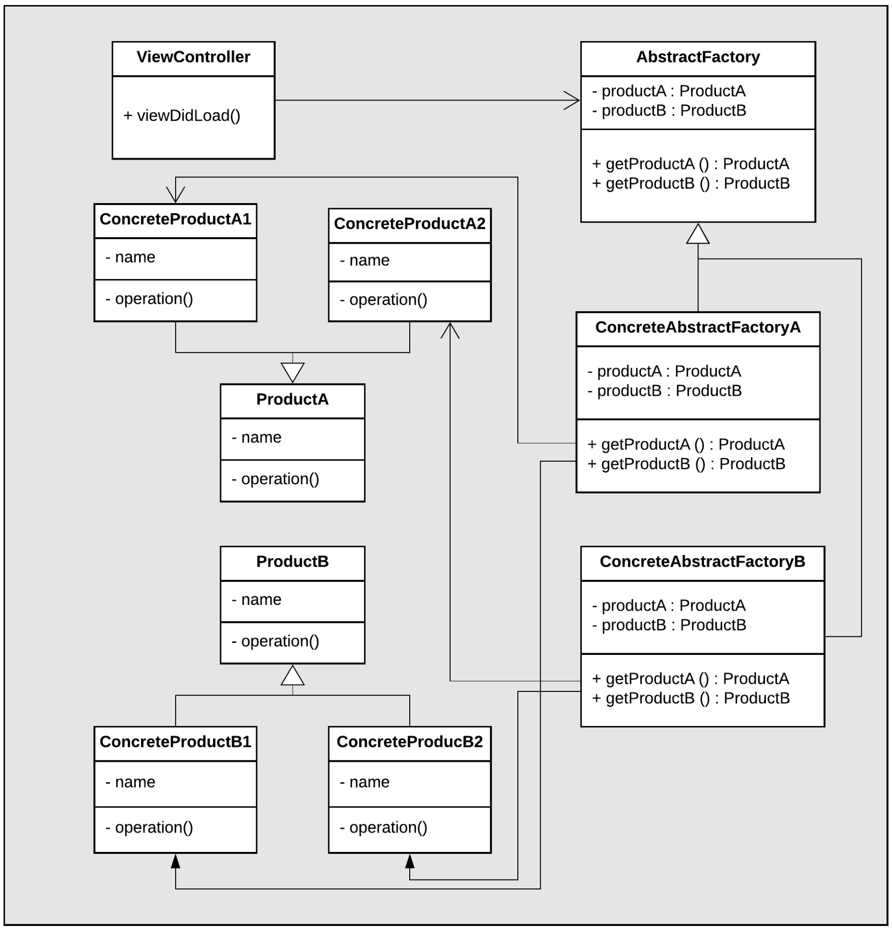


Figure 7.3
Abstract Factory - Class diagram
Client: ViewController, Pattern: AbstractFactory, Data: Product

Concrete example:

Let us assume that we want to buy a car with insurance; we can choose between the different type of cars and insurances based on our needs. The car could be the product A and the insurance the product B, if we relate to the fig. 7.3 on page 41. To create a set of packages containing both a car and insurance, we can use the abstract factory pattern. The concrete abstract factory is responsible for choosing the proper subclass of product A and B to create the package, let us say BMW car with super insurance. The concrete abstract factory may in this case named *BMWSuperInsuranceFactory*.

Extension scenarios:

A new concrete product will force us to create a new subclass inside the data component, either of product A or B; then we need to update the factory to add support for it. This may also cause us to create a new abstract factory for a unique combination that also includes the new product in which will lead us to update the client code as well; this is more alike based on the needs and the design choice and therefore marked as optional in the table 7.2 on page 40.

Type	Addition of classes			Modification of classes		
	Client	Pattern	Data	Client	Pattern	Data
New concrete product	N	N	R	N	R	N
New product type	N	O	R	O	R	N
Average						
Required	0	0	1	0	1	0
Optional	0	0,5	0	0,5	0	0

Table 7.3
Abstract Factory - Scenarios table
Client: ViewController, Pattern: Factory, Data: Product
R: Required, O: Optional

Introducing new product type needs new classes in the data component and most likely modification of the concrete abstract factory classes; like the new concrete product, this extension may also lead us to create a new factory as well as updating the client code to make use of it. A new class in data component and modification in the pattern is required; new factory and client update are optional. Unlike with factory method, the need for a new abstract factory is more or less present in this pattern design. A unique combination of a package of products needs new factory. In such case, we need to create a new class in the pattern component and update the client to make use of it.

Justification:

As we see it, this pattern is just a continuation of the factory method, but products creation is more encapsulated and abstracted inside the factories. Potential extension scenarios are also higher than the factory method, and the sum of average required and optional changes are three, like the Factory method. However, this design introduces more classes in the pattern component, the middleware classes that contribute to decouple the client and the data component. There will be a factory method logic inside the abstract factory to for example chose a concrete class of product A.

What we clearly see here is that decoupling in this situation makes the design more complicated, which in turn leads to increased work when extending. The question is all about where we need less rework, in the client or other components.

7.3.1.3 Builder

Motivation:

Builder pattern encapsulates default configuration values; it is typically used when default values are many and must not often be changed. Handling an object with a lot of initializer arguments is not very productive, neither it decouples callers and data model. Instead, we can use builder pattern that provides an interface for constructing an object using simple steps. Due to this, it is not necessary to change the code on the caller component when the default configuration changes on the object. By changes, we mean adding or removing variables from the object. Such behaviour decouples components in an application from the design phase.

The fig. 7.4 shows how this pattern is constructed using director and builder roles. Director's role is to take the order from the view controller and use the correct builder to build the requested product.

Extensibility pattern: Yes

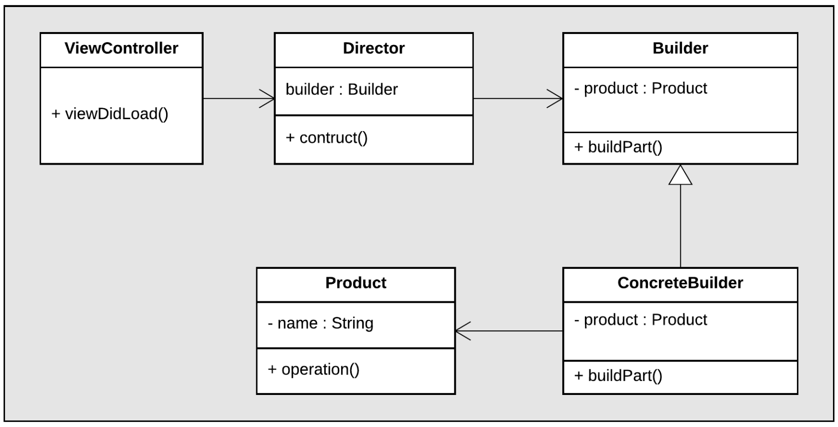


Figure 7.4
Builder - Class diagram
Client: ViewController, Pattern: Director and Builder, Data: Product

Concrete example:

Let us assume that we have a class that represents a laptop, and the variables inside the class are information about the parts inside, such as CPU power, memory capacity and the rest of the building blocks. We might need different specifications based on the demands; an office laptop does not need to be a high-end product, on the other hand, a gamer laptop needs higher specifications. We do not want the client to give all the specifications for the different types. In such case, we can take advantage of the builder design pattern.

From the fig. 7.4 on page 44, the product is the laptop, and the builder is where the office or gamer laptop is built using the required specification. Director will construct the builder and return a laptop object from it.

Extension scenarios:

Typical extension when using this pattern is new or extension in existing constructor of a product. Both scenarios create required modification in the data and pattern component, as shown in table 8.3 on page 86. Note that there is no needed action on the client component because the configuration of an object is wholly encapsulated by the builder.

Type	Addition of classes			Modification of classes		
	Client	Pattern	Data	Client	Pattern	Data
New constructor in product	N	N	N	N	R	R
Extension in product's constructor	N	N	N	N	R	R
Average						
Required	0	0	0	0	1	1
Optional	0	0	0	0	0	0

Table 7.4
Builder - Scenarios table
Client: ViewController, Pattern: Director and Builder, Data: Product
R: Required, O: Optional

Justification:

We interpret this pattern in a way that there is no needed for separate concrete builders for each concrete product. This is due to the fact that default configuration is normally standardized when classes share a common interface. We are aware of that doing this way makes the one-to-many relationship between the concrete builder and the concrete products, which may be seen as tightly coupling. However, an extension is much more comfortable compared to the Factory method and Abstract factory because there is no need for new classes in the pattern component.

7.3.1.4 Prototype

Motivation:

Prototype design pattern is used to create a new object by copying an existing object. It is also beneficial to avoid expensive initialization. Prototypes are useful when object initialization is expensive, and you anticipate few variations on the initialization parameters. In this context, the prototype can avoid expensive creation from scratch, and support cheap cloning of a pre-initialized prototype. The fig. 7.5 illustrates the relationship between the view controller and the pattern and how clone method is used in the prototype classes. We do not find this as an extensible design pattern due to its characteristics. This pattern is normally used to save memory and increase the performance of an application, which means that this is not an extensibility pattern.

Extensibility pattern: No

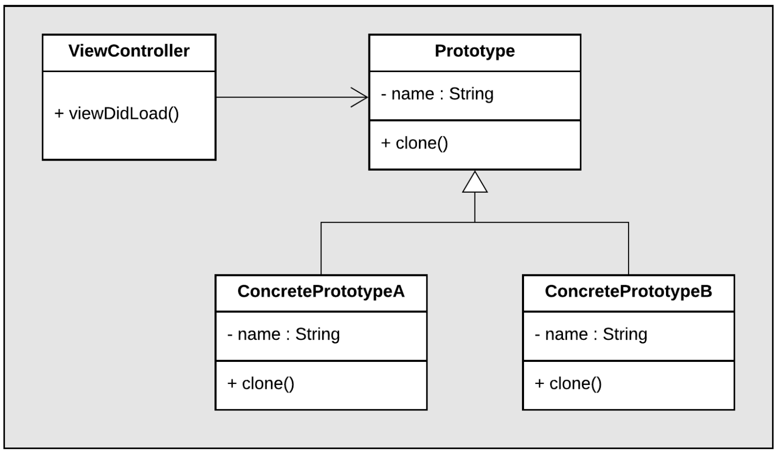


Figure 7.5
Prototype - Class diagram

Concrete example:

Let us assume that we have a car object with several properties and we want to create another car with almost same features, such as engine, and other parts; only differences is the color. Without this pattern, we would most likely create this from scratch by initializing a new object. Instead of doing that we can clone it and only change the color, which saves us for some computing power.

7.3.1.5 Singleton

Motivation:

Motivation for using singleton pattern is to ensure that only one instance of a class exist at any time and there is a global pointer to that instance. The class diagram on fig. 7.6 shows relation between the view controller and the Singleton class. Singleton pattern will always return the same reference of the shared instance to it's callers, in this situation, any view controllers in the application. The *getSharedInstance* method will ensure this behaviour. We mean this pattern is best suitable when one wants to manage a shared instance. Thus, we do not find this as an extensible design pattern. The pattern neither decouples components in the design phase nor helps to extend a tightly coupled system. Due to that we do not look more deeper into this pattern in this study.

Extensibility pattern: No

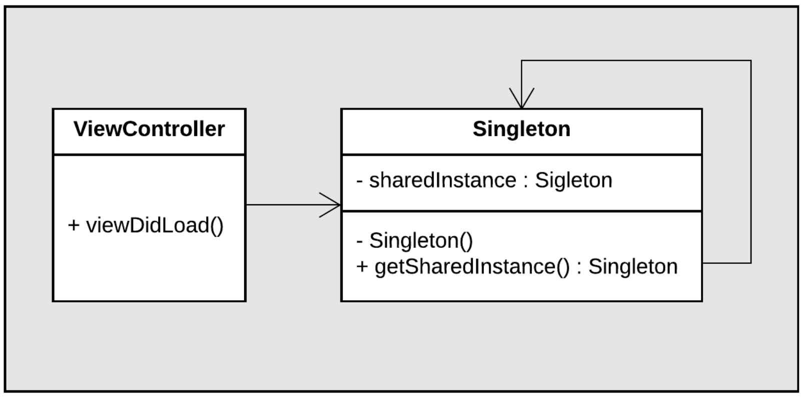

ViewController
+ viewDidLoad()
Singleton
- sharedInstance : Sigleton
- Singleton()
+ getSharedInstance() : Singleton

Figure 7.6
Singleton - Class diagram

Concrete example:

A good example is the main application on a mobile device; you will only have one instance of the main screen at any time. Having multiple instances of an application on such devices is unnecessary and memory consuming, in some cases also lead to user frustration.

7.3.2 Structural

7.3.2.1 Adapter

Motivation:

Adapter pattern is typically used to integrate two incompatible components when the components itself cannot be modified. The adapter extends or wraps incompatible components that must be connected. It is an extensible pattern but not utilized in the design phase of an application. Instead, it extends tightly coupled components. As shown in fig. 7.7, the Adapter class is integrating two incompatible components. For example, a parameter sent from a client to Adaptee must be converted by the Adapter. Our focus is on the patterns that decouple from the beginning, not the patterns that fix a problem that already exists.

Extensibility pattern: No

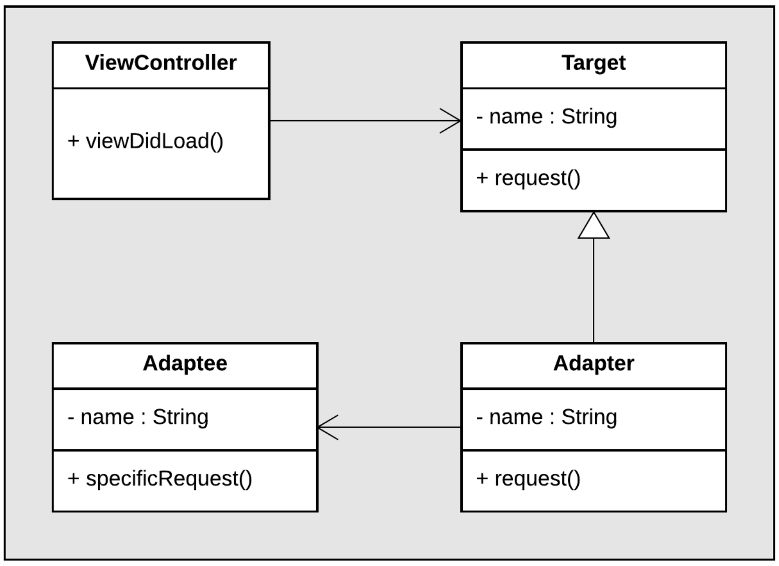


Figure 7.7
Adapter class diagram

Concrete example:

We have a Class A and B that needs to communicate, but their interfaces are entirely different, so they cant understand each other. To make the communication possible, we can introduce an adapter that does the translation work between these classes. Target could be the class A, adaptee the class B and then we have the adapter to do the translation.

7.3.2.2 Bridge

Motivation:

A bridge pattern is generally used to the exploding class hierarchy problem. The advantage is when adding a new feature, it only requires a single class. It also decouples an abstraction from its implementation so that the two can vary independently. The client component can only access the abstract class without being worrying about the implementation part, as shown in fig. 7.8. This shows us that we can add concrete classes to abstraction or implementor without needing to inform each other or the view controller class. Such kind of decoupling makes an extensible solution from the beginning of application life-cycle.

Extensibility pattern: Yes

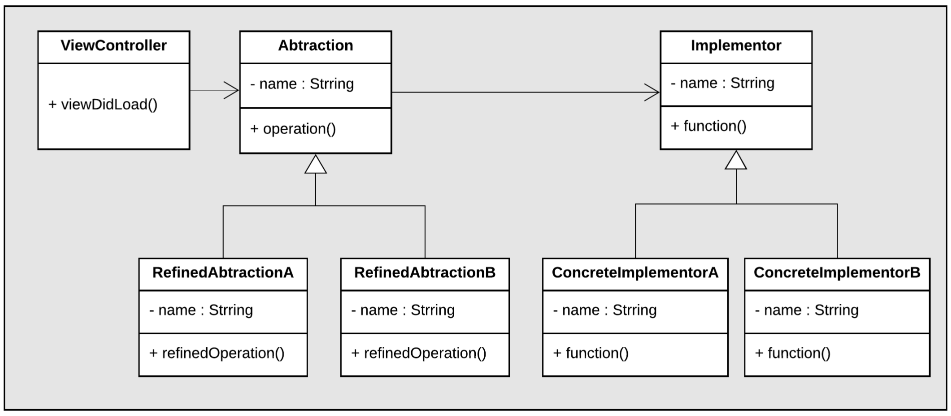


Figure 7.8
Bridge - Class diagram
Client: ViewController, Pattern: N/A, Data: Element

Concrete example:

In case we have concrete classes of shape (circle, triangle, square, etc.) that share a common interface, and we need to fill these shapes with different color, which mean that we have an interface for color as well. A typical exploding class hierarchy will be if we create classes such as CircleRed, CircleBlue, TriangelRed, TriangelBlue, etc. To avoid such hierarchy, we can use the bridge pattern that makes it possible to have a relationship between the shape and the color. The shape is the abstraction layer and color is the implementor, according to the fig. 7.8.

Extension scenarios:

Using this pattern, we have two potential extensible scenarios. We can either add more classes to abstraction layer or implementor part. Both scenarios need required action in the data component, as shown in table 7.5. Notice that there is no pattern component in this design, pattern component is merged into the data component. To take advantage of the new classes, the client needs to know that these exist, but it is optional to use it.

Type	Addition of classes			Modification of classes		
	Client	Pattern	Data	Client	Pattern	Data
New refined abstraction	N	N	R	O	N	N
New concrete implementor	N	N	R	O	N	N
Average						
Required	0	0	1	0	0	0
Optional	0	0	0	1	0	0

Table 7.5
Bridge - Scenarios table
Client: ViewController, Pattern: N/A, Data: Element
R: Required, O: Optional

Justification:

In our point of view, this pattern is a good extensible pattern, classes in both abstraction or implementor part can be extended without needing to do any work on the other components. As descried in the concrete example we can extend either shape or color by adding concrete classes to the different parts independently. Share can use any type of color, and it is usually clients responsibility to connect these together.

7.3.2.3 Composite

Motivation:

Composite design pattern creates a tree structure of a group of object and treats them as a single object. This tree structure is mostly not modified once it has been created. This pattern is only useful when you need to structure objects. That being said, this pattern lets clients treat a group of objects uniformly, which means that we have a loose coupling between client and data components. However, it is not entirely supporting extensibility due to the static tree structure. View controller has a relation to the component and does not need to know anything about the Composites and Leafs. This means that there is a loose coupling between the client and the data model, but not inward the data model, as shown in fig. 7.9. We feel that the purpose is not bright enough to say that it will contribute to support extensibility from the initial stage of an application life-cycle.

Extensibility pattern: No

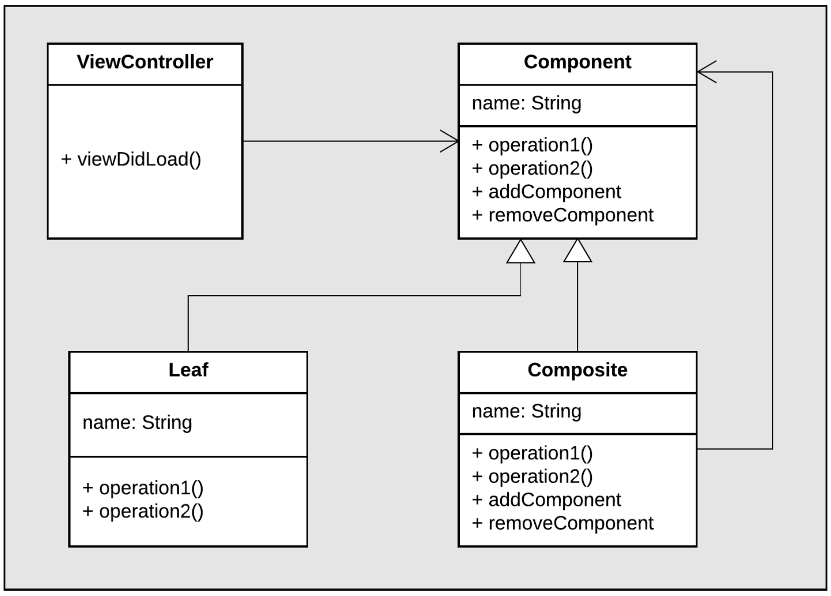


Figure 7.9
Composite class diagram

Concrete example:

A good real-world example that reflects this pattern is the hierarchy structure of a car's parts. The car itself will be the root component and engine as a subcomponent that again has several components or leafs, such as electric parts and valves.

7.3.2.4 Decorator

Motivation:

Decorator design pattern adds new additional functionality to an object without rewrite or alters existing code, at the same time keeps the new functionality separate. It is also used to change the behavior of an object dynamically at runtime. In short, it adds new features to the object by wrapping it. Like Adapter pattern, this is used to wrap an existing object and do not decouples components from the design phase. This pattern is used for extension, but in most of the cases to extend an existing object at the runtime, it is not meant to be used in the design phase. Thus, we mark this as none extensibility pattern.

Extensibility pattern: No

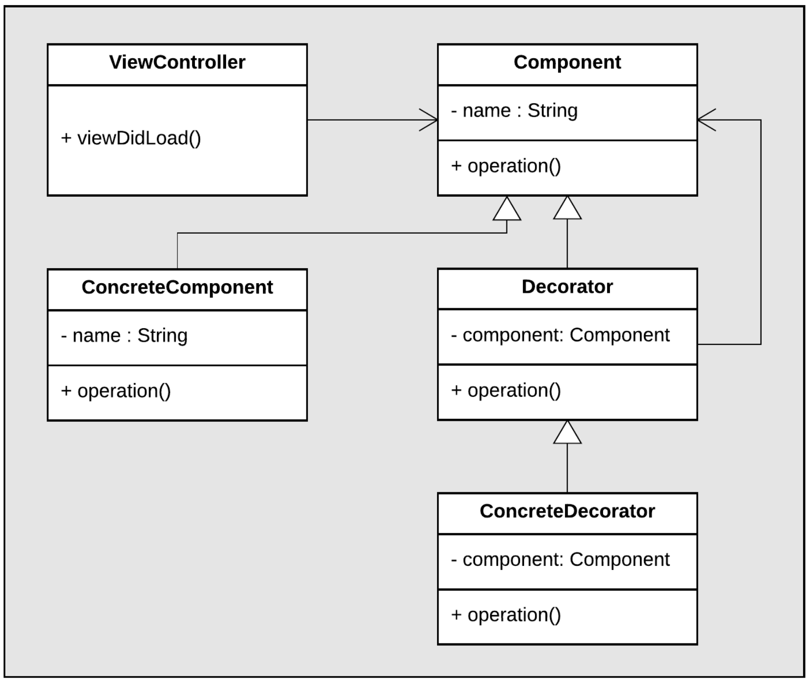


Figure 7.10
Decorator - Class diagram

Concrete example:

This pattern is very straightforward in a manner in which it decorate an existing class without modifying the class. Let say we have Date class that prints date information in central European format. We want this class to include support for American format but are don't able to modify the original class, in such case, we can decorate the class by adding new function that prints date information in American format.

7.3.2.5 Facade

Motivation:

The facade design pattern provides one straightforward interface to a complex underlying system. It may wrap a single or multiple classes, and the wrapped classes are isolated from the client. Any extension or changes to the wrapped part will not affect the Facade interface that is exposed to the client components. This means one can add new classes or functionality inside the wrapped interface without informing the client components.

Extensibility pattern: Yes

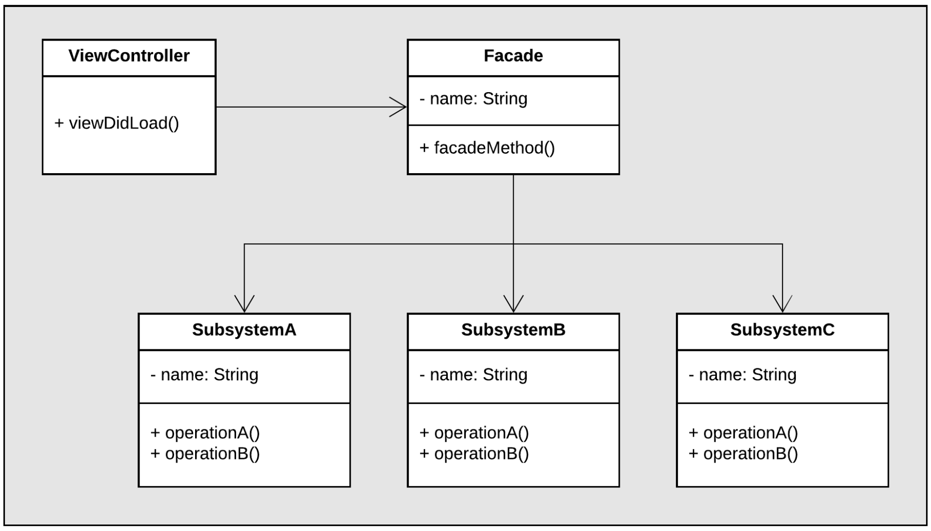


Figure 7.11
Facade - Class diagram
Client: ViewController, Pattern: Facade, Data: Subsystem

Concrete example:

An internet API is an excellent example of Facade design pattern utilized in that system. The API itself is exposed to potential clients, and the back-end systems are wrapped around it. Any addition or changes to the backend components may not generate any work on the client side. The API provider has the flexibility to add subsystems inside wrapped interface without breaking the communication from the clients that already exists.

Extension scenarios:

When utilizing this pattern, we have two potential extensible scenarios. We can either add new subsystem or extend functionality in the existing system without needing to inform the clients. Note that the pattern component in this design is the facade class itself that wraps the underlying systems.

Adding new subsystem requires addition of classes in the data component and sometimes modification in the pattern components in order to include the new subsystem. Extension in subsystem only need modification in the data component.

If the client needs to be updated in some situations, this patterns is probably used in wrong way.

Type	Addition of classes			Modification of classes		
	Client	Pattern	Data	Client	Pattern	Data
New subsystem	N	N	R	N	R	N
Extension in subsystem	N	N	N	N	N	R
Average						
Required	0	0	0,5	0	0,5	0,5
Optional	0	0	0	0	0	0

Table 7.6
Facade - Scenarios table
Client: ViewController, Pattern: Facade, Data: Subsystem
R: Required, O: Optional

Justification:

Our view on this pattern is that it encapsulates the inner work by putting a wrapped interface around a possible complex system. The client needs only to have knowledge about the facade interface and its functions. We mean this pattern is a good extensibility pattern when to comes to separate the client from the data component and can be implemented easily because the purpose is obvious.

7.3.2.6 Flyweight

Motivation:

Flyweight design pattern is used when there is need to create a large number of objects of almost similar nature. A large number of objects consume a large amount of memory, and the flyweight design pattern provides a solution for reducing the load on memory by sharing object. The Flyweight factory that is shown in the fig. 7.12 is responsible for not creating a new object if the same type already exists.

This pattern is used to decrease memory footprint and increase performance, and not decouples components as we expect, that means this pattern is not designed to be used for extensibility.

Extensibility pattern: No

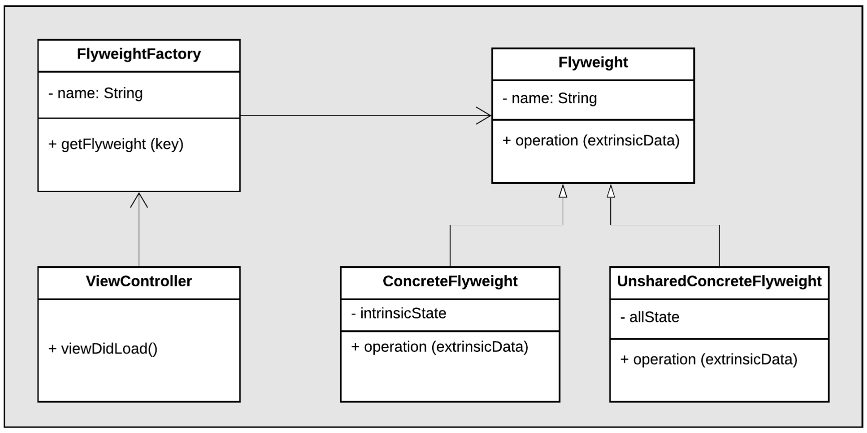


Figure 7.12
Flyweight - Class diagram

Concrete example:

If we have an application where we, for example, need to create 100 circles with different colors, the circle object must be initialized 100 times, which will consume a large amount of memory. Using Flyweight pattern, we can share the circle object and only change the color, because the circle is a static object.

7.3.2.7 Proxy

Motivation:

Proxy design pattern acts as a place-holder for another object to control access to it. Proxy also means "on behalf of." It is usually used when we want to shield a class from the client. In some sense, it sounds like Facade pattern, but the intent of using it is completely different. In this case we are shielding a class, but in Facade, we are wrapping a complex subsystem. The table 7.1 on page 38 shows how an object is hide behind the proxy. This pattern is used to protect or hide components and we do not see it as an extensibility pattern.

Extensibility pattern: No

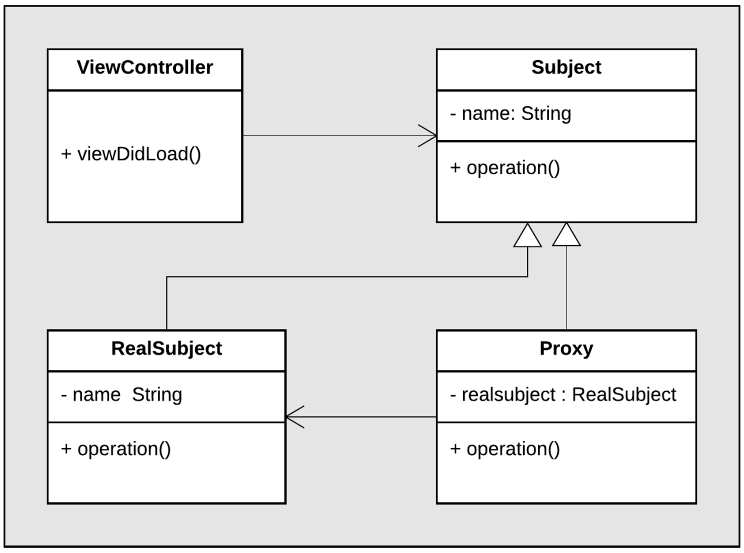


Figure 7.13
Proxy class diagram

Concrete example:

A good example is if we have a back-end system that needs to be shielded for clients due to security purposes, a proxy can be put between the critical system and the client component; through the proxy, we only expose the functions clients need.

7.3.3 Behavioral

7.3.3.1 Interpreter

Motivation:

Interpreter pattern is a design that contains a logic in which can convert a context to an understandable format using an expression, as shown in fig. 7.14. This pattern is typically used for translation purposes. This pattern is famous for its existence in SQL (Structured Query Language) parsing and programming compilers. We do not see this as an extensible design pattern due to its intent rather as a pattern for translation purposes.

Extensibility pattern: No

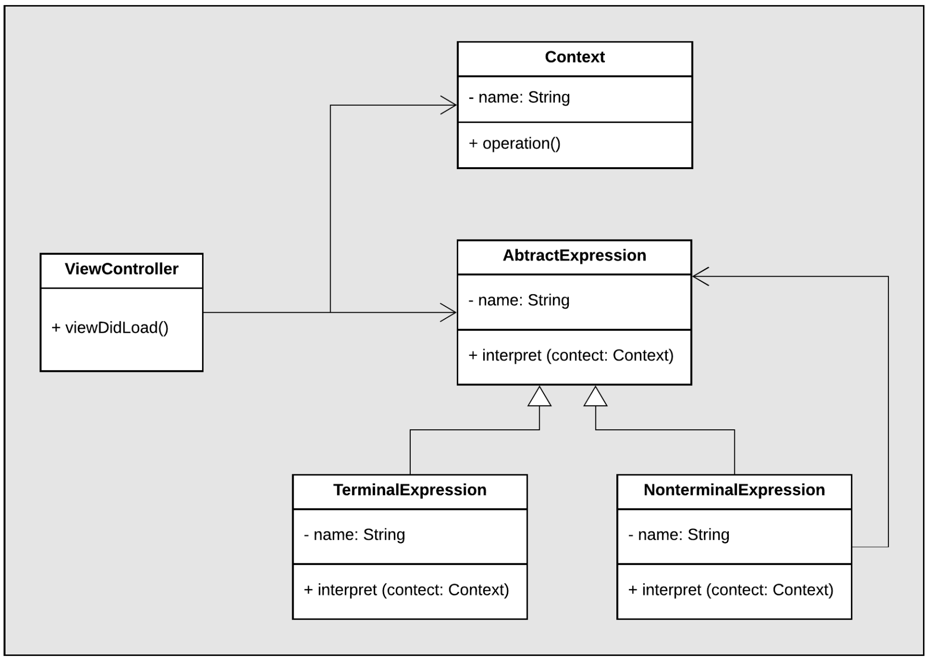


Figure 7.14
Interpreter - Class diagram

Concrete example:

A date string functioning as context using expressions can be interpreted to more understandable format. Other examples of interpreters are: A translator that is allowing people to understand a foreign language and a programming compiler that interprets the source code into bytecode that is understandable by the machine.

7.3.3.2 Template Method

Motivation:

The template method pattern is used when we want the implementation classes to either replace or extend the functionality that is available through an interface. This will permit changing the algorithm in a method without changing the base class or the caller component code. An example is if you are making a framework and you want to allow other developers to change some of the algorithms. Using this patten we encapsulate the methods in a abstract class that can be changed by the concrete classes, as shown in the fig. 7.15. This mechanism decouples in a way that data model can be extended without making changes to the caller component.

Extensibility pattern: Yes

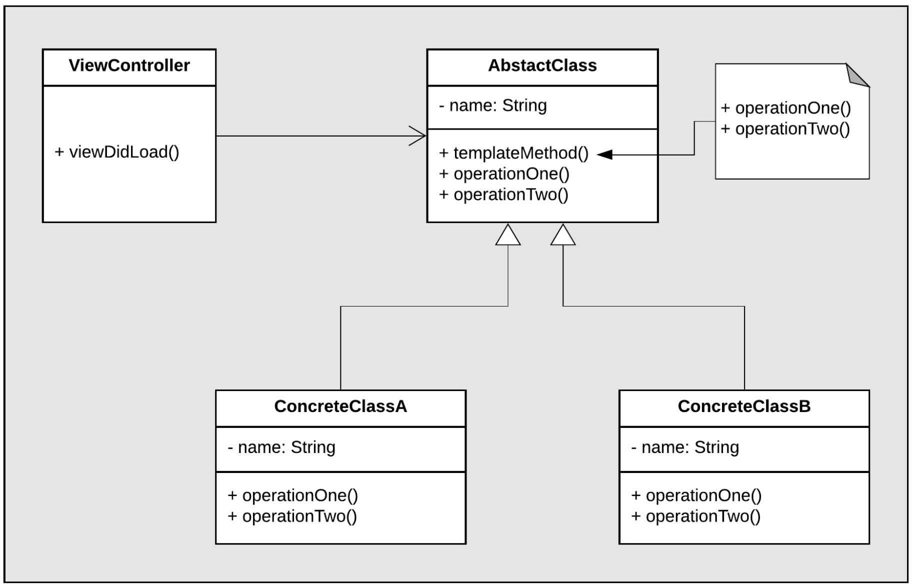


Figure 7.15
Template Method - Class diagram
Client: ViewController, Pattern: N/A, Data: Class

Concrete example:

Let say that the abstract class from the fig. 7.15 is pizza, and concrete class A and B are different types of pizzas that share the common interface. The template method operation's behaviour can be modified by the concrete classes as they override the operations that are bundled in the *templateMethod* operation, which mean that Pizza A can contain different logic inside the other functions. The *templateMethod* function will be the same for the clients.

Extension scenarios:

A new concrete class that overrides some of the operations needs required addition in the data component in the form of creation of new classes. Note that, we do not have any pattern (supporting classes) component in this design. Only required action happens in the data component.

Type	Addition of classes			Modification of classes		
	Client	Pattern	Data	Client	Pattern	Data
New concrete class	N	N	R	O	N	N
Average						
Required	0	0	1	0	0	0
Optional	0	0	0	1	0	0

Table 7.7
Template method - Scenarios table
Client: ViewController, Pattern: N/A, Data: Class
R: Required, O: Optional

Justification:

This pattern does not have any pattern component; the pattern logic is merged into the data component. In our point of view, this is one of the simplest patterns in the GoF list. If one wants to let the concrete classes to change the behavior of several methods without generating lot work in other components than data component, this is the pattern to go for. Functionalities can be easily extended.

7.3.3.3 Chain of Responsibility

Motivation:

The chain of responsibility pattern creates a chain of receiver objects for a request. Each receiver holds a pointer to another receiver. In this pattern, each receiver keeps information about another receiver; the request is passed between objects until it reaches the end of a chain. We do not find this as an extensible design pattern in a manner that decouples components for extensibility; it is normally used to make a chain of requests.

fig. 7.16

Extensibility pattern: No

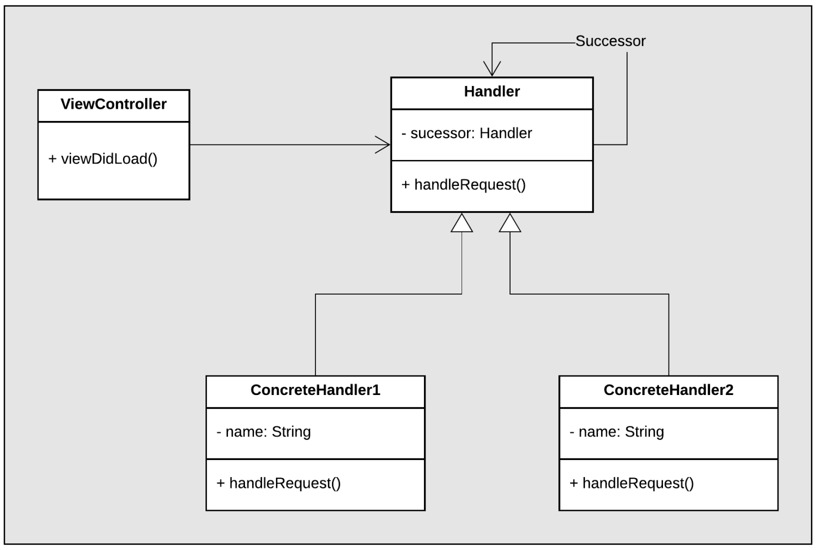


Figure 7.16
Chain of Responsibility - Class diagram

Concrete example:

When we make an ATM withdrawal of 955 dollars, the ATM must put together several bills because single 955 dollar bill does not exist. For such situation, we can use this pattern by having handlers for 100, 50, 10 and one dollars. When withdrawing 955 dollars, request will be fist sent to 100 dollar handler that will collect nine 100 dollar, next request will be sent to 50 dollar handler which will add one 50 dollars to the money pile followed by the 10 dollar handler that will without doing anything forward the request to one dollar handler that will finalize the request by adding five one dollar to the pile.

7.3.3.4 Command

Motivation:

In short, Command pattern is used to decouple a request for an action from the object which initially has the action method. By request, it means the command that is to be executed. With such decoupling, the sender does not know anything about the receiver's interface. UML diagram, as shown in fig. 7.17 demonstrate how classes interact in a Command pattern design. The caller component creates instances of the receiver and command class and sends it to the invoker to perform an action. This sort of mechanism allows us to change or add functions in the receiver without touching much code on the client side. However, some changes to the concrete command classes might be needed.

Extensibility pattern: Yes

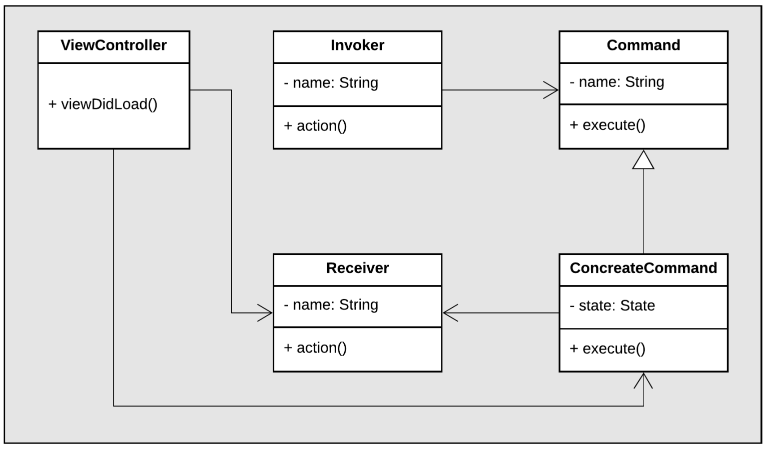


Figure 7.17
Command - Class diagram
Client: ViewController, Pattern: Command and Invoker, Data: Receiver

Concrete example:

Receiver here could be a fan or light, command is typically ON or OFF command made specific for a receiver, and the invoker is the actual switch. This pattern decouples the switch from the receiver interface itself. The switch can operate both light and fan even they have different interface. Swith can only flip up or down and have no knowledge about what is happening inside the commands.

Extension scenarios:

New receiver requires addition of classes in the data and pattern component. We need to create a new receiver and corresponding concrete commands. An optional change is needed in the client component if we want to use the new receiver and the corresponding commands.

New command specific for an existing receiver requires a new class in the pattern component and optional change in the client component. We do not see the point of introducing new invoker is this pattern design.

Type	Addition of classes			Modification of classes		
	Client	Pattern	Data	Client	Pattern	Data
New receiver	N	R	R	O	N	N
New command	N	R	N	O	N	N
Average						
Required	0	1	0,5	0	0	0
Optional	0	0	0	1	0	0

Table 7.8
Command - Scenarios table
Client: ViewController, Pattern: Command and Invoker, Data: Receiver
R: Required, O: Optional

Justification:

Our view on this pattern is that it is one of the most challenging patterns to understand and implement because the addition of new receiver creates some work on the command and afterward must be connected to an invoker. Note that we have two classes in the pattern component, command, and invoker. Builder pattern from the creational category also has two classes, builder, and director, but the extension does not create that much work ones it is implemented correctly.

7.3.3.5 Iterator

Motivation:

Using Iterator design pattern we can assess different collection of objects in an uniformal way. You can for example treat arrays, lists or hash-tables as the same way through the iterator interface, without exposing these internal implementation. Iterator makes it possible for clients components to iterate through any collection of the data. It is also easy to add new data collection to the iterator without making any changes outside of the collections. We encapsulate the traversing tasks and expose an uniformal interface to client, which make it easy to add new collections to the system, thus it supports extensibility.

Extensibility pattern: Yes

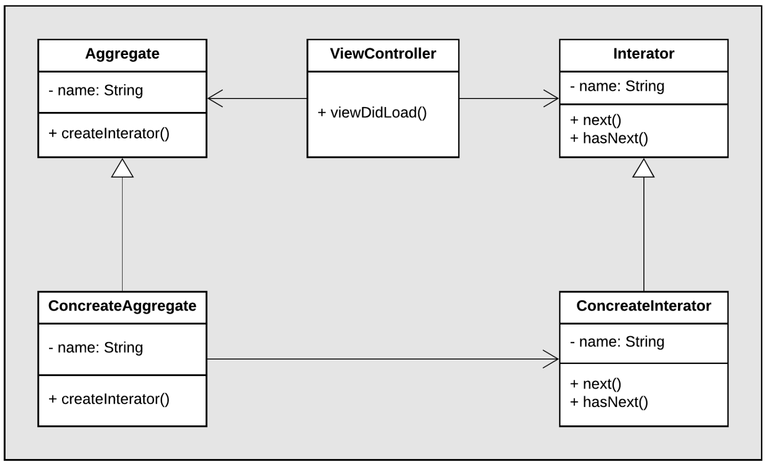


Figure 7.18
Iterator - Class diagram
Client: ViewController, Pattern: Iterator, Data: Aggregate

Concrete example:

Aggregate, in this case, is nothing other than a collection of objects. An iterator contains looping mechanism for the aggregate. Let us assume that we have a collection of strings that we want to loop through. Without using the pattern, we need to implement the looping mechanism at the client, and it needs a lot of code work. Using this pattern, we can avoid the looping at the client and move the responsibility to the Iterator pattern.

Extension scenarios:

A new concrete aggregate or a collection needs new classes in the data and pattern component, and if the client wants to loop through that collection, an optional modification is needed on the client component. New iterator generates a new class in the pattern component, but we do not see the point of introducing new concrete iterator without having a new type of aggregate.

Type	Addition of classes			Modification of classes		
	Client	Pattern	Data	Client	Pattern	Data
New concrete aggregate	N	R	R	O	N	N
Average						
Required	0	1	1	0	0	0
Optional	0	0	0	1	0	0

Table 7.9
Iterator - Scenarios table
Client: ViewController, Pattern: Iterator, Data: Aggregate
R: Required, O: Optional

Justification:

This pattern leaves the looping work to iterator pattern from the client. This is a good practice, but in case we introduce new type aggregate, so must the iterator extend as well, which seems to be quite excessive in a small size of application design. We think that one iterator that can loop through the different collection is more efficient and easily extensible. However, our thinking will make a one-to-many relationship between a single iterator and the concrete aggregates, which can be seen as tightly coupling. Note that, in some patterns, we ague that increased decoupling (one-to-one relationship) may make an extension complex.

7.3.3.6 Mediator

Motivation:

Mediator design pattern provides loose coupling by taking responsibility to handle communications between different objects. It is a communication center for the objects that want to interact with each other. It allows you to not having to manage links between objects. This pattern is used to have control over the communications that need to established between object, instead of having to keep track of all object individually. Using this pattern simplify adding new objects to the communication system without introducing them individually to all involved objects. The extension can be done without a lot of changes to the components of the system.

Extensibility pattern: No

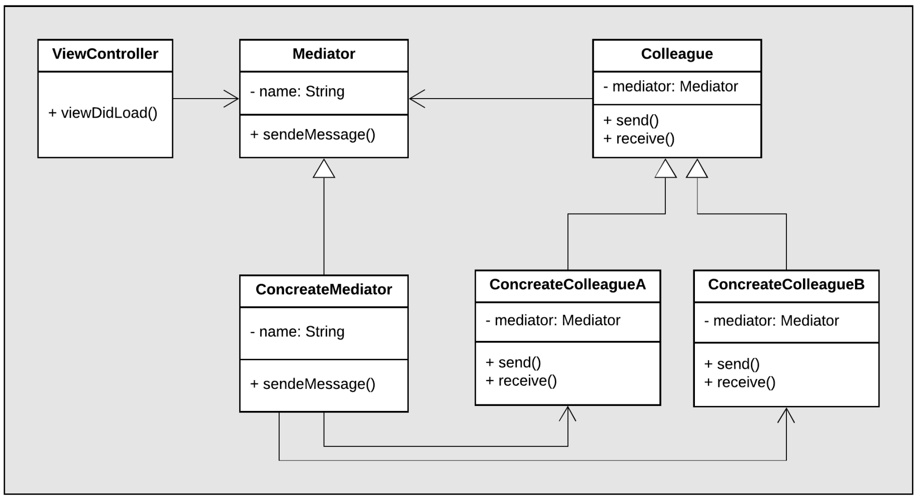


Figure 7.19
Mediator - Class diagram
Client: ViewController, Pattern: Mediator, Data: Colleague

Concrete example:

A good example of a mediator is an air traffic control tower that is responsible for handling communications between flights related to the tower. In such case, the Mediator class is the tower, and the flights are the colleague classes.

Extension scenarios:

A new concrete colleague requires new class in the data component and optional few line of codes in the client. New mediator needs new class in the pattern component and optional modification in the client. It is not that often a new concrete mediator needs to be introduced, for example, a traffic control tower is not changed once it is established.

Type	Addition of classes			Modification of classes		
	Client	Pattern	Data	Client	Pattern	Data
New concrete colleague	N	N	R	O	N	N
Average						
Required	0	0	1	0	0	0
Optional	0	0	0	1	0	0

Table 7.10
Mediator - Scenarios table
Client: ViewController, Pattern: Mediator, Data: Colleague
R: Required, O: Optional

Justification:

We mean this pattern is a good candidate that should be considered during the design phase of an application. Extension does not seems to be that challenging due to that we can have different concrete classes of type colleague and a single type of concrete subject in the design.

7.3.3.7 Memento

Motivation:

Memento design pattern takes a snapshot of an object's state that can be used to do a roll-back to the previous state. It allows a complete roll-back of an object's state without the need to know the specific undo tasks. This pattern is used when one needs to return an object to an earlier state. This pattern provides a recovery mechanism and does not be used for extensibility purpose.

Extensibility pattern: No

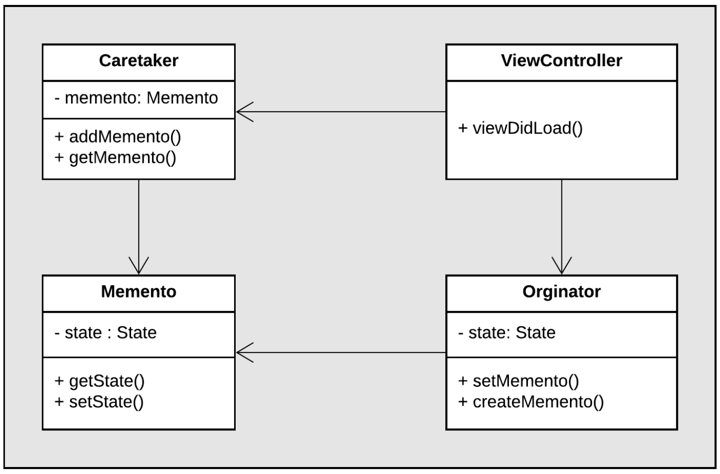


Figure 7.20
Memento - Class diagram

Concrete example:

A simple real-world example is the undo operation we are bale to in a word processing application, such a Microsoft word. File restore and database restore are also can also build using Memento pattern.

7.3.3.8 Observer

Motivation:

Observer pattern gives the responsibility of monitoring changes to an independent object that should update all necessary objects. The receiving object register self to the independent object to be notified. This pattern is used whenever one object needs to be notified of any changes in an another, but the sender does need to know anything about the receiver. This pattern simplifies adding a receiver to the notification center without altering much code due to the decoupling of sender and receiver.

Extensibility pattern: Yes

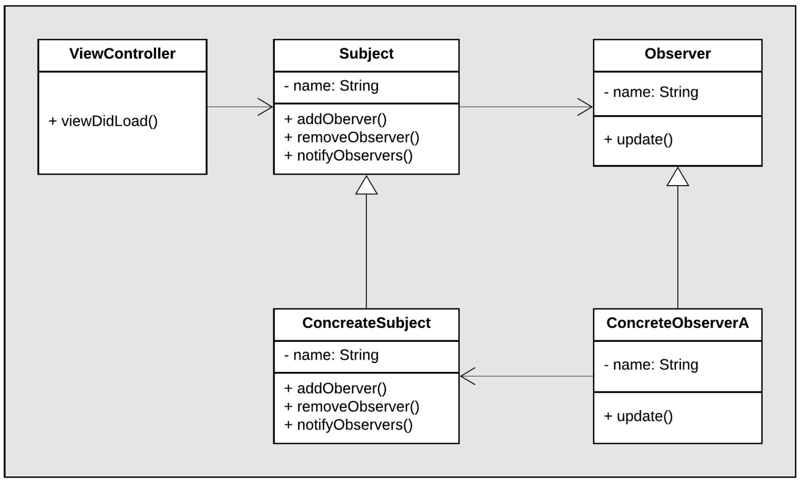


Figure 7.21
Observer - Class diagram
Client: ViewController, Pattern: N/A, Data: Subject and Observer

Concrete example:

Let us assume that we have a background process in an application that regularly checks for any updates that must need to be informed to other components, for example, GUI or database. In such case, the background process is the Subject, and the GUI or database are the observers.

Extension scenarios:

Extending either subject or observer needs an only addition of classes in the data component. It can be explained by that both subject and observer are in the data component. This pattern is used to establish contact between objects without coupling the sender to a reviver. Observer or the reviver will indicate the subject for interest if needed.

Type	Addition of classes			Modification of classes		
	Client	Pattern	Data	Client	Pattern	Data
New concrete subject	N	N	R	O	N	N
New concrete observer	N	N	R	O	N	N
Average						
Required	0	0	1	0	0	0
Optional	0	0	0	1	0	0

Table 7.11
Observer - Scenarios table
Client: ViewController, Pattern: N/A, Data: Subject and Observer
R: Required, O: Optional

Justification:

In this pattern, a concrete subject of type A needs a corresponding type of X observers. If we introduce a new concrete subject of type B, we need to create corresponding observers of type Y. However, both abstract classes are in the same component as mentioned above. We can in this case see that even patten component or supporting classes are absent, we still have to create concrete classes that belongs to different interfaces. This make the extension more complex then for example the template method that also does not have pattern component.

7.3.3.9 State

Motivation:

The state design pattern allows an object to change its behavior when its internal state changes. In this pattern, the object's internal state is the controller for its behavior. This pattern is about remembering the start of an object, and we do not see it as an extensible pattern that needs to be taken into consideration at the initial stage of application development.

Extensibility pattern: No

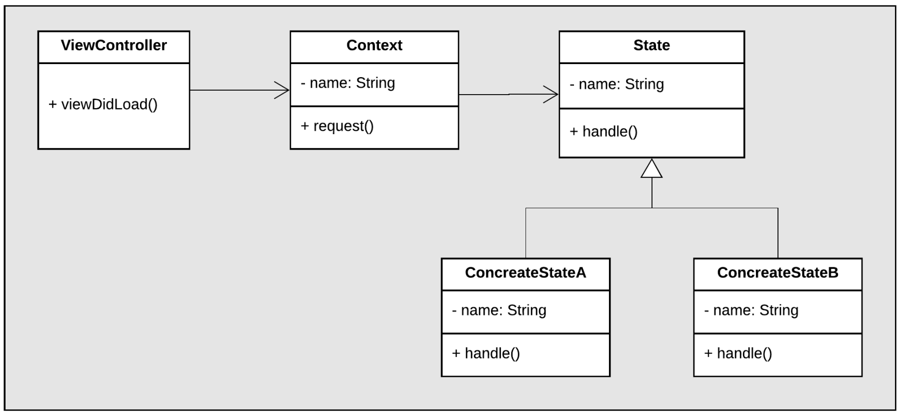


Figure 7.22
State pattern class diagram

Concrete example:

A real-world example, in this case, would be an ATM machine with is an internal state. It will typically have two states, *No card inserted* and *Card inserted*. Based om this state, it will change the ATM objects behavior..

7.3.3.10 Strategy

Motivation:

Strategy pattern allows us to change the behavior of a method at the runtime. This pattern is used whenever one needs classes that must be extended without being modified; this is, for example, useful when there is a need for exposing frameworks for clients and give them the possibility to extend functionalities without modifying or sub-classing the original object.

Extensibility pattern: Yes

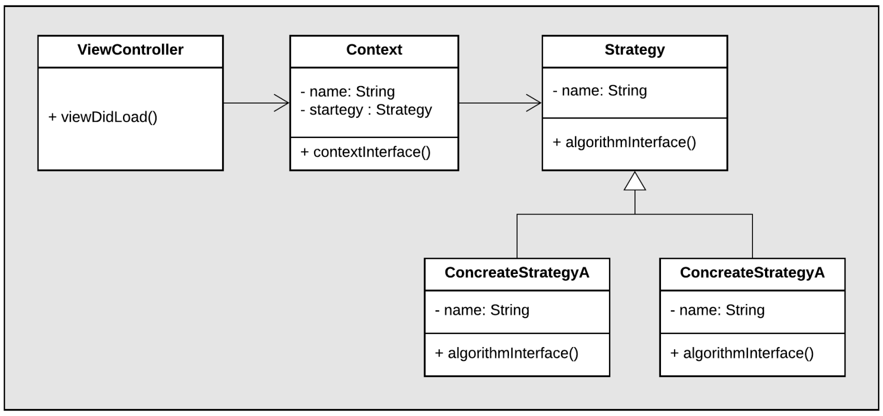


Figure 7.23
Strategy - Class diagram
Client: ViewController, Pattern: Strategy, Data: Context

Concrete example:

Let us assume that a person needs to travel from location A to B and can choose between different transport. Travelling itself is the context and the different transport methods is the strategy.

Extension scenarios:

A new concrete strategy requires only addition of classes in the pattern component. Even we have both the data and pattern component in this design; an extension only creates the addition of classes in the one component.

Type	Addition of classes			Modification of classes		
	Client	Pattern	Data	Client	Pattern	Data
New concrete strategy	N	R	N	O	N	N
Average						
Required	0	1	0	0	0	0
Optional	0	0	0	1	0	0

Table 7.12
Strategy - Scenarios table
Client: ViewController, Pattern: Strategy, Data: Context
R: Required, O: Optional

Justification:

This pattern is a good extensible pattern to use in the design phase due to that it contributes to extend the data component's behavior without recompiling the associated classes.

7.3.3.11 Visitor

Motivation:

The visitor pattern is used when we want to change the executing algorithm of an element object, which means that, we are decoupling the operation from the element object itself. In other words, the behavior of element algorithm can be manipulated using the visitor. The fig. 7.24 illustrate the relationship between the element and the visitor object..

Extensibility pattern: Yes

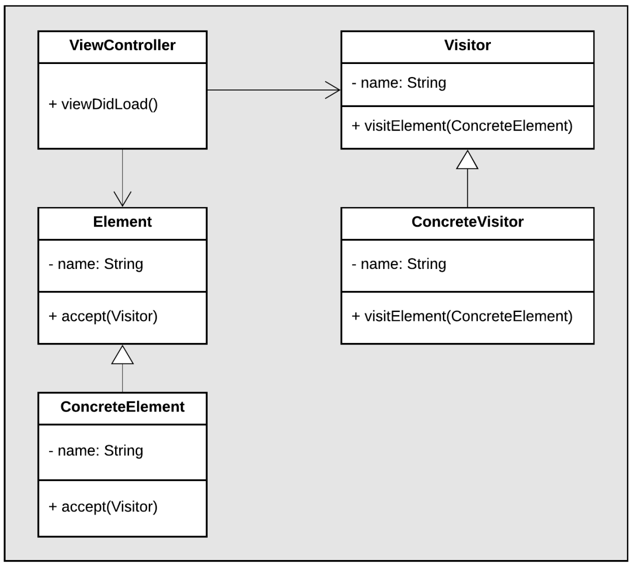


Figure 7.24
Visitor - Class diagram
Client: ViewController, Pattern: Visitor, Data: Element

Concrete example:

A real-world example will be that if we compare a postman with the visitor and the mailboxes as an element. It is actually up to the postman to decide where to visit and put what kind mail in a mailbox. The mailboxes (element) content will vary depending on postman's (visitor's) behavior.

Extension scenarios:

The table 7.13 on the next page shows the amount of work that needs to be done when extending entities while using this pattern. Extending by new concrete element needs an only addition of class in the data component. However, if the client needs to visit an element, it needs to add the element to the visitor's collection, but this is optional. Extend functionality in visitor requires modification only in the pattern. A new concrete visitor can be added without affecting any other components, but client component needs to know about it if there is an interest in using the new visitor.

Type	Addition of classes			Modification of classes		
	Client	Pattern	Data	Client	Pattern	Data
New concrete element	N	N	R	O	N	N
New concrete visitor	N	R	N	O	N	N
Average						
Required	0	0,5	0,5	0	0	0
Optional	0	0	0	0,5	0	0

Table 7.13
Visitor - Scenarios table
Client: ViewController, Pattern: Visitor, Data: Element
R: Required, O: Optional

Justification:

This pattern can change the data components behavior without needing to recompile anything of the data class files. A new visitor can be added any time to the system, and it can visit many elements objects in the data component. Thus, we consider this as a good extensibility pattern.

7.4 Summary

In this chapter, we selected patterns that have extensibility characteristics and can be used in the design phase of an application life-cycle. The primary goal was to point out those patterns that support extensibility by loosely coupling components from the beginning of an application life-cycle, the design phase. We selected pattern based on their purpose. In some patterns, we could see that increased decoupling may impact the extension and make it bit more complicated in a manner that demands more work.

The selected patterns are:

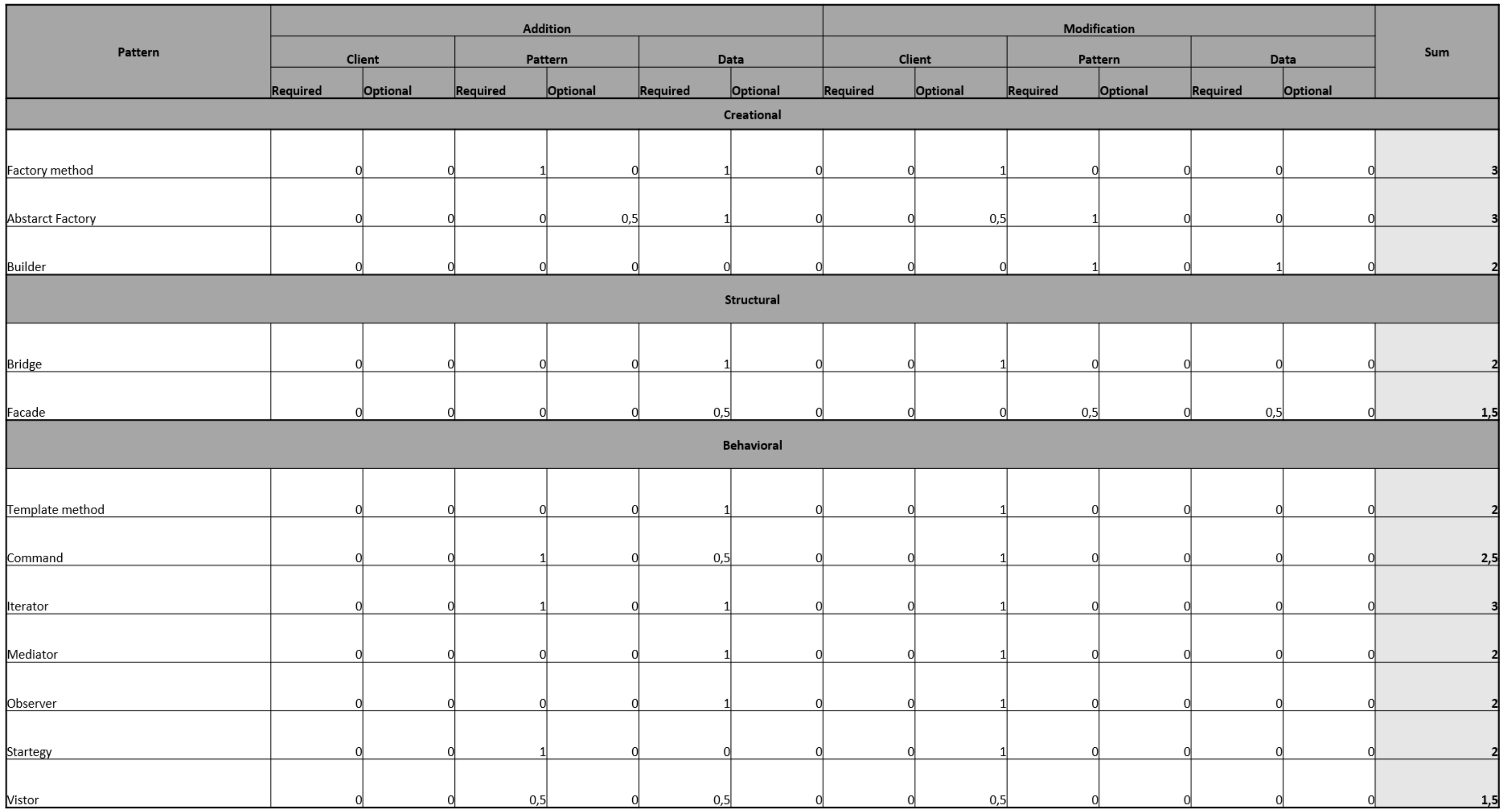
- **Creational:**
  - Factory Method, Abstract Factory and Builder
- **Structural:**
  - Bridge and Facade
- **Behavioral:**
  - Template Method,Command, Iterator, Mediator, Observer, Strategy and Visitor

We excluded patterns where the intent was to increase the performance, minimize the memory allocation, extend a tightly coupled software and other purposes that we meant do not relate to extensibility.

The excluded pattern are:

- **Creational:**
  - Prototype and Singleton
- **Structural:**
  - Adapter, Composite, Decorator, Flyweight and Proxy
- **Behavioral:**
  - Interpreter, Chain of Responsibility, Memento and State

On the facing pages, we have attached results from the extension scenarios measurement, both a table overview and a corresponding chart that we will discuss more in detail in the evaluation part. In the next chapter, we implement two of the patterns form each category to demonstrate how they provide support for extensibility in a real-world application. We build a simple application from scratch using some selected patterns that we mean suites for the particular application.



Pattern	Addition						Modification						Sum
	Client		Pattern		Data		Client		Pattern		Data		
	Required	Optional	Required	Optional	Required	Optional	Required	Optional	Required	Optional	Required	Optional	
Creational													
Factory method	0	0	1	0	1	0	0	1	0	0	0	0	**3**
Abstarct Factory	0	0	0	0,5	1	0	0	0,5	1	0	0	0	**3**
Builder	0	0	0	0	0	0	0	0	1	0	1	0	**2**
Structural													
Bridge	0	0	0	0	1	0	0	1	0	0	0	0	**2**
Facade	0	0	0	0	0,5	0	0	0	0,5	0	0,5	0	**1,5**
Behavioral													
Template method	0	0	0	0	1	0	0	1	0	0	0	0	**2**
Command	0	0	1	0	0,5	0	0	1	0	0	0	0	**2,5**
Iterator	0	0	1	0	1	0	0	1	0	0	0	0	**3**
Mediator	0	0	0	0	1	0	0	1	0	0	0	0	**2**
Observer	0	0	0	0	1	0	0	1	0	0	0	0	**2**
Startegy	0	0	1	0	0	0	0	1	0	0	0	0	**2**
Vistor	0	0	0,5	0	0,5	0	0	0,5	0	0	0	0	**1,5**

Figure 7.25
Extensibility design patterns - Average actions overview

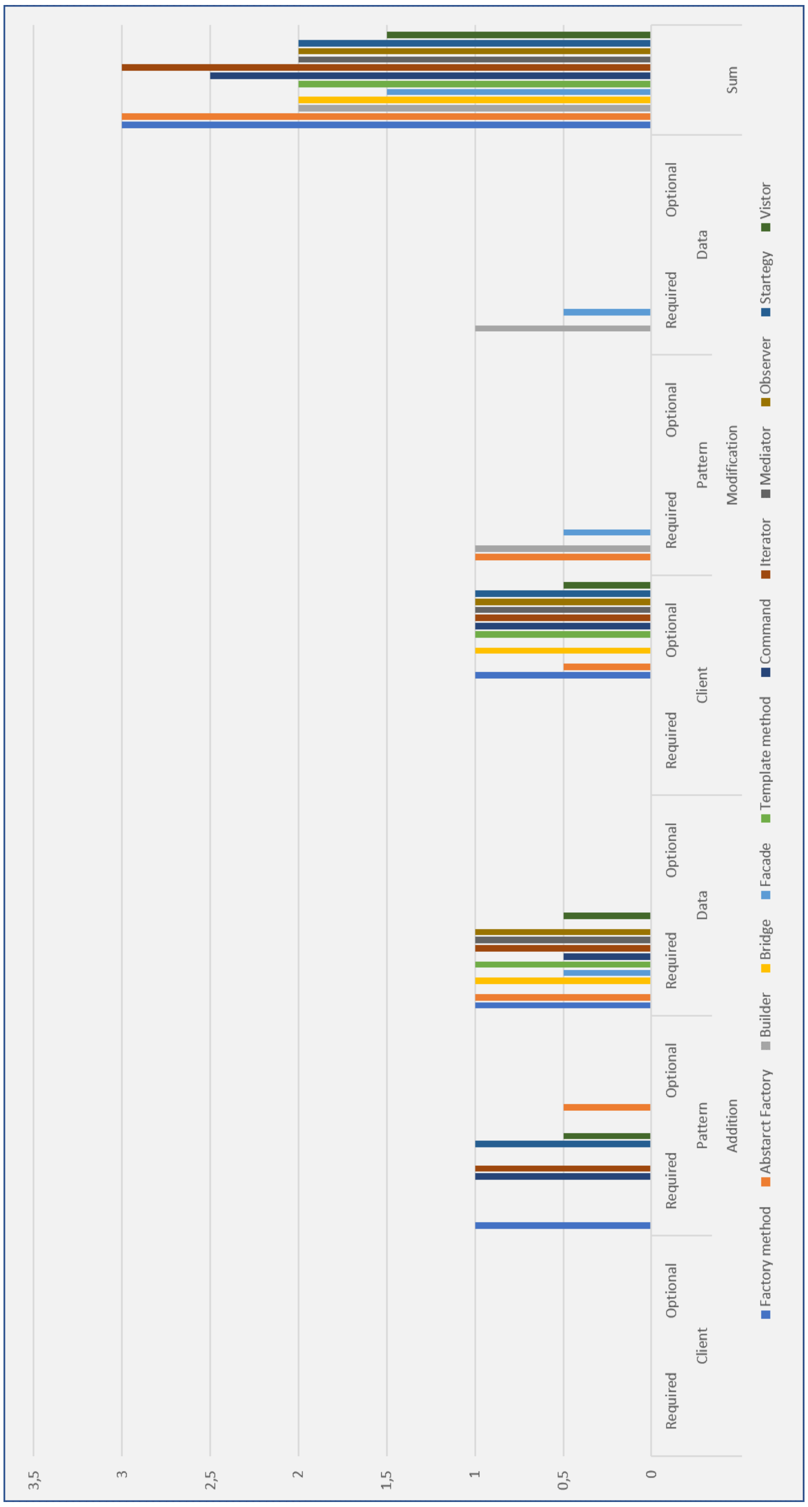


Figure 7.26
Extensibility design patterns - Average actions chart

8

Part III: Research

Implementation of Extensibility Design Patterns

8.1 Introduction

In this chapter, we implement six of the pointed out design patterns from the previous chapter. Implementation is done using a case-study application named eFuel that is briefly described in the next section. Purpose of this case-study is to examine the extensibility pattern from a practical perspective.

Our primary goal throughout this case-study is to examine the theoretical research from the previous chapter and demonstrate any benefits or drawbacks by implementing some the claimed extensibility patterns. This is also due to verify or reject our expectations after examining the patterns theoretically. For each implemented pattern, we will describe the motivation of choosing it as well as some code samples and possible extensible scenarios. We also use class diagrams to show the involved components, but unlike chapter 7, we use classes from the application itself.

Based on the finding from the previous chapter and the needs for our project, we picked two design patterns from each pattern category that we found to fit into our case study application. We believe these design patterns can solve some of the extensibility challenges we may meet in the future. The application we create is only to discuss the design patterns, which means that, the focus is not on the quality of the application, but the patterns that are demonstrated using the application.

The selected patterns for implementation are:

- **Creational**

  Factory method and Builder

- **Structural**

  Bridge and Facade

- **Behavioral**

  Template Method and Iterator

8.2 eFuel application

In this section, we describe briefly the application we built using the selected design patterns. We decided to call the application by name eFuel. In this context, eFuel is the abbreviation for electric fuel. It is a simple application that shows nearby charging stations for electric vehicles within a radius of 10 km. We are using a public internet API provided by Norsk elbilforening (Norwegian Electric Vehicle Association) to fetch the data from the internet. The majority of nowadays mobile applications receive and consume data via APIs, for that reason, we chose to interact with an API in this case-study. This is also due to that API tends to change or provide additional functionality over time, in such case, designing application with extensibility in mind is a huge benefit.

High-level design of application is shown in fig. 8.1 on the facing page. The rectangle on the left side is the start view of the application and shows a table that contains stations found nearby. The table is filled with data from the internet API. Right side view will appear on the screen when a table row is tapped, and contains more details about the station that was tapped by the user.

From the detail pane, a user can share the station information to different media, for example to social media such as Facebook or Twitter. A station itself includes a variety of information, but it also holds references to car types that are supported by the station.

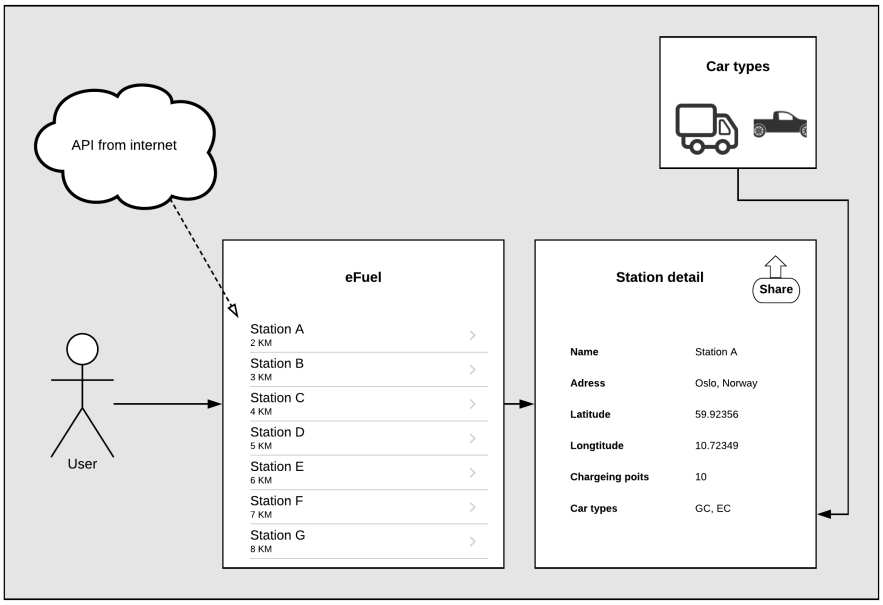


Figure 8.1
eFuel application - High level design

8.3 Building eFuel

In this section, we demonstrate how the selected patterns are implemented in the application. We describe the intent and motivation of using these patterns specific to this application. Addition to the textual explanation, we use class diagrams, and code examples to present the work. Our primary goal throughout this section to show how these design patterns are implemented using the case-study. The table 8.1 shows the areas where the patters are used.

Design Pattern	**Area of usage**
Factory method	This pattern is used to create the station objects.
Builder	Used to configure the station object.
Bridge	Relation between car types and stations is built using this pattern.
Facade	Used to wrap the API data download tasks.
Template Method	Sharing station information to other media is designed using this pattern.
Iterator	Used to loop through the data that is provided by API.

Table 8.1
Usage area of the patterns in the case-study

We mostly illustrate how the patterns are loosely coupling the client, pattern and the data model components; however, in the evaluation part, we will also discuss the decoupling inward the data model itself.

Evaluation of the implemented patterns is described in the next part.

8.3.1 Creational

8.3.1.1 Factory method

Motivation:

To store information about the electric charging stations that we retrieve from the internet API, we need to create one object for each station; this is due to the object-oriented encapsulation mechanism that is very common in software development. Thus, we need a class for charging station with several variables, such as the name of the station, address, position and amount of charging points at the station. We decided to use factory method pattern to create the objects; this is because we must be able to extend the scope with more type of station in the future. We use a factory to create a concrete station objects that could be either an electric station, gas station or some other type of stations.

As shown in the fig. 8.2, we have two abstract classes, *StationFactory.swift* and *Station.swift*. Both classes have its own concrete classes which in turn hold a reference to each other. Note that each type of station has a separate factory.

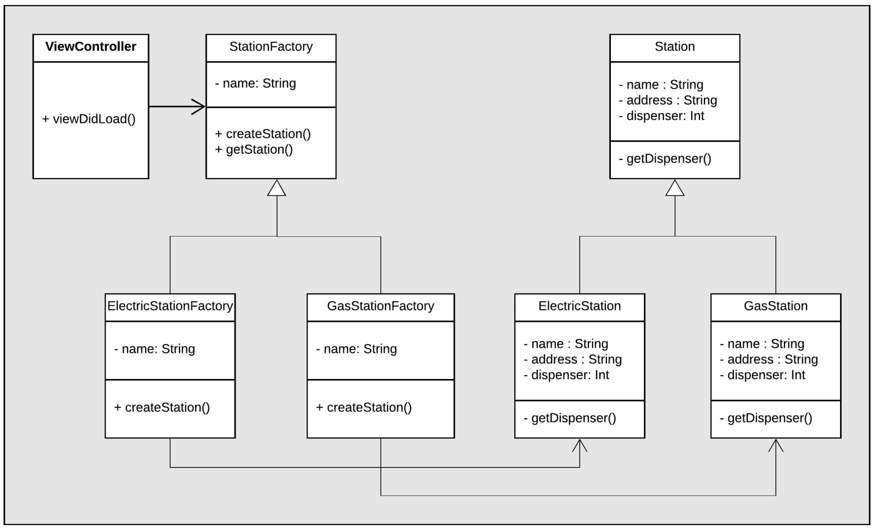


Figure 8.2
Case-study: Factory method - Class diagram
Client: ViewController, Pattern: StationFactory, Data: Station

Code samples:

The code below shows that we need to create classes in the data and pattern component when introducing a new type of station. This is loose coupling because we have a separate fabric for each type of station.

```
    /*ViewConroller class*/

    var efactory, gfactory, newFactory : StationFactory
    var eStation,gStation, newStation : Station

    //Requesting a concrete station object of type electric
    //station
    eFactory = ElectricStationFactory()
    eStation = factory.getStation()

    //Requesting a concrete station object of type gas station
    gFactory = GasStationFactory()
    gStation = factory.getStation()

    //Requesting a concrete station object of type new station
    newFactory = NewStationFactory()
    newStation = factory.getStation()

```

Listing 8.1
Case-study: Factory method - Client code

Extension scenarios:

Extension scenario table below verify our theoretical explanation from chapter 7.

Type	Addition of classes			Modification of classes		
	Client	Pattern	Data	Client	Pattern	Data
New concrete station	N	R	R	O	N	N
Average						
Required	0	1	1	0	0	0
Optional	0	0	0	1	0	0

Table 8.2
Case-study: Factory method - Scenarios table
Client: ViewController, Pattern: StationFactory, Data: Station
R: Required, O: Optional

Lesson learned:

The implementation of Factory method shows that we can change the logic inside the concrete factory without informing the view controller. However, adding a new type of station leads to a new concrete factory in the pattern component as well. After the implementation of this pattern in this case-study, we have the feeling that using a single factory to chose between different concrete stations is much more efficient. We see that customizing the pattern based on the needs is beneficial compared to implementing it as-is.

8.3.1.2 Builder

Motivation:

To create a station object, we need values such as name, street, house-number, zip-code, city, charging points, position latitude and position longitude. We cannot always be sure that we have all the values available at the creation time, and in fact that we accept an object to be created without having all the values, using Builder pattern might be beneficial.

In the previous section we used factory pattern to create an empty concrete station object in which had no values in the variables. Using Builder pattern, we can fill the object with necessary data and let the builder set default values to other variables that we do not care about. To implement the builder pattern, we need to create more layers between the view controller and the station object, as shown in fig. 8.3.

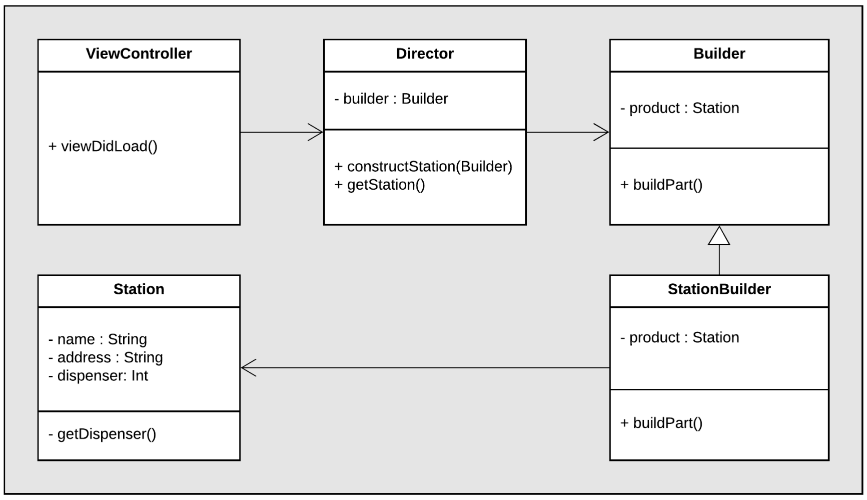


Figure 8.3
Case-study: Builder method - Class diagram
Client: ViewController, Pattern: Director and Builder, Data: Station

Instead of initializing the station object directly on the View-controller, we go through a director that uses the builder object to build a station object, as shown in listing 8.3. In this case, we are combining the factory method and builder pattern in a manner that makes it sense for our case study, but it can also be done in other ways. That being said, evaluation of combining patterns is out of scope for this study.

Code samples:

```
    /*ViewConroller class*/

    var gfactory : StationFactory
    var gStation: Station
    var sBuilder : StationBuilder

    //Requesting a "null" concrete station object
    //of type gas station
    gfactory = GasStationFactory()
    gStation = gfactory.getStation()

    //Creating the builder
    sBuilder = StationBuilder(gStation)

    //Preparing the builder
    var stationDirector = StationDirector(sBuilder)
    stationDirector.constructStation()

    //Getting the station object from the director
    var gstation = stationDirector.getStation()

```

Listing 8.2
Case-study: Builder - Client code

Extension scenarios:

Type	Addition of classes			Modification of classes		
	Client	Pattern	Data	Client	Pattern	Data
New constructor in station	N	N	N	N	R	R
Extension in station constructor	N	N	N	N	R	R
Average						
Required	0	0	0	0	1	1
Optional	0	0	0	0	0	0

Table 8.3
Case-study: Builder - Scenarios table
Client: ViewController, Pattern: Director and Builder, Data: Station
R: Required, O: Optional

Lesson learned:

After implementing this pattern, we have learned that combining patterns could be challenging, especially if the intent is almost the same. We use the factory to pick a concrete station and then use the builder to set default configurations. We could also have encapsulated the builder pattern behind the factory, which would have decoupled the client completely from the Builder pattern. Combining patterns is not in the scope of this study. Thus, we do not look deeper into it. However, we can see the benefit of hiding as much as possible from the client.

We could also have used separate concrete builders for particular stations. In that case, it is all about the design choice or how a developer interprets the pattern. Of course, it would have complicated this pattern somewhat more.

8.3.2 Structural

8.3.2.1 Bridge

Motivation:

The charging stations we retrieve from the internet API may not support fuelling of all type of cars; some may for example not support Tesla which needs a proprietary charger. An only electric station will never be able to fuel gasoline cars. Thus, we need a relationship between stations and the supported cars. To solve this problem, we decided to use bridge pattern that is used to avoid the exploding class hierarchy problem. We would like to avoid creating a for example classes such a *TeslaElecticStation.swift* or *NissanElecticStation*, instead vi create a relation between station and the supported cars, as shown in fig. 8.4.

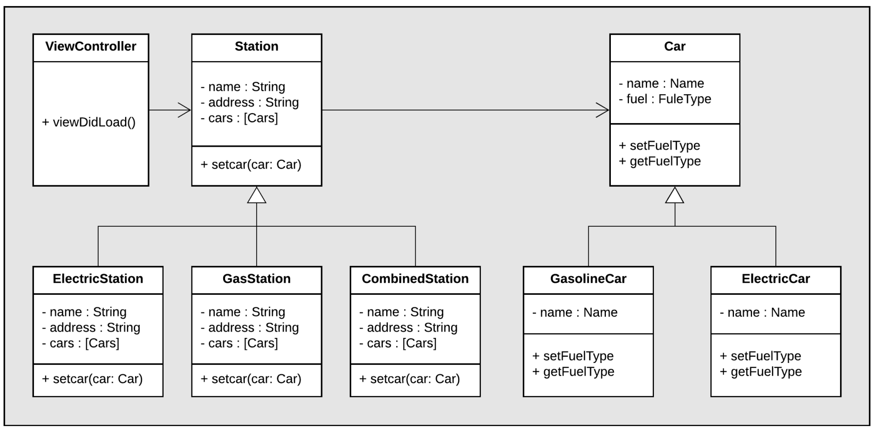


Figure 8.4
Case-study: Bridge - Class diagram
Client: ViewController, Pattern: N/A, Data: Station and Car
R: Required, O: Optional

The listing 8.3 and line 24 and 27 show this pattern in action on the client side. There are no supporting pattern classes in this design.

Code samples:

```
/*ViewConroller class*/

var gfactory : StationFactory
var sBuilder : StationBuilder
var gStation : Station
var car : Car

//Requesting a "null" concrete station object
gfactory = GasStationFactory()
gStation = gfactory.getStation()

//Creating the builder
sBuilder = StationBuilder(gStation)

//Preparing the builder
var stationDirector = StationDirector(sBuilder)
stationDirector.constructStation()

//Getting the station object from the director
var station = stationDirector.getStation()

//Creating a ElectricCar (not using factory or builder for
//car objets)
car = ElectricCar()

//Bridge patten method to add car types to station
station.setCar(car)


```

Listing 8.3
Case-study: Bridge - Client code

Extension scenarios:

Type	Addition of classes			Modification of classes		
	Client	Pattern	Data	Client	Pattern	Data
New concrete station	N	N	R	O	N	N
New concrete car	N	N	R	O	N	N
Average						
Required	0	0	1	0	0	0
Optional	0	0	0	1	0	0

Table 8.4
Case-study: Bridge - Scenarios table
Client: ViewController, Pattern: N/A, Data: Station and Car
R: Required, O: Optional

Lesson learned:

The usage of this pattern in this case-study verify our theoretical explanation from chapter 7, there is no supporting classes/pattern component. The pattern logic is merged into the data component; both Car and Station classes belong to the data component.

Station holds a reference to supported cars, and extension of any of them need only required action in the data component, as shown in table 8.4 on page 89.

8.3.2.2 Facade

Motivation:

This application gets its core data from an internet API source. We need to download the data and store it to a file or memory before using it in the application. This work contains several complex tasks that could be shied from the client. Facade pattern is a good candidate to solve this problem, and in addition, it may contribute to easy extension inside the wrapped interface without needing to alter the code on the client side.

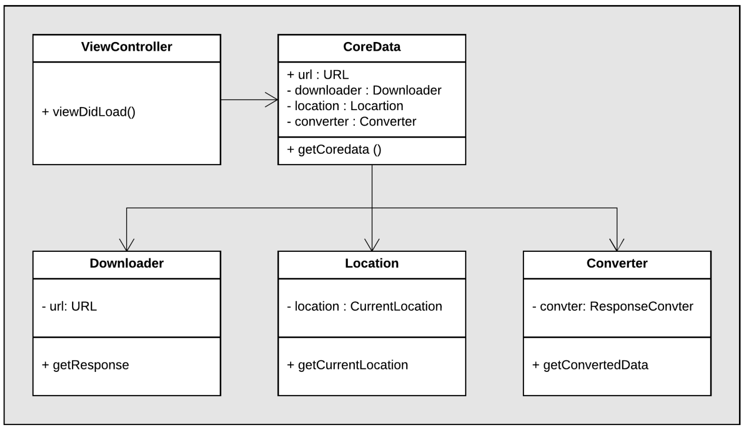


Figure 8.5
Case-study: Facade - Class diagram
Client: ViewController, Pattern: Facade, Data: Downloader, Location and Converter

Code samples:

```
/*ViewConroller class*/

var coredata : Coredata
var stationList : List

//Preparing the facade interface
```

```
    coredata = Coredata (url: URL)

    //Usning the "facade method"
    stationList = StaionListDictionary
      (list: coredata.getCoredata())


```

Listing 8.4
Case-study: Facade - Client code

Extension scenarios:

Type	Addition of classes			Modification of classes		
	Client	Pattern	Data	Client	Pattern	Data
Database support (Save downloaded data)	N	N	R	N	R	N
Extension in Location (support 10 km+)	N	N	N	N	N	R
Average						
Required	0	0	0,5	0	0,5	0,5
Optional	0	0	0	0	0	0

Table 8.5
Case-study: Facade - Scenarios table
Client: ViewController, Pattern: Facade, Data: Downloader, Location and Converter
R: Required, O: Optional

Lesson learned:

In this case-study, we downloaded data from an internet API, and there are a lot of tasks that need to be done in order to retrieve the data, such as, establish the connection, send a request, receive the response and translate the received data and put that into a collection of type dictionary. We wrapped all these tasks using the facade pattern and at the same time removed the complexity from the client. An extension inside the wrapped interface as shown in the table 8.5 does not create any required action in the client.

8.3.3 Behavioral

8.3.3.1 Template Method

Motivation:

We decided to implement a function that shares station information to social media. For the initial phase, we only want to use Facebook and Twitter. For such implementation, using the template method might be beneficial regarding any future extension.

We use an abstract class named *Share.swift* and two concrete classes, *FacebookShare.swift* and *TwitterShare.swift*. On the concrete classes, we override the methods to customize the sharing mechanism to meet the requirements of a specific social media. The *share()* method include a collection of the methods that can be overridden by the concrete classes, and the *share()* itself is untouched by the concrete classes, as show in fig. 8.6.

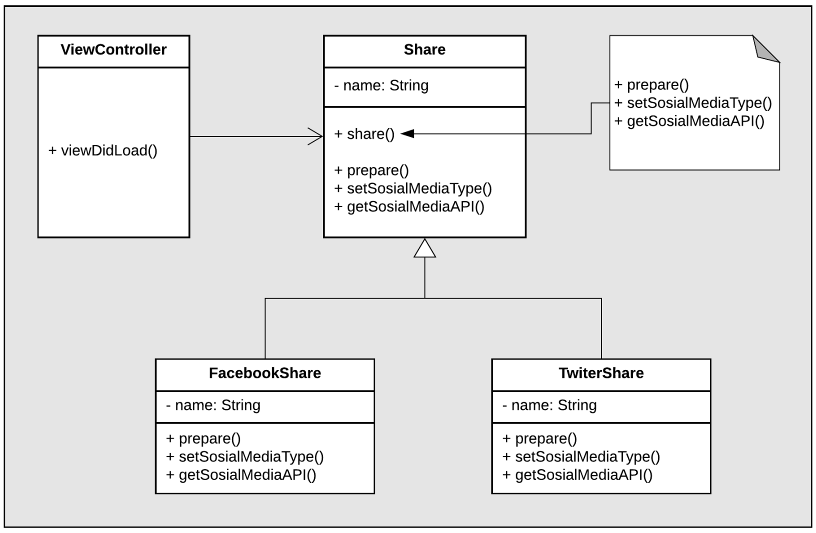


Figure 8.6
Case-study: Template Method - Class diagram
Client: ViewController, Pattern: N/A, Data: Share

Code samples:

The code samples below illustrate the implementation in different components. We can clearly see that introducing new social media require optional actions in the client component.

```
/*ViewConroller class*/

  //Defining the type
  var facebook, twitter : Share

  //Sharing to Facebook
  facebook = FacebookShare(station: Station)
  facebook.share();

  //Shareing to Twitter
  twitter = new TwitterShare(station: Station)
  twitter.share();

  //Shareing to new social media
  newSocialMedia = new NewSocialMediaShare(station: Station)
  newSocialMedia.share();

```

Listing 8.5
Case-study: Template Method - Client component code

```
/*Facebook share class*/

class NewSocialMediaShare: Share {

  override func prepare() {
    //New scoial media related code
  }

  override func setSocialMediaType()
   {
    //New scoial media related code
  }

  override func getSocialMediaAPI() {
    //New scoial media related code
  }

}

```

Listing 8.6
Case-study: Template Method - Data component code

Extension scenarios:

To verify our explanation of Template Method in chapter 7, we illustrate the possible extension scenarios in this case study.

Type	Addition of classes			Modification of classes		
	Client	Pattern	Data	Client	Pattern	Data
New social media support	N	N	R	O	N	N
Average						
Required	0	0	1	0	0	0
Optional	0	0	0	1	0	0

Table 8.6
Case-study: Template method - Scenarios table
Client: ViewController, Pattern: N/A, Data: Share
R: Required, O: Optional

Lesson learned:

Our experience with this pattern after putting it into action in this case-study is that this is one of the simplest design patterns in the GoF list. We support this claim with the implementation as shown in listing 8.5 and 8.6. Making a new concrete class of share and overriding some of the methods will change the behavior of the social media share functionality that is available through the Share interface.

8.3.3.2 Iterator

Motivation:

We use the facade pattern to download the data from Internet API, which gives us a collection of type dictionary in return. We must loop through the dictionary to store the information into an object array of type Station. This is where the iterator pattern can be used. We encapsulate the dictionary into a concrete aggregate class called *StationListDictionary.swift* and put a reference to the corresponding iterator named *StationListDictionaryInterator.swift*, as shown in fig. 8.7

StationListDictionaryInterator.swift contains the logic for the dictionary looping. The drawback we see here is that *StationListDictionaryInterator.swift* can only handle the collection of type dictionary. For every type of collection, we must create a new iterator, which seems to be in some situation quit complex, even it highly decoupling the collection and the iterator itself.

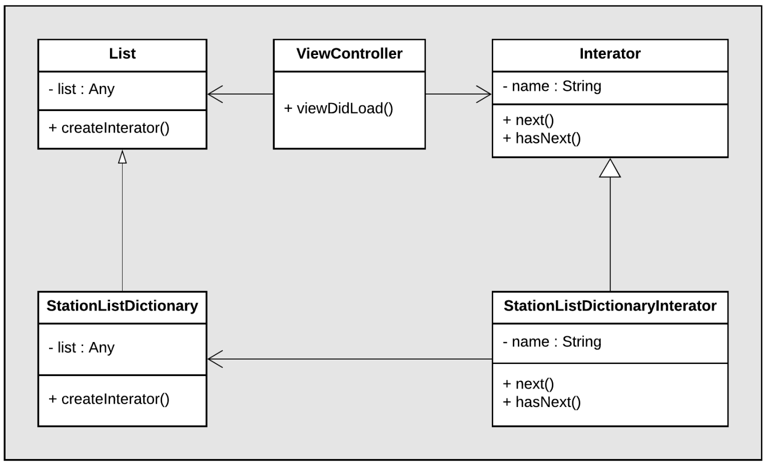


Figure 8.7
Case-study: Iterator - Class diagram
Client: ViewController, Pattern: Iterator, Data: List

The code and extension scenarios on the facing page illustrate that if we need to support a new type of collection, a new iterator must be built to loop through the new logic. That means both data and the pattern components demand required addition of classes. Client component needs to be informed about the new list but not the iterator itself

Code samples:

```
/*ViewConroller class*/

//Creating the list
var stationList : List
stationList = StationListDictionary(list: station_list)

//Creating the interator
var stationListDictionaryIterator =
  stationlist.createInterator()

//Looping
for count in stationListDictionaryIterator {
  //Add to station object array"
}


```

Listing 8.7
Case-study: Iterator - Client component code

Extension scenarios:

Type	Addition of classes			Modification of classes		
	Client	Pattern	Data	Client	Pattern	Data
New type of station list	N	R	R	O	N	N
Average						
Required	0	1	1	0	0	0
Optional	0	0	0	1	0	0

Table 8.7
Case-study: Iterator - Scenarios table
Client: ViewController, Pattern: Iterator, Data: List
R: Required, O: Optional

Lesson learned:

We can realize that letting the client do the looping work is definitely against the encapsulation practice. After implementing this pattern, we see that a collection of objects can be iterated without exposing its underlying representation. We also noted that having a one-to-one relationship between the concrete list and the concrete iterator decouple the collection classes and looping algorithms.

The one-to-one relationship makes the extension of the collection of classes bit more complicated than necessary. Thus, we could have handled several types of collections in a single concrete iterator by detecting the incoming type in that class.

8.4 Summary

In this chapter, we used the case-study application to demonstrate the extensibility patterns in action. We believe that we managed to follow the pattern design and rules as it was described in chapter 7. The extension scenarios from the previous chapter were also able to relate to this case study's situations.

We realized that it could be challenging when two patterns are about to collide because they are designed for almost the same purpose, such as Factory Method and Builder. We combined these two in our way that made it possible to function in our case-study.

We also could realize that patterns can be customized based on the needs, for example, in the Factory pattern, we could have used a single concrete factory to chose between the concrete stations. Builder pattern design could have had separate builders for each type of stations instead of having a common. In the Iterator pattern, we see that one-to-one relationship between the station list and iterator is maybe excessive, but we are aware that doing this decrease the decoupling.

Part IV

Evaluation

9
Part IV: Evaluation
Introduction

In the previous part, we pointed out extensibility patterns from the GoF list. Furthermore, using our case-study, we demonstrated implementation of two design patterns form each category of the pointed out patterns. In this part, we discuss and review our work and findings.

9.1 Outline

We have divided this part into two chapter as the following bullet points explain.

- **Chapter 10: Evaluation of the extensibility patterns**
  In this chapter, we evaluate the pointed out extensibility patterns as well as the lessons from the implemented six patterns.
- **Chapter 11: Conclusion**
  In this chapter, a summary of the key findings from the research will be described, such as expectation at the start compared to results, the importance of this study and recommendations for future research.

10

Part IV: Evaluation

Evaluation of the extensibility patterns

10.1 Introduction

In this chapter, we evaluate the patterns that we discussed and pointed out as extensible from the GoF list. We review these pattern category by category due to that they are grouped based to their purpose. We look into extensibility characteristics based on our subjective view, data from the extension scenarios tables and the lessons from implementation of the six patterns using the case-study.

Decoupling and encapsulation, as mentioned in the background part are also taken into consideration as evaluation criteria. However, we do not have any measurement for these due to the limitations. Thus, we rely on our subjective view of decoupling and encapsulation.

Before the evaluation section, we do a quick recap of the criteria:

Decoupling is ensuring that two or more components are not depend on each other, loosely coupling in other hand is trying to minimize the tightly coupling between components. Encapsulation is hiding the inner workings of a software component behind a defined interface.

10.2 Creational

10.2.1 Result introduction

In this category, we pointed out following design patterns as extensible:

- Factory Method
- Abstract Factory
- Builder

The fig. 10.1 shows the extensibility chart for the creational patterns based on the extension scenarios.

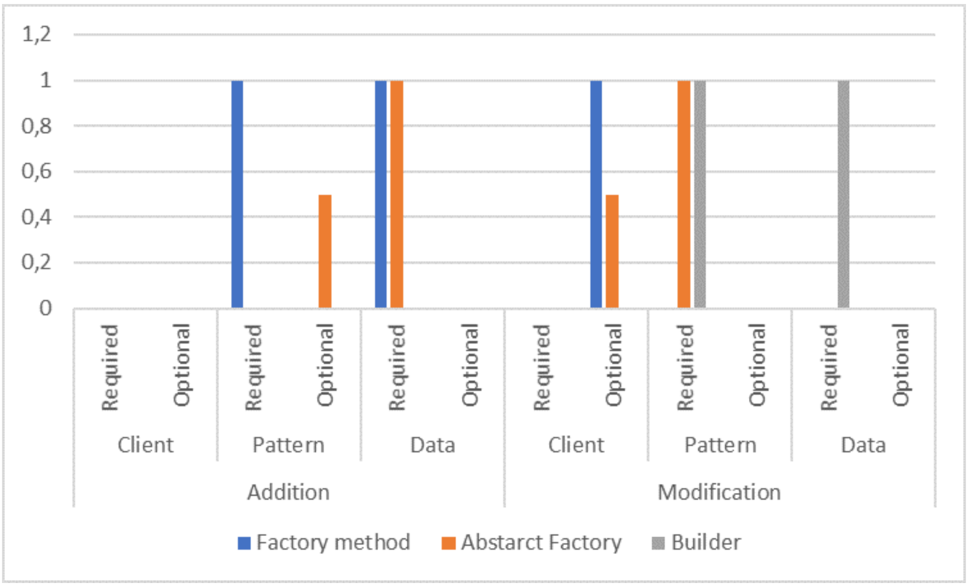


Figure 10.1
Creational patterns - Extensibility chart

10.2.2 Evaluation

Factory method:

Factory method pattern generates work in all three components when business logic is extending. We can see from the chart in fig. 10.1 that it requires addition of classes in both patterns and data component. An optional work is also needed in the client component to take advantage of the new extension. The extending part here is the concrete product that belongs to the data component. The pattern component is where the concrete factory classes are placed, as shown in fig. 7.2 on page 39. Creating of an object is encapsulated using the factory. However, for every new concrete class, we need to build a separate factory. This is to avoid the tight coupling (one-to-many relationship) between the factory and the product, which also increase the amount of work when extending.

Abstract Factory:

Abstract Factory in other hand is less aggressive than the Factory method because it encapsulates the creating much more by adding more layers between the client and the final product. Encapsulation must not be confused with decoupling, as described in the introduction of this chapter. Extending with the new concrete product does not require new abstract factory in the pattern component, as shown in fig. 10.1 on page 104, instead a modification is needed to add support for it in the existing factories. A new product type or abstract product will generate new classes in the data component, and required modification in pattern classes to update the existing factories; it may also force us to create new factory if there is a need to build a unique factory with a new combination of abstract classes.

Builder:

Builder pattern seems to be a winner in the category. Extension related to this pattern does not require any actions in the client component and only modification in the pattern component. This is due to the high encapsulation of the object creation. We interpreted this in a way that does not make it necessary to create concrete builder when data component is extending by new concrete product. It makes us believe that there is no answer to how one should interpret a pattern because patterns can be customized and interpreted in different ways.

10.2.3 Review of the implementation

Factory method:

Factory method pattern from the implementation shows that every time we want to extend by a new concrete station in the data component, we also need to create a class in the pattern component in which has a reference in the view controller class. Creating a new type of station generate two new classes, one concrete station class, and a corresponding factory. The corresponding factory needs to be referenced in the view controller, as shown in listing 8.1. We mean that customizing this pattern to use a single factory (one-to-many relation between factory and product) is much more efficient when extending.

Builder:

Builder pattern was used to configure the default values for the station. We see that building a station in our case-study is highly encapsulated, all of the station configurations and creation are completely hidden. This is the benefit of the builder pattern. It is easier to extend the station object with variables without needing to inform the client. We could have had specific builders for each type of station, but in our point of view, Builder pattern is normally used to encapsulate the default configuration which should be common for all concrete class of type Station. We could have had separate builders for each type of stations instead of having a common builder, but we chose this approach because default values in our case-study are et to be the same for all type of stations.

10.2.4 Summary

In this category, Builder pattern seems to be a good design pattern when it comes to extensibility; it encapsulates the creation and the configuration part. It is important to emphasize that we interpreted builder in a way that does not force us to create a one-to-one relation between the concrete builder and concrete product. It reflects what we see on the graph 10.1.

Factory method and Abstract Factory are also encapsulating the creation, but put a bit more complex to the design due to the one-to-one relationship between the concrete factory and concrete product classes, which also means that, the decoupling is more in present here.

However, the builder pattern can also be used with more decoupling or one-to-one relationship as Factory method and Abstract Factory which may increase the complexity. This shows also that design patterns can be implemented in different ways or customized based on the needs, which makes it very challenging to evaluate.

In our point of view, Builder pattern deserves an assessment in an initial phase of any application design. Factory method and Abstract Factory is more suitable if one see the need for it in relation to the problem one want to solve. With the experience we have from the implementation, we can see that combining these patterns is challenging because all three are used for creation purpose.

10.3 Structural

10.3.1 Result introduction

In this category, we pointed out following design patterns as extensible:

- Bridge
- Facade

The fig. 10.2 shows the extensibility chart for the structural patterns.

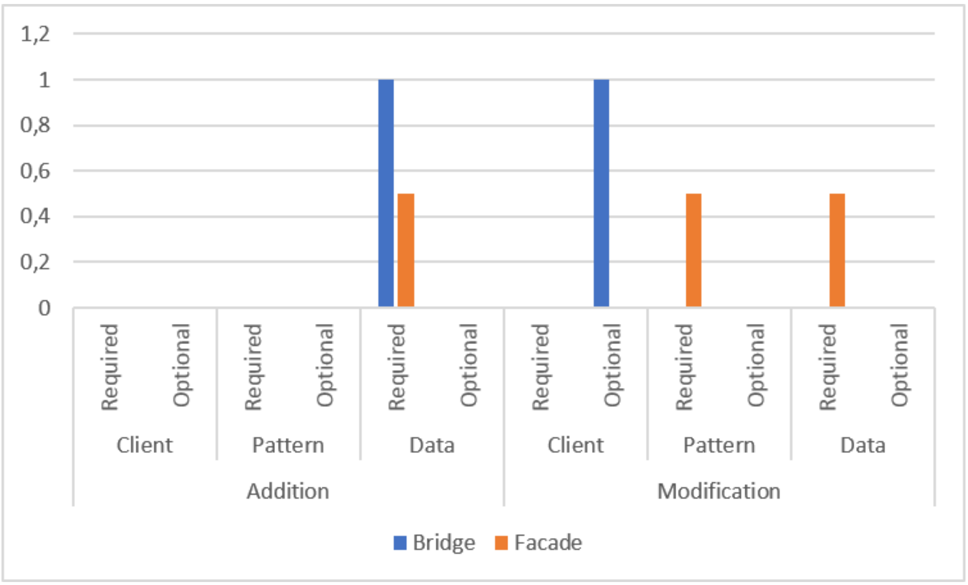


Figure 10.2
Structural patterns - Extensibility chart

10.3.2 Evaluation

Bridge:

Bridge pattern needs the least actions in this category when extending, for the reason that pattern and data component are merged. From the fig. 7.8 on page 49, we can see that both *abstraction* and *implementor* are part of the data component. We do not have any other supporting classes to make this design complete. When extending, we only need to add classes to the data component and let the client use it which is optional action seen from the clients perspective.

Facade:

Facade pattern also seems to do a good job when it comes to encapsulating the inner work by putting a wrapped interface around a possible complex system. We can see from the chart that any extension on the data component does not require any actions on the client side that uses the wrapped interface. This behavior may not apply if the wrapped interface needs to be changed, but our assumptions and understanding are that it stays unchanged for a long period.

10.3.3 Review of the implementation

Bridge:

Bridge pattern was used to a have a relation between station and car. A new type of station or car may appear in the future and to easily support this kind of extension, we used the bridge pattern to avoid the so-called exploding class hierarchy. Unlike many other patterns, this has a special design where pattern and data components are merged.

Facade:

The facade was implemented in our case-study to put a wrapped interface around the classes and tasks that were involved in the download of data from the Internet API. Extending inside the wrapped interface, for example including support for new API types will not affect the functions the view controllers already are using. View controller needs only to run a method that is defined in the Facade class, in our case-study application, it is called Downloader.getStations() that returns a dictionary of all stations in a given area. Inside the wrapped class there are a lot of tasks that need to be completed before creating the stations dictionary, but the client does not need to know anything about these tasks.

10.3.4 Summary

Both patterns in this category are useful extensibility patterns. The builder has data and pattern components bundled, and Facade does a good job when it comes encapsulation of complex tasks.

The fig. 10.2 on page 107 shows that facade pattern really shield the client from updating it's data when extension happens behind the facade interface, no action on the client side is needed. Bridge demands only required action in the data component. In our point of view, both design patterns are useful extensibility pattern with less complexity that needs to be considered for use in a design phase of an application.

10.4 Behavioral

10.4.1 Result introduction

In this category, we pointed out following design patterns as extensible:

- Template Method
- Command
- Iterator
- Mediator
- Observer
- Strategy
- Visitor

The fig. 10.3 shows the extensibility chart for the behavioral patterns.

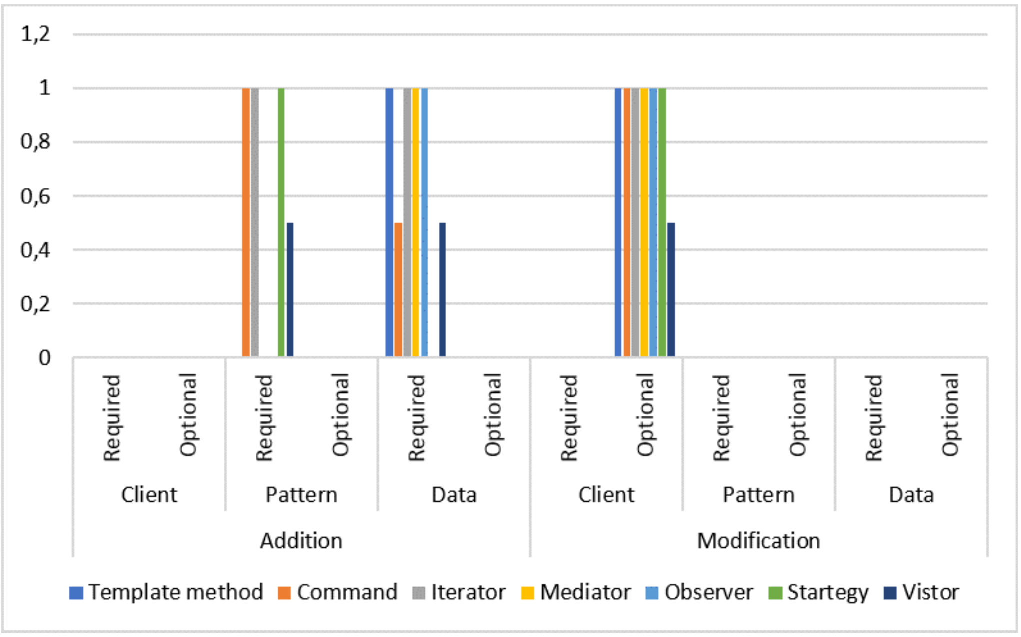


Figure 10.3
Behavioral patterns - Extensibility chart

The chart illustrates that all the behavioral patterns generate optional actions in the client component when an extension happens in the data or pattern component. Hence, it does not break any existing function due to the non-existence of required actions in the client. This behaviour is the same as what we saw in the creational and behavioral patterns.

10.4.2 Evaluation

Template method:

Template method does not demand any required or optional work in the pattern component; this is due to that this pattern logic is merged into the data component. This is visible in the fig. 10.3 on page 109. In our point of view, this is one of the simplest patterns in the GoF list. If one wants to allow the concrete classes to change the behavior of several methods without generating lot work in other components than data component, this pattern is very useful.

Command:

Command pattern's average number of extension scenarios require actions in the pattern component as well as the data component, as shown in fig. 10.3 on page 109. The reason for this behavior is the decoupling of command, receiver, and the invoker, as shown in fig. 7.17 on page 61. Creation of new class in data component force us to create classes in the pattern component because of the need for one-to-one relationship.

Iterator:

Iterator does not vary a lot compared to Command; it requires addition of classes in both data and pattern component as well as optional action in the client. The latter is only if the extension needs to be used by the client. For every new collection in the system, we need to create a new iterator to meet the best practice design of this pattern, which seems to overkill in some situations.

Mediator:

Mediator design has a unique design because the extension in data component might not require any actions in the pattern component even these two are not merged as in Template method. We can have several numbers of instances of a different type of classes that can have a reference to the shared instance of Mediator.

Observer:

Observer seems to be more aggressive than the Mediator when it comes to the needed actions in different components. A concrete class of type A can have zero or more Observer of only type B registered with it; this forces us to create a new type of observer when a data component extends with a new type of class.

Strategy:

Strategy pattern is, in fact, fascinating, and likewise Mediator it has both pattern and data component separately. However, it differs when it comes to the extension. There is only needed required action in the pattern component, as shown in fig. 7.23 on page 71.

Visitor:

Visitor data from the chart is measured based on two potential extension scenarios. However, it require least actions in this behavioural category. A new visitor can be added any time to the system, and it can visit many elements objects in the data component. Thus, we consider this as a good extensibility pattern.

10.4.3 Review of the implementation

Template Method:

Template Method in the case-study was used to the social media share functionality where we only needed to create a concrete class of the Share interface and override some of the methods. For the client component, it is still a Share interface, as shown in listing 8.5. In Template method design, data and pattern component are merged. Thus, we only need a new concrete class of the interface Share.

Interator:

Iterator pattern was used to iterate the collection of type dictionary. We built an iterator for that called *StationListDictionaryInterator*. We realized that extending by a new type of collection creates some amount of work in the pattern component that can be avoided if we customize the pattern to use only one concrete iterator to handle multiple type collection.

10.4.4 Summary

Evaluation of the behavioral extensibility patterns shows us that Command and Iterator patterns are the most challenging patterns in terms of the work that needs to be done when extending.

Template method, Mediator, Observer Strategy, and Visitor patterns seem to be in a group that needs less work when the extension happens. In our view, Strategy and Visitor are patterns that not only require the least actions when extending, but can also contribute to change the behavior of an object at runtime without changing anything in the data component.

11

Part IV: Evaluation

Conclusion

11.1 Summary

Our primary goal from the beginning of this study was to find design patterns that can be used in the design phase of an application life-cycle to achieve extensibility. To limit the scope of this study, we decided to only look into GoF design patterns. Throughout this empirical study, we discussed all the GoF design patterns and pointed out those we considered as extensibility patterns based on their purpose. We excluded patterns where the intent was to increase performance, minimize the memory allocation, extend a tightly coupled software and other purposes that were not in the scope or did not relate to extensibility.

We investigated the pointed out patterns by putting them up against extensibility characteristics based on our subjective view, data from the self-designed extension scenarios tables and the lessons from implementation of the six patterns using the case-study. Decoupling and encapsulation are also taken into consideration as evaluation criteria. Quantitative measurement for these is not in the scope of this study. Thus, we rely on our subjective view of decoupling and encapsulation.

We measured the required and optional actions that were needed on the different components when extending some of the parts, actions in this situation refers to either addition or modification of classes.

Furthermore, we used the Swift programming language to experiment six of the pointed out patterns, two from each design pattern category, using a case-study application. This is due to do the practical work to verify our theoretical research.

Evaluation of the extensibility patterns shows us that pattern design and logic differs a lot and used for different purposes. Due to this, it is difficult to point out patterns that are better than the others. We also devolved an understanding that patterns can be customized, combined and used in different ways even GoF book has some guidelines to follow. There is no one way of implementing a pattern, and in this study, we discuss and implement as we interpret it, without any customization.

In some extensibility patterns, an extension generates work in only one component, which makes it less complex to implement, understand and extend. We found that in some cases complexity grows by increased decoupling, which in turn will lead more amount of work when extending, especially when the creation of new classes in the data component need to have a one-to-one relationship with the pattern component classes. Increased decoupling may be beneficial for code maintainability and understandability, but it requires quite a lot of work when extending. This finding is the opposite of the exportation we had at the beginning of this study. We believed that decoupling had only positive intentions on extensibility.

As the result, we have divided the extensibility patterns into two groups, one with the less complex patterns where extension generates less work than the other group in which contains more challenging patterns when it comes to extending.

11.1.1 Less complex patterns

In this group, the average number of needed actions when extending is less then 2,5.

- **Creational:** Builder
- **Structural:** Bridge and Facade
- **Behavioral:** Template method, Mediator, Observer, Strategy and Visitor

11.1.2 Challenging patterns

In this challenging group average number of needed actions exceeds 2,5.

- **Creational:** Factory Method and Abstract Factory

- **Behavioral:** Command and Iterator

During this study, we made up one favorite from each category that we believe has the advantage of implementing or should at least taken into consideration at the design phase of application life-cycle. Builder, Facade, and Visitor are patterns that not only require the least actions when extending, but they also solve some conmen problems every application may have, such as, encapsulating configuration of an object, hide complexity behind wrapped interface and change behavior of an object at runtime. These may not sound like extensible features, but in fact, these flexibilities make it easier to expand the system, which is described in the scenarios tables in chapter 7 and 8.

11.2 Further work

We excluded some patterns and considered these as not extensibility patterns due to their purpose. This decision was done based on our assumptions. However, it would be interesting to explore some of the excluded patterns that had in our point view another purpose then solving extensibility problems. The future research may then focus on the excluded patterns to see if they also can be used as extensibility patterns.

We are divided the excluded patterns into three groups:

- **Patterns for extension after design phase:**
  - Adatper and Decorator
- **Performance related patterns:**
  - Singleton, Prototype and Flyweight
- **Other patterns:**
  - Composite, Interpreter, Chain of Responsibility, Memento and State

The interesting patterns to look into when it comes to extensibility is the group of other patterns that we struggled to decide its characteristics during this study.

Based on the scope of this study, we looked only into the Go4 list and implemented six of the twelve pointed out patterns. Further work may also look into other design patterns or implement the remaining the six patterns using the same case-study application.

Part V

Appendix

A
Code

The case-study experimental code can be obtained from the following URL:

https://github.com/tkarthig/eFuel

Bibliography

[1] Christopher Alexander. *The timeless way of building*. Vol. 1. New York: Oxford University Press, 1979.

[2] B Annappa et al. "Analyzing Design Patterns for Extensibility". In: *Computer Networks and Intelligent Computing*. Springer, 2011, pp. 269–278.

[3] Artur Kamil Barczynski. "Assessing the Impact of Using Design Patterns of Enterprise Application Architecture". MA thesis. 2014 - Universitetet i oslo.

[4] Carlo Chung. *Pro Objective-C Design Patterns for iOS*. 1st. Berkely, CA, USA: Apress, 2011. ISBN: 9781430233305.

[5] James O Coplien. "Software design patterns: Common questions and answers". In: *The Patterns Handbook: Techniques, Strategies, and Applications* (1998), pp. 311–320.

[6] John Deacon. "Model-view-controller (mvc) architecture". In: *Online][Citado em: 10 de março de 2006.] http://www. jdl. co. uk/briefings/MVC. pdf* (2009).

[7] Adam Freeman. *Pro Design Patterns in Swift*. 1st. Berkely, CA, USA: Apress, 2015. ISBN: 148420395X, 9781484203958.

[8] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Softwareeee*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[9] Andrew Hoog and Katie Strzempka. *iPhone and iOS forensics: Investigation, analysis and mobile security for Apple iPhone, iPad and iOS devices*. Elsevier, 2011.

[10] John Hunt. "Gang of four design patterns". In: *Scala Design Patterns*. Springer, 2013, pp. 135–136.

[11] Apple INC. *The Role of View Controllers*. 2016. URL: `https://developer.apple.com/library/content/featuredarticles/ViewControllerPGforiPhoneOS/` (visited on 12/28/2017).

[12] Chandramohan Lingam (Intel). *Performance Testing and Tuning - Part II - Coupling*. 2012. URL: `https://software.intel.com/en-us/articles/performance-testing-and-tuning-part-ii` (visited on 01/02/2017).

[13] Martin Røed Jacobsen. “A Possible Composite Design Pattern for Immature REST APIs.” MA thesis. 2016 - Universitetet i oslo.

[14] Julien Lange. *Swift 2 Design Patterns*. Packt Publishing Ltd, 2015.

[15] Gaute Bernhard Sveen Lyngstad. “Design Patterns In A Smartphone Environment: An Empirical Study using Android. Universitetet i oslo”. MA thesis. 2013 - Universitetet i oslo.

[16] Josephine Micallef. “Encapsulation, reusability and extensibility in object-oriented programming languages”. In: (1987).

[17] Snesh Prajapati. *Factory Patterns - Simple Factory Pattern*. 2016. URL: https://www.codeproject.com/Articles/1131770/Factory-Patterns-Simple-Factory-Pattern (visited on 12/28/2017).

[18] Kai Qian, Jigang Liu, and Frank Tsui. “Decoupling metrics for services composition”. In: *Computer and Information Science, 2006 and 2006 1st IEEE/ACIS International Workshop on Component-Based Software Engineering, Software Architecture and Reuse. ICIS-COMSAR 2006. 5th IEEE/ACIS International Conference on*. IEEE. 2006, pp. 44–47.

[19] Steve. *OO Principles: Encapsulation and Decoupling*. 2009. URL: http://www.bryantwebconsulting.com/blog/index.cfm/2009/7/9/OO-Principles-Encapsulation-and-Decoupling (visited on 01/02/2018).

[20] Walter F Tichy. “A catalogue of general-purpose software design patterns”. In: *Technology of Object-Oriented Languages and Systems, 1997. TOOLS 23. Proceedings*. IEEE. 1997, pp. 330–339.

[21] Kim W Tracy. “Mobile application development experiences on Apple’s iOS and Android OS”. In: *Ieee Potentials* 31.4 (2012), pp. 30–34.

[22] Weik-Tek Tsai et al. “Testing extensible design patterns in object-oriented frameworks through scenario templates”. In: *Computer Software and Applications Conference, 1999. COMPSAC’99. Proceedings. The Twenty-Third Annual International*. IEEE. 1999, pp. 166–171.

[23] Wikipedia. *About Swift*. 2012. URL: https://swift.org/about/#swiftorg-and-open-source) (visited on 12/27/2017).

[24] Wikipedia. *Swift (programming language)*. 2012. URL: https://en.wikipedia.org/wiki/Swift_(programming_language) (visited on 01/12/2017).