DESIGN PATTERN DETECTION FOR UML INTEGRATED META-MODEL

BY

ABDULLAH ALWI AL-BAITY

A Thesis Presented to the
DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In
SOFTWARE ENGINEERING

JANUARY 2018

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN- 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **Abdullah Alwi Hussein Al-Baity** under the direction of his thesis advisor and approved by his thesis committee, has been presented and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN SOFTWARE ENGINEERING.**

M Alshayeb 7/1/2018

Dr. Mohammad Alshayeb
(Advisor)

Dr. Khalid A. Al-Jasser
Department Chairman

7/1/2018

Dr. Mahmood Niazi
(Member)

Dr. Salam A. Zummo
Dean of Graduate Studies

Sajjad Mahmood 7/01/2018

Dr. Sajjad Mahmood
(Member)

عمادة الدراسات العليا
DEANSHIP OF GRADUATE STUDIES

10/1/2018
Date

Dedicated to

My Beloved Mother

ACKNOWLEDGMENTS

All praises and worship are for ALLAH SWT alone; we seek His help and ask for His forgiveness, we thank Him for providing us with knowledge and we always strive to achieve His reward. I feel privileged to glorify His name in the sincerest way through this small accomplishment. Peace and blessings be upon the greatest human being that ever walked on this earth, the last and the final Prophet, Muhammad (peace be upon him), upon his family, his companions, and all those who follow him until the day of judgment.

I would like to thank my parents, Mr. Alwi Al-Baity and Mrs. Alwyah Ba-Alwi, from the bottom of my heart for their immense patience and the motivation they had given me and also for teaching me the true meaning of hard work through their lives. My two brothers Khalid and Majed, my sisters Nawal and Fawzyah, and my beloved wife, too share the same acknowledgments for their support. It would not be wrong to say that they have worked more than me, in making me what I am today. I ask Allah to reward all of them the best in this world and the hereafter, Ameen.

I would like to pay a high tribute to my thesis advisor Dr. Mohammad Alshayeb for his invaluable guidance and helpful ideas throughout this research. His appreciation and words of motivation gave a new life to my efforts in hard times. I am also indebted to him for his valuable time, efforts, his continuous support, and inspiration. I always admired his knowledge, intuition, and vision. I feel very great to say, he has successfully instilled in me a passion for scientific research, which will continue to guide me for many more years to come.

I would like to acknowledge Dr. Mohammed Misbhauddin for helping me in understanding the Integrated Metamodel and for guiding me to the appropriate way to define the design patterns, I am also indebted to him for his valuable time, efforts, his continuous support, and inspiration.

I owe a deep appreciation to my committee members, Dr. Mahmood Niazi and Dr. Sajjad Mahmood, who have spared their valuable times and given their thoughtful suggestions and they have been a source of constant help and encouragement. I thank all the other teachers who taught me in the university.

I acknowledge King Fahd University of Petroleum & Minerals for supporting my M.S. studies. I am thankful to Dr. Khalid A. Al-Jasser the chairman of Information and Computer Science Department, for providing an excellent environment for learning and research in the department.

جزاكم الله خير

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

GoF	:	Gang of Four
DP	:	Design Patterns
DPDT	:	Design Patterns Detection Tool Technique
SC	:	Sequence Diagram
CD	:	Class Diagram
UD	:	Use Case Diagram
UD	:	Not Specified
UD	:	Not Applicable

ABSTRACT

Full Name : [Abdullah Alwi Hussein Al-Baity]
Thesis Title : [Design Pattern Detection for UML Integrated Meta-Model]
Major Field : [Software Engineering]
Date of Degree : [January, 2018]

Design patterns are general reusable solutions that can be applied to a commonly occurring problem within a given context in software design. Design patterns are not finished designs that can be transformed blindly into the source code. They are descriptions or templates for how to solve a problem that can be used in many different situations. Taking care of such area will improve the software development processes and reduce the maintenance cost as well. In this research, we propose a new technique to detect design patterns from a UML integrated meta-model which is built from three UML views, the structural view represented by a class diagram, the behavioral view represented by sequence diagram and functional view represented by use case diagram. In the proposed approach, first, we conduct a systematic literature review to collect and review all the design pattern detection techniques proposed in the literature. Second, we represent design patterns using the UML integrated metamodel to have all the different views features in one concrete XML file. Third, we validate the representation of design patterns using the integrated metamodel to see if the integrated metamodel gives more information about the design patterns than the other individual representation forms. Finally, we develop a tool that detects the design patterns in the integrated metamodel to show that the integrated metamodel representation is giving more information about the design pattern, which will increase the level of accuracy. The manual and the automatic validation of our technique showed that the

integrated metamodel representation of the design patterns gives more information about the design pattern to be detected, which will decrease the level of accuracy of design pattern detection

ملخص الرسالة

الاسم الكامل: عبدالله علوي حسين البيتي

عنوان الرسالة: تقنية إيجاد أنماط التصاميم البرمجية للنماذج المدمجة في لغة التصميم الموحدة

التخصص: هندسة البرمجيات

تاريخ الدرجة العلمية: يناير 2018

تعتبر نماذج التصميم فرعا مهمّا من فروع وعلوم هندسة البرمجيات. إن الهدف الرئيسي لنماذج التصميم هو ان يقوم المصمم باستخدام الحلول المتكررة لحل مشاكل متكررة وجدت في عمليات التصميم البرمجية. ولكن يجب ان نعلم ان هذه الحلول المتكررة ليست عبارة عن قوالب جاهزة يمكن استخدامها مباشرة، فهي لا تعدو إلا أن تكون نماذج للحل، فكما هو مبين من اسم هذه النماذج فأنها تحتاج للتعديل والتكييف فلابد أن نحدد ونكيف هذه النماذج لكي تتوافق مع المشاكل البرمجية التي نريد حلها. ومن الجدير بالذكر إن أغلب هذه النماذج التصميمية تعتمد بشكل أساسي في البرمجة والتصميم على التوجه الكائني في. لذلك فإن نماذج التصاميم دايما ما تضع تصورا" كاملا ورسوما" مبنية على العلاقات والتفاعلات بين كل مكون من مكونات البرنامج والفئات الخاصة به. اذا فالاهتمام بهذا الجانب من هندسة البرمجيات وتطويره من شأنه أن يطور البرمجيات ويزيد من جودتها ويقلل من تكلفة تصميم هذه البرمجيات وصيانتها .

لذلك قمنا في هذه الرسالة وهذا البحث، بتقديم تقنية جديدة للكشف عن نماذج التصاميم في مرحلة التصميم قبل ان نبدأ بتشفير البرنامج، وذلك باستخراجها من النماذج المدمجة في لغة التصميم الموحدة، حيث اننا سوف نستكشف نماذج التصميم بالنظر الى ثلاث نواحي وجهات مهمة وهي: نماذج التصميم الإنشائية أو الخلقية، ونماذج التصميم السلوكية، ونماذج التصميم الهيكلية.

في هذا البحث سوف نقوم بدمج عدة تقنيات للحصول على تقنية اكتشاف انماط او نماذج التصميم، مثل: قواعد البيانات، الذكاء الاصطناعي، المقاييس البرمجية، تقنية النقاط.

ماجستير في هندسة البرمجيات
جامعة الملك فهد للبترول والمعادن
الظهران, المملكة العربية السعودية
2018

CHAPTER 1

INTRODUCTION

Design patterns are general reusable solutions that can be applied to widely occurring problems within a given context in software design. Design patterns are not completely ready designs that can be used directly inside source code to solve the problem. They are descriptions that can be used in many different situations to solve commonly occurring problems. They provide best practices in a formal way to the designer and programmer to solve the commonly occurring problems. The complex software system design needs complex tools and tasks in order to generate high quality software design.

One of the simplest, yet more powerful, techniques to improve a design is to use patterns whenever possible and to follow some well-known rules to realize them. The application of this technique to an existing design is tedious because it requires finding all pattern realizations used in the design. He design patterns plays a major role in improving the software design [1].

Design pattern detection is not only beneficial to forward engineering it also can help and aid to the reverse-engineering, as well as helping in code comprehension. Design patterns are very important that it can reflect the designers' intents, and code maintenance would be easier.

In the last decades, Object Oriented concept have gained immense popularity and strength over the other programming languages concepts and paradigms. One of the main reasons behind this acceptance is that the Object Orient Paradigm gives the priority to the component and modeling aspects and concepts. This issue is very important from the problem domain's perspective. "With this huge growing and popularity of the Object-Oriented concepts and paradigm the needs were raised to provide a general standard for Object-Oriented Analysis and Design, the Object Management Group (OMG) adopted UML as a standard language for the design and analysis of Object-Oriented Programs. UML is a graphical language that provides notations and action semantics to describe and design software systems" OMG group[2].

The integrated meta-model is composed of meta-models of the three UML meta-model diagrams, which are the meta-model of the class diagram, the meta-model of the sequence diagram and the meta-model of the use case diagram. The integrated meta-model integrated the three UML structural (Class Diagram), behavioral (Sequence Diagram), and function (Use Case Diagram) views to provide more understandability of the overall UML [24].

Software design patterns gained a lot of attention in the literature review. Recently, many approaches and methods have been proposed to detect Design Patterns from the different software artifacts. Some of these methods are based on the source code [4], while other methods are based on diagrams or graph trees such as UML or Directed Graph [5]. However, most of the previous studies were based on only one single view either structural, behavioral, or functional view. Other design pattern detection methods use the class diagram to detect the structural behavioral and sequence or statechart diagram to capture the behavioral of the design patterns. This may affect the accuracy of the Design Pattern detection since it will be biased by one single view as different views may provide more information.

1.1 Problem Description

Although the concept of detecting design patterns at the model level is extensively researched, several problems remain. The research of design pattern detection techniques is still under development surrounded by different open gaps and challenges. The main issue and challenge is due to the lack of design pattern detection in the integrated meta-model or inter-related models: Based on our literature review, there was no study conducted for design pattern detection at any integrated meta-model or inter-related views. All the previous studies were focusing on detecting design pattern at a single view such as [6, 7] or separated two views such as [4, 8].

1.2 Motivation

Design patterns detection at the model level is more crucial and accurate than at the low level (Code-level) because the model level has multiple views to represent the software system's functionality and aspects. A typical software system design is represented by using different diagrams from the three main UML views (Structural, Behavioral, and Functional), each view is capturing major and crucial characteristics of the software system. Multiple views have been used by researchers in different areas since the UML-based techniques got more attention. Most of the research studies published on design pattern detection techniques based mainly on a single view, some prominent studies proposed a design detection technique based on only a class diagram (Structural view) [5], some are based only on sequence diagram (behavioral) [6]. Only very view studies proposed a design pattern detection technique based on two views (Structural view, and Behavioral view) [3, 7]. The main motivation behind using multiple views in design pattern detections are [24]:

1. There is a complementary relationship between all the UML views. Detecting design pattern from one or two views would ignore the other supplementary information that could be obtained from the other views.
2. Detecting designs patterns from the inter-view relationship, multiple or integrated views is more accurate than detecting design patterns using a single view at a time.
3. Detecting design patterns at the model level is much better than detecting it at the code level since we can detect the design pattern earlier and improve the design before going to the code level.

1.3 Research Contribution

A typical software system design is represented by using different diagrams from the three main UML views (Structural, Behavioral, and Functional); each view is capturing major and crucial characteristics of the software system. Multiple views have been used by researchers in different areas since the UML-based techniques got more attention. However, these views are not connected or integrated to each other, instead, they are separated views.

In our research we are going to use UML integrated metamodel of the three main views: functional view, behavioral view and structural view to build up a design pattern detection technique that is based on taking the advantages of the integrated metamodel, since all the different views features will be placed in one UML integrated metamodel file, instead of taking each one separately.

We are going to use the class diagram to represent the structural view features, and use case diagram to represent the functional view features and the sequence diagram to represent the behavioral view features. However, this is the first research study that discussed the design patterns from the functional point of view. We have implemented the design patterns by the three main views, the structural view, the behavioral view and the functional view.

Each view will provide us with different features, these features are integrated used the UML integrated metamodel. We have also conducted a systematic literature review for state of art of design patterns detection approaches to have a deep understanding of the design patterns detection techniques. The systematic literature review survived 7,403 different papers in total that matched our search string.

The systematic literature review helped us to answer different research questions to classify and categorize the different techniques based on the type of technique and the representation form they have used in their proposed approach. The systematic literature review also opened a lot of different future gaps that haven't yet fulfilled by the researchers.

1.4 Thesis Structure

The rest of this thesis is structured as follows:

Chapter 2: This section discussed the background knowledge of what is our research was based on. This section describes all the design patterns categories and what each design patterns intended to do. It also describes UML (Unified Modeling language) different views, and how each view is described and defined in it. The declarative specification language (OCL) is discussed also in this section, as well as the integrated metamodel.

Chapter 3: This section discussed the traditional literature review where all the previous detection techniques were discussed and defined. This section also contains the systematic literature review that discussed and reviewed more than 7,403 different papers and categorized all the different techniques based on the detection techniques and representation forms they have used.

Chapter 4: This section discussed our proposed methodology, this section contains the functional representation of design patterns, the traditional representations of design pattern compared with the design pattern representation used the three main views, and the UML integrated metamodel representation.

Chapter 5: This section discussed the validation of our proposed approach; it contains the visual validation, the tool support, and the empirical study.

Chapter 6: This section discussed the conclusion of our proposed technique as well as the future work.

CHAPTER 2

Background

2.1 Design Patterns

Design Pattern is one of the most important areas in software engineering recently. Design patterns are general reusable solutions that can be applied to widely occurring problems within a given context in software design. Design patterns are not completely ready designs that can be used directly inside source code to solve the problem [8]. They are descriptions that can be used in many different situations to solve commonly occurring problems. They provide best practices in a formal way to the designer and programmer to solve the commonly occurring problems [9]. For instance, in many cases in the real world situations we might need to create just one instance of a specific class. This pattern is called Singleton pattern. In Other cases in software development a programmer needs to have only one database connection that can be shared by many objects because creating a separate database connection for every object is very expensive. Another example of a real world situation is the Adapter design pattern. Adapter design pattern is a structural design pattern that used to repurpose a specific class with a many different interfaces. Another term to reference an adapter class is a wrapper, which lets you "wrap" actions into a class and reuse these actions in the correct situations. A classic example might be when you create a domain class for table classes. Instead of calling the different table classes and calling up their functions one by one, you could encapsulate all of these methods into one method using an adapter class. This would not only allow you to reuse whatever action you want, it also keeps you from having to rewrite the code if you need to use the same action in a different place.

Design Patterns was first proposed in 1994 by four authors (Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides) they published a book titled as a Design Patterns - Elements of Reusable Object-Oriented Software [9]. This book initiated the concept of a design pattern as a solution description in Software implementation and development. Later on, researchers called these authors as Gang of Four (GoF).

The Gang of Four proposed 23 Designed Patterns that can be classified into three different categories, Creational, Structural and Behavioral patterns. Every category of Design Pattern contains many different Design Patterns that can be used as a reusable solution in order to solve some widely according to problems. Every Design Pattern has a terminology to a specific problem scenario.

Many researchers have been accomplished by different institutes and universities to improve the original draft of GoF Design Patterns, as well as introducing new design patterns that could be added to the 23 GoF Design Patterns.

Creational Patterns provide the developer with a flexible way to create an object whilst allowing the developer to hide the object creation logic, instead of directly creating and instantiating objects using some new operators. This feature gives a software flexibility of choosing which object to be created or selected for a given use case. Table 1 Creational Design Patterns [9] lists the GoF Creational Design Patterns.

Table 1 Creational Design Patterns [9]

No.	Design Pattern	Description
1	Abstract Factory	Creates an instance of several families of classes.
2	Builder	Separates object construction from its representation.
3	Factory Method	Creates an instance of several derived classes.
4	Object Pool	Avoid expensive acquisition and release of resources by recycling objects that are no longer in use.
5	Prototype	A fully initialized instance to be copied or cloned
6	Singleton	A class of which only a single instance can exist.

Structural Patterns provide the way of composition between class and objects. Structural Design Patterns are used to ease the software design by simplifying the relationship between the software design different entities. Structural Design Patterns used the concept of inheritance in order to establish the composition between the interfaces, and provide the ways to establish the composition between different objects to come up with different functionality. Table 2 Structural Design Patterns [9] lists the GoF Structural Design Patterns.

Table 2 Structural Design Patterns [9]

No.	Design Pattern	Description
1	Adapter	Match interfaces of different classes
2	Bridge	Separates an object's interface from its implementation
3	Composite	A tree structure of simple and composite objects
4	Decorator	Add responsibilities to objects dynamically
5	Facade	A single class that represents an entire subsystem
6	Flyweight	A fine-grained instance used for efficient sharing
7	Proxy	An object representing another object
8	Private Class Data	Restricts accessor/mutator access

Behavioral Patterns are mainly concerned with the management of communications between different objects in a software design. Table 3 behavioural Design Patterns [9] lists the GoF Behavioral Design Patterns.

Table 3 behavioural Design Patterns [9]

No.	Design Pattern	Description
1	Chain of responsibility	A way of passing a request between a chain of objects
2	Command	Encapsulate a command request as an object
3	Interpreter	A way to include language elements in a program
4	Iterator	Sequentially access the elements of a collection
5	Mediator	Defines simplified communication between classes
6	Memento	Capture and restore an object's internal state
7	Null Object	Designed to act as a default value of an object
8	Observer	A way of notifying change to a number of classes
9	State	Alter an object's behavior when its state changes
10	Strategy	Encapsulates an algorithm inside a class
11	Template method	Defer the exact steps of an algorithm to a subclass
12	Visitor	Defines a new operation to a class without change

2.2 UML: Unified Modeling Language

In the last decades, Object Oriented concept have gained immense popularity and strength over the other programming languages concepts and paradigms. One of the main reasons behind this acceptance is that the object orient paradigm gives the priority to the component and modeling aspects and concepts. This issue is very important from the problem domain’s perspective. “With this huge growing and popularity of the Object-Oriented concepts and paradigm the needs were raised to provide a general standard for Object-Oriented Analysis and Design, the Object Management Group (OMG) adopted UML as a standard language for the design and analysis of Object-Oriented Programs. UML is a graphical language that provides notations and action semantics to describe and design software systems" OMG group [1].

UML is a graphical language type that used to provide a way to describe a software systems design with graphical notations and semantic procedures and actions. UML is a result of combining different graphical models that were proposed by Ivar Jacobson [10, 11], Rumbaugh et al. [12] and Booch [13]. UML is also used in many software tools to simulate software [14].

Since OMG group had set the UML as an open standard in 1997, it has been a continued improvement to keep UML up with the new criticisms [15]. These improvements made UML more expressive, accurate and precise model graphical language. In the most recent UML specification, UML 2.4, UML describes the software under design by 14 formal diagrams. A UML view is a group of diagrams that used to illustrate and explain the software system similar characteristics. The original UML taxonomy classifies it software design diagrams into structural and behavioral views. There have been many tries to classify UML views such as the four + one classification view which proposed by Kruchten [16], and the three views classification (Functional, Structural, and Behavioral) which was proposed by Iivari [17]. The typical classification of the UML diagrams mainly classified into three different views: Functional, Structural, and Behavioral views, as illustrated in Figure 1: Hierarchical Classification of UML Diagrams [1]Figure 1.

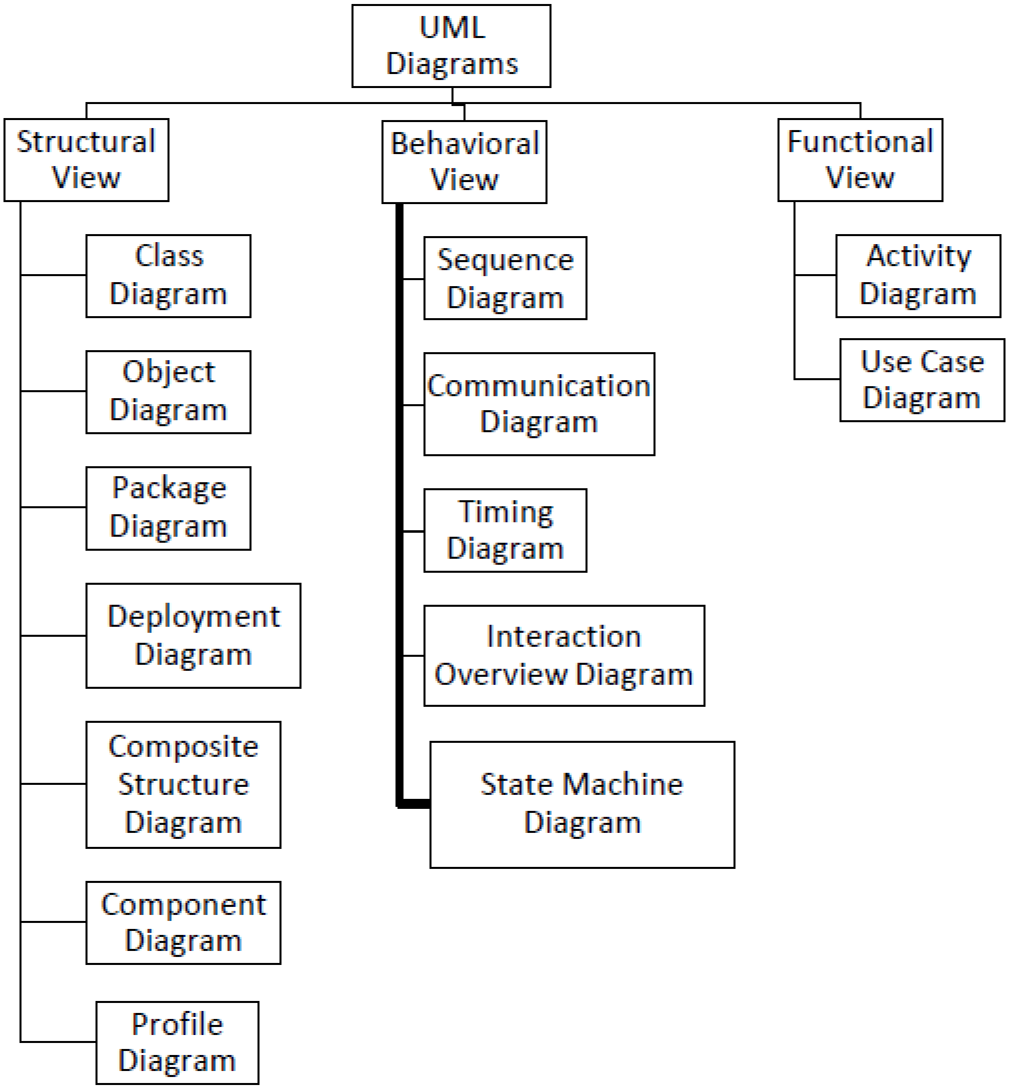


Figure 1: Hierarchical Classification of UML Diagrams [1]

2.2.1 Structural View

The structural view is the most important view in the UML views. Since objects and classes are the basic building blocks in any software system design. The Structural view is concerned about provides diagrams that would help in capturing the physical organization of the software building blocks. The structural view provides a description of the static structure and organization of any software system. The class diagram is the most prominent and widely used diagram in structural view, which will be considered as a representative diagram of the structural view in our proposed thesis.

2.2.2 Behavioral View

Behavioral view diagrams are concerned with showing the dynamic behavior of all the structural classes and objects in the software system. The behavioral view used to illustrate and show the list of series changes that made to any software system over the time. A sequence diagram is a mostly used diagram to describe the behavioral view of any software system, which will be considered as a representative diagram of the behavioral view in our proposed thesis.

2.2.3 Functional View

The functional view is supported by different diagrams, these diagrams provide a picture of how a software system is supposed to act. The functional view grasps the software system information from the user's point of view. Use case diagram is the most vital diagram in functional view which provides a way of modeling the software system's functional requirements, which will be considered as a representative diagram of the functional view in our proposed thesis.

2.2.4 UML Class Diagram

The structural view is represented by the class diagram in an object-oriented software system. It consists of a set of classes that are used to represent and design all the important entities of the modeled software system. As well as, class diagram contains many ways to represent the different relationships between these classes. In the modeling object-oriented software systems, the class diagram considered as the most important and commonly used diagram [1].

Each class contains set of objects that share the same attributes, methods (operations) and the associations between class objects. Attributes are defined as unique features of any class, while methods (operation) are defined as the means of how a class can show up its own functionality to other class of the modeled software system.

The class diagram in UML is represented as a rectangular box that contains three different sections and partitions. The top section of the box contains the name of the class, the middle section of the box defines the list of all class attributes, and the bottom section of the box contains and defines the list of all class methods (operations).

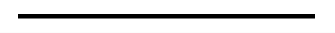
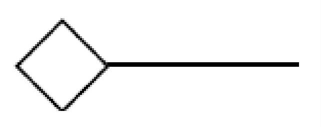
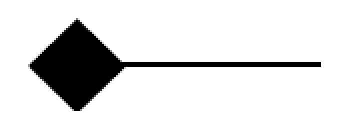
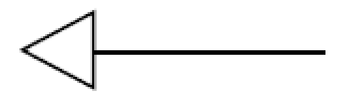
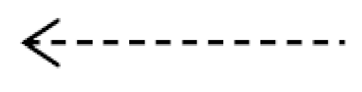
One of the most concepts regarding class diagram called visibility. Visibility defines the way whether class members are allowed to see the corresponding class member of a given class. There are three kinds of visibility is defined in UML, as following:

Table 4 Class Diagram Operation [1]

Notation	**Name**	**Meaning**
+	Public	It allows all the other class objects to access any class member.
-	Private	It allows accessing the class member only from the owner class.
#	Protected	It allows accessing the class member from the class subclasses only.

As we mentioned earlier, all classes in the UML class diagram are communicated to each other using different types of relationships. UML class diagram defines the relationships into four different categories: Association, Generalization, Dependency, and Composition. The description of the UML class diagram relationships is as follows:

Table 5 Hierarchical Classification of UML Diagrams [1]

Notation	Name	Meaning
	Association	Association is used to connect two or more classes to each other.
	Aggregation	Aggregation represents a specific type of association, as a class is a part of or has a relationship. E.g. a doctor "has a "patient to take care of.
	Composition	The composition has the same meaning of aggregation but it stronger that aggregation in which the lifetime of the part class is dependent on the lifetime of the whole class.
	Generalization	Generalization is used to establish the relationship between the parent class and its subclasses, it is also known as inheritance. Where the subclass inherits the common functionalities that are defined in the superclass.
	Dependency	Dependency is the most complicated relationship type of class diagram, where it concerned about establishing a semantic connection between two classes.

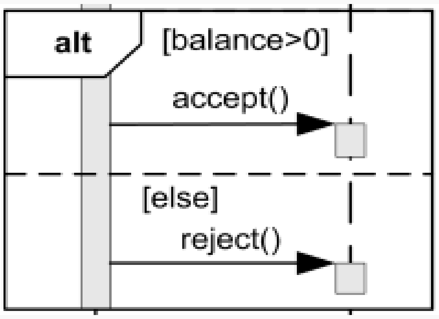
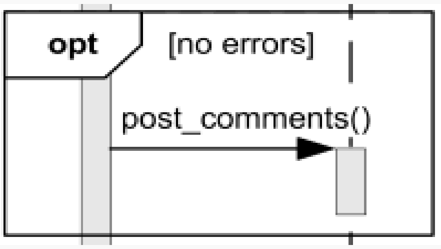
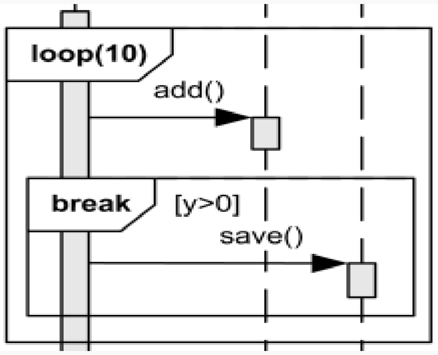
2.2.5 UML Sequence Diagram

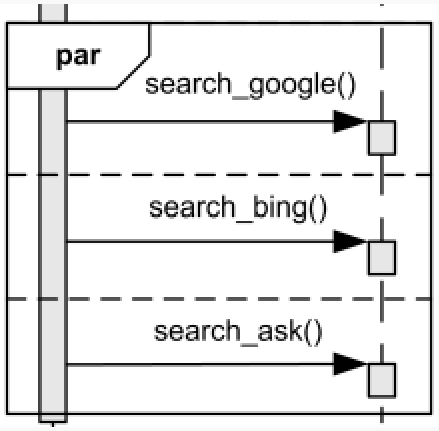
Dynamic view is represented by sequence diagram in an object-oriented software system. The sequence diagram main purpose is to capture the dynamic behavior of a software system. The sequence diagram shows how the class objects interact with each other for a specific scenario for a particular use case. The class objects are interacting with each other and share vital information through using messages, as well as the order of the occurrence of the objects. Sequence diagram conveyed this vital information using two dimensions: virtual and horizontal. Moving from left to right in the vertical dimensions in sequence diagram identifies and shows the objects through which messages are exchanged while moving from the top to the bottom in the horizontal dimension provides us with time sequence and lifetime of these messages. Objects in sequence diagram are represented by a dotted line lies along with virtual dimension, and this line will be extended as long as the interaction exists. Messages in sequence diagram are represented with arrows that move from source object to the destination object. Each message in sequence diagram has two different events: a send event which occurred at the end of the sender, and a receive event which occurs at the end of the receiver [1].

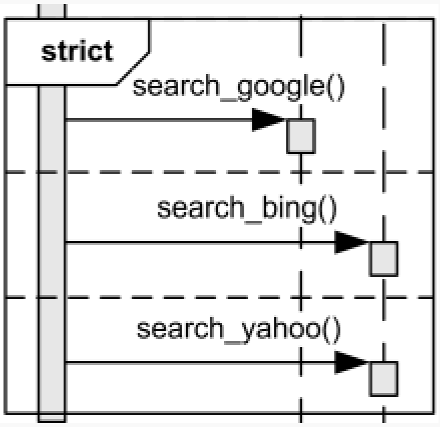
One important notation feature of UML sequence diagram is the Combined Fragment. Combined fragment provides the conditional flow in the series of sequences in the sequence diagram. The combined fragment consists of two elements: an operand and a guard. The operand could be defined as a sub-sequence diagram which constitutes the body of the combined fragment. One or more operand can belong to a combined fragment depends on its type. Every operand is associated with a guard, a guard is a Boolean condition that should be evaluated either to "true" or "false", if "true" then it will execute the sequence, otherwise it will not. The main concerned of a guard is about execution the sequence of the operand it belongs to [1].

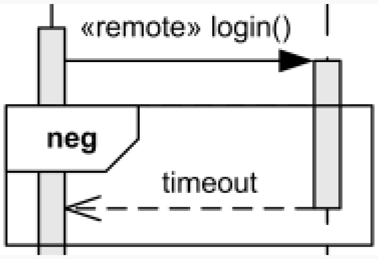
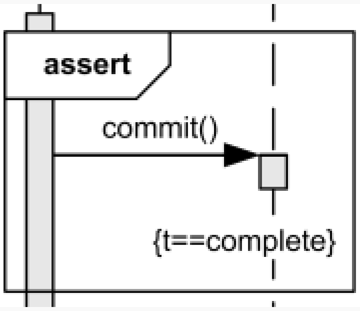
Twelve kinds of combined fragments identified in UML specification as following [18]:

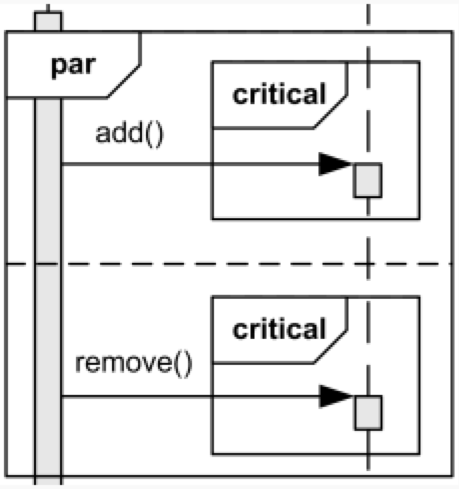
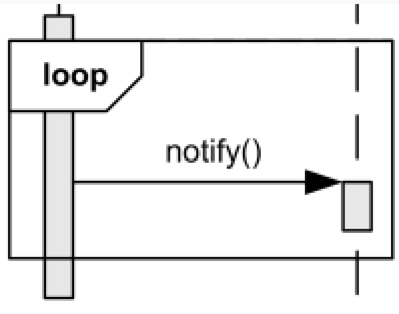
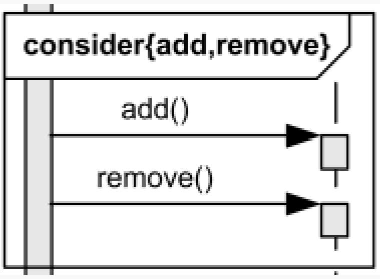
Table 6 Sequence Diagram Fragments [17]

Notation	Fragment Name	Description
	Alt (alternatives)	The alternative combined fragment is used to choose between two different behaviors. It has multiple operands, at most one of the operands should be chosen. The alternative combined fragment can be realized in programming logic by "*if – else* ".
	Opt (optional)	The optionally combined fragment is used to specify a behavior that might or might not occur. The optionally combined fragment can be realized in programming logic by "*if – else*".
	Break	Break combined fragment is used to break up a behavioral scenario. It consists of a single operand which executes when the guard is true. It is mostly used to handle exception behaviors.

	Par (parallel)	The parallel combined fragment is used to define the parallel execution of multiple operands at the same time without exposing the integrity of the results.

 strict
search_google()
search_bing()
search_yahoo()
	Seq (Sequencing)	Sequencing combined fragment is used to settle the order of execution between multiple operands. The message is enforced to execute within an operand before the next operand starts.

	Neg (negative)	The negatively combined fragment is used to identify an invalid behavior. It has only one guarded operand that represents an invalid sequence, while the other behavioral sequences are positive.
	Assert	Assert combined fragment is used to identify the assertion sequence of a behavior while the other sequences are not valid.

	Critical	The critical combined fragment is used to identify the group of sequence messages as critical.
	Loop	Loop combined fragment is used to identify a repletion of a sequence. It has a single operand that contains the number of times of reparations depends on the maximum and minimum iteration number in the loop guarded operator.
	Consider / Ignore	Consider combined fragment is used to identify the messages that should be considered with a sequence of a combined fragment while ignoring combined fragment defines these messages that should be ignored.

The following diagram shows all UML sequence diagram graphical notations:

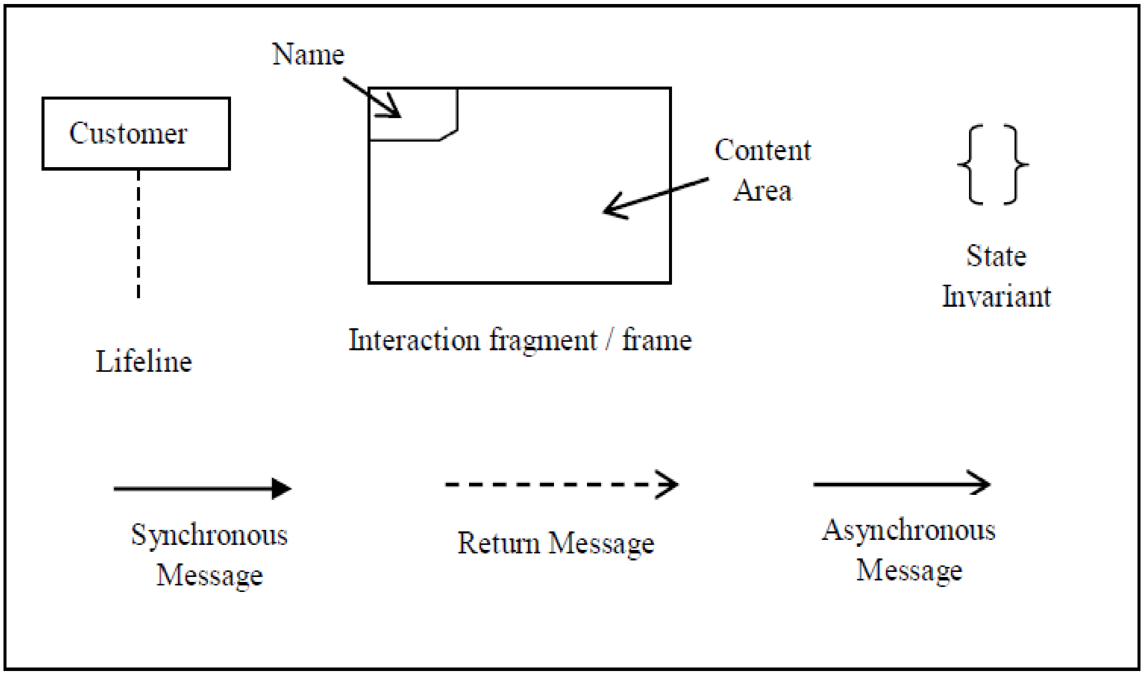


Figure 2: UML Sequence Diagram Graphical notations [17]

2.2.6 UML Use Case Diagram

The concept of use case diagrams is initially introduced by Jacobson et al. [10], later on, the OMG group adopted the use case concept and conceded it as a part UML (Unified Modeling Language). The main purpose of use case diagram is to represent the object-oriented system's functional view. Use case diagram is playing a critical role in the collection and modeling the software system requirements. Requirement engineers describe and represent the requirements of a system using a set of use cases in the UML use case diagram. The specification of the system is represented by use cases that operate and interacts with the actors in order come up with a result that is valuable to both the stakeholders and the actors of the system.

Mainly use case contains four different items that show how the system works [10]:

The first element is the software system itself, the uses cases that describe the system's service that should be performed, and the actors who contribute to the software system, and the different relationships between the system's elements.

We can classify the relationships in the UML use case diagram into three main categories as follows:

The actor to Use Case Relationship.

The Actor to another Actor Relationship.

The Use Case to another Use Case Relationship.

There are also three relationships between use cases in UML:

Generalization: In uses case generalization relationship when one use case functionality is separated to different use cases. Generalization is similar to the concept of inheritance where a use case inherits other's use case functionality and add it to their own specific functionality [10].

Inclusion: In use case inclusion two use cases are related to each other by one of the use cases is offered by the other one. The use case which included the other use case is not a complete use case in its own functionality, and that supports the idea of reusability since one use case can use other use case functionality without repeating [10].

Extension: In use case extension relationship when one of the use cases wants to employ and profit the other use case functionality. The disparity of the inclusion use case relation, the use case that extends the other use case functionality is complete by its own functionality.

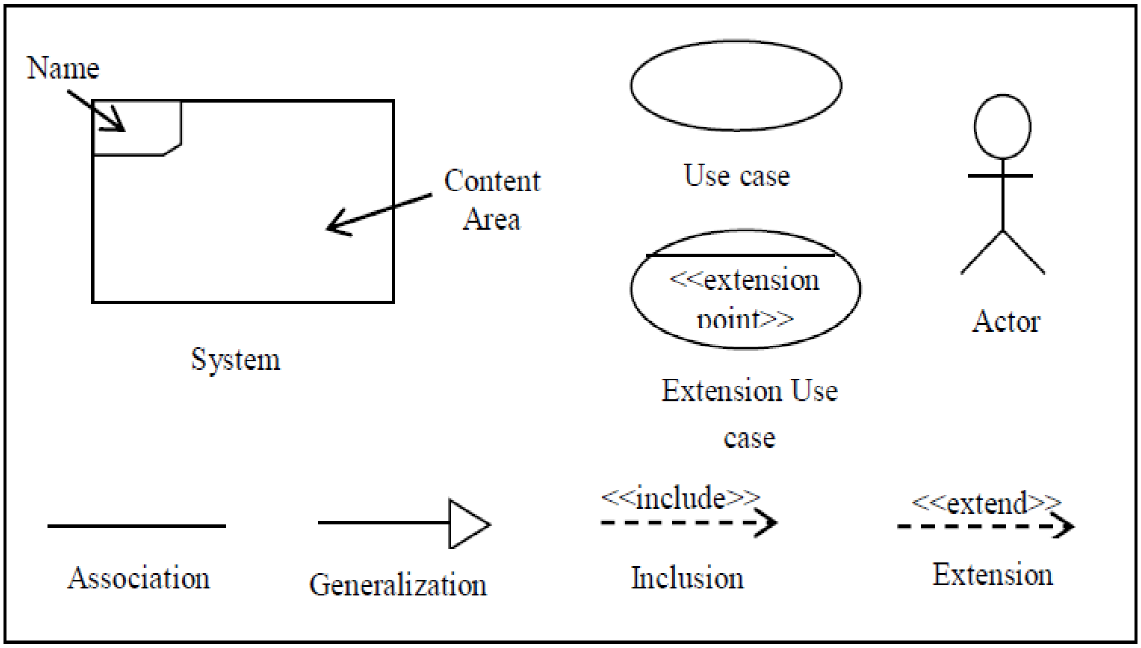
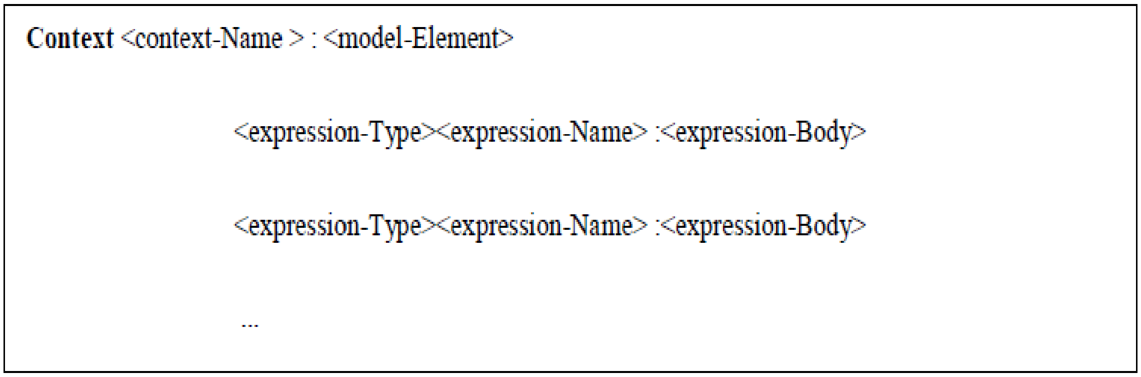


Figure 3: UML Use case diagram Graphical notations [17]

2.3 UML Meta-Model

The UML notation provides designers with information in a graphical way. OMG group proposes a declarative specification language (OCL) [1] to express and present the software properties that can't be represented in a graphical way such as invariants and constraints. The OCL metamodel language represents and traces the software design elements and its constraints using conditions, post-condition, and invariants. The UML metamodel provides the designer with a well-formedness and preciseness of the software design models. It also assists in the validation and certification processes. The following figure outlines OCL metamodel language specifications.



```
Context <context-Name > : <model-Element>

                  <expression-Type><expression-Name> :<expression-Body>

                  <expression-Type><expression-Name> :<expression-Body>

                  ...
```

Figure 4: Outline of an OCL Constraint Specification [1]

2.4 UML Integrated Meta-Model

In order to model a complex software design, it requires the software designer to pay more attention to system's different aspects and look to it from different views. The system design different views should be considered the structural (static) view (methods and attributes), the behavioral (dynamic) view (invariants, and scenarios), and the functional view (the rights of access, requirements). Since there are core functionalities and aspects are shared by different views for the same metamodel, the need for the integrated metamodel is essential. The definition of the integrated metamodel is to create links between different views, processes or services [19]. The concept of integrated metamodel has been used and applied for a massive number of applications is the domain of Model-driven software engineering [20-23].

The Object Management Group currently defines the UML language using a metamodel.

The three parts of defines the metamodel in the UML specification docmunent are:

1. **Abstract Syntax**: A class diagram describes the abstract syntax of UML, which is composed of meta-classes and meta-associations. The syntax of UML is well defined and unambiguous.
2. **Well-formedness Rules**: Specification of constraints on instances of the meta-classes (that represent the UML language constructs) is through a set of well-formedness rules.
3. **Semantics**: Semantics describe the meanings of the meta-classes introduced in the abstract syntax.

Misbhauddin and Alshayeb[3, 24] proposed an integrated metamodel composed of three main metamodel diagrams class diagram metamodels, sequence diagram metamodels, and use case diagram metamodels see figure 5.

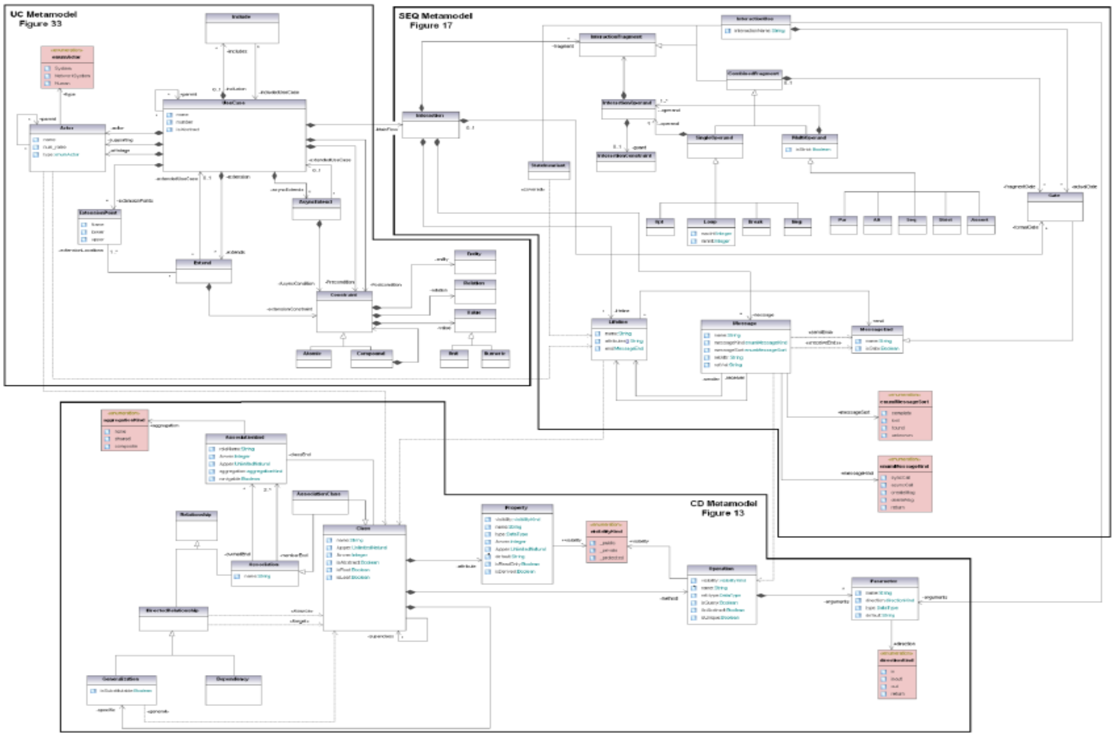


Figure 5: (Intermediate) Metamodel [3, 24]

The integrated metamodel proposed by Misbhauddin and Alshayeb [3, 19, 24] provides a multi-view integrated approach to model-driven refactoring using UML models. They selected a single model from each UML view at metamodel level to construct an integrated metamodel. They selected class diagram to represent the structural view, sequence diagram to represent the behavioral view and use case diagram to represent the functional view. Misbhauddin and Alshayeb [3, 19, 24] in their proposed integrated metamodel used the class diagram to represent the structural view without any extension, however, they extended metamodel of the sequence diagram to represent the behavioral view and the extended metamodel of the use case diagram to represent the functional view. To ensure complete modeling of information, the integrated metamodel also incorporates the OCL metamodel so that constraints (from class diagrams), invariants and guards (from sequence diagrams) and pre- and post-conditions (from use case diagrams) are structurally represented.

They validated the proposed approach by comparing integrated refactoring approach with refactoring applied to models individually in terms of quality improvement through UML model metrics.

The main objective of the integrated metamodel proposed by Misbhauddin and Alshayeb [3, 19, 24] is to identify refactoring opportunities within the software design using model information from multiple views. They identified a total of seven refactoring opportunities that can be detected from the integrated model.

In our design pattern detection technique, we going to use the integrated metamodel that proposed by Misbhauddin and Alshayeb [3, 19, 24].

CHAPTER 3

LITERATURE REVIEW

In this chapter, we are going to discuss two types of literature reviews, the first is the traditional literature review and the second is the systematic literature review.

The traditional literature review highlights the previous different design pattern detection techniques. The revision had been conducted without analyzing or investigation the results. In the other way around, we had conducted a systematic literature review for almost 7,403 papers to investigate and discuss the state of art of design pattern detection techniques, as well as spotting the future gaps in the existing work.

3.1 Related Work

This section highlights the attempts for design patterns detection techniques proposed in the past. Different design pattern detection techniques have been proposed in the literature review, some are based on Database queries, metrics, matrices, graph based and constraint satisfaction problem based.

3.1.1 Database Queries

Different techniques and methods use database queries in order to detect the occurrence of Design Patterns [23-27]. Keller et al. [23] proposed a tool called SPOOL to detect and identify design patterns. The purpose of this tool is to confirm that the pattern-based reverse engineering is a very valuable approach for software understandability and comprehension. The tool was applied to C++ case studies. SPOOL tool first converts the C++ source code into UML meta-model, then it applies a query mechanism to the UML meta-model that can recognize and detect design patterns in three different modes: automatic design recovery, manual design recovery, and semi-automatic design pattern recovery. Lee et al. [24] proposed a design pattern detection algorithm to detect the 23 Gang of Four (GoF) design patterns. The purpose of this algorithm is to reduce the maintenance costs in reverse-engineering by reusing the design patterns to solve the commonly occurred problem in software design. Their proposed algorithm was applied to different well-known open source systems. Their technique first converts the source code into AST (Abstract Syntax Tree), or ASG (Abstract Syntax Graph), and then to XMI. Second, a query mechanism applied to XMI to detect the design patterns. Rasool et al. [25] proposed a design pattern detection technique based on database queries, regular expressions, and annotations. Their annotations are added to the source code to profile more understandability. Their annotation can be used by both the human and the configurable machine. The annotation is directly applied on the source code instead of converting it to any intermediate representation. Their approach examines the source code annotations with the specific design pattern annotation using database queries and regular expressions. Stencel and Wegrzynowicz [26] proposed a new method to detect the occurrence of design patterns automatically. The purpose of their proposed method is to increase the level of understandability and to help in the reverse engineering process. Their method can detect the standard implementation of design patterns as well as the non-standard implementations. They provided the proof-of-concept tool for their proposed method. The tool was based on three main phases. The first phase is parsing, the tool converts the Java source code into an Abstract Syntax Tree (AST) and then it builds the main core parts of the software system based on AST. The second phase is analyzing, the tool computes the transitive closures of relationships between the core elements, and then it stores the core parts with their relations in a rational database. The third phase is detecting, the tool executes SQL queries on the rational databases to detect the occurrence of the design patterns. Marek Vokác [27] developed a tool to detect the occurrence of design patterns. The tool is based on C++ open source systems to detect only five design patterns with a high precision and speed. The tool goes through three different stages. The first stage is to convert the C++ source code into UML metadata. The second stage is to link between the entities and references of metadata then store it in an SQL database. The third stage is to perform SQL queries to detect the occurrence of design patterns.

These approaches and techniques first transform the source code into some intermediate representations such as XMI, AST, ASG, UML structures, and metadata etc. Then they use SQL queries as a next step to detect and extract patterns that have related information from particular representations. The main advantage of using database queries is the performance of the queries to extract and detect related features and information of design patterns can be directly bound to the database in use and can be scaled very well, but such method of using queries has some disadvantages. The major disadvantage is that they are limited to the information which is available in the intermediate representations. One of the main limitations of intermediate languages is that they cannot represent the non-functional requirements of the system. Based on the literature review, there is no currently available intermediate representation format which could be used to store all the information and features presented in the source code. Another disadvantage of these SQL queries based approaches is that they are restrictive to structural and creational design patterns so far and they do not fully support behavioral design pattern recovery.

Wang and Tzerpos [28] proposed an REQL query design pattern detection technique called Design Pattern Verification toolKit (DPVK). REQL is a query language (RethinkDB query language) used to manipulate JSON files. The technique fist creates a repository database to store all the variants of design patterns. This technique is done in two main stages. The first stage is to store and describe all the classes and methods in the system using some java parser tools. The second stage is to compare the candidate design pattern instances with the file description in stage 1.

3.1.2 Metrics

Metrics were used by different design pattern detection techniques. These techniques calculate the software related metrics like (associations, generalizations, interface hierarchies, aggregations etc.) from the different source code representations and then the tool will use other techniques to validate and compare the design pattern definition metric values with source code metrics. Paakki et al. [29] proposed a metric tool to detect the design pattern occurrence. The tool is based on constraint satisfaction problem (CSP). CSP has been applied to many different areas such as program understanding, machine vision, and scheduling. The main idea behind the CSP algorithm is to formulate a large number of the central problems as a single set of constraints (predicates) in a particular domain over variables. The tool first converts the source code into the UML intermediate representation then it applies the proposed metric to detect the occurrence of the design patterns. Another metric based tool is called Fujaba, Fujaba Tool Suite [FUJABA] [17] is an Eclipse Plug-In for detecting design patterns in source code. The tool suite is based on two parts, the structural part, and the behavioral part. UML intermediate representation represents the structural part of the tool, while the Story Driven Modeling (SDM) represents the behavioral part. SDM allows the user to identify the complete part of method bodies in the activity diagram in UML environment. In the first phase, the tool specifies the design pattern by specifying it in two main parts the structural part and the behavioral part. The second phase is to compare the specified design pattern with an actual design of the software system. Antonio et al. [30] proposed a tool based on java to automatically detect the occurrence of design patterns. The tool first maps the source code into Abstract Syntax Tree (AST) intermediate representation. Then it parses the AST tree using Abstract Object Language (AOL) to generate the structure properties and components of software systems. A set of matrices then applied to AOL to detect the instances of design patterns. Lucia et al. [3] proposed a new design pattern detection technique based on structural and behavioral analysis techniques. This technique detects only behavioral design patterns. The first phase is to extract the information of class relationships and method with their calls from the source code system. Then a source code analyzer is used to check if the identified design pattern instances are confirmed to the predefined design pattern instances. The second phase is to capture the behavioral of the classes and methods that were collected from the static phase, and if the identified design pattern description matches the actual one then the tool claims that it caught the design pattern.

The main advantage of metric based techniques is that they are computationally efficient. However, the metric-based approaches have many drawbacks, the first one is the experiments were performed on very few sets of patterns, so the generalization cannot be made towards all the types of the GoF patterns. Furthermore, these techniques are not considered to be interactive. These techniques reported a low precision and recall as well.

3.1.3 Matrices

This technique uses the matrices to represent the structural and behavioral information of software systems. These approaches build corresponding matrices for each system to store all the classes and their interrelations. They applied different techniques and algorithms to examine and match the predefined design pattern templates with the matrices generated by the software system.

Tsantalis et al. [4] proposed a technique based on graph similarity algorithm. Their approach uses similarity scoring mechanism. This technique gives scores to each node in the similarity graph depending on the structural information that was collected from the class diagram (association, aggregation, abstract method invocation, and etc). This approach uses a set of predefined matrices that represents all of the static structure of design patterns of interest. Dong et al. [31] introduced a new pattern detection technique. The technique is based on weights and matrices. This approach has three main analyses models, structural, behavioral, and semantic modes. The Structural analysis main job is to parse the XMI files that were generated from the UML diagram, and then build up a square matrix that stores each class as rows and columns. Each row represents a class and each corresponding column represents another corresponding class. Each cell in this matrix stores the relationship between the both classes in the row and in the column. The behavioral analysis main concern is to check if the interested method invocation really exists in the class within its right signature. Finally, the semantic analysis here is to distinguish between the similar designs patterns such as Bridge and Strategy.

These techniques and approaches have many different advantages since they are computationally efficient, and they have very good precision and recall rates, however, these techniques are not well interactive as they are not able to distinguish and extract the different implementation variants of the similar design patterns. In addition, matrix-based techniques and approaches are limited and restrictive to only a few number of design patterns and they cannot recover the whole complete set of the GoF design patterns which does not make it a reliable technique.

3.1.4 Graph Based

This section presents the other techniques proposed in the literature.

PTIDEJ team [32] developed a tool suite called Pride, this tool was used as a reverse engineering framework to detect and identify macro-patterns, idioms, design defects and design patterns using a meta-model called PADL (Pattern and Abstract-Level Description Language). PADL meta-model represents the software at different abstraction levels. PADL meta-model provides components like Relationships, Methods, Classes, and Model. Therefore, by using PADL they can build a whole representation for the software. This tool detects design pattern by representing the relationships among roles as constraints among variables. PTIDEJ group focuses on gaining and ensuring a 100% recall rate, but they sacrificed the precision and the detection performance is very low.

Pierre et al. [33] proposed a fuzzy weight based technique to detect design patterns. The patterns of interest are identified by graph transformation rules. A graph transformation rule is a UML-alike collaboration diagram. These graph transformation rules are defined as collaborative diagrams to detect the annotation of abstract syntax graph

(ASG) patterns, and each rule is given a specific weight. Abstract syntax graph (ASG) is generated using JavaCC source code parser (JCC).

Shi and Olsson [34] proposed a data flow and control flow based technique to detect the occurrence of design patterns using ASTs. Their proposed technique uses data-flow analysis to analyze the entire AST (Abstract Syntax Tree) of a specific method body. The tools build a group of related methods as building blocks using a control-flow graph (CFG). Then the technique compares the behavioral of each building block with specific behavioral of the desirable design pattern.

Beyer and Noack [35] proposed a new design pattern detection technique based on directed graph. This technique uses the (BDD) data structure binary decision diagram to represent the relationships between classes and hierarchical representations between them. The tool first converts the desired system into directed graph then it applies a binary relationship between the implemented design patterns and the system directed graph in order to detect the design patterns.

Heuzeroth et al. [36] proposed a new design pattern detection technique based on the static and dynamic analysis. The static analysis converts the source code into AST (Abstract Syntax Tree) to collect the static information of the system (classes, methods, and relationships). The dynamic analysis of this technique takes the static information set as an input. Then it executes the nodes of the AST (Abstract Syntax Tree) to monitor the behavioral of the system. Then it tracks the execution of the system to check if the candidate design pattern satisfies the rules that identified at the dynamic analysis. This technique can’t detect the design if it did not execute at the dynamic phase.

Balanyi and Ferenc [37] proposed a new design pattern detection technique based on XML matching. The matching is done in two main stages. In the first stage, the technique analyzes and converts the source code into ASG (Abstract Semantic Graph). In the second stage, DPML (Design Pattern Mark-Up Language), is used to define the description of the design patterns. The selected DPML design pattern description is taken as an input into XML DOM file. Then the proposed technique checks and matches the ASG tree with the DMPL design pattern description.

Wang and Tzerpos [28] proposed an REQL query design pattern detection technique called Design Pattern Verification toolKit (DPVK). REQL is a query language (RethinkDB query language) used to manipulate JSON files. The technique fist creates a repository database to store all the variants of design patterns. This technique is done in two main stages. The first stage is to store and describe all the classes and methods in the system using some java parser tools. The second stage is to compare the candidate design pattern instances with the file description in stage 1.

Ferenc et al. [38] proposed a machine learning based design pattern detection technique. The machine learning technique is used to improve the results of the detection method by using predictors. Predictors are the metrics related to each design pattern that can be used to detect the pattern instance. This technique has two main stages. The first stage is to convert the source code into Abstract Semantic Graph (ASG) using Columbus framework. The design pattern descriptions are stored in the DPML (Design Pattern Markup Language). Applying DPML design pattern description to the ASG graph to detect the design pattern instance is the second stage.

Huang et al. [39] introduced a new pattern detection technique based on runtime behavioral capturing. They proposed a prolog tool to represent the design pattern descriptions called Hrycej. The idea is similar to the previously mentioned approach. The technique first captures all the structural description of the system by converting the source code to the UML intermediate representation. Then it captures the behavioral parts at runtime. The algorithm then checks the design pattern description stored in Hrycej with the structural and behavioral information.

Arcelli and Cristina [40] proposed a data mining based design pattern detection technique. This technique uses Weka data mining environment to increase the correctness of the detection method. This technique first divides the source components into a set of subcomponents to make it easier to deal with. The next step is to collect all the design pattern structure components using their proposed tool MARPLE (Metrics and Architecture Reconstruction Plug-in for Eclipse). The false positives of the results are improved using data mining technique (Weka data, and neural network). Table 7 summarizes the current design pattern techniques.

3.1.5 CSP Based

Guyomarc'h and Sahraoui [41] proposed a numerical signature based technique to detect the occurrence of design patterns. The technique uses the constraint satisfaction problem (CSP) algorithm to identify the source code classes and the relationships between these classes. The numerical signatures are used to reduce the research latency of detection. The idea behind the numerical signatures is to give each class, playing a role in the design motif, a specific number and then remove every class that is not playing any role in design motifs.

Table 7 Comparison of design pattern detection techniques

Paper No.	**Detection Technique**	**No. Of Views**	**Applied on**
Keller et al. 1999 [25]	Data Base Queries	1-View	Class Diagram and Source Code
Lee et al. 2007 [8]	Data Base Queries	1-View	Class Diagram and Source Code
Rasool et al. 2010 [26]	Data Base Queries	1-View	Class Diagram and Source Code
Stencel and Wegrzynowicz. 2008 [27]	Data Base Queries	1-View	Class Diagram and Source Code
Marek Vokác.[28]	Data Base Queries	1-View	Class Diagram and Source Code
Paakki et al. 2000 [29]	Metrics	1-View	UML
UJABA 1997 [30]	Metrics and matrices	1-View	SDM and UML
Antoniol et al. 1998 [31]	Metrics	1-View	AST and AOL
Lucia et al. 2009 [4]	Metrics	1-View	Source Code
Tsantalis et al. 2006 [32]	Matrices (scoring mechanism)	2-View	UML
Dong et al. 2009 [6]	Matrices	1-View	XMI
PTIDEJ 2003 [33]	PADL	1-View	Source Code
Niere et al.2002 [34]	Fuzzy Reasoning	1-View	ASG and graph transformation rule
Guyomarc'h and Sahraoui. 2009 [34]	Numerical Signature	1-View	CSP
Shi and Olsson. 2006 [35]	Data flow and control flow	1-View	CFG and AST
Beyer and Noack. 2007 [36]	Directed Graph	1-View	BDD

Heuzeroth et al. 2003 [37]	Static and Dynamic Analysis	1-View	AST
Balanyi and Ferenc. 2003 [38]	XML matching	2-View	ASG and DPML
Wang and Tzerpos. 2005 [39]	REQL query	1-View	Source Code
Ferenc et al. 2005 [40]	Machine Learning	1-View	ASG and DPML
Huang et al. 2005 [41]	Runtime Behavioral Capturing	1-View	UML
Arcelli and Cristina 2007 [42]	Data Mining	1-View	Source Code
The proposed approach	**XML Integrated Metamodel**	**3- Views**	**UML**

We noticed from Table **7** that most previous studies are based on only one single view. Only a few number of studies have conducted their proposed approach in two views. There has been no study used the main three views (Functional, Behavioral, and Structural). Furthermore, there has been no research study used any integration model to detect design patterns. We can also notice that most of these methods focused on detecting design patterns from UML and source code.

3.2 Systematic Literature Review

We have conducted a systematic literature review is to state the art of the existing design pattern techniques in the last two decades as well as providing a classifications and categorization review of these techniques to the design pattern detection research area. In order to meet our goal, we conducted a systematic literature review to cover all the primary studies conducted in design pattern recovery area. Based on our research question provided different customized research terms to determine on literature on design patterns detection techniques.

In this systematic literature review, we categorized and classified the design patterns detection techniques based on the representation model used to build up the software design model as well as the detection techniques used to detect the design patterns. Furthermore, we provided a review of the views used in the detection processes. The automation of the techniques also considered and reviewed.

The first attempt to review the different design pattern techniques in a comprehensive way was made by Rasool, and Streit [43]. The authors studied the different design pattern techniques and provided a detailed observation lessons as a benchmark and guidelines and directions for this discipline.

Another review had been conducted by Kamatchi Priya [44] in a comparative analysis. The author provided a detailed list of different design pattern detection approaches that show the different aspects of each technique.

To the best of our knowledge, this is the first systematic literature review in reviewing the design pattern detection approaches. A systematic review process requires precise definition and documentation of the whole process. So for the sake of space, in this section, we are going limit the discussion for the only research question, research string, and the discussion of the results.

3.2.1 Research Question

The objective of this review is to analyze and use the results of the survey to answer the research question (RQ) discussed below. This research question is subdivided into five different sub-questions. The rationale about why these specific sub-questions were selected for the review is included for each research sub-question.

(RQ) What are the available design pattern detection techniques in software development lifecycle?

The sub-questions of the research question are listed as follows:

(RQ).1 what are the different software design model representations used in DPDT?

Rationale: Software design model representation is the vital rule in DPDT since each approach used a different type of representation. The main motivation behind this research sub-question is identified the different representation approaches that have been used in in order to detect the design patterns.

(**RQ).2** what are the different software design patterns detection algorithms and methods used in DPDT?

Rationale: In order to detect the design patterns from an existing software design, different algorithms have been applied to search and parse the design pattern features. Each technique built their own algorithm to detect design patterns either at the code level or design level. The main aim of this research sub-question is to identify what are the different algorithms and methods used in the literature to detect the design patterns.

(RQ).3 what are the different software design views used in the DPDT?

Rationale: A software model is built up from many different views, a view is a collection of models that illustrate similar characteristics of the system. There are three main views for every software design model (Functional, Structural, and Behavioral) views. The main goal of this research sub-question is to identify what are the different model views are used in every DPDT to detect the design patterns.

(RQ).4 is the DPDT seamlessly integrated with the existing software CASE tool or providing prototype tools to simplify the DPDT?

Rationale: DPDT automation is one of the most important issues related to this discipline. The automation plays a vital rule in software development lifecycle. The main motivation of this research sub-question is to show which of the proposed DPDT was implemented to be adapted to existing CASE tools or which of it used a prototype tool to assist the detection process.

(RQ).5 which design pattern categories have been used in the DPDT?

Rationale: There are different types of design patterns proposed in the literature such as GoF and Web Design patterns. The main motivation of this research sub-question is to identify which design patterns categories are used in a specific DPDT proposed in the literature.

3.2.2 Data Sources and Search Strategy

The objective of this review is to analyze and use the results of the survey to answer the research question (RQ) discussed below. This research question is subdivided into five different sub-questions. The rationale about why these specific sub-questions were selected for the review is included for each research sub-question.

Research articles published in literature related to the field of design pattern detection were extracted from pertinent scientific databases and considered for review. Scientific databases considered in this review process include:

1. IEEE Explore (http://ieeexplore.ieee.org)

2. ACM Digital Library (http://dl.acm.org/)

3. Science Direct (http://www.sciencedirect.com/)

4. Springer Link (http://link.springer.com/)

5. John Wiley Online Library (http:// http://onlinelibrary.wiley.com/)

Depending on the search Database we can use Boolean operator 'AND' for concatenation of the major term and Boolean operator 'OR' for the concatenation of alternative spellings and synonyms. The search strings for the specific electronic Databases are given below with the screenshots present in Appendix A respectively for each database:

RQ1)

((" Design pattern* " AND (Detect* OR Recovery OR Identif* OR Min* OR Recognition OR Discovering OR Revealing OR Retrieval OR Searching OR Research OR Extraction OR Miner)))

IEEExplore- 561 Proper Results returned

RQ1)

"Design Pattern" in All Fields AND "Detection " OR " Recovery " OR " Identification " OR " Mining " OR " Recognition " OR " Discovering " OR " Revealing " OR " Retrieval " OR " Extraction " OR " Representation " OR " Identifying " OR " Detecting " OR " Miner " OR " Detector " OR " Discover" in All Fields between years 1998 and 2016

John Wiley Online Library- 2037 Proper Results returned

RQ1)

("Design pattern") and ("Detection " OR " Recovery " OR " Identification " OR " Mining " OR " Recognition " OR " Discovering " OR " Revealing " OR " Retrieval " OR " Extraction " OR " Identifying " OR " Detecting " OR " Miner " OR " Detector " OR " Discover")[All Sources(Computer Science)].

Science Direct- 2,186 proper results returned

RQ1)

'("Design Pattern") and ("Detection " OR " Recovery " OR " Identification " OR " Mining " OR " Recognition " OR " Discovering " OR " Revealing " OR " Retrieval " OR " Extraction " OR " Identifying " OR " Detecting " OR " Miner " OR " Detector " OR " Discovering ") ' within Computer Science English Article

Springer Link- 1,399 proper results returned

RQ1)

(+"design pattern" Detection Recovery Detector Miner Detecting Identifying Extraction Research Searching Retrieval Revealing Discovering Recognition Mining Identification)

ACM- 1,220 proper results returned

3.2.3 Citation Retrieval and Management

The number of the returned citations in **Stage 1** is 7,403, these citations were maintained and organized using a citation management tool called EndNote [46]. Then the citations were recorded in a spreadsheet using the author, the source of citation, the year of publication and type.

Both authors shared the filtering and classification processes independently; in case if a disagreement raised up, the common decision achieved.

In **Stage 2**, the citations titles of Stage 1 were studied to the scope of the proposed systematic review. We excluded the studies that are not related to the design pattern detection techniques. For instance, surveys and imperial studies were excluded. The vague and unclear citations in this stage were included to be studied in the next stage. In Stage 2 we ended up having 2,262 out of 7,403.

In **Stage 3**, by studying the articles abstracts we excluded the studies that are not related to the design pattern detection techniques such as surveys and empirical studies, the articles that have unclear abstract were included to the next stage (detailed quality assessment) for further investigations. In this stage, we left up with 210 citations. In **Stage 4** (detailed quality assessment) we ended up with 91 articles.

Stage 5 (Snowballing Stage) was performed to cover as many articles as we can, since some articles might be missed by our proposed research string, by looking to the citations and references of the 91 articles of (detailed quality assessment stage) we found other 8 articles that were not found by our research string. The Snowballing stage was made by applying Stage 2 to Stage 4 processes for the final number of the articles in stage 4.

The citations duplication found by the citation management tool were studied by both authors based on the content of the whole article and then removed. Intra-database duplicates were reviewed and the ones secondarily indexed were removed. For instance, an article published in Springer was retrieved from Springer, Scholar, and ACM. Hence, its instance from Scholar and ACM was deleted leaving behind the one in its primary index database.

Table 8 the detailed summary of all the articles and their stage

	IEEE	ACM	Springer	Science Direct	John Wiley	Total number
Stage 1	561	1,220	2,186	1,399	2,037	7,403
Stage 2	300	266	620	442	900	2,262
Stage 3	95	23	47	26	19	210
Stage 4	56	12	16	4	3	91
Stage 5	2	0	0	6	0	8

3.2.4 Results Analysis and Discussion

In this part of the review, the studies are reviewed and analyzed in order to analysis the future gaps and challenges, drawbacks, and challenges based on the main research question and its sub-questions. Table 9 shows the summary of the systematic review, the table summarized each study with the results corresponding to each one.

Table 9 Summary of all design pattern detection techniques

No.	Reference	Journal Name	Initial Representation	Final Representation	Detection Technique	Design Pattern Type	Views	Automation
1	Iacob, Claudia [45]	ACM	-	Manual Text Discerption of DPs	A Query Algorithm and Metrics Based (DoR)	N/A	1- View (Structural)	N/A
2	Arnold and Corporaal [46]	ACM	-	Directed Graph	Graph Matching Algorithm (Incremental Matching)	N/A	1- View (Structural)	Automated
3	Alnusair et al. [47]	ACM	-	Ontology-Based	First-Order Predicate Logic	GoF	1- View (Structural)	Automated
4	Pappalardo and	ACM	UML Class Diagram, Concern and	XML	Graph Matching	N/A	2- Views (Structural and Behavioral	Automated

	Tramontan [48]		Pattern Concern View				'Concern view')	
5	J Dong et al.[49]	ACM	-	Matrix	Template Matching and Similarity Score calculation	Composite, Adapter, State, Decorator	1 - View (Structural)	Automated
6	Dabain, et al[50]	ACM	-	Source Code	Query Based Algorithm	GoF	1 - View (Structural)	Automated
7	Stephan and Cordy[51]	ACM	UML Class Diagram	XML	Model Clone Detection	N\A	1 - View (Class Diagram)	Automated
8	Seemann and Gudenberg [52]	ACM	UML Class Diagram	XML	Graph Matching	Composite, Bridge, and Strategy	1 - View (Class Diagram)	Manual
9	Wendehals and Orso [53]	ACM	UML	Pattern Catalog and Finite Automata	Static Analysis Algorithm	Behavioral Patterns	2- Views (Structural and Behavioral Sequence Diagram and FA')	Automated
10	Sudhakar and Gyani [54]	ACM	-	Intent Aspects(IA's)	Selection Based Algorithm	Prototype, Builder, and Singleton	1 - View (Class Diagram)	Automated
11	Moha and Guéhéneuc [54]	ACM	-	PADL meta-model	Matching Algorithm	GoF	1 - View (Class Diagram)	Automated
12	Bernardi et al. [55]	JW	-	Domain Specific Language (DSL)	Graph Matching	GoF	2- Views (Class and Sequence Diagram "Method Call")	Automated
13	Martino and Esposito [56]	JW	UML	Web Ontology Language (OWL), XML, and Logic Rules	Prolog Facts Search Algorithm	GoF	2- Views (Class and Behavioral interaction diagram)	Automated
14	Ng and Guéhéneuc [57]	JW	UML Scenario Diagrams	Scenario Diagrams	Matching Algorithm and CSP	Behavioral and Creational	2- Views (Class and Sequence Diagram)	Automated
15	D Yu et al. [58]	Science Direct	Class-Relationship Directed Graphs	XML	Graph Matching	GoF	2- Views (Structural and Behavioral Method Signature')	Automated
16	Tekin and Buzluca [59]	Science Direct	UML and Abstract Syntax Trees (AST)	XML	The Subgraph Mining Algorithm	GoF	1- View (Structural)	Semi-Automatic
17	Huang et al. [41]	Science Direct	UML	XML	Prolog Rules and Selection Based Algorithm	GoF	2- Views (Structural and Behavioral At Runtime)	Automated

18	**Fontana and Zanoni [60]**	**Science Direct**	**AST(abstract syntax tree) and UML**	**XML**	**Metrics and Basic Elements Detector (BED)**	**GoF**	**1 - View (Structural)**	**Automated**
19	**Gaitani et al. [61]**	**Science Direct**	**UML and AST**	**XML**	**Prolog Rules Queries**	**NULL OBJECT design pattern**	**1 - View (Structural)**	**Automated**
20	**Christopoulou et al. [62]**	**Science Direct**	**UML**	**XML**	**Identification Algorithm and Logic Metrics**	**Strategy Design Pattern**	**1 - View (Structural)**	**Automated**
21	**Rasool et al. [26]**	**Science Direct**	**UML with Annotation and Regular expressions**	**XML**	**Database queries**	**Singleton, Factory method, Adapter, Composite, Proxy, Observer, Visitor**	**1 - View (Structural)**	**Automated**
22	**De Lucia et al.[63]**	**Science Direct**	**UML**	**XML**	**XPG formalism and LR-based parsing**	**GoF**	**1 - View (Structural)**	**Automated**
23	**Wang and Tzerpos [64]**	**Science Direct**	**UML and Abstract Syntax Trees (AST)**	**XML**	**Database queries**	**GoF**	**1 - View (Structural)**	**Automated**
24	**Kaczor et al. [65]**	**Science Direct**	**UML**	**Finite Automata**	**Automata Simulation processing Algorithm and bit-vector processing algorithm**	**GoF**	**1 - View (Structural)**	**Automated**
25	**Fabry and Mens [66]**	**Science Direct**	**-**	**Logic parse tree**	**Logic Based and Query-Based**	**GoF**	**1 - View (Structural)**	**Automated**
26	**Antoniol et al. [67]**	**Science Direct**		**Abstract Objects Language (AOL)**	**Metrics Based**	**GoF**	**1 - View (Structural)**	**Manual**
27	**Zanoni et al.[68]**	**Science Direct**	**AST**	**XML**	**Machine Learning and Matching Algorithm**	**Singleton Adapter Composite Decorator Factory method**	**1 - View(Structural)**	**Automated**

28	**Haitzer and Zdun[69]**	**Science Direct**	**UML**	**Structural Primitives and Annotation**	**Selection Based Algorithm**	**Architectural Patterns**	**1 - View (Structural)**	**Semi-Automatic**
29	**Chihada et al. [2]**	**Science Direct**	**-**	**Association Graph and Matrix**	**Searching Algorithm and Machine Learning**	**Adapter Builder Composite Factory method Iterator Observer**	**1 - View (Structural)**	**Semi-Automatic**
30	**Fontana et al. [70]**	**Science Direct**	**-**	**Micro-Structures**	**Selection Based Algorithm**	**Singleton, Abstract Factory, Template Method, State, Composite, and Decorator**	**1 - View (Structural)**	**Automated**
31	**Philippow et al. [71]**	**Springer**	**UML**	**XML**	**Minimal key Structures**	**GoF**	**1 - View (Structural)**	**Automated**
32	**Qing-hua et al. [72]**	**Springer**	**UML**	**XML**	**minimal key structures**	**N/A**	**1 - View (Structural)**	**Semi-Automatic**
33	**Alnusair et al. [73]**	**Springer**	**UML**	**Ontology-Based**	**Graph Matching**	**Composite, Factory Method, and Visitor**	**1 - View (Structural)**	**Automated**
34	**Issaoui et al. [74]**	**Springer**	**UML**	**Metrics Based**	**Search Based Algorithm MAPeD (Multi-phase Approach for Pattern Discovery)**	**GoF**	**1 - View (Structural)**	**Automated**
35	**Issaoui et al.[75]**	**Springer**	**UML**	**Matrix and Metrics-Based**	**Query Based Algorithm and Normalized Algorithm**	**N\A**	**1 - View (Structural)**	**Automated**
36	**Kim and Boldyreff [76]**	**Springer**	**UML**	**Metrics Based**	**Searching Algorithm**	**GoF**	**1 - View (Structural)**	**Automated**
37	**Pande et al. [77]**	**Springer**	**Directed Graph and UML**	**Matrix**	**Graph Matching**	**GoF**	**1 - View (Structural)**	**N/A**
38	**Bouassida Ben-Abdallah [78]**	**Springer**	**UML**	**Matrix (CRMatrix and Normalized CRMatrix)**	**Identification Algorithm**	**Bridge**	**1 - View (Structural)**	**Automated**
39	**Kirasić and Basch[79]**	**Springer**	**AST**	**XML and Ontology Based**	**XML Query Algorithm**	**N/A**	**1 - View (Structural)**	**N/A**

40	Bouassida and Ben-Abdallah [80]	Springer	UML	XML	XML Query Algorithm	Behavioral and Creational	2- Views (Structural and Behavioral ' Method Signature')	Automated
41	Bernardi et al. [81]	IEEE	-	Domain Specific Language (DSL)	Graph-Matching	GoF	1 - View (Structural)	Automated
42	Zhang et al. [82]	IEEE	UML	Matrix	Graph-Matching	NS	1 - View (Structural)	Manual
43	Paydar and Kahani [83]	IEEE	-	Ontology-Based	Query Based Algorithm	GoF	1 - View (Structural)	NS
44	De Lucia et al. [84]	IEEE	UML	XML	visual language parsing	GoF	1 - View (Structural)	Automated
45	Muangon and Intakosum [85]	IEEE	-	Formal Concept Analysis (FCA)	Case Based Reasoning (CBR)	GoF	1 - View (Structural)	Manual
46	Chen and Qiu [86]	IEEE	-	State Space Graph	Mining Algorithm	GoF	1 - View (Structural)	Automated
47	Ren and Zhao [87]	IEEE	-	Ontology-Based	Selection Based Algorithm	Observer	2- Views (Structural and Behavioral ' Sequence Diagram')	Automated
48	Heuzeroth et al. [37]	IEEE	AST	XML	Searching Algorithm	Observer, Composite, Mediator, Chain of Responsibility and Visitor	2- Views (Structural CD and Behavioral ' Dynamic Rules')	Semi-Automatic
49	Gupta et al. [88]	IEEE	UML	Matrix	Normalized cross correlation (NeC)	GoF	1 - View (Structural)	Manual
50	Gupta et al. [89]	IEEE	UML	Matrix	Matching Algorithm	NS	1 - View (Structural)	Semi-Automatic
51	Ferenc et al. [90]	IEEE	UML and Abstract Semantic Graph (ASG)	XML	Graph Matching Algorithm and Machine Learning	Adapter Strategy	1 - View (Structural)	Automatic
52	Pande et al. [91]	IEEE	UML	Matrix	Matching Algorithm	Composite	1 - View (Structural)	Semi-Automatic
53	Basu et al. [92]	IEEE	XMPL	XML	Graph Matching	GoF	1 - View (Structural)	NS
54	Alhusain et al. [93]	IEEE	UML	ANFIS architecture	Search Algorithm and	Adapter, Command, Composite, Decorator,	1 - View (Structural)	Manual

					Machine Learning	Observer and Proxy		
55	Antoniol et al.[31]	IEEE	UML	Abstract Object Language (AOL)	Metrics and Parsing	GoF	1 - View (Structural)	Automatic
56	Kramer and Prechelt [94]	IEEE	-	Prolog Rules	Logic Query Algorithm	Structural Design Patterns	1 - View (Structural)	Automatic
57	Binun and Kniesel [95]	IEEE	UML	XML	Search Algorithm	GoF	2- Views (Structural CD and Behavioral control flow graph CFG)	Automatic
58	Washizaki et al. [96]	IEEE	-	Source Code	Searching Algorithm	NS	1 - View (Structural)	Manual
59	Stoianov and Sora [97]	IEEE	-	Prolog Rules	Logic Query Algorithm	Observer, Singleton, Strategy, Adapter, Decorator.	1 - View (Structural)	Automatic
60	Thongrak and Vatanawood [98]	IEEE	-	Semantic Query-Enhanced Web Rule Language (SQWRL)	Query Based Algorithm	GoF	1 - View (Structural)	Automatic
61	Dongjin et al. [99]	IEEE	-	Directed and weighted Graph	Search Algorithm	NS	1 - View (Structural)	NS
62	Pradhan et al. [100]	IEEE	UML	Matrix	Normalized Cross Correlation and Graph Matching	NS	1 - View (Structural)	Automatic
63	Stencel and Wegrzynowicz [27]	IEEE	-	logic Rules	SQL queries and Parsing Algorithm	GoF	2- Views (Structural CD and Behavioral ‘Message Call’)	Automatic
64	Thankappan and Patil [101]	IEEE	UML	XML	Similarity Scoring Algorithm	GoF	1 - View (Structural)	NS
65	Pandel et al. [102]	IEEE	-	Directed Graph	Graph Matching	GoF	1 - View (Structural)	NS
66	Dong et al. [103]	IEEE	UML	Matrix	Matching Matrix algorithm	GoF	2- Views (Structural CD and Behavioral 'Control Flow Graph (CFG)')	Automatic
67	Nguyen and Pooley [104]	IEEE	UML	XML	Similarity Scoring Algorithm	GoF	1 - View (Structural)	Automatic

					and Fuzzy Method			
68	Kaczor et al. [11]	IEEE	-	Eulerian mode and String representation	bit-vector algorithm	GoF	1 - View (Structural)	Automatic
69	Arcelli et al. [105]	IEEE	AST	XML	Parsing Algorithm	GoF	2- Views (Structural CD and Behavioral 'Message Call')	Automatic
70	Arcelli and Christina [42]	IEEE	AST	XML	Metrics Neural Networks	Elemental Design Patterns	1 - View (Structural)	Automatic
71	De Lucia et al. [106]	IEEE	UML	XML	Searching Algorithm	GoF	2- Views (Structural CD and Behavioral 'Sequence Diagram')	Automatic
72	Sandhu et al. [107]	IEEE	UML	XML	Metrics and Searching Algorithms	GoF	1 - View (Structural)	NS
73	Lebon and Tzerpos [108]	IEEE	-	fine-grained detection rules	Searching Algorithms	GoF	1 - View (Structural)	Automatic
74	Rasool and Mäder [109]	IEEE	-	Feature Types and Regular expressions	Different Search Techniques	GoF	1 - View (Structural)	Automated
75	Yu et al. [110]	IEEE	Class-Relationship Directed Graph and A Structural Feature Model	XML	Searching Algorithms	Structural DP	1 - View (Structural)	Automated
76	Heuzeroth et al. [111]	IEEE	-	Prolog Rules	Searching Algorithm and query based	GoF	2- Views (Structural CD and Behavioral temporal logic of actions (TLA))	Automated
78	He et al. [112]	IEEE	-	(Object Constraint Language) and Java Annotation	Searching Algorithms	GoF	1 - View (Structural)	Semi-Automatic
79	Stephan and Cordy[51]	IEEE	UML	Prolog Rules	Model clone detection (MCD)	NS	1 - View (Structural)	Automated
80	De Lucia et al. [113]	IEEE	UML	XML	Model Checking phase analyzes	Behavioral DP	2- Views (Structural CD and Behavioral SD)	Automated
81	Albin-Amiot et al. [114]	IEEE	-	Pattern Description Language	CSP	GoF	1 - View (Structural)	Automated
82	Haqqie and Shahid [115]	IEEE	UML	XML	Search Algorithm	NS	1 - View (Structural)	Semi-Automatic

83	Balanyi and Ferenc [38]	IEEE	Design Pattern Markup Language (DPML)	XML	Mining Algorithm	GoF	1 - View (Structural)	Automated
84	Bernardi and Di Lucca [116]	IEEE	UML	XML	Matching Algorithm	GoF	1 - View (Structural)	Automated
85	Nagy and Kovari[117]	IEEE	UML	XML	Matching Algorithm	GoF	1 - View (Structural)	Automated
86	Fayad et al. [118]	IEEE	UML	XML	Searching Algorithm	Stable Design Patterns	2- Views (Structural CD and SD)	NS
87	Wegrzynowicz and Stencel [119]	IEEE	UML	Prolog Rules	Logic queries	GoF	1 - View (Structural)	Automated
88	Li et al. [120]	IEEE	UML		Matching Algorithm	GoF	2- Views (Structural CD and Behavioral MCT)	Automated
89	Muangon and Intakosum [85]	IEEE		Formal Concept Analysis (FCA)	Case Based Reasoning (CBR)	GoF	1 - View (Structural)	Manual
90	N Shi and RA Olsson [35]	IEEE	AST	XML	Machine Learning and Searching Algorithm	GoF	2- Views (Structural CD and Behavioral CFG)	Automated
91	Miao et al.[121]	IEEE	UML	XML	Relational Calculus	GoF	1 - View (Structural)	Automated
92	Smith and Stotts [122]	IEEE	UML	XML	Searching Algorithm	GoF	1 - View (Structural)	Automated
93	Zhu et al. [123]	IEEE	Descriptive Semantics of UML	XML	Query Based	GoF	1 - View (Structural)	Automated
94	Alhusain et al.[93]	IEEE	UML	XML	artificial neural network (ANN) and Machine Learning	Adapter, Decorator, Observer, Proxy, Composite, and Command	1 - View (Structural)	Automated
95	Niere et al. [34]	IEEE	UML and abstract syntax graph (ASG)	XML	Parsing Algorithm	GoF	1 - View (Structural)	Automated
96	Tsantalis et al. [32]	IEEE	UML	Matrix	Using Similarity Scoring	GoF	1 - View (Structural)	Automated

97	**Guéhéneuc and Antoniol [124]**	**IEEE**	**-**	**Pattern and Abstract-level Description Language (PADL)**	**CSP**	**GoF**	**1 - View (Structural)**	**Automated**
98	**Antoniol et al. [125]**	**IEEE**	**AST**	**XML**	**Metrics**	**GoF**	**1 - View (Structural)**	**Automated**

Not specified fields “NS” in Table 9 means that the study didn’t mention a clear evidence about the part of the study mentioned in the table.

In the following we are going to discuss each research question based on the results showed in table 9:

Design Pattern Representations (RQ).1?

The effectiveness of the design pattern detection techniques is mainly based on the model representation of the design patterns. Based on the results there are different model representation forms have been used to represent design patterns to detect design patterns effectively. Most of the studied techniques used graph-based representations such as UML, AST, ASG and Directed Graph. The main advantage of using a graphical based representation is that the information of design patterns can be stored as graphs which make it easy to understand and to detected features.

The other representations model is based on logic. A logic-based representation such as first-order logic, and prolog rules. Ontology-based representation is another representation model that has been used in representing and defining design patterns.

There are some studies proposed design patterns detection techniques based on matrix representation form.

There are also other variants of representation models have been used in order to define and represent the different characteristics of design patterns, such as formal concept, relational calculus, Pattern Description Language, etc.

The following table and figure illustrated the percent of each design patterns representation form covered in the systematic literature review:

Table 10 Representations forms statics

Representation Form	**No.**
Matrix	11
Graph-Based	43
Logic-Based	5
Others	39

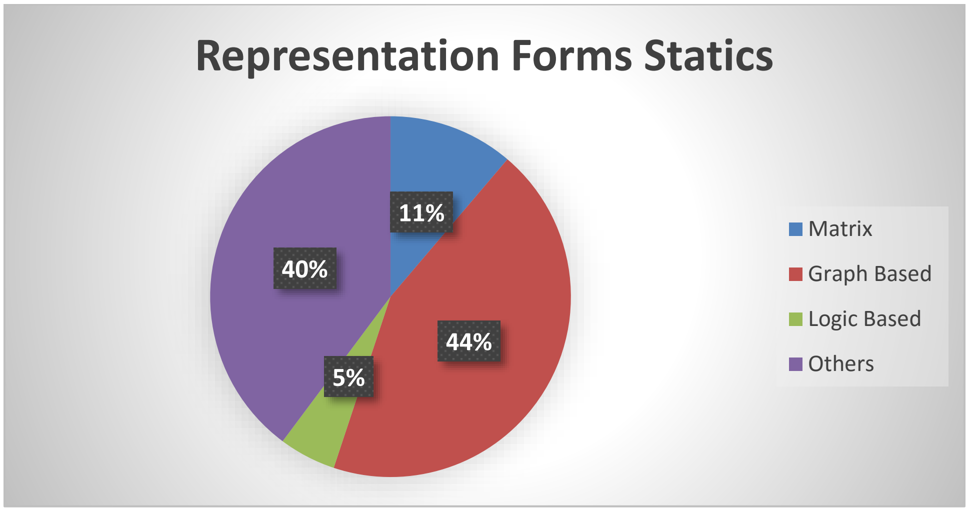


Figure 6 Representations forms statics

The previous table, and figure showed the different statics of each representation covered in the systematic literature review based on Table 9.

Matrix-based representation form takes 11% of the total representation forms used, while graph-based representation takes 40% in total, logic-based represents only 5%, and the 44% represents the other presentation forms like metrics and ontology.

Detection algorithms and methods of DPDT (RQ).2?

To detect design patterns effectively, the researchers should implement and introduce an algorithm that looks for a design pattern based on the representation forms selected. Most of the previous design pattern detection techniques used a search algorithm to search for a design pattern at the code, text, or metamodel.

Some used a matching algorithm to compare the design model with the predefined design pattern model. Some research studies used SQL queries algorithm to detect design pattern. The following table and figure illustrated the different statists of design pattern detection algorithms used in the literature review:

Table 11 Design pattern detection algorithms statics

Algorithm	**No.**
Graph Matching	25
Searching Algorithm	49
SQL Queries	10
Others	14

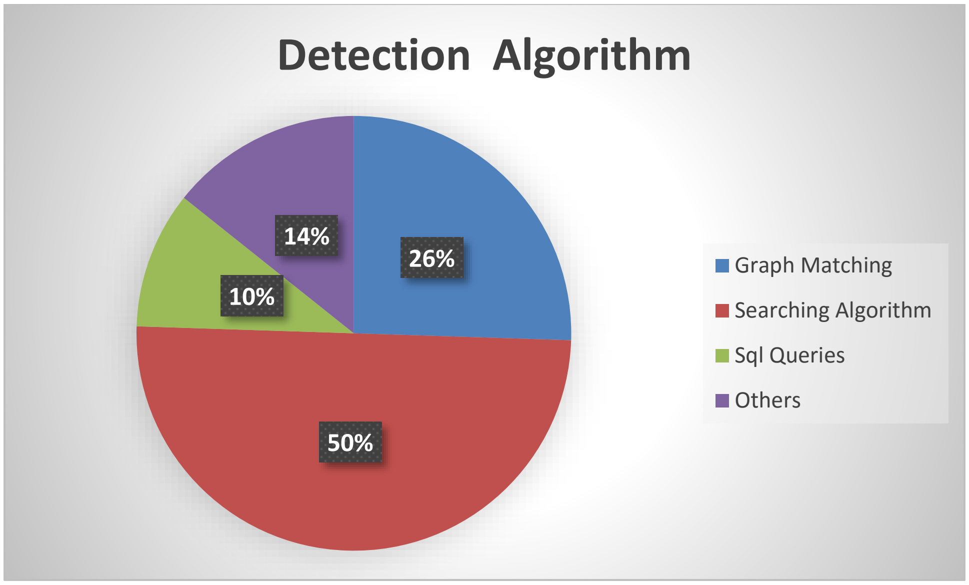


Figure 7 Design pattern detection algorithms statics

The searching algorithm takes 50% of the total design pattern detection algorithms. Whilst Graph matching algorithm is taking 26%, SQL queries are taking 10%, and other design pattern detection algorithms are taking 14%.

What are the different software design views are used in the DPDT?

From the summary Table 9, we can realize that easily most of the previous techniques were conducted only in one view the structural view, there are few studies were conducted in two views (structural and behavioral).

The following table and figure show the statics of different views used in the literature:

Table 12 Design pattern detection views based statics

Views	**No.**
1 View	77
2 Views	20
3 Views	0
Integrated Views	0

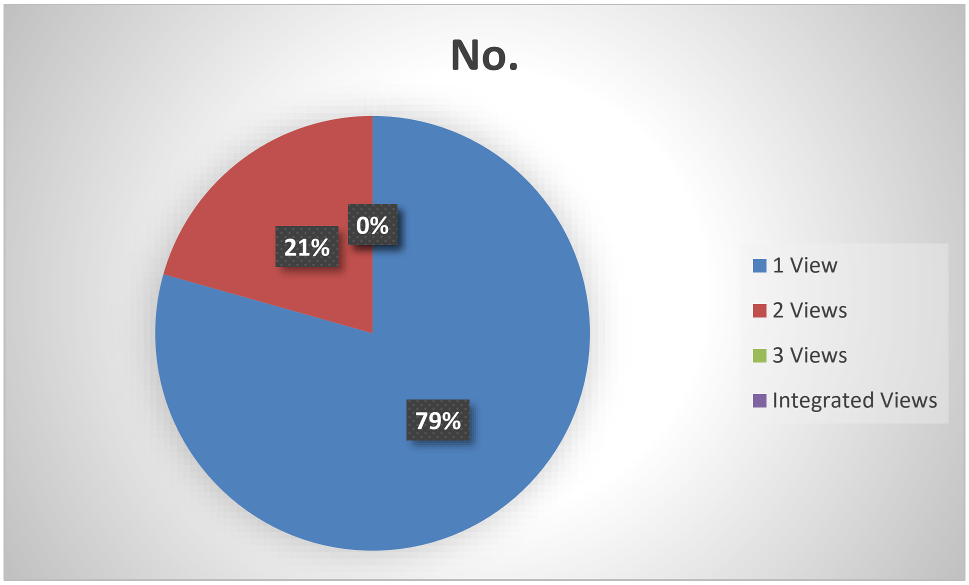


Figure 8 Design pattern detection algorithms statics

The previous figure and table showed that 79% of the previous design pattern detection techniques were based only on one view, while only 21% are based on two views. No research study was conducted on three views or an integrated metamodel.

Is the DPDT seamlessly integrated with the existing software CASE tool or providing prototype tools to simplify the DPDT?

The summary table 9 showed that there is a variety of the automation process of detection design patterns. Some studies were conducted manually or in a semi-manual manner, whilst the most techniques used different CASE TOOLS to fully automate the design pattern detection process.

The following table and figure will list the statics of how much design pattern detection techniques are automated:

Table 13 Design pattern detection automation statics

Views	No.
Automated	70
Manual	18
Semi-Automated	9

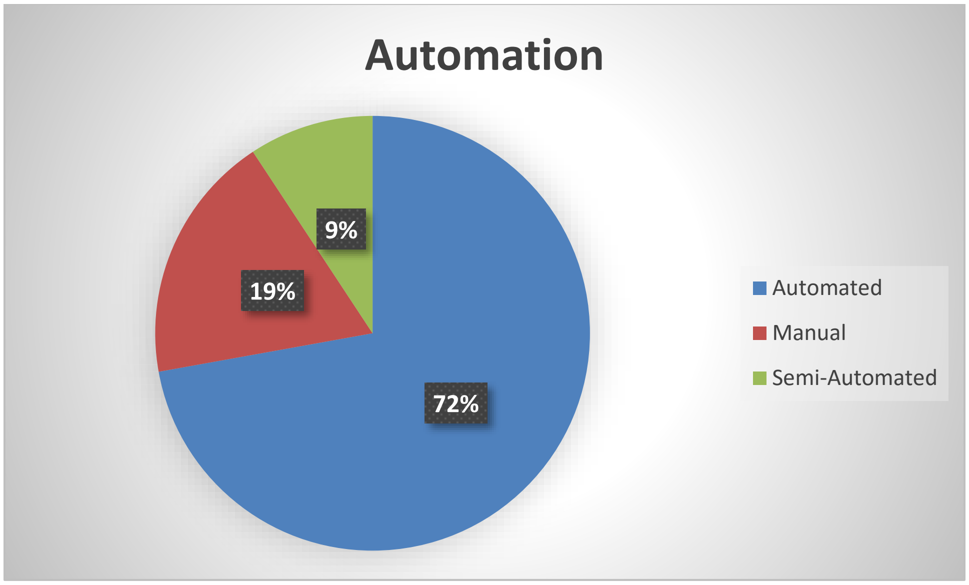


Figure 9 Design pattern automation statics

The statics showed that 72% of design pattern detection techniques proposed in the literature review are automated, whilst 19% are manually conducted, and 19% of the previous detection techniques are semi-automated.

Which design pattern categories have been used in the DPDT?

The summary Table 9showed that most of the previous techniques were conducted in GoF design patterns. Some techniques were conducted in specific design patterns of GoF. There are some studies hadn't mention the type of GoF they have conducted their results on.

Table 14 Design pattern type statics

Design Patterns	No.
GoF	95
Others	3

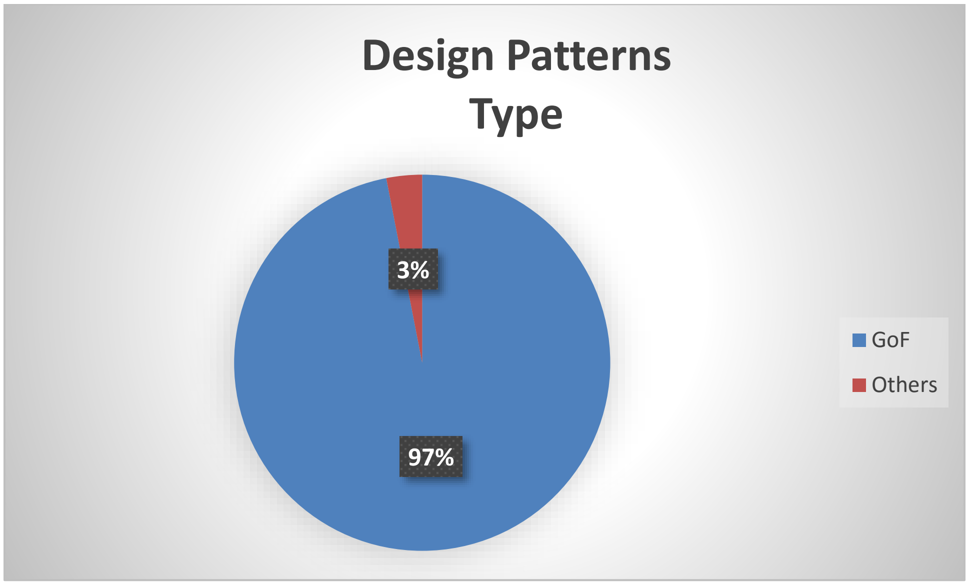


Figure 10 Design pattern type statics

The previous table and figure showed that 97% of the previous design patterns detection techniques were conducted on GoF design patterns, while 3% of the studies conducted their proposed technique in other types of design patterns.

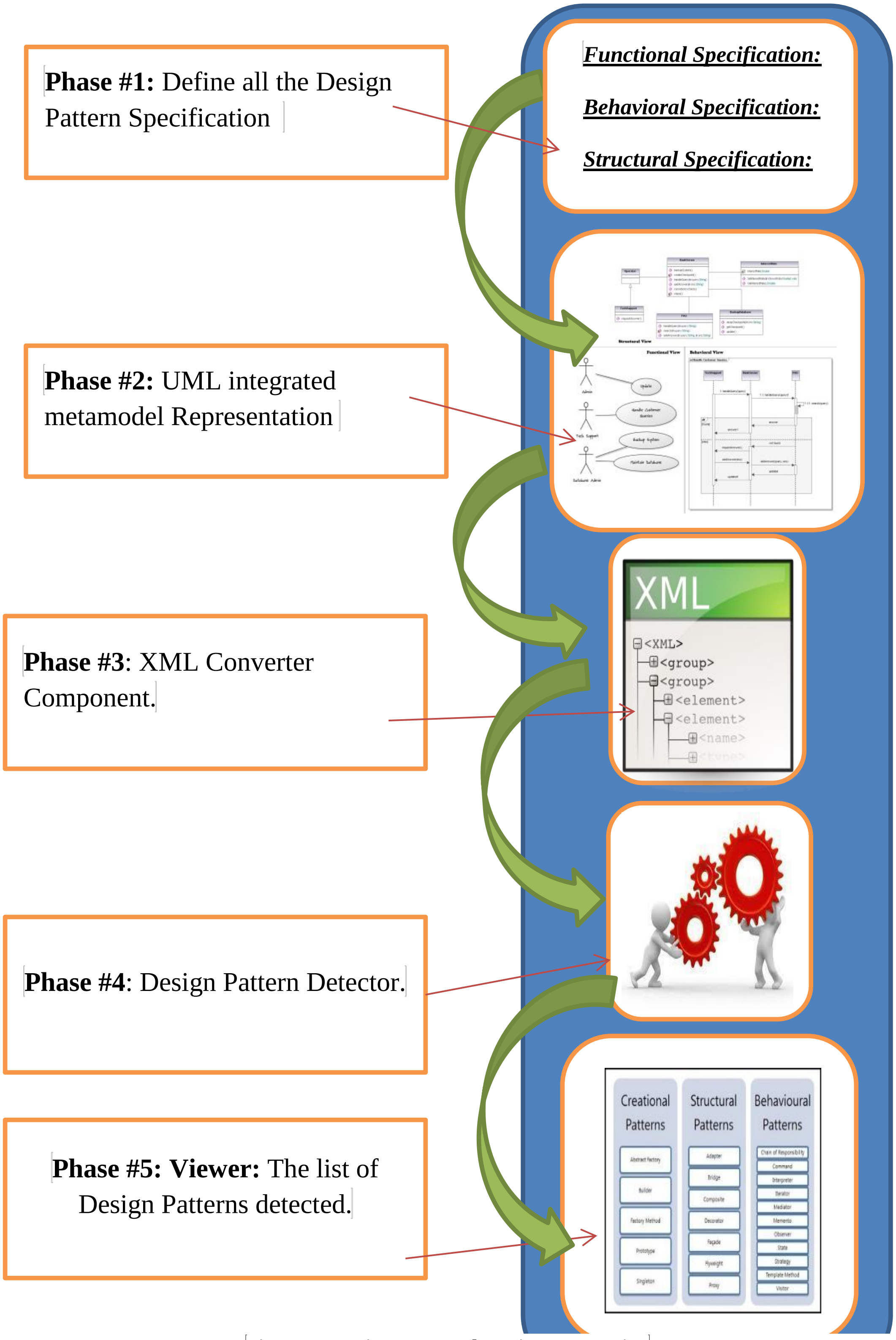
CHAPTER 4

RESEARCH METHODOLOGY

In order to address our research objectives mentioned earlier, we propose to use the integrated metamodel representation to represent the design patterns. Since were using UML to represent the design pattern specification we used the concept of views. The UML model classified into three main views: functional view, behavioral view, and the structural view. Each view represents a major aspect of the software system, when combining all the different views it provide a comprehensive description of the software system. Each view can be represented by different diagrams, we have selected the popular and the most efficient diagram for each view.

Use case diagram represents the functional aspects of the software system, sequence diagram represents the behavioral aspects of the software system, and class diagram represents the structural aspects of the software system.

The outline of our research methodology is illustrated in Figure 11. Figure 11 shows the steps of our proposed representation and detection techniques of design patterns process in a graphical form.



Phase #1: Define all the Design Pattern Specification

Phase #2: UML integrated metamodel Representation

Phase #3: XML Converter Component.

Phase #4: Design Pattern Detector.

Phase #5: Viewer: The list of Design Patterns detected.

Figure 11 Design pattern detection automation

Phase #1: Define all the Design Pattern Specification

In this stage, we are going to define each design pattern different views features. Each design pattern will be detected only if the XML file satisfies all the different views features and specifications. The definition of each design pattern will be discussed for each view.

Phase #2: UML integrated Metamodel Representation

The first step in the automated detection process is representing the design patterns using the UML integrated metamodel proposed in [24]. Every design pattern has been defined and represented by the three main views (Functional, Structural, and behavioral).

Phase #3: XML Converter Component.

We have used the proposed representation tool in [24] in order to convert all the different views of the design pattern to an XML file. We have also used a tool called XML spy in order to validate our representation with UML integrated metamodel proposed in [24].

Phase #4: Design Pattern Detector.

In this phase, we have developed a detection tool by using Python programming language. The algorithm of detection procedure had been taken from the design pattern specification. The tool reads an XML file and checks every tag based on the design pattern specifications, if the XML file satisfies all the different view specification of the design pattern, then the tool claims that it found the design pattern. Otherwise, if the XML file does not satisfy all the different design pattern views specification the tool will claim that there is no design pattern detected.

Phase #5: Display the Detected Design Pattern.

In this phase, the tool displays a list of design patterns detected from the XML file and the number of occurrences of each design patterns.

4.1 Design Pattern Specifications

In this section, we are going to define the design patterns specification. In order to generate an integrated metamodel representation for every design pattern, the design pattern three main views specification should be defined first. The definition of the design pattern specifications will be discussed per each view as follows:

Table 15 Builder Design Pattern Specifications

Building the Complex Object

Functional Specification:

- **Use Case Name**: Building the Complex Object
- **Actor Name**: Client\Programmer\Designer

Behavioral Specification:

- **The first Life Line: Director**
  Operations: director (ConcreteBuilder).
  Operations: construct ().
- **The Second Life Line**: ConcreteBuilder
  Operations: buildPartA().
  Operations: buildPartB().
  Operations: buildPartC().

Structural Specification:

- **The first Class**: Director
  Operations: director (ConcreteBuilder).
  Operations: construct ().
- **The Second Class**: ConcreteBuilder
  Operations: buildPartA().
  Operations: buildPartB().
  Operations: buildPartC().

Inherits: Builder

- **The Third Class**: Builder
  Operations: buildPartX().
  Compose: Director

Table 16 Prototype Design Pattern Specifications

Clone Object

Functional Specification:

- **Use Case Name**: Clone Object
- **Actor Name**: Client\Programmer\Designer

Behavioral Specification:

- **The first Life Line**: ConcretePrototype1
  Operations: doneItSelf().
- **The Second Life Line**: ConcretePrototype2
  Operations: doneItSelf().

Structural Specification:

- **The first Class**: Prototype
  Operations: done().
- **The Second Class**: ConcretePrototype1
  Operations: done().
  Inherits: Prototype
- **The Third Class**: ConcretePrototype2
  Operations: done().
  Inherits: Prototype

Table 17 Singleton Design Pattern Specifications

Get Instance

Functional Specification:

- **Use Case Name**: Get Instance
- **Actor Name**: Client\Programmer\Designer

Behavioral Specification:

- **The first Life Line**: Singleton
  Operations: getInstance().

Structural Specification:

- **The first Class**: Singleton
  Operations: getInstance().
  Operations: singelton().
  Association: itSelf

Table 18 Decorator Design Pattern Specifications

Add Behavior

Functional Specification:

- **Use Case Name**: Add Behavior
- **Actor Name**: Client\Programmer\Designer

Behavioral Specification:

- **The first Life Line**: Decorator1
  Operations: operation().

Operations: addBehaviorr().

- **The Second Life Line**: Decorator2

Operations: operation ().

Operations: addBehaviorr().

- **The Third Class**: ConcreteComponenet

Operations: operation ().

Structural Specification:

- **The first Class**: Component

Operations: operation().

- **The Second Class**: ConcreteComponenet

Operations: operation ().

Inherits: Component

- **The Third Class**: Decorator

Operations: operation ().

- **The Fourth Class**: Decorator1

Operations: operation ().

Operations: addBehaivor ().

Inherits: Decorator

- **The Fifth Class**: Decorator2

Operations: operation ().

Operations: addBehaivor ().

Inherits: Decorator

Table 19 Proxy Design Pattern Specifications

Requesting the Functionality of Subject

Functional Specification:

- **Use Case Name**: Requesting the Functionality of Subject
- **Actor Name**: Client\Programmer\Designer

Behavioral Specification:

- **The first Life Line**: Subject

Operations: operation().

- **The Second Life Line**: RealSubject

Operations: operation ().

Structural Specification:

- **The first Class**: Subject

Operations: operation().

- **The Second Class**: RealSubject

Operations: operation ().

Inherits: Subject

- **The Third Class**: Proxy

Operations: operation ().

Inherits: Subject

Association: RealSubject

Table 20 Adapter Design Pattern Specifications

Adapting the Request

Functional Specification:

- **Use Case Name**: Adapting the Request
- **Actor Name**: Client\Programmer\Designer

Behavioral Specification:

- **The first Life Line**: Adapter
  Operations: requiredMethod().
- **The Second Life Line**: Adaptee
  Operations: specificedMethod().

Structural Specification:

- **The first Class**: Target
  Operations: requiredMethod ().
- **The Second Class**: Adapter
  Operations: requiredMethod ().
  Inherits: Target
  Uses: Adaptee
- **The Third Class**: Adaptee
  Operations: specificedMethod ().

Table 21 Bridge Design Pattern Specifications

Decouple the Abstraction from The Implementation

Functional Specification:

- **Use Case Name**: Decouple the Abstraction from The Implementation
- **Actor Name**: Client\Programmer\Designer

Behavioral Specification:

- **The first Life Line**: Abstraction
  Operations: operation ().
- **The Second Life Line**: ConcreteImpelementor1
  Operations: operationImpl ().
- **The Third Life Line**: ConcreteImpelementor2
  Operations: operationImpl ().

Structural Specification:

- **The first Class**: Abstraction
  Operations: operation ().
- **The Second Class**: Implementor
  Operations: operationImp ().
  Compose: Abstraction
- **The Third Class**: ConcreteAbstraction
  Operations: operation ().
- **The Fourth Class**: ConcreteImpelementor1
  Operations: operationImp ().
  Inherits: Implementor
- **The Fifth Class**: ConcreteImpelementor2

Operations: operationImp ().
Inherits: Implementor

Table 22 Flyweight Design Pattern Specification

Reduce Memory Load

Functional Specification:

- **Use Case Name**: Reduce Memory Load
- **Actor Name**: Client\Programmer\Designer

Behavioral Specification:

- **The first Life Line**: Flyweight
  Operations: create().
  Operations: operationExtrinsicState().
- **The Second Life Line**: FlyweightFactory
  Operations: getFlyweight ().
  Operations: findFlyweight ().

Structural Specification:

- **The first Class**: Flyweight
  Operations: operationExtrinsicState().
  Compose: FlyweightFactory
- **The Second Class**: FlyweightFactory
  Operations: getFlyweight ().
  Inherits: Component
- **The Third Class**: ConcreteFlyweight
  Operations: operationExtrinsicState().
  Inherits: Flyweight
- **The Third Class**: UnsharedConcreteFlyweight
  Operations: operationExtrinsicState().
  Inherits: Flyweight

Table 23 Chain of Responsibility Design Pattern Specifications

Handling Request by Controller

Functional Specification:

- **Use Case Name**: Handling Request by Controller
- **Actor Name**: Client\Programmer\Designer

Behavioral Specification:

- **The first Life Line**: Controller
  Operations: HandleRequest().
- **The Second Life Line**: Mediator 1
  Operations: None.
- **The Third Life Line**: Mediator 2
  Operations: None.
- **Alternative Name**: Handling the request by the next Mediator

The first Life Line: Controller

Operations: ForwardRequest().

Structural Specification:

- **The first Class**: Controller
  - **Operations**: HandleRequest().
  - **Operations**: ForwardRequest().
- **The Second Class**: Mediator 1
  - **Operations**: HandleRequest().
  - **Operations**: ForwardRequest().
  - **Inherits:** Controller
- **The Third Class**: Mediator 2
  - **Operations**: HandleRequest().
  - **Operations**: ForwardRequest().
  - **Inherits:** Controller

Handling Request by a Mediator

Functional Specification:

- **Use Case Name**: Handling Request by a Mediator
- **Actor Name**: Client\Programmer\Designer

Behavioral Specification:

- **The first Life Line**: Controller
  - **Operations**: ForwardRequest().
- **The Second Life Line**: Mediator 1
  - **Operations**: HandleRequest().
- **The Third Life Line**: Mediator 2
  - **Operations**: None.
- **Alternative Name**: Handling the request by the next Mediator
  - **The first Life Line**: Controller
    - **Operations**: ForwardRequest().
  - **The first Life Line**:: Mediator 1
    - **Operations**: ForwardRequest().
  - **The first Life Line**: Mediator 2
    - **Operations** HandleRequest().

Structural Specification:

- **The first Class**: Controller
  - **Operations**: HandleRequest().
  - **Operations**: ForwardRequest().
- **The Second Class**: Mediator 1
  - **Operations**: HandleRequest().
  - **Operations**: ForwardRequest().
  - **Inherits:** Controller
- **The Third Class**: Mediator 2
  - **Operations**: HandleRequest().
  - **Operations**: ForwardRequest().
  - **Inherits:** Controller

Handling Request Partially

Functional Specification:

- **Use Case Name**: Handling Request Partially
- **Actor Name**: Client\Programmer\Designer

Behavioral Specification:

- **The first Life Line**: Controller
  - **Operations**: ForwardRequest().
  - **Operations**: HandlePartially().
  - **Operations**: ForwardPartially().
- **The Second Life Line**: Mediator 1
  - **Operations**: HandlePartially().
  - **Operations**: ForwardPartially().
- **The Third Life Line**: Mediator 2
  - **Operations**: HandlePartially().
  - **Operations**: ForwardPartially().
- **Alternative Name**: Handling the request by the next Mediator
  - **The first Life Line**: Controller
    - **Operations**: Exception ().
  - **The first Life Line**: Mediator
  - **The first Life Line**: Mediator

Structural Specification:

- **The first Class**: Controller
  - **Operations**: HandleRequest().
  - **Operations**: ForwardRequest().
- **The Second Class**: Mediator 1
  - **Operations**: HandleRequest().
  - **Operations**: ForwardRequest().

  Inherits: Controller
- **The Third Class**: Mediator 2
  - **Operations**: HandleRequest().
  - **Operations**: ForwardRequest().

  Inherits: Controller

Table 24 Observer Design Pattern Specification

Watching Item

Functional Specification:

- **Use Case Name**: Watching Item
- **Actor Name**: Client\Programmer\Designer

Behavioral Specification:

- **The first Life Line**: ConcreteSubject
  - **Operations**: watchItemState().
  - **Operations**: registerObcerver().
- **The Second Life Line**: Observer 1
  - **Operations**: None.
- **The Third Life Line**: Observer 2
  - **Operations**: None.

<u>Structural Specification:</u>

➢ **The first Class**: ConcreteSubject

Operations: notify ().

Operations: watchItemState().

Operations: registerObcerver().

Operations: removeObcerver().

Inherits: Subject

➢ **The Second Class**: Observer 1

Operations: update ().

Inherits: Observer

➢ **The Third Class**: Observer 2

Operations: update ().

Inherits: Observer

Item State Changed

<u>Functional Specification:</u>

➢ **Use Case Name**: Item State Changed

➢ **Actor Name**: Client\Programmer\Designer

<u>Behavioral Specification:</u>

➢ **The first Life Line**: ConcreteSubject

Operations: notify().

➢ **The Second Life Line**: Observer 1

Operations: update ().

➢ **The Third Life Line**: Observer 2

➢ **Operations**: update ().

<u>Structural Specification:</u>

➢ **The first Class**: ConcreteSubject()

Operations: notify().

Inherits: Subject

➢ **The Second Class**: Observer 1

Operations: update().

Inherits: Observer

➢ **The Third Class**: Observer 2

Operations: update ().

Inherits: Observer

Table 25 Strategy Design Pattern Specifications

Building the Complex Object

<u>Functional Specification:</u>

➢ **Use Case Name**: Building the Complex Object

➢ **Actor Name**: Client\Programmer\Designer

<u>Behavioral Specification:</u>

➢ **The first Life Line**: Director

Operations: director (ConcreteBuilder).

Operations: construct ().

- **The Second Life Line**: ConcreteBuilder
  Operations: buildPartA().
  Operations: buildPartB().
  Operations: buildPartC().

<u>Structural Specification:</u>

- **The first Class**: Director
  Operations: director(ConcreteBuilder).
  Operations: construct ().
- **The Second Class**: ConcreteBuilder
  Operations: buildPartA().
  Operations: buildPartB().
  Operations: buildPartC().
  Inherits: Builder
- **The Third Class**: Builder
  Operations: buildPartX().
  Compose: Director

Table 26 Mediator Design Pattern Specifications

Handle the Objects Communications

<u>Functional Specification:</u>

- **Use Case Name**: Handle the Objects Communications
- **Actor Name**: Client\Programmer\Designer

<u>Behavioral Specification:</u>

- **The first Life Line**: ConcreteMediator
  Operations: mediate().
- **The Second Life Line**: Colleague1
  Operations: action ().
  Operations: getState().
- **The Third Class**: Colleague2
  Operations: action ().
  Operations: getState().

<u>Structural Specification:</u>

- **The first Class**: Mediator
  Operations: mediate().
  Use: Colleague
- **The Second Class**: ConcreteMediator
  Operations: mediate().
  Inherits: Mediator
  Associate: Colleague1, Colleague2
- **The Third Class**: Colleague
  Operations: action ().
  Operations: getState ().
- **The Forth Class**: Colleague1
  Operations: action ().

Operations: getState ().

Inherits: Colleague

- **The Fifth Class**: Colleague2

  Operations: action ().

  Operations: getState ().

  Inherits: Colleague

Table 27 State Design Pattern Specifications

Change Object State

Functional Specification:

- **Use Case Name**: Change Object State
- **Actor Name**: Client\Programmer\Designer

Behavioral Specification:

- **The first Life Line**: Context

  Operations: request().
- **The Second Life Line**: State1

  Operations: handleRequestState1 ().
- **The Third Class**: State2

  Operations: handleRequestState2 ().

Structural Specification:

- **The first Class**: Context

  Operations: request ().
- **The Second Class**: State1

  Operations: handleRequestState1 ().

  Inherits: State
- **The Third Class**: State2

  Operations: handleRequestState1 ().

  Inherits: State
- **The Forth Class**: State

  Operations: handleRequestState1 ().

  Compose: Context

Table 28 Visit Design Pattern Specifications

Visit Class Elements to Perform Operations

Functional Specification:

- **Use Case Name**: Visit Class Elements to Perform Operations
- **Actor Name**: Client\Programmer\Designer

Behavioral Specification:

- **The first Life Line**: ElementA

  Operations: acceptVisitor().

  Operations: operation().
- **The Second Life Line**: ElementB

Operations: acceptVisitor ().

Operations: operation ().

- **The Third Class**: Visitor

  Operations: visitElementA ().

  Operations: visitElementB ().

Structural Specification:

- **The first Class**: Visitor

  Operations: visitElementA ().

  Operations: visitElementB ().

- **The Second Class**: ConcreteVisitor

  Operations: visitElementA ().

  Operations: visitElementB ().

  Inherits: Visitor

  Uses= ElementA, ElementB

- **The Third Class**: Element

  Operations: acceptVisitor ().

- **The Forth Class**: ElementA

  Operations: operation ().

  Operations: acceptVisitor ().

  Inherits: Element

- **The Fifth Class**: ElementB

  Operations: operation ().

  Operations: acceptVisitor ().

  Inherits: Element

Table 29 Template Method Design Pattern Specification

Define Algorithm Skelton

Functional Specification:

- **Use Case Name**: Define Algorithm Skelton
- **Actor Name**: Client\Programmer\Designer

Behavioral Specification:

- **The first Life Line**: AbstractClass

  Operations: templateMethod().

- **The Second Life Line**: ConcreteClass

  Operations: premitiveOperation1 ().

  Operations: premitiveOperation2 ().

Structural Specification:

- **The first Class**: AbstractClass

  Operations: templateMethod ().

- **The Second Class**: ConcreteClass

  Operations: premitiveOperation1 ().

  Operations: premitiveOperation2 ().

  Inherits: AbstractClass

Table 30 Command Design Pattern Specifications

Encapsulate a Request as an Object

Functional Specification:

- **Use Case Name**: Encapsulate a Request as an Object
- **Actor Name**: Client\Programmer\Designer

Behavioral Specification:

- **The first Life Line**: Command
  - **Operations**: createCommand().
  - **Operations**: execute().
- **The Second Life Line**: Invoker
  - **Operations**: storeCommand ().
  - **Operations**: executeCommand ().
- **The Third Class**: Receiver
  - **Operations**: action ().

Structural Specification:

- **The first Class**: Command
  - **Operations**: createCommand ().
  - **Operations**: execute ().
  - **Compose**: Invoker
- **The Second Class**: Invoker
  - **Operations**: createCommand ().
  - **Operations**: executeCommand ().
- **The Third Class**: Receiver
  - **Operations**: action ().
- **The Fourth Class**: ConcreteCommand
  - **Operations**: createCommand ().
  - **Operations**: execute ().
  - **Inherits**: Command

4.2 Functional Representation of Design Patterns

To the best of our knowledge and the literature review we have done, there is no research study tried to represent the design patterns functionality using any modeling diagram such as Use Case diagram. Since all the previous studies were focusing only on studying design patterns from the two views only (Behavioral and Structural). In our study we used Use Case diagram to represent the design patterns functionality features. We have found that it is better to describe the functionality of each design pattern using some modeling diagram for the following reasons:

1. Defining the functionality of each design pattern will give a concrete base to cover all the different instances and implementation of the design patterns. We think that the best practice is to describe all the different possible functionality and instances for a design pattern using a modeling diagram. Every use case will describe a concrete instance or a functionality of a design pattern.
2. Defining the design patterns functionality using some modeling diagrams will increase the level of documentation because all the different implementation and instances will be described with each Use Case, each Use Case will have its description, sequence diagram, and class diagram.
3. Defining design pattern functionality using some modeling diagrams will help in detecting the design patterns in a semantic way because each use case contains the description of each design pattern instance and implementation.

4.3 Traditional Representations and definition of design patterns

In this part of the research, we are going to define and represent the design patterns using the traditional representation based on Gang of four definitions.

4.3.1 Builder

Builder design patterns is a creational design pattern. The intention behind builder design pattern is to Separate the construction of a complex object from its representation so that the same construction process can create different representations [9].

Applicability

Use the Builder pattern when

1. The algorithm for creating a complex object should be independent of the parts that make up the object and how they are assembled.

Structure View

Here is the class diagram of builder design pattern that represented the structural view features.

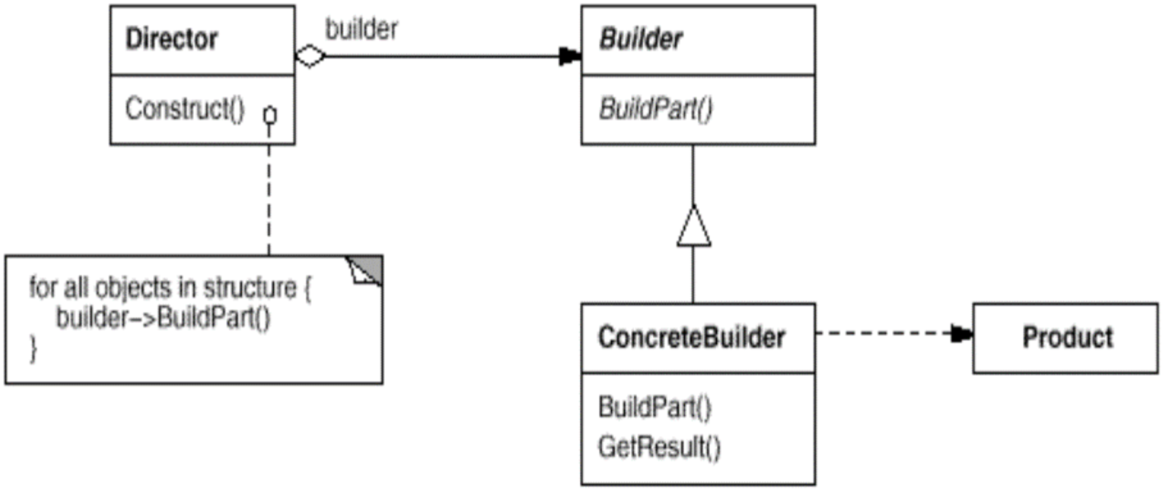


Figure 12 Builder class diagram

The class diagram in Figure 12 showed that the Builder design pattern contains one superclass and two subclasses. The subclasses inherit the superclass and implement its functionality.

The participants in Builder are as following:

- **Builder**

1. Specifies an abstract interface for creating parts of a Product object.

- **ConcreteBuilder**

1. Constructs and assembles parts of the product by implementing the
2. Builder interface.

- **Director**

1. Constructs an object using the Builder interface.

- **Product**

1. Represents the complex object under construction. ConcreteBuilder builds the product's internal representation and defines the process by which it is assembled.

Collaborations

1. The client creates the Director object and configures it with the desired Builder object.
2. Director notifies the builder whenever a part of the product should be built.
3. Builder handles requests from the director and adds parts to the product.
4. The client retrieves the product from the builder.

The following showing the cooperative procedure between Builder and Director.

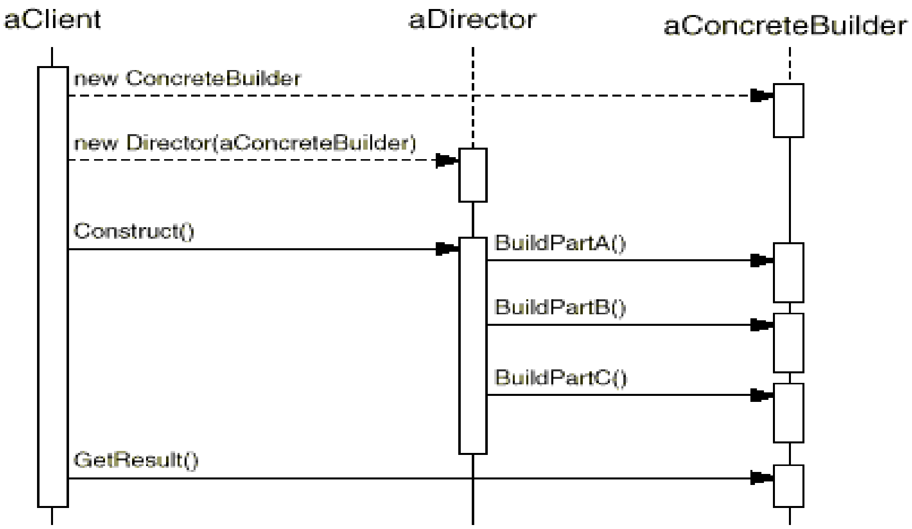


Figure 13 Builder Collaboration Diagram

4.3.2 Adapter

Adapter design patterns is a structural design pattern. The intention behind Adapter design pattern is to convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

- **Applicability**

Use the Adapter pattern when

  - You want to use an existing class, and its interface does not match the one you need.
  - You want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.

- **Structure View**

Here is the class diagram of Adapter design pattern that represented the structural view features.

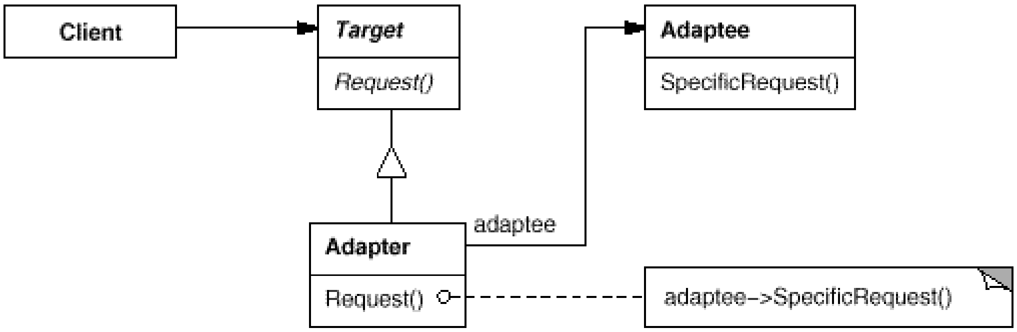


Figure 14 Adapter Diagram [9]

The following class diagram shows the structural view features of Adapter design pattern.

- **Target**

  1. Defines the domain-specific interface that Client uses.

- **Client**

  1. Collaborates with objects conforming to the Target interface.

- **Adaptee**

  1. Defines an existing interface that needs adapting.

- **Adapter**

  1. Adapts the interface of Adaptee to the Target interface.

4.3.3 Chain of Responsibility

Chain of responsibility design patterns is a behavioral design pattern. The intention behind Chain of responsibility design pattern is to avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along with the chain until an object handles it.

- **Applicability**

Use Chain of Responsibility when:

- More than one object may handle a request, and the handler is not known a priori. The handler should be ascertained automatically.
- You want to issue a request to one of the several objects without specifying the receiver explicitly.
- The set of objects that can handle a request should be specified dynamically.

- **Structure View**

Here is the class diagram of a chain of responsibility design pattern that represented the structural view features.

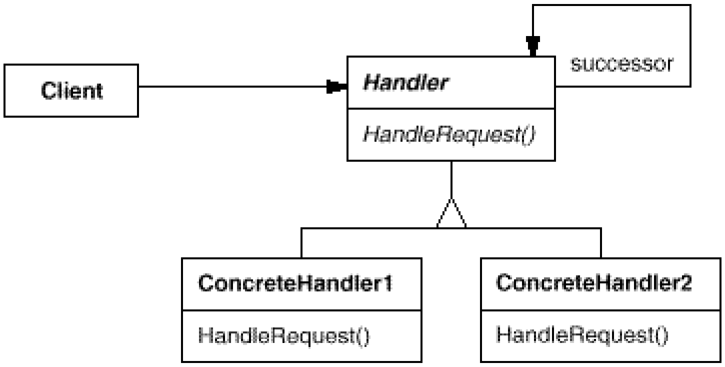


Figure 15 Chain of Responsibility Class Diagram [9]

The class diagram in Figure 15 showed that the chain of responsibility design pattern contains one superclass and two subclasses. The subclasses inherit the superclass and implement its functionality.

A typical object structure might look like this:

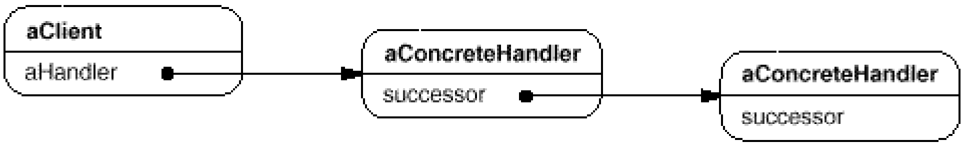


Figure 16 Chain of Responsibility Collaboration Diagram [9]

Figure 16 showed that how the different classes are structured, a client establishes a handler then the handler maintains the connection between the two subclasses.

The participants in the chain of responsibility are as follows:

- **Handler**

2. Defines an interface for handling requests.
3. Optional) implements the successor link.

- **ConcreteHandler**

1. Handles requests it is responsible for.
2. Can access its successor.
3. If the ConcreteHandler can handle the request, it does so; otherwise, it forwards the request to its successor.

- **Client**

1. Initiates the request to a ConcreteHandler object on the chain.

4.3.4 Observer

The intention of observer design pattern is to define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically [9].

- **Applicability**

Use the Observer pattern in any of the following situations:

- When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
- When a change to one object requires changing others, and you don't know how many objects need to be changed?

- **Structure View**

The following class diagram shows the structural view features of Observer design pattern.

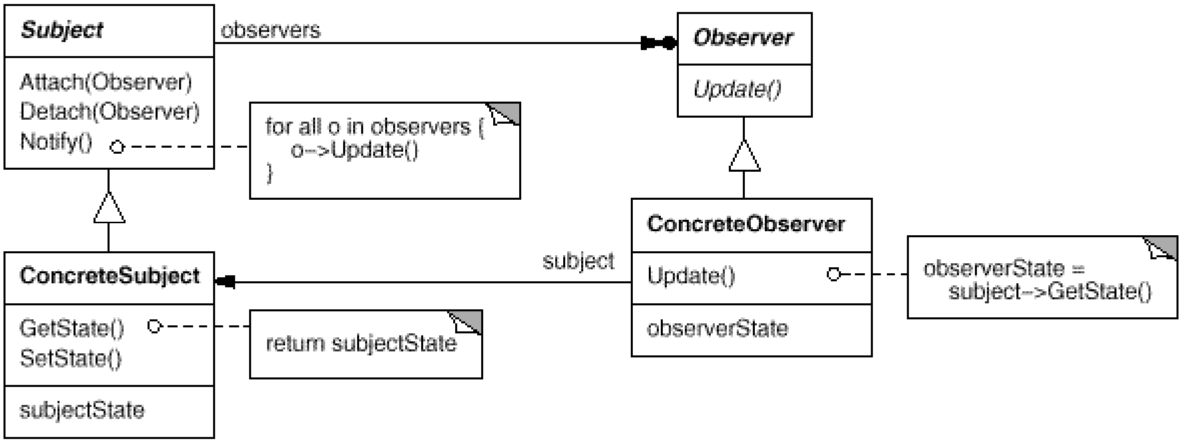


Figure 17 Observer Class Diagram [9]

The class diagram in Figure 17 shows that the observer design pattern contains four main classes. Two of these classes are superclasses while the other two classes are the subclasses. Each subclass implemented its superclass functionality.

The participants in the chain of responsibility are as follows:

- **Subject**

1. Knows its observers. Any number of Observer objects may observe a subject.

2. Provides an interface for attaching and detaching Observer objects.

- **Observer**

  1. Defines an updating interface for objects that should be notified of changes in a subject.

- **ConcreteSubject**

  1. Stores state of interest to ConcreteObserver objects.
  2. Sends a notification to its observers when its state changes.

- **ConcreteObserver**

  1. Maintains a reference to a ConcreteSubject object.
  2. Stores state that should stay consistent with the subjects.
  3. Implements the Observer updating interface to keep its state consistent with the subjects.

- **Collaborations**

  1. ConcreteSubject notifies its observers whenever a change occurs that could make its observers' state inconsistent with its own.
  2. After being informed of a change in the concrete subject, the aConcreteObserver object may query the subject for information. ConcreteObserver uses this information to reconcile its state with that of the subject.

The following interaction diagram shows the behavioral procedure between subject and two observers:

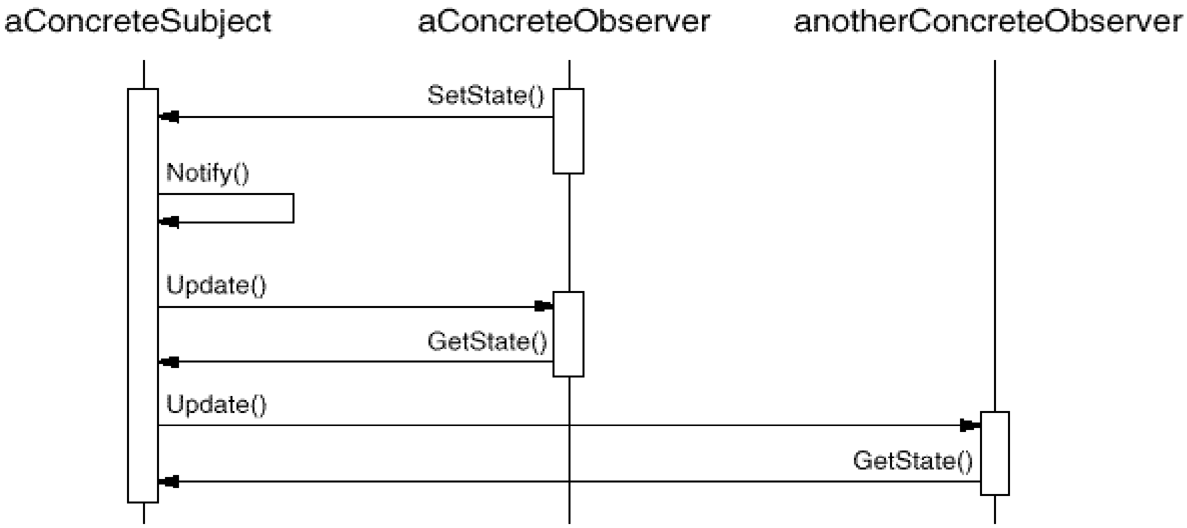


Figure 18 Observer Collaboration Diagram [9]

To know more about the traditional way of representing and defining the rest of design patterns we recommend the reader to refer to GoF book [9].

4.4 Design Patterns Definition and Representation Using the Integrated Metamodel

In this section we will discuss the design patterns from the three main views (Functional, Behavioral, and Structural Diagrams), the discussion here will be different from the traditional representation and definition of design patterns since we added the functional view and the behavioral view as well to support the integrated metamodel representation of the design pattern. We will also consider the different implementations of a design pattern if there is any.

The actor for all the design patterns will be a programmer or a designer and we are going to call it a client.

The importance of defining the functional view of the design patterns using use case diagram is to clarify the main functionality of any design pattern since some design patterns have more than one functionality. Then based on that functionality we can build up the different stages of design patterns defection and representation (behavioral and structural) views. The integrated metamodel XML representation of the design patterns should have all the behavioral and structural features included within one use case. Each use case belongs to one instance of a design pattern.

The main advantage of using the integrated metamodel to represent the design patterns is to have one concrete and integrated XML file that contains all features of the three main views of the design pattern. Each view provides the other views with a complementary information, so the design of the detection made after the design pattern specifies all the three main views features. This advantage helps the programmer and developer to detect the design patterns is a more accurate way. In the other way around, representing the design patterns with individual views will not provide a full description of the design patterns from the three main views.

We are going also to show how the design patterns can be represented using the UML integrated metamodel. We numbered each line of the XML code to make it easy to discuss each section of the code and show what it provides. Two main tools we used to generate the XML file, XML Spy and Enterprise Architecture.

4.4.1 Creational Design Patterns

- **Builder**

We will represent the builder design pattern with an only one-use case that represents its main functionality. Builder design pattern has three main classes: director, builder and concrete builder. The three views representation of builder design pattern is as follows:

Functional View Using Case Diagram

In the functional view of builder design pattern, we will consider only one implementation implemented by only one use case. The use case will include a use case description that explains the main scenario and the functionality of the use case.

The use case diagram of Builder design pattern as follows:

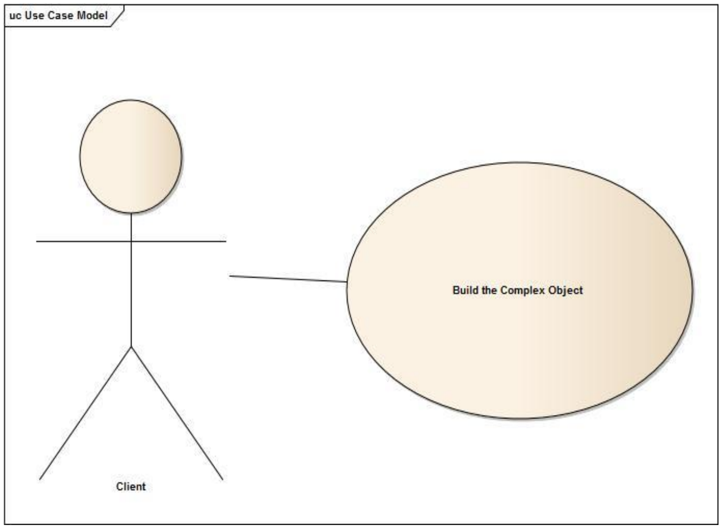


Figure 19 Builder Design Pattern Use Case Diagram

Figure 19 shows the use case diagram of builder design pattern. The only main use case is: Building the Complex Object. The use case has its own description and sequence diagram. The following is the description of the use case:

Table 31 the description of Building the Complex Object Use Case

Use Case Name	Building the Complex Object.
Actor	Client
Main Flow	1- The client creates the Director object and configures it with the desired builder. 2- Director notifies the builder whenever a part of the product should be built. 3- Builder handles requests from the director and adds parts to the product. 4- The client retrieves the product from the builder.

Table 31 shows the description of the Builder uses case 1: Building the Complex Object. The description showed the different steps of the use case in the main flow. The structural and behavioral views representations are similar to the traditional views representation and definition.

Behavioral View Using Sequence Diagram

Since we have only one use case for Builder design pattern then we will have only on sequence diagram that shows the flows of scenario steps from the object and to object and actors of the use case.

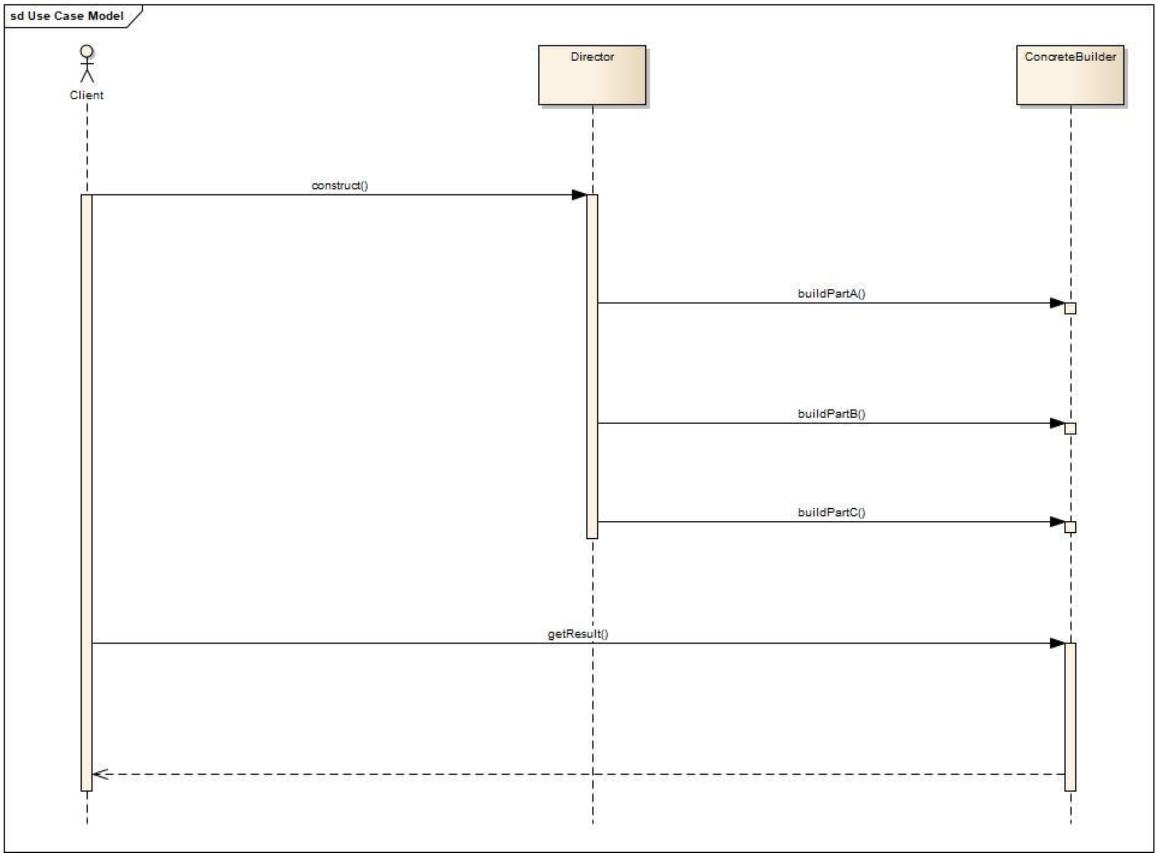


Figure 20 Sequence Diagram of Builder Design Pattern

The sequence diagram in Figure 20 shows the flow of action of objects to create and build a complex object. Where small parts of the object are collected together to build up the final one.

Structural View Using Class Diagram

In the structural view of Builder, there are no changes in the traditional representation since the structural features of the Builder design pattern have been well maintained and defined in the literature review.

The following is the class diagram of Builder:

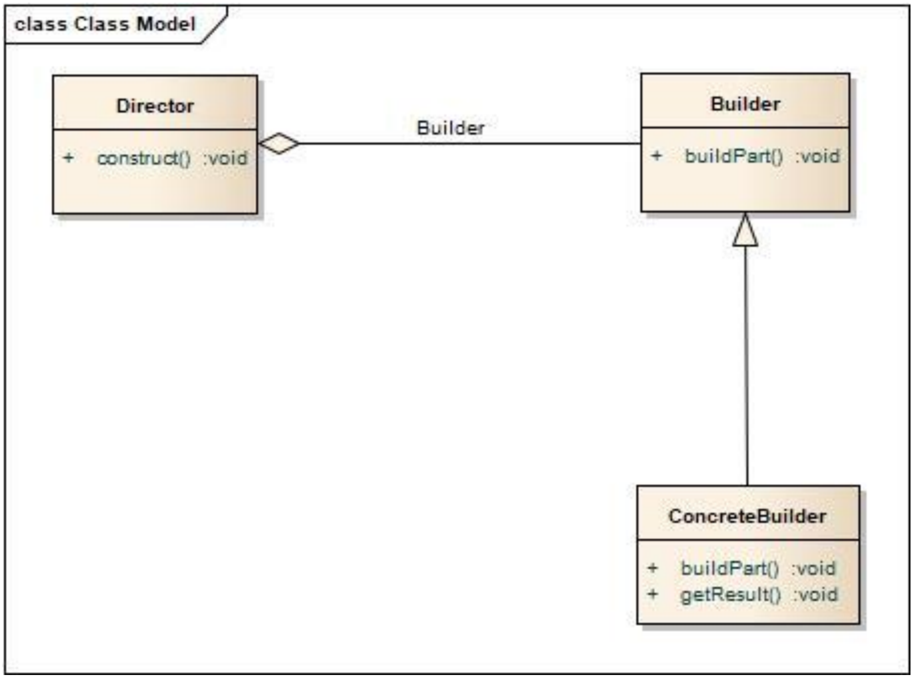


Figure 21 Class Diagram of Builder Design Pattern

Builder UML integrated Representation

The following XML code is the UML integrated metamodel representation of Builder design pattern. The integrated metamodel contains all the three main views features in on concrete file. See Appendix B.

❖ Prototype

We will consider one use case for Prototype design pattern. Prototype design pattern is used for creating new objects (instances) by cloning (copying) other objects and in this way we can improve the performance by not creating the objects from scratch. The prototype is used when the creation of an object is costly or complex. For instance, creating an object after we have a costly database operation.

The three views representation is as follows:

Functional View Using Use Case Diagram

Similarly, as what we have done with previous design patterns, Prototype design pattern we will have use case diagram that describes its functionality, the use case will describe the main functionality of strategy design pattern.

The use case diagram of prototype design pattern as follows:

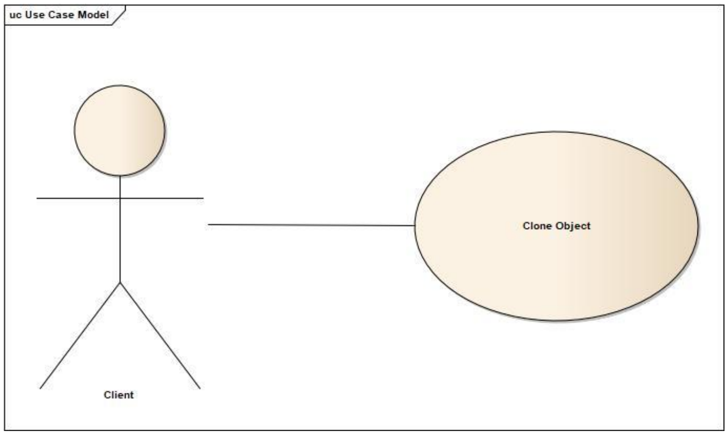


Figure 22 Prototype Design Pattern Use Case Diagram

Figure 22 shows that Prototype design pattern has one main use case: Clone. This main use case represents the main flow of Prototype. The following are the description of each use case:

Table 32 the Description of Clone Object Use Case

Use Case Name	Clone Object
Actor	Client
Main Flow	1- The client sends a request to the prototype to clone itself. 2- The specified concrete class clones itself. 3- The cloned prototype created and send back to the client. 4- End.

Table 32 shows the description of State main uses case: Clone Object. The description showed the different steps of the use case in the main flow.

Behavioral View Using Sequence Diagram

Each use case has its separate sequence diagram that showing the flows of scenario steps from each and to each object and actors of the use case.

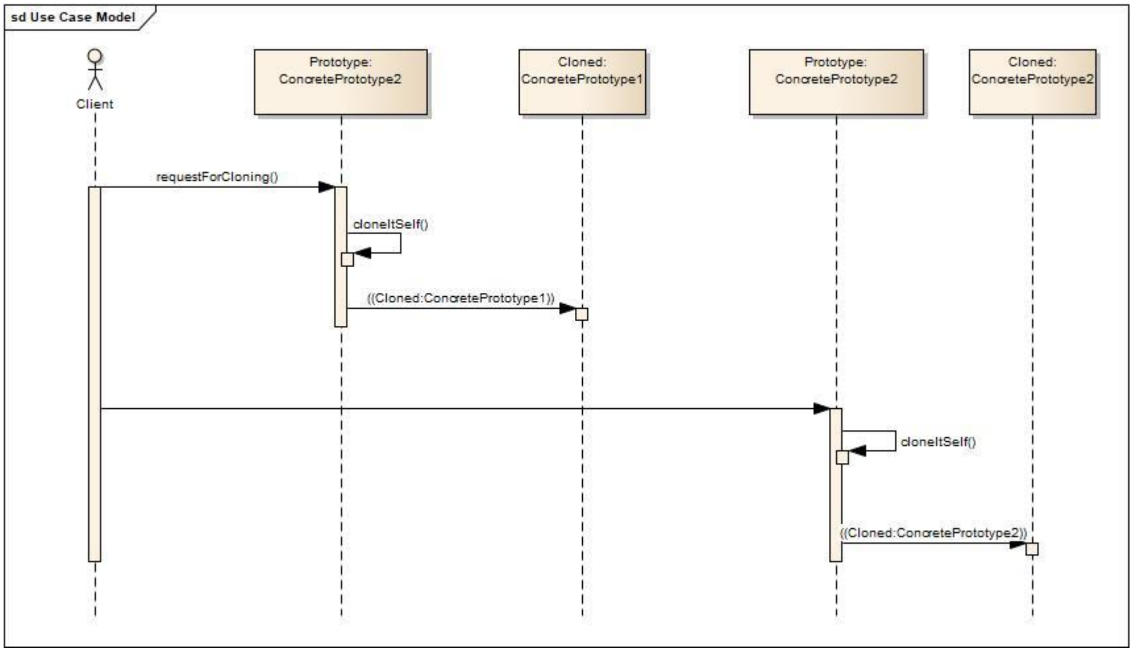

sd Use Case Model
Client
Prototype: ConcretePrototype2
Cloned: ConcretePrototype1
Prototype: ConcretePrototype2
Cloned: ConcretePrototype2
requestForCloning()
cloneItSelf()
((Cloned:ConcreteProtoytpe1))
cloneItSelf()
((Cloned:ConcreteProtoytpe2))

Figure 23 Prototype Design Pattern Sequence Diagram

Figure 23 shows the sequence flow of Prototype main use case: Clone Object. The sequence behavior between the Client and the two different concrete classes ConcretePrototype1 and 2, shows how the flow of action happens if the Client sends a request to clone one of the concrete classes.

Structural View Using Class

In the structural view of Prototype design pattern, the class diagram similar to the traditional class diagram. Where we have the client class and the different Prototype classes.

The following is the class diagram of Prototype:

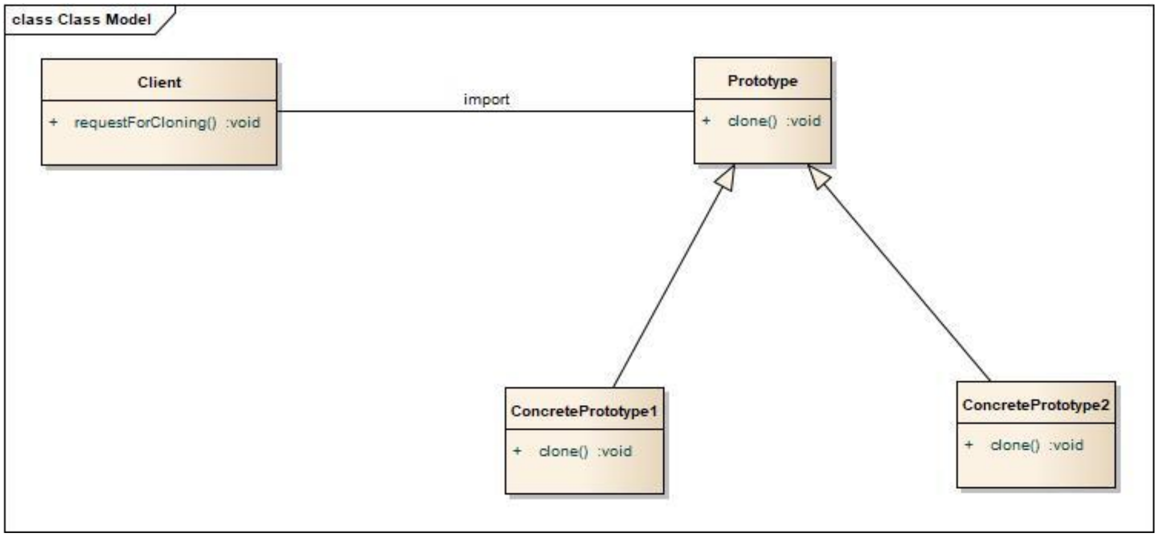


Figure 24 Prototype Design Pattern Class Diagram

Figure 24 shows the class diagram of Prototype design pattern. The class diagram illustrates all the different structural view features. The class diagram contains four classes: Client, Prototype, and two concrete Prototype classes. The client sends a request to one of the concrete classes to clone itself. The specified concrete class clones itself and creates a copy of itself.

Prototype UML integrated Representation

The UML integrated metamodel representation of Prototype design pattern, See Appendix B, contains all the three main view features in the concrete file.

❖ Singleton

We will consider one use case for Singleton design pattern. Singleton design pattern is used for Ensuring a class only has one instance, and provide a global point of access to it.

The three views representation is as follows:

Functional View Using Use Case Diagram

Similarly, as what we have done with previous design patterns, Singleton design pattern we will have use case diagram that describes its functionality.

The use case diagram of Singleton design pattern as follows:

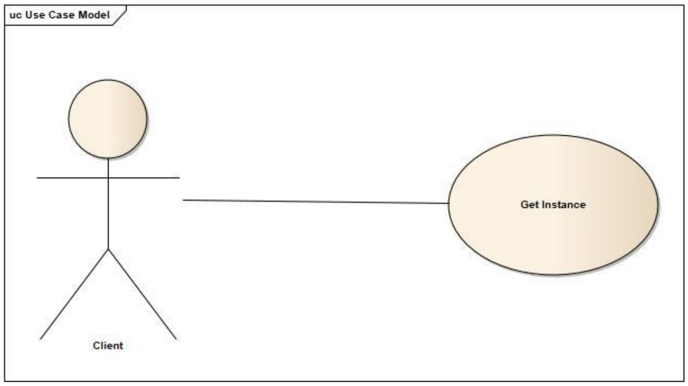


Figure 25 Singleton Design Pattern Use Case Diagram

Figure 25 shows that Singleton design pattern has one main use case: Get Instance. This main use case represents the main flow of Singleton. The following are the description of each use case:

Table 33 the Description of getting Instance Use Case

Use Case Name	Get Instance
Actor	Client
Main Flow	1- The client sends a request to the Singleton to create an instance of itself. 2- Singleton checks if the instance is empty. 3- Singleton creates a new instance of itself. 4- The instance returns to the client 5- End.

Table 33 shows the description of Singleton main uses case: Get Instance. The description showed the different steps of the use case in the main flow.

Behavioral View Using Sequence Diagram

Each use case has its separate sequence diagram that showing the flows of scenario steps from each and to each object and actors of the use case.

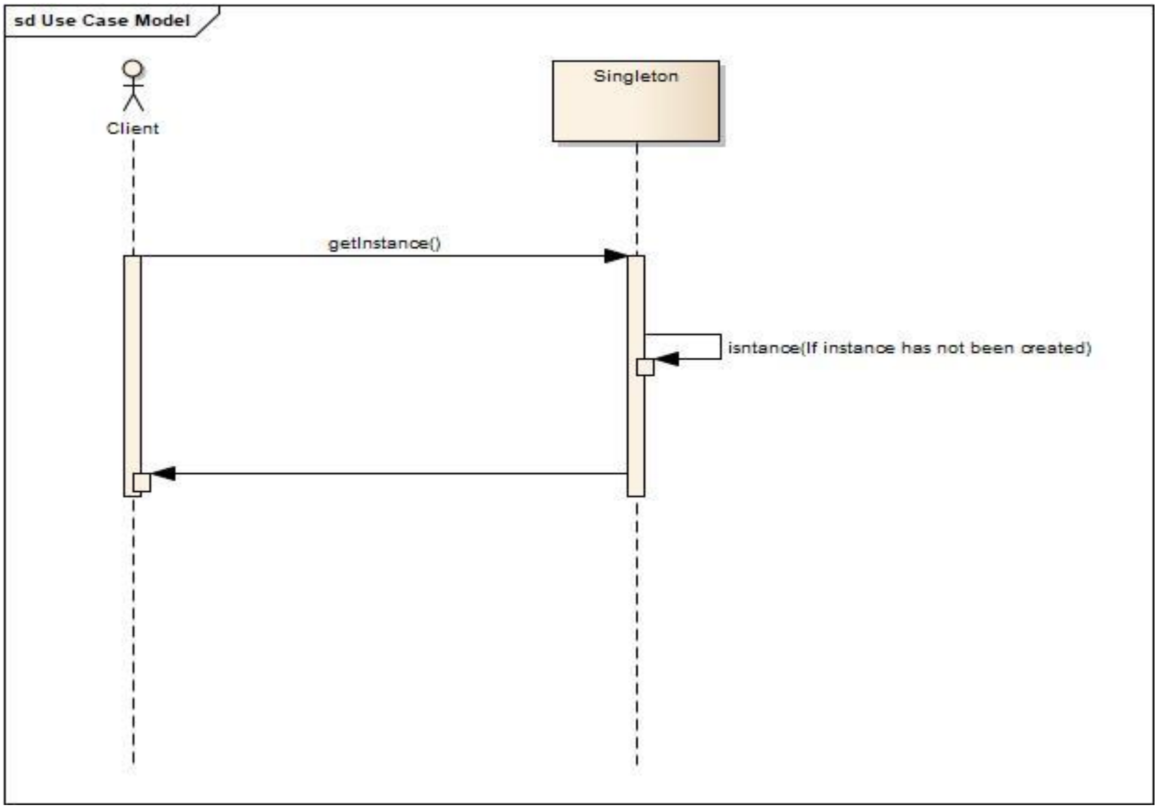


Figure 26 Singleton Design Pattern Sequence Diagram

Figure 26 shows the sequence flow of Singleton main use case: Get Instance. The sequence behavior between the Client and the Singleton classes shows how the flow of action happens if the Client sends a request to Singleton to create an instance of the class.

Structural View Using Class

In the structural view of Singleton design pattern, the class diagram similar to the traditional class diagram. Where we have the client class and the different Singleton classes.

The following is the class diagram of Singleton:

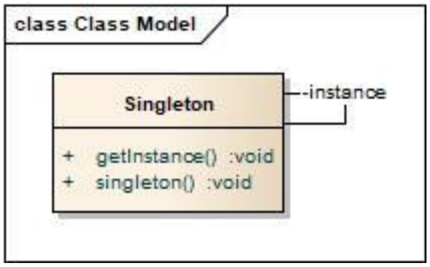


Figure 27 Singleton Design Pattern Class Diagram

Figure 27 shows the class diagram of Singleton design pattern. The class diagram illustrates all the different structural view features. The class diagram contains two classes: Client, and Singleton. The client sends a request to Singleton to create one instance of the class. The Singleton ensures that only one instance is created from the specified class.

Singleton UML integrated Representation

The UML integrated metamodel representation of Singleton design pattern, See Appendix B, contains all the three main view features in the concrete file.

4.4.2 Structural Design Patterns

❖ Decorator

We will consider one use case for Decorator design pattern. Decorator design pattern is used for Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclasses for extending functionality.

The three views representation is as follows:

Functional View Using Use Case Diagram

Similarly, as what we have done with previous design patterns, Decorator design pattern we will have use case diagram that describes its functionality.

The use case diagram of Decorator design pattern as follows:

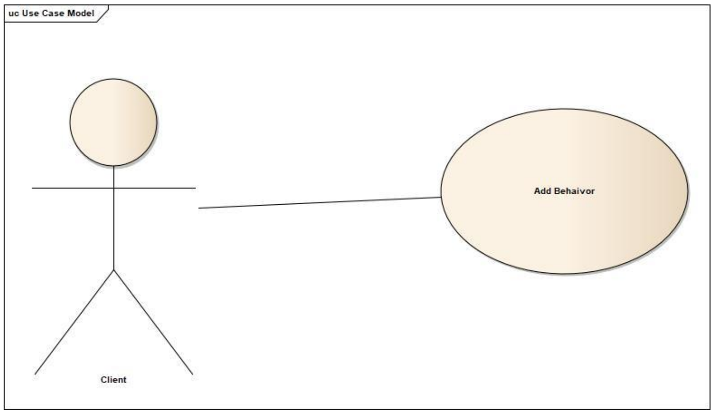


Figure 28 Decorator Design Pattern Use Case Diagram

Figure 28 shows that Decorator design pattern has one main use case: Add Behavior. This main use case represents the main flow of Decorator. The following are the description of each use case:

Table 34 the Description of Add Behavior Use Case

Use Case Name	Add Behavior
Actor	Client
Main Flow	1- The client sends a request to the Decorator. 2- Decorator forwards requests to its Component object. 3- The Decorator performs additional operations before and after forwarding the request. 4- End.

Table 18 shows the description of State main uses case: Add Behavior. The description showed the different steps of the use case in the main flow.

Behavioral View Using Sequence Diagram

Each use case has its separate sequence diagram that showing the flows of scenario steps from each and to each object and actors of the use case.

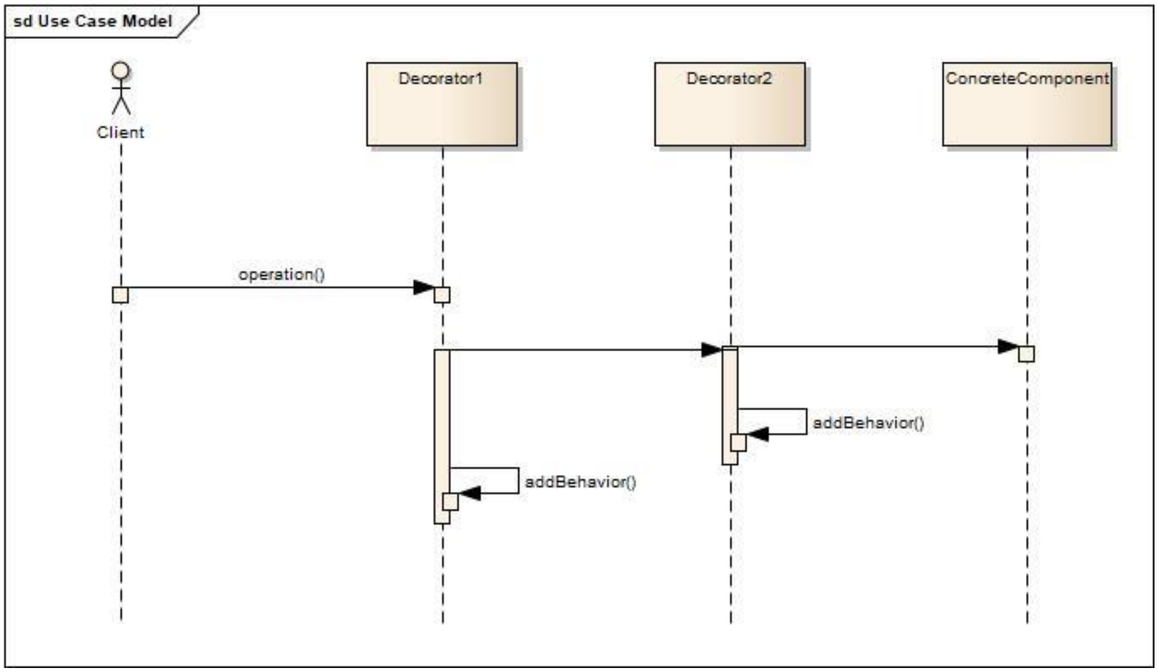


Figure 29 Decorator Design Pattern Sequence Diagram

Figure 29 shows the sequence flow of Decorator main use case: Add Behavior. The sequence behavior between the Component and Decorator classes shows how the flow of action happens if the Client sends a request to Decorator to add behavior.

Structural View Using Class

In the structural view of Decorator design pattern, the class diagram similar to the traditional class diagram. Where we have the client class and the different Decorator classes.

The following is the class diagram of Decorator:

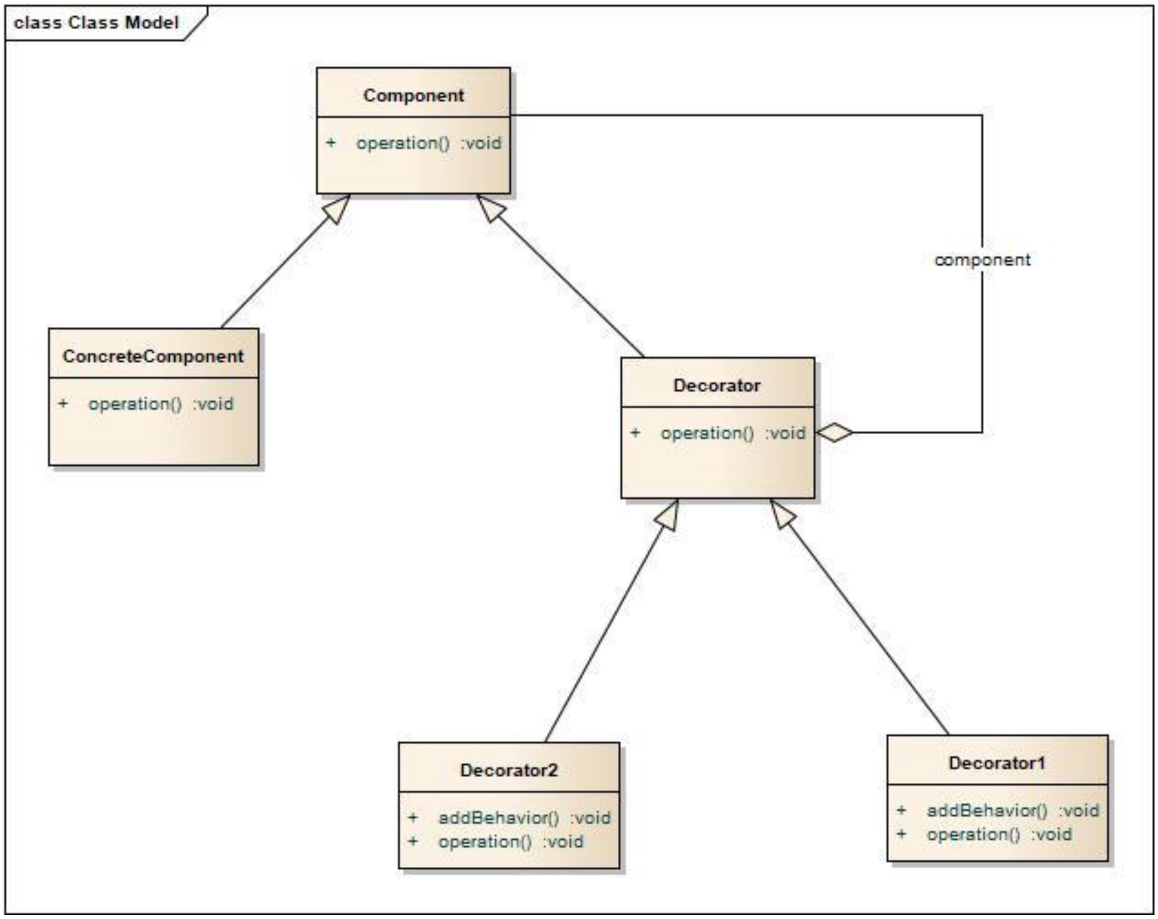


Figure 30 Decorator Design Pattern Class Diagram

Figure 30 shows the class diagram of Decorator design pattern. The class diagram illustrates all the different structural view features. The class diagram contains five classes: Component defines the interface for objects that can have responsibilities added to them dynamically. ConcreteComponent defines an object to which additional responsibilities can be attached. Decorator maintains a reference to a Component object and defines an interface that conforms to Component's interface. ConcreteDecorator adds responsibilities to the component.

Decorator UML integrated Representation

The UML integrated metamodel representation of Decorator design pattern, See Appendix B, contains all the three main view features in the concrete file.

❖ Proxy

We will consider one use case for the Proxy design pattern. A proxy design pattern provides a surrogate or placeholder for another object to control access to it.

The three views representation is as follows:

Functional View Using Use Case Diagram

Similarly, as what we have done with previous design patterns, Proxy design pattern we will have use case diagram that describes its functionality.

The use case diagram of a Proxy design pattern as follows:

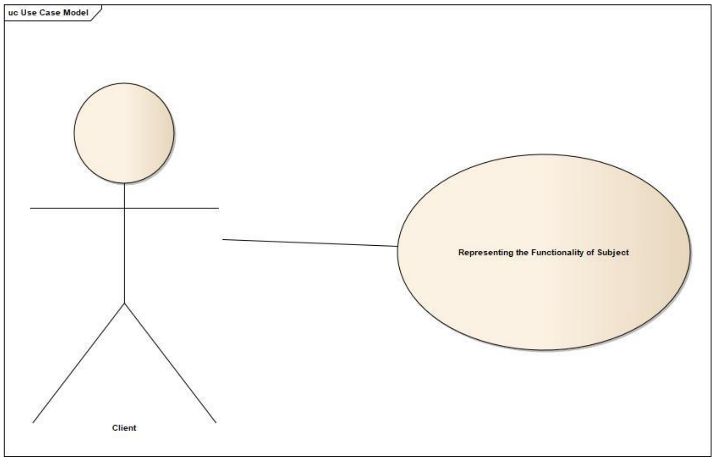


Figure 31 Proxy Design Pattern Use Case Diagram

Figure 31 shows that Proxy design pattern has one main use case: Representing the Functionality of Subject. This main use case represents the main flow of Proxy. The following are the description of each use case:

Table 35 the Description of Representing the Functionality of Subject Use Case

Use Case Name	Representing the Functionality of Subject
Actor	Client
Main Flow	1- The client sends a request to the Decorator.

	2- Decorator forwards requests to its Component object. 3- The Decorator performs additional operations before and after forwarding the request. 4- End.

Table 19 shows the description of Proxy main uses case: Representing the Functionality of Subject. The description showed the different steps of the use case in the main flow.

Behavioral View Using Sequence Diagram

Each use case has its separate sequence diagram that showing the flows of scenario steps from each and to each object and actors of the use case.

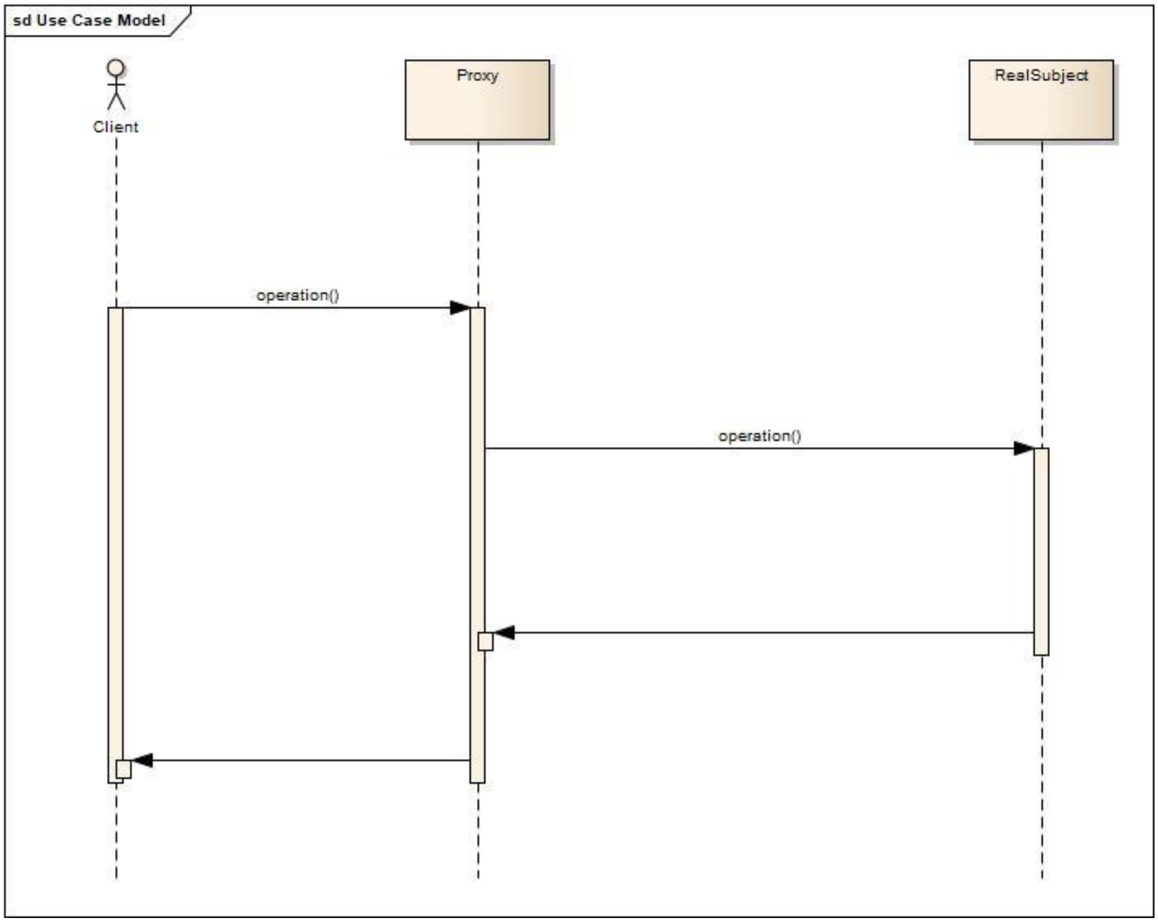


Figure 32 Proxy Design Pattern Sequence Diagram

Figure 32 shows the sequence flow of Proxy main use case: Representing the Functionality of Subject. The sequence behavior between the Proxy and RealSubject classes.

Structural View Using Class

In the structural view of Proxy design pattern, the class diagram similar to the traditional class diagram. Where we have the client class and the different Proxy classes.

The following is the class diagram of Proxy:

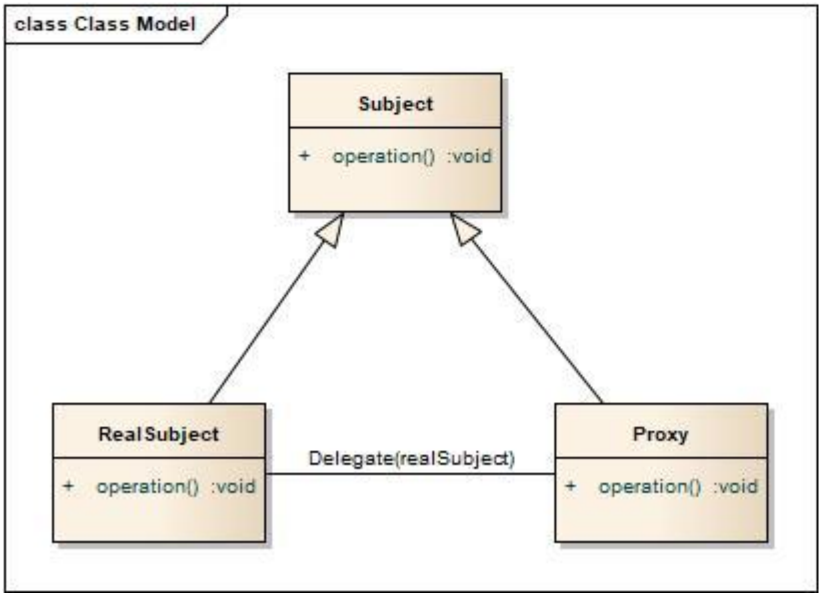


Figure 33 Proxy Design Pattern Class Diagram

Figure 33 shows the class diagram of the Proxy design pattern. The class diagram illustrates all the different structural view features. The class diagram contains three classes: Proxy maintains a reference that lets the proxy access the real subject. A proxy may refer to a Subject if the RealSubject and Subject interfaces are the same. Subject defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected. RealSubject defines the real object that the proxy represents.

Proxy UML integrated Representation

The UML integrated metamodel representation of Proxy design pattern, See Appendix B, contains all the three main view features in the concrete file.

❖ Adapter

We will represent the adapter design pattern with only one use case that represents its main functionality. Adapter design pattern has three main classes: director, builder and concrete builder. The three views representation of Adapter design pattern is as follows:

Functional View Using Use Case Diagram

In the functional view of adapter design pattern, we will consider only one implementation implemented by only one use case. The use case will include a use case description that explains the main scenario and the functionality of the use case.

The use case diagram of adapter design pattern as follows:

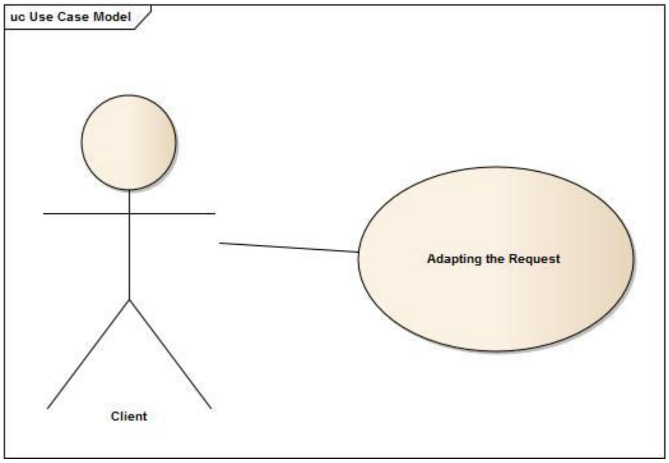


Figure 34 Adapter Design Pattern Use Case Diagram

Figure 34 shows the use case diagram of Adapter design pattern. The only main use case is: Adapting the Complex Object. The use case has its own description and sequence diagram. The following is the description of the use case:

Table 36 Adapter Design Pattern Use Case Diagram

Use Case Name	Adapting the Request.

Actor	Client
Main Flow	1- Client calls operations on an Adapter instance. 2- The adapter calls Adaptee operations that carry out the request. 3- The Adapter replies the results to the client.

Table 20 shows the description of the Adapter use-case 1: Adapting the Request. The description showed the different steps of the use case in the main flow. The structural and behavioral views representations are similar to the traditional views representation and definition.

Behavioral View Using Sequence Diagram

Since we have only one use case for Adapter design pattern then we will have only on sequence diagram that shows the flows of scenario steps from the object and to object and actors of the use case.

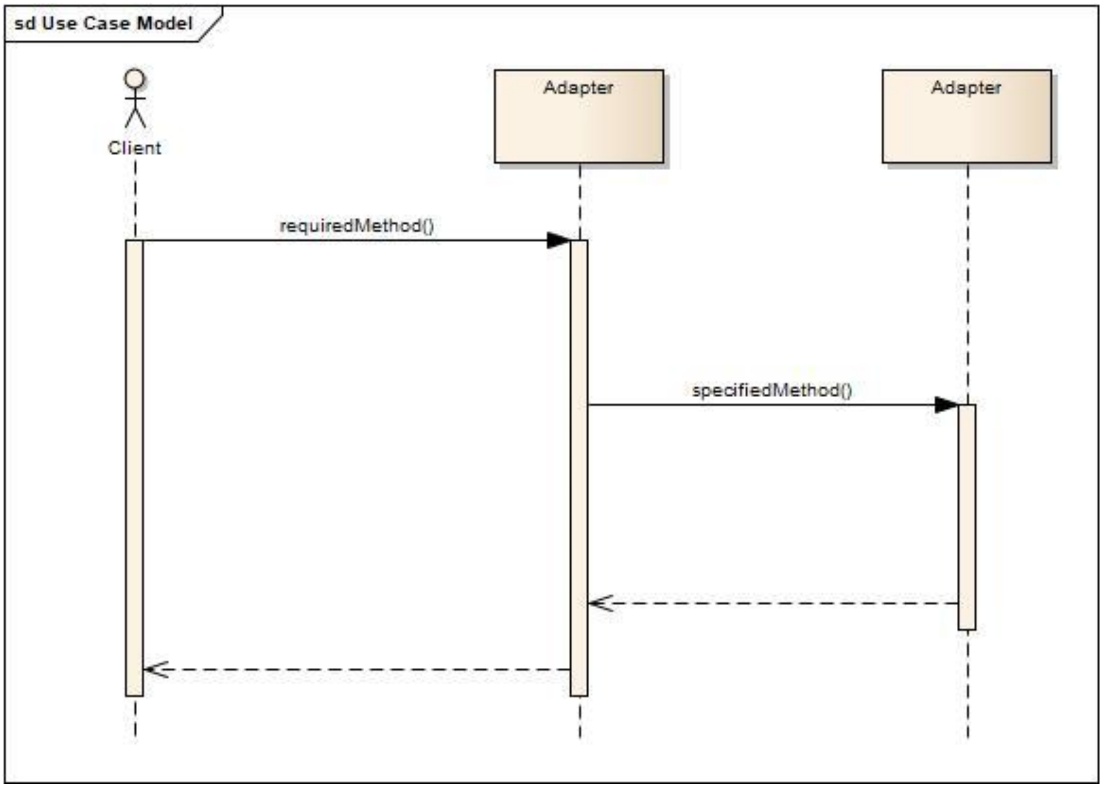


Figure 35 Sequence Diagram of Adapter Design Pattern

The sequence diagram in Figure 35 shows the flow of action of objects to adapt an object to be suitably used by a client since the object can’t be used in its initial formal by the client.

Structural View Using Class Diagram

In the structural view of Adapter, there are no changes in the traditional representation since the structural features of Adapter design pattern have been well maintained and defined in the literature review.

The following is the class diagram of Adapter:

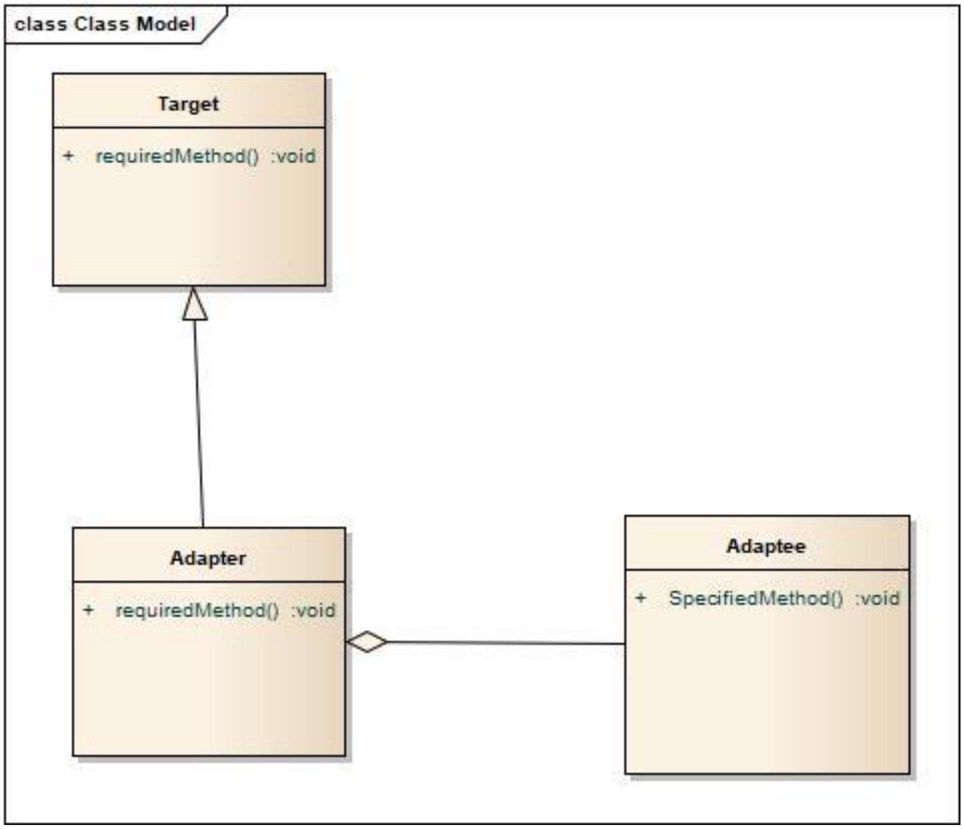


Figure 36 Class Diagram of Adapter Design Pattern

Adapter UML integrated Representation

The following XML code is the UML integrated metamodel representation of Adapter design pattern. The integrated metamodel contains all the three main views features in on concrete file. See Appendix B.

❖ Bridge

We will consider one use case for Bridge design pattern. Bridge design pattern is used for Ensuring a class only has one instance, and provide a global point of access to it.

The three views representation is as follows:

Functional View Using Use Case Diagram

Similarly, as what we have done with previous design patterns The Bridge Design Pattern is a structural design pattern used to completely decouple an abstraction from its implementation so that both of them can change independently.

The use case diagram of Bridge design pattern as follows:

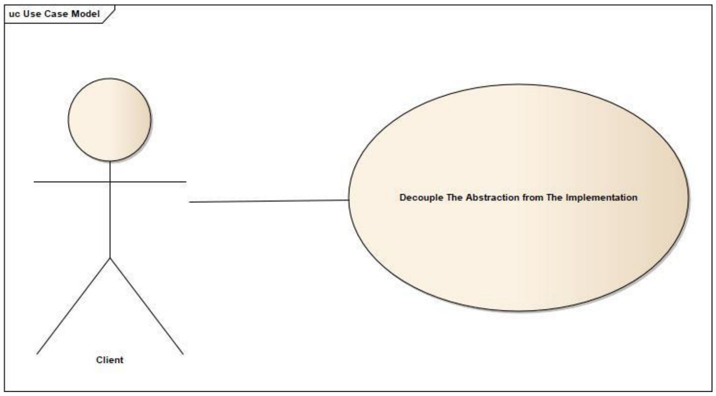


Figure 37 Bridge Design Pattern Use Case Diagram

Figure 37 shows that Bridge design pattern has one main use case: Decouple the Abstraction from The Implementation. This main use case represents the main flow of Bridge. The following are the description of each use case:

Table 37 the Description of Decouple the Abstraction from the Implementation Use Case

Use Case Name	Decouple the Abstraction from The Implementation
Actor	Client
Main Flow	1- The client sends the request to the Abstraction to perform a specific operation. 2- The Abstraction sends the request to the appropriate implementation. 3- The Abstraction returns the implementation to the client. 4- End.

Table 21 shows the description of Bridge main use case: Decouple the Abstraction from The Implementation. The description showed the different steps of the use case in the main flow.

Behavioral View Using Sequence Diagram

Each use case has its separate sequence diagram that showing the flows of scenario steps from each and to each object and actors of the use case.

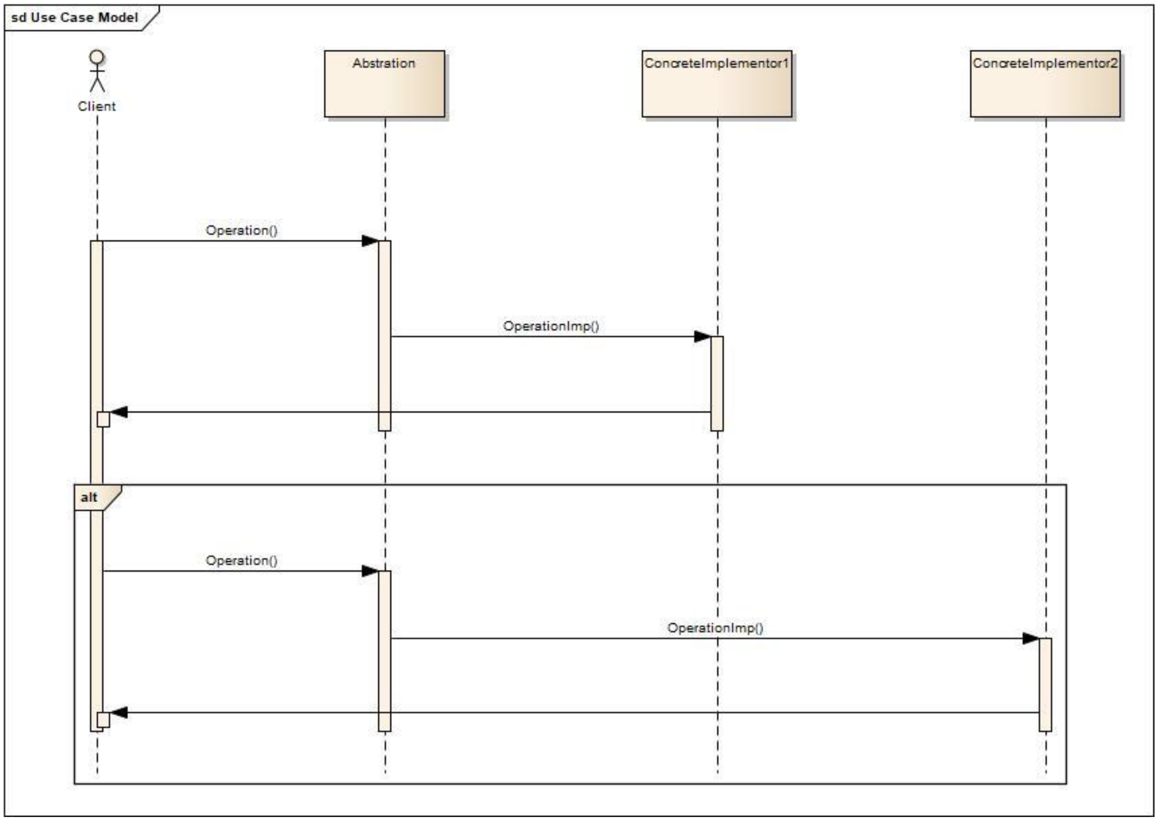


Figure 38 Bridge Design Pattern Sequence Diagram

Figure 38 shows the sequence flow of Bridge main use case: Decouple the Abstraction from The Implementation. The sequence behavior between the Abstract Class and the two concrete Implementor classes, it shows how the flow of action happens if the client requested a specific implementation for an operation.

Structural View Using Class

In the structural view of Bridge design pattern, the class diagram similar to the traditional class diagram. Where we have the client class and the different Bridge classes.

The following is the class diagram of Bridge:

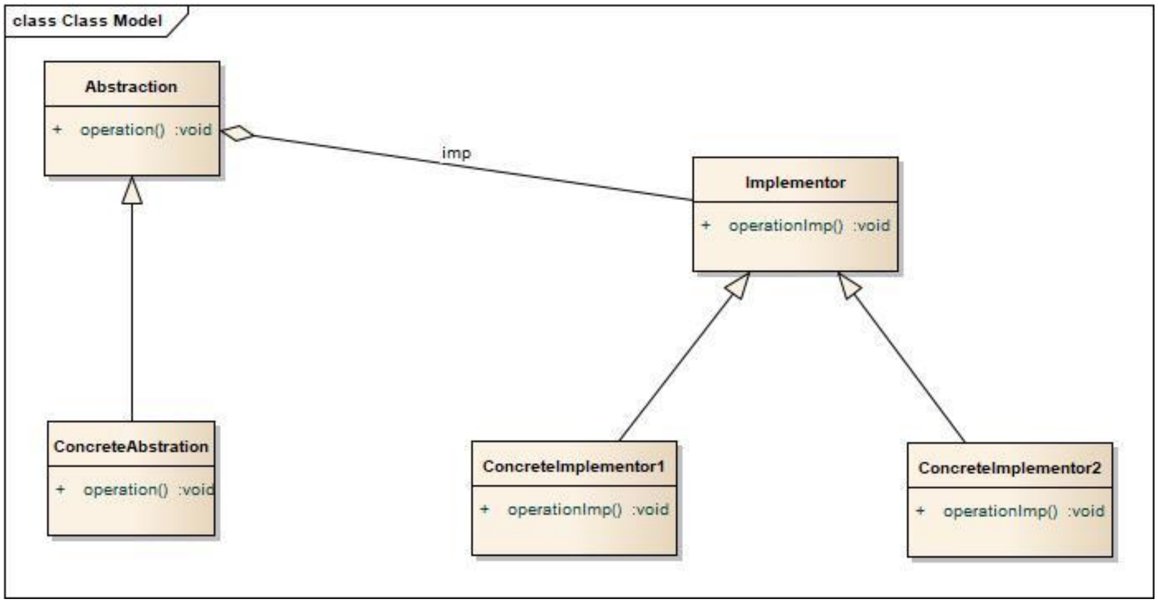


Figure 39 Bridge Design Pattern Class Diagram

Figure 39 shows the class diagram of Bridge design pattern. The class diagram illustrates all the different structural view features. The class diagram contains four classes: Abstraction (abstract class) which defines the abstract interface maintains the Implementor reference. ConcreteAbstraction (normal class) extends the interface defined by Abstraction Implementor (interface) which defines the interface for implementation classes ConcreteImplementor (normal class) that implements the Implementor interface.

Bridge UML integrated Representation

The UML integrated metamodel representation of Bridge design pattern, See Appendix B, contains all the three main view features in the concrete file.

❖ Flyweight

We will consider one use case for Flyweight design pattern. Use sharing to support large numbers of fine-grained objects efficiently.

The three views representation is as follows:

Functional View Using Use Case Diagram

Similarly, as what we have done with previous design patterns The Bridge Design Pattern is a structural design pattern used to completely decouple an abstraction from its implementation so that both of them can change independently.

The use case diagram of Bridge design pattern as follows:

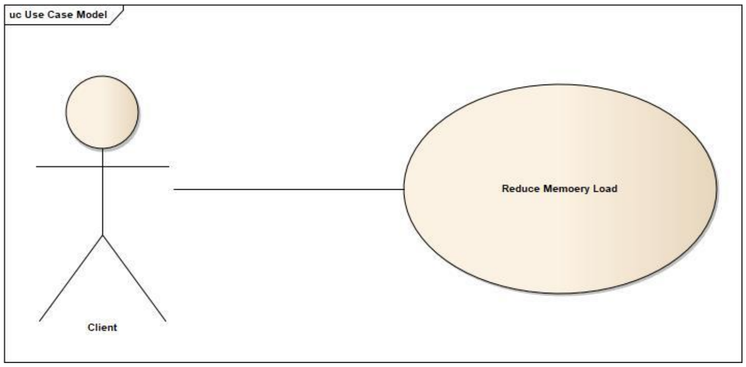


Figure 40 Flyweight Design Pattern Use Case Diagram

Figure 40 shows that Flyweight design pattern has one main use case: Reduce Memory Load. This main use case represents the main flow of Flyweight. The following are the description of each use case:

Table 38 the Description of Reduce Memory Load Use Case

Use Case Name	Reduce Memory Load
Actor	Client
Main Flow	1- The client sends a request to get a copy from an object. 2- The FlyweightFactory checks if the Flyweight exists. 3- If it exists then it shares it. 4- Otherwise, it creates it and then shares it. 5- End.

Table 22 shows the description of Flyweight main use case: Reduce Memory Load. The description showed the different steps of the use case in the main flow.

Behavioral View Using Sequence Diagram

Each use case has its separate sequence diagram that showing the flows of scenario steps from each and to each object and actors of the use case.

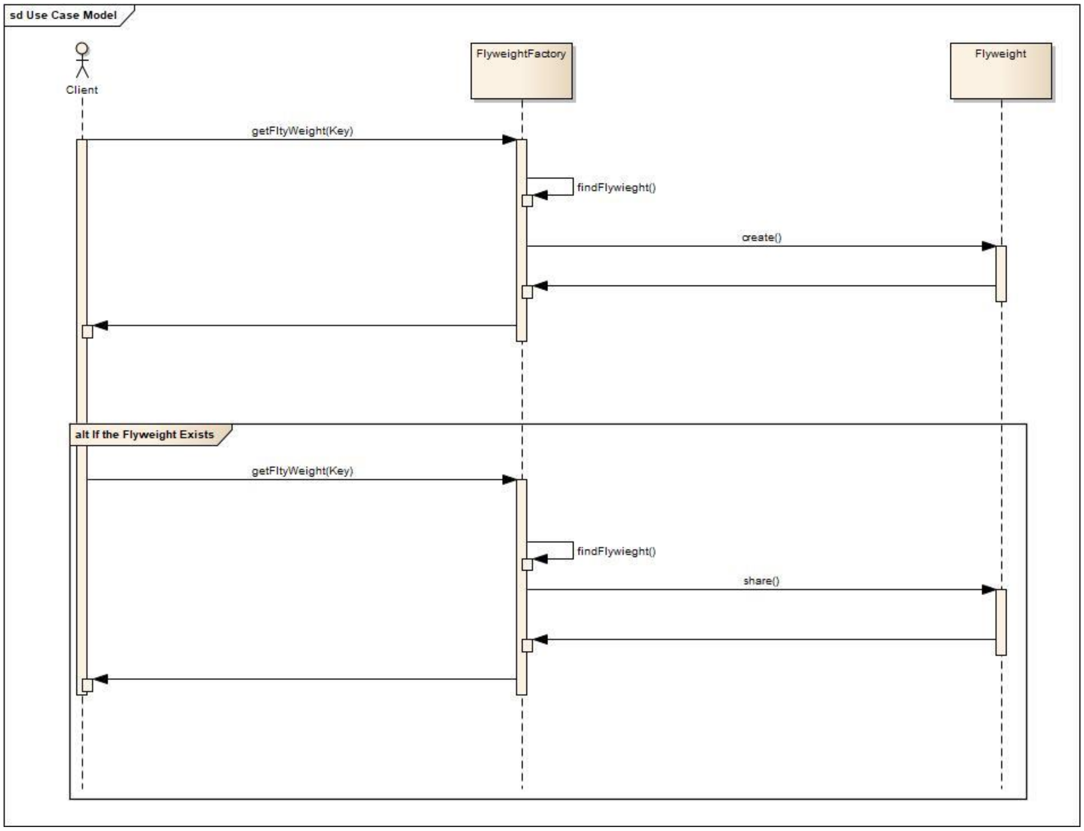


Figure 41 Flyweight Design Pattern Sequence Diagram

Figure 42 shows the sequence flow of Flyweight main use case: Decouple the Abstraction from The Implementation. The sequence behavior between the FlyweightFactory and the Flyweight class, it shows how the flow of action happens when the FlyweightFactory creates and shares the Flyweight.

Structural View Using Class

In the structural view of Flyweight design pattern, the class diagram similar to the traditional class diagram. Where we have the client class and the different Flyweight classes.

The following is the class diagram of Flyweight:

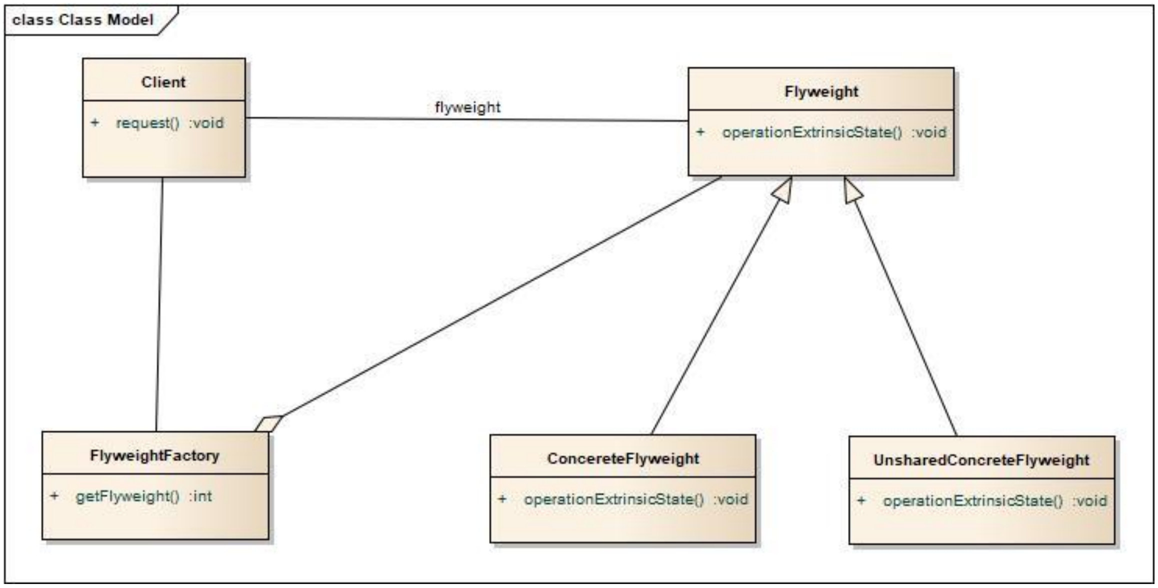

class Class Model
Client
+ request() :void
flyweight
Flyweight
+ operationExtrinsicState() :void
FlyweightFactory
+ getFlyweight() :int
ConcereteFlyweight
+ operationExtrinsicState() :void
UnsharedConcreteFlyweight
+ operationExtrinsicState() :void

Figure 42 Flyweight Design Pattern Class Diagram

Figure 42 shows the class diagram of Flyweight design pattern. The class diagram illustrates all the different structural view features. The class diagram contains five classes: Flyweight declares an interface through which flyweights can receive and act on the extrinsic state. ConcreteFlyweight implements the Flyweight interface and adds storage for the intrinsic state if any. A ConcreteFlyweight object must be sharable. It stores must be intrinsic; that is, it must be independent of the ConcreteFlyweight object's context. UnsharedConcreteFlyweight, not all Flyweight subclasses need to be shared. The Flyweight interface enables sharing; it doesn't enforce it. It is common for

UnsharedConcreteFlyweight objects to have ConcreteFlyweight objects as children at some level in the flyweight object structure. FlyweightFactory creates and manages flyweight objects. It ensures that flyweights are shared properly. When a client requests a flyweight, the FlyweightFactory object supplies an existing instance or creates one, if none exists. The client maintains a reference to flyweight(s). It computes or stores the extrinsic state of flyweight(s).

Flyweight UML integrated Representation

The UML integrated metamodel representation of Flyweight design pattern, See Appendix B, contains all the three main view features in concrete file.

4.4.3 Behavioral Design Patterns

- **Chain of Responsibility**

We will consider a chain of responsibility case where we have one controller and two mediators. The controller is the main object which responsible for forwarding and receiving requests to and from mediators and response to the client. The three views representation of the chain of responsibility is as follows:

Functional View Using Use Case Diagram

In the functional view of the chain of responsibility design pattern, we will consider all the possible implementations and scenarios. Each use case will represent one possible scenario or instance. Each use case will include a use case description that explains the main scenario and the functionality of the use case.

The use case diagram of a chain of responsibility design pattern as follows:

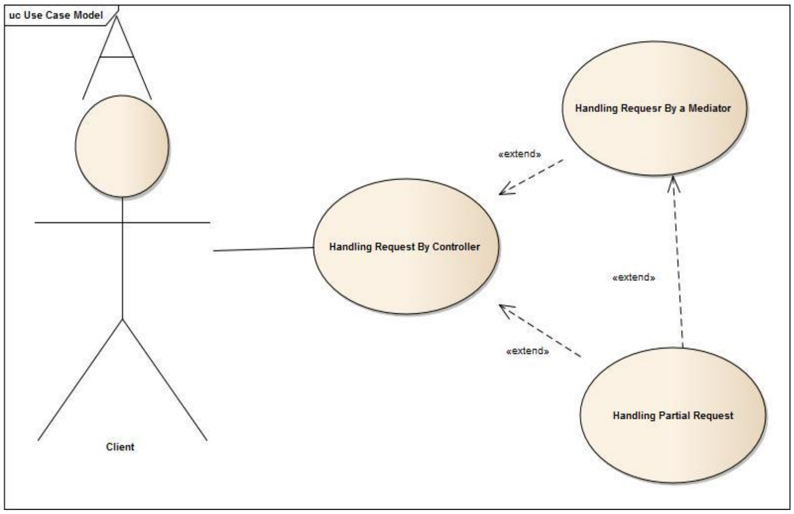


Figure 43 Chain of Responsibility Design Pattern Use Case Diagram

Figure 43 shows that Chain of responsibility has three main use cases: Handling Request by Controller, Handling Request by a Mediator, and Handling Partial Request. Each use case has a specific scenario different from the other one. Some use cases have an alternative section as well as some extends other use cases. The following are the descriptions of each use case:

Table 39 the description of Handling Request by Controller Use Case

Use Case Name	Handling Request by Controller
Actor	Client
Main Flow	1- Receive request from Client. 2- Check if the request can be done completely. 3- Response to the Client.
Alt	**a. If the request can't be done completely:** a.i Forward request to the next mediator. a.ii End.

Table 23 shows the description of the chain of responsibility use case 1: Handling Requests by Controller. The description showed the different steps of the use case in the main flow. The alt section shows the alternative scenario of the use case.

Table 40 the description of Handling Request by a Mediator Use Case

Use Case Name	Handling Request by Mediator
Actor	Client
Main Flow	1- Receive request from the controller. 2- Check if the request can be done completely. 3- Response to the Controller. 4- Response to the Client
Alt	**a. If the request can't be done completely by a Mediator:** a.i Forward request to the next mediator. a.ii End.

Table 24 shows the description of the chain of responsibility use case 2: Handling Request by Mediator. The description showed the different steps of the use case in the main flow. The alt section shows the alternative scenario of the use case.

Table 41 the description of Handling Partial Request Use Case

Use Case Name	Handling Partial Request
Actor	Client
Main Flow	1- If the request can't be handled completely by Controller or a Mediator. 2- Check if the request can be done partially by the controller. 3- Forward request to next mediator. 4- Check if the request can be done partially by a Mediator. 5- Forward request to next mediator. 6- Response to the Controller. 7- Response to the Client
Alt	**1- If the request can't be done partially:** a.i Exception Handler.

Table 25 shows the description of the chain of responsibility use case 3: Handling Partial Request. The description showed the different steps of the use case in the main flow. The alt section shows the alternative scenario of the use case.

Behavioral View Using Sequence Diagram

Each use case has its separate sequence diagram that showing the flows of scenario steps from each and to each object and actors of the use case.

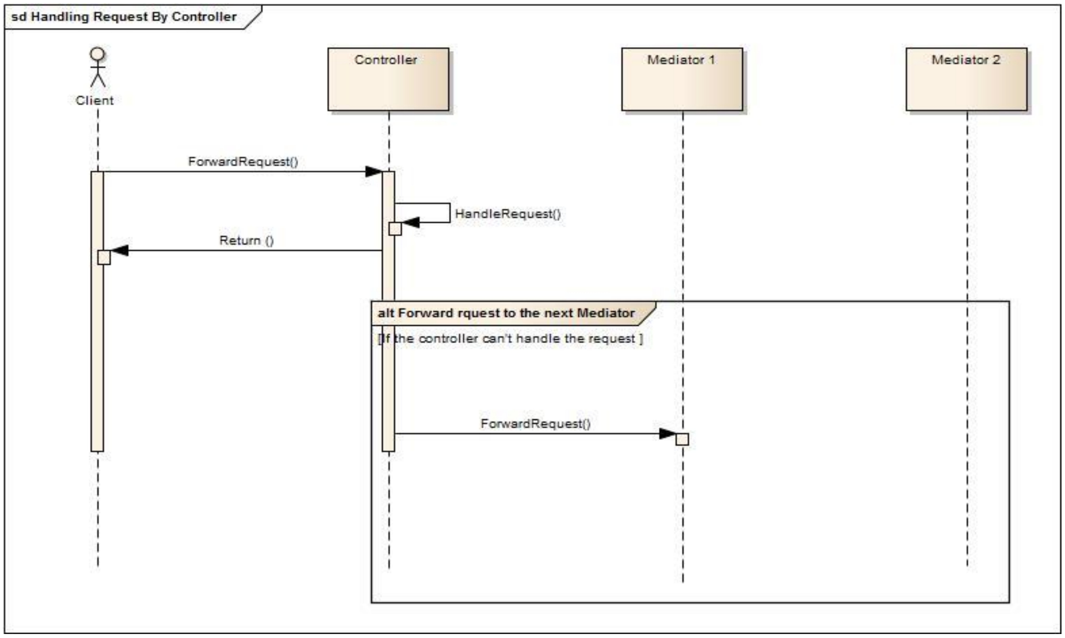

sd Handling Request By Controller
Client
Controller
Mediator 1
Mediator 2
ForwardRequest()
HandleRequest()
Return ()
alt Forward rquest to the next Mediator
[If the controller can't handle the request]
ForwardRequest()

Figure 44 the sequence diagram of the Handling Request by Controller Use Case

Figure 44 shows the sequence flow of chain of responsibility use case 1: Handling Requests by Controller. The sequence behavior between Controller, Mediator 1 and Mediator 2 are described in a sequential manner.

The alternative section sequence behavior also described and the lower part of the sequence diagram.

All the behavioral view features and how the sequence is managing between the different classes of use case 1 is shown in a visual point of view.

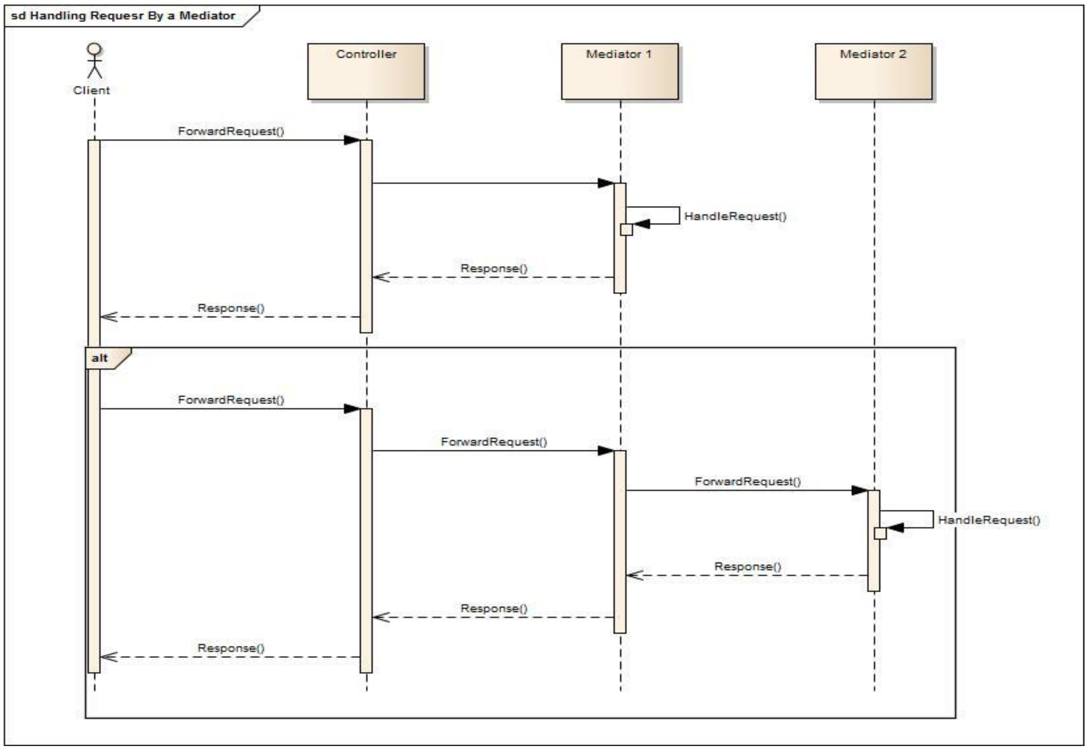


Figure 45 the sequence diagram of the Handling Request by a Mediator Use Case

Figure 45 shows the sequence flow of chain of responsibility use case 2: Handling Request by Mediator. The sequence behavior between Controller, Mediator 1 and Mediator 2 are described in a sequential manner.

The alternative section sequence behavior also described and the lower part of the sequence diagram. All the behavioral view features and how the sequence is managing between the different classes of use case 2 is shown in a visual point of view.

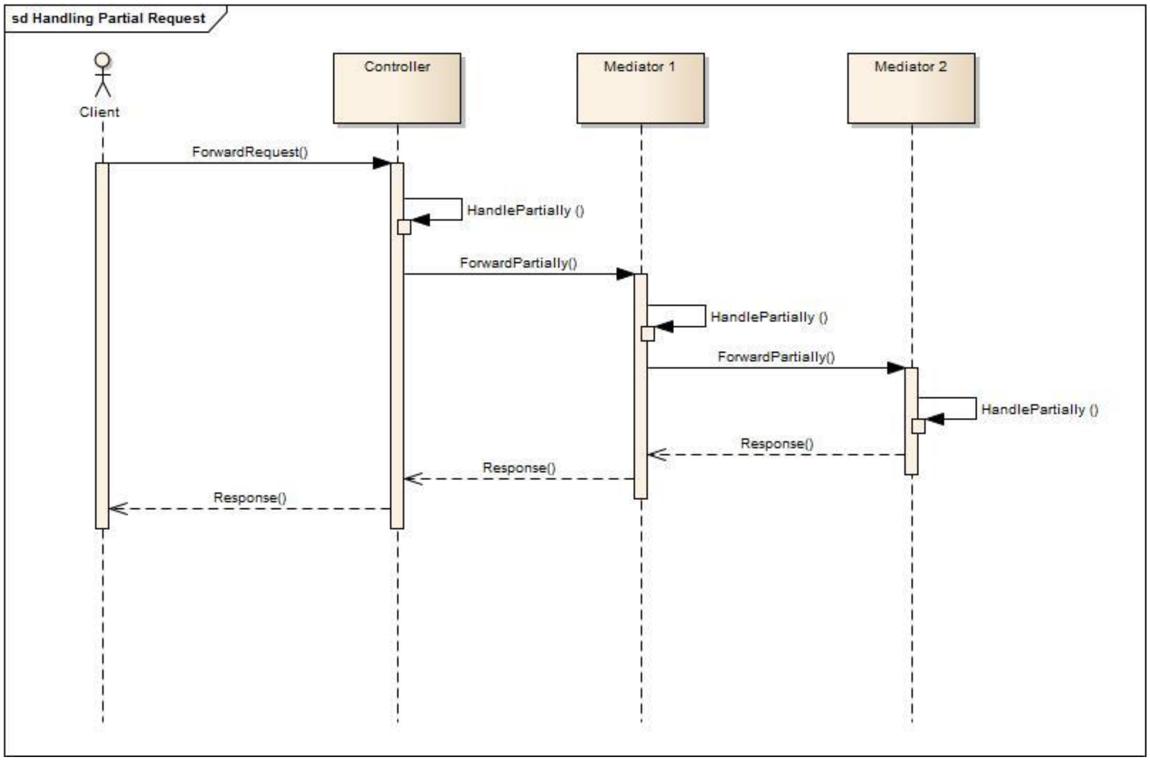


Figure 46 the sequence diagram of the Handling Partial Request Use Case

Figure 46 shows the sequence flow of chain of responsibility use case 3: Handling Partial Request. The sequence behavior between Controller, Mediator 1 and Mediator 2 are described in a sequential manner.

The alternative section sequence behavior also described and the lower part of the sequence diagram. All the behavioral view features and how the sequence is managing between the different classes of use case 3 is shown in a visual point of view.

Structural View Using Class

In the structural view of the chain of responsibility, there are no changes in the traditional representation since the structural features of a chain of responsibility have been well maintained and defined in the literature review.

The following is the class diagram of chain responsibility:

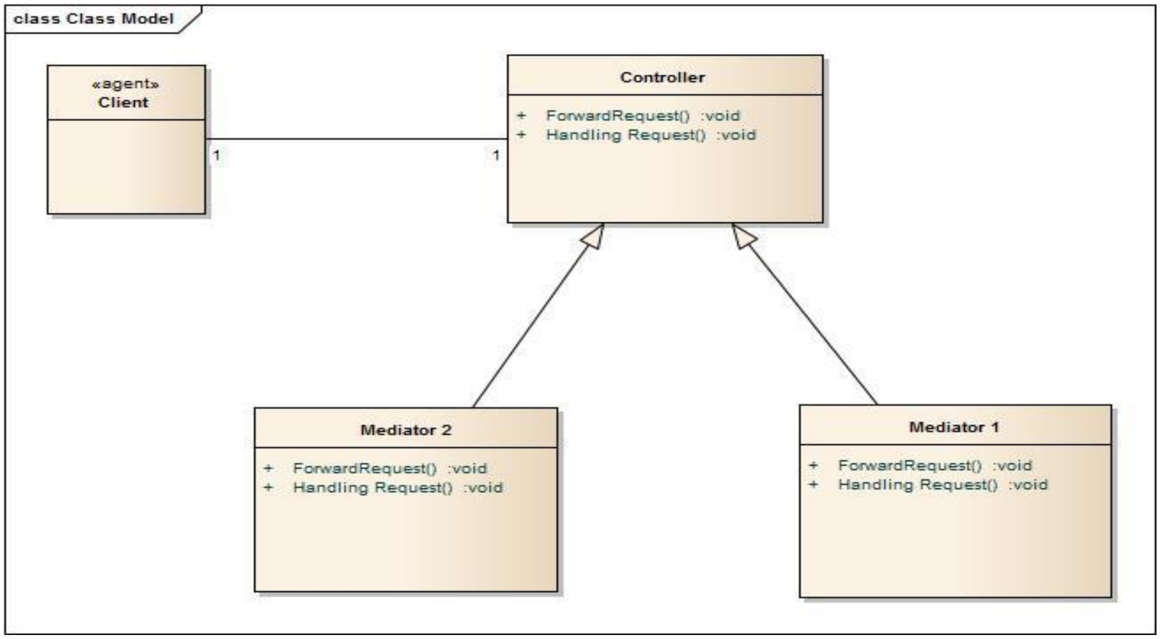


Figure 47 Class Diagram of Chain of Responsibility

Figure 47 shows the class diagram of a chain of responsibility design pattern. The class diagram illustrates all the different structural view features. The class diagram contains three classes' controller, mediator 1, and mediator 2.

Chain of responsibility UML integrated Representation

The following XML code is the UML integrated metamodel representation of the chain of responsibility design pattern. The integrated metamodel contains all the three main views features in on concrete file. See Appendix B.

❖ Observer

We will consider an observer case where we have two observers. The subject maintains the state of the object and notifies the observers whenever the state of the object changed. The observers need to register in the subjects to get the notifications. The three views representation is as follows:

Functional View Using Use Case Diagram

Similarly, as a chain of responsibility, observer design pattern use case diagram will have different uses cases, each use case will represent one possible scenario for instance. Each use case will include a use case description that explains the main scenario and the functionality of the use case.

The use case diagram of a chain of responsibility design pattern as follows:

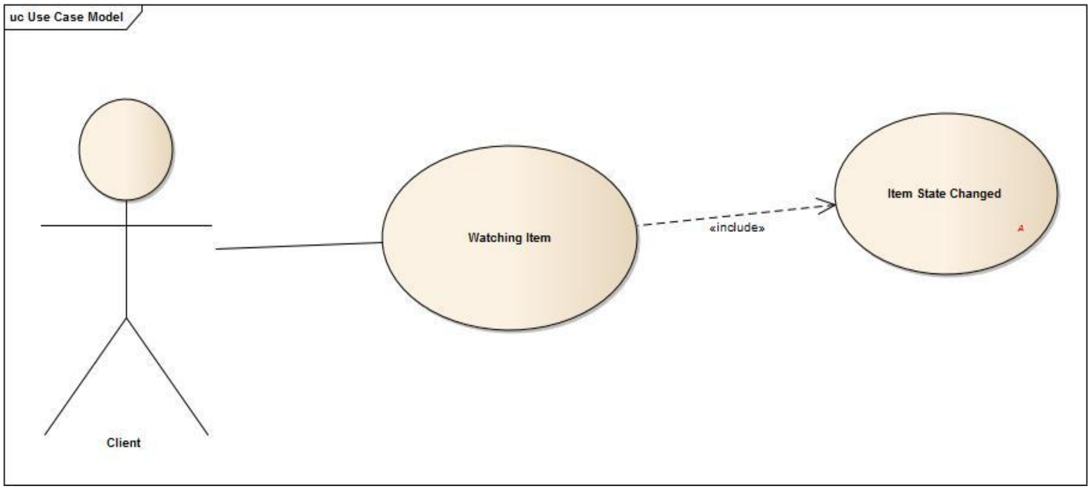


Figure 48 Observer Design Pattern Use Case Diagram

Figure 48 shows that Chain of responsibility has two main use cases: Watching Item, and Item State Changed. Each use case has a specific scenario different from the other one. Some use cases have an alternative section as well as some extends other use cases. The following are the description of each use case:

Table 42 the description of Watching Item Use Case

Use Case Name	Watching Item
Actor	Client
Main Flow	1- Register the Observers to the Subject. 2- Check the state of the Item. 3- If Item is available to set Item State =0; 4- Keep watching the Item.

	5- **If the Item State Changes:** a.i Call Use Case 2. a.ii End.

shows the description of Observer uses case 1: Watching Item. The description showed the different steps of the use case in the main flow.

Table 43 the description of Item State Changed Use Case

Use Case Name	Item State Changed
Actor	Client
Main Flow	1- Notify the Observers. 2- Send Information to the Observers. 3- Update the State of the Item;

Table 43 shows the description of Observer uses case 2: Item State Changed. The description showed the different steps of the use case in the main flow.

Behavioral View Using Sequence Diagram

Each use case has its separate sequence diagram that showing the flows of scenario steps from each and to each object and actors of the use case.

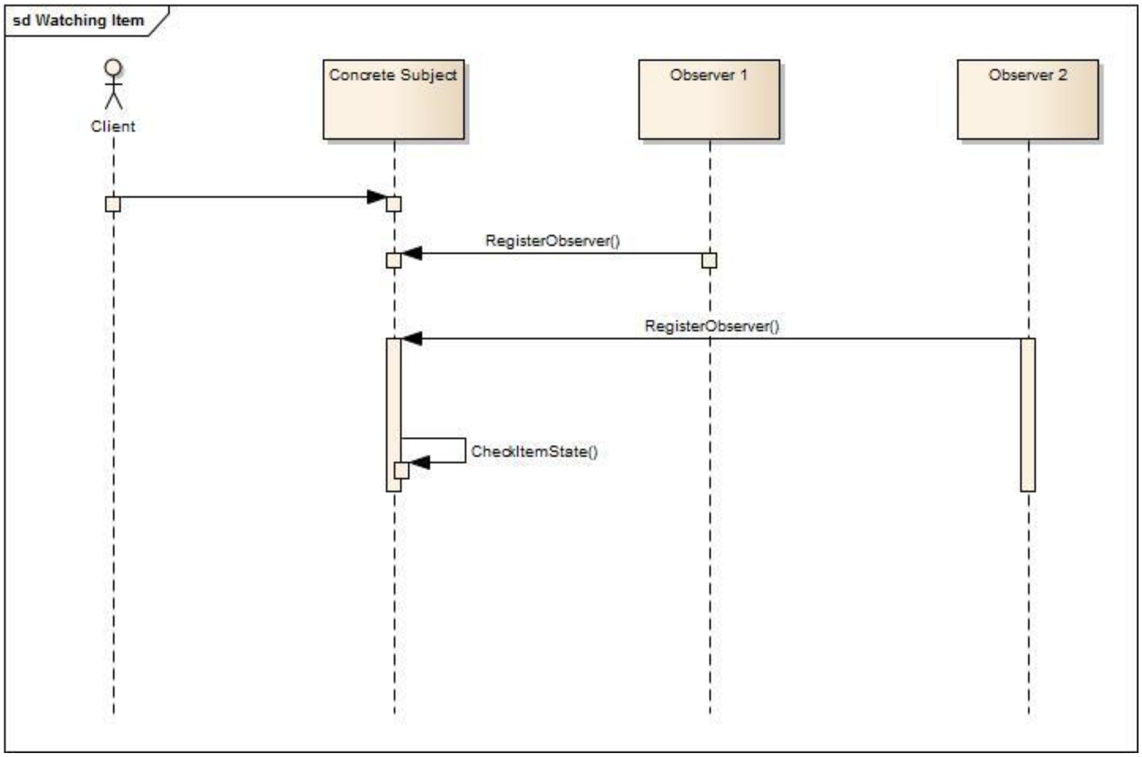


Figure 49 the sequence diagram of Watching Item Use Case

Figure 49 shows the sequence flow of Observer uses case 1: Watching Item. The sequence behavior between concrete subject, observer 1 and observer 2 are described in a sequential manner.

All the behavioral view features and how the sequences are managing between the different classes of use case 1 is shown in a visual point of view.

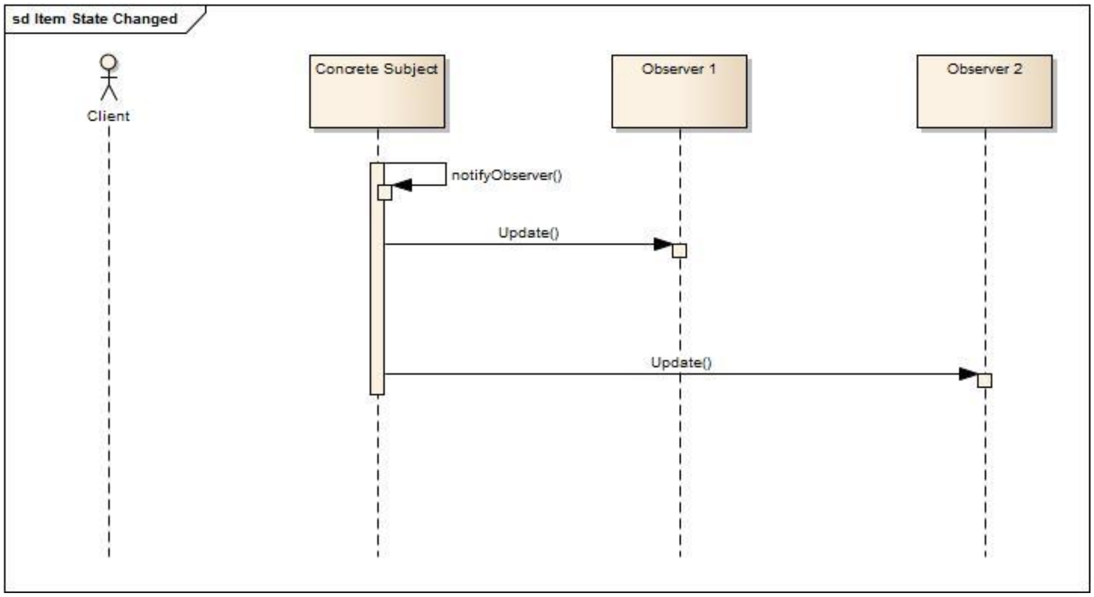


Figure 50 the sequence diagram of Item State Changed Use Case

Figure 50 shows the sequence flow of Observer uses case 2: Item State Changed. The sequence behavior between concrete subject, observer 1 and observer 2 are described in a sequential manner.

All the behavioral view features and how the sequence is managing between the different classes of use case 1 is shown in a visual point of view.

Structural View Using Class

In the structural view of Observer design pattern, we have made some changes to the traditional representation to support more than two observers and to cover all the different implementations of the observer.

The following is the class diagram of chain responsibility:

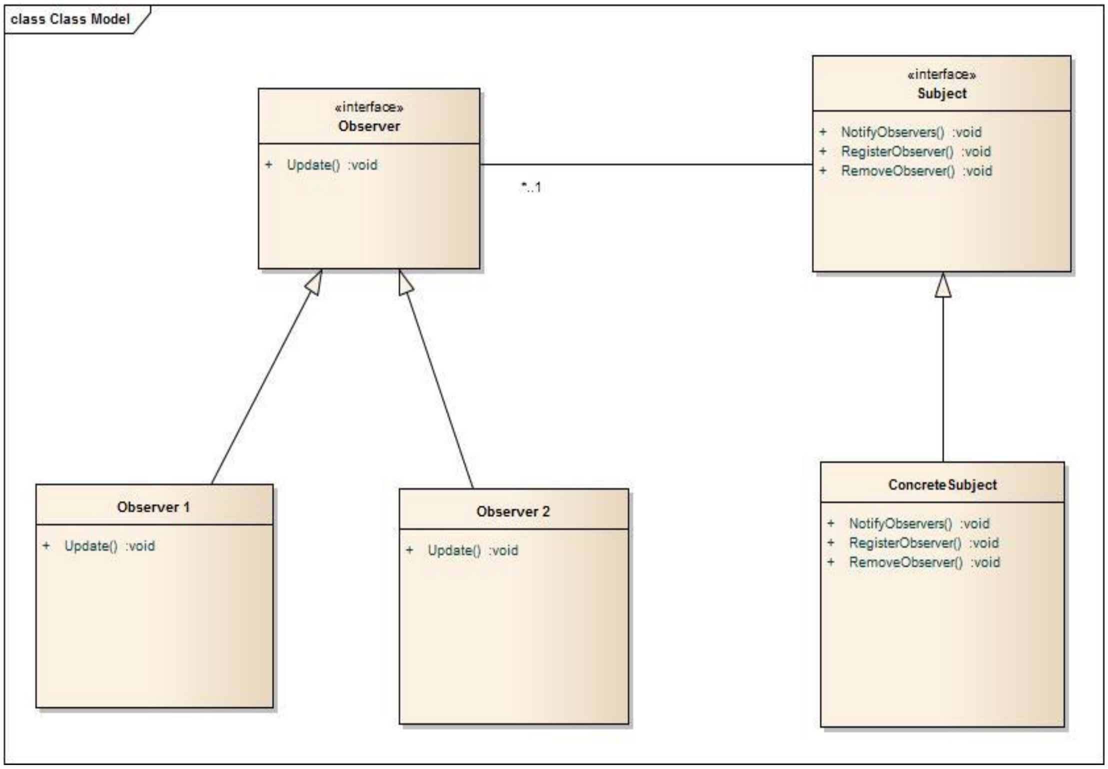


Figure 51 the class diagram of Item Observer Design Pattern

Figure 51 shows the class diagram of a chain of responsibility design pattern. The class diagram illustrates all the different structural view features. The class diagram contains three classes Subject, ConcreteSubject, Observer, Observer 1, and Observer 2. Observer 1 and Observer 2 inherits Observer, while ConcreteSubject inherits Subject. Subject uses Observer 1…* multiplicity.

Observer UML integrated Representation

The following XML code is the UML integrated metamodel representation of observer design pattern. The integrated metamodel contains all the three main view features in the concrete file. See Appendix B.

❖ Strategy

We will consider one use case for Strategy design pattern. Strategy pattern is used to have multiple algorithms for a specific task and client decides the actual implementation to be used at runtime. The Strategy design pattern attempts to solve the issue where you need to provide multiple solutions for the same problem so that one can be selected at runtime. The three views representation is as follows:

Functional View Using Use Case Diagram

Similarly, as what we have done with previous design patterns, strategy design pattern we will have use case diagram that describes the functionality of strategy design pattern, the use case will describe the main functionality of strategy design pattern.

The use case diagram of strategy design pattern as follows:

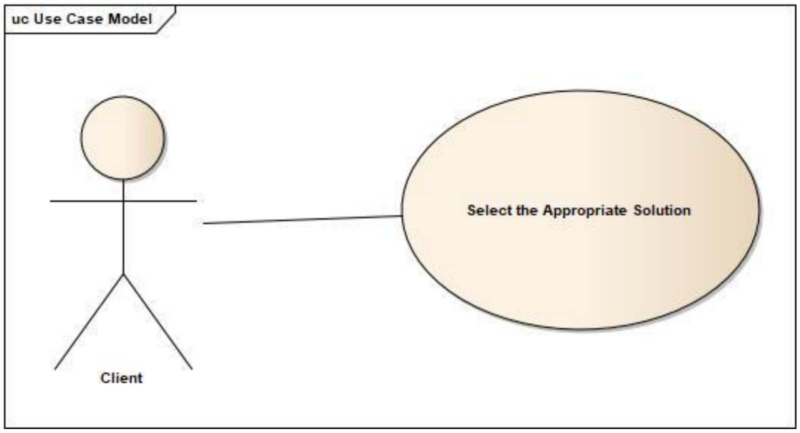


Figure 52 Strategy Design Pattern Use Case Diagram

Figure 52 shows that Strategy design pattern has one main use case: Select the appropriate solution. This main use case represents the main flow of strategy. The following are the description of each use case:

Table 44 the description of Select the Appropriate Solution Use Case

Use Case Name	Select the Appropriate Solution
Actor	Client
Main Flow	1- The context takes the request from the clients. 2- The context forwards the request to its concrete strategy classes. 3- The context choose the appropriate solution. 4- The

Table 44 shows the description of Strategy main uses case: Select the appropriate solution. The description showed the different steps of the use case in the main flow.

Behavioral View Using Sequence Diagram

Each use case has its separate sequence diagram that showing the flows of scenario steps from each and to each object and actors of the use case.

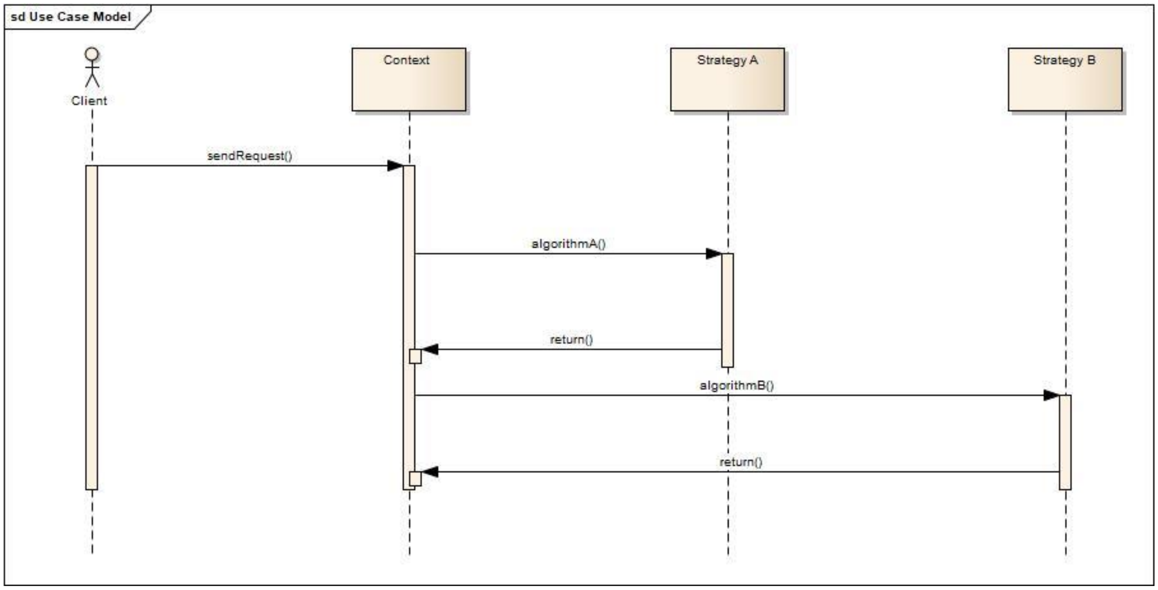


Figure 53 Strategy Design Pattern Sequence Diagram

Figure 53 shows the sequence flow of Strategy main uses case: Select the appropriate solution. The sequence behavior between the context and the two different concrete classes

Strategy A and B shows how the flow of action happens where the context choose the appropriate solution at runtime.

Structural View Using Class

In the structural view of Strategy design pattern, the class diagram similar to the traditional class diagram. Where we have the client and the context classes that have many different strategy concrete classes.

The following is the class diagram of Strategy:

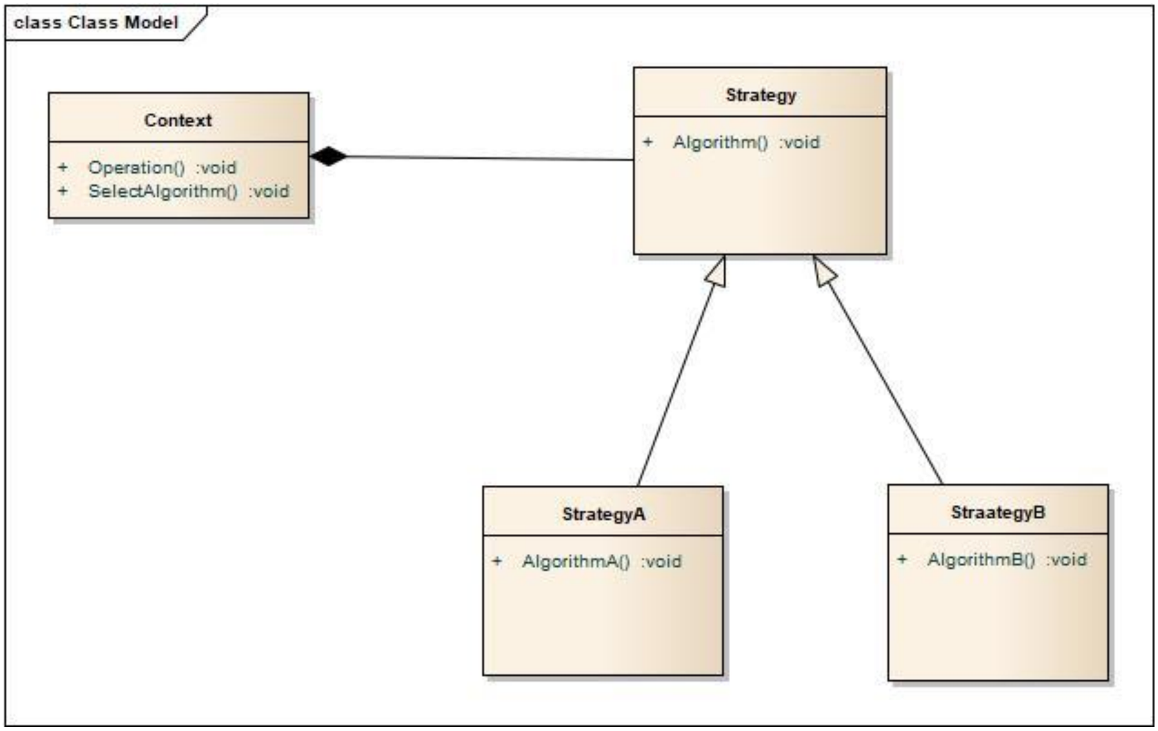


Figure 54 the class diagram of Strategy Design Pattern

Figure 54 shows the class diagram of Strategy design pattern. The class diagram illustrates all the different structural view features. The class diagram contains four classes: Context, Strategy, Strategy A and Strategy B. Strategy A and Strategy B inherit Strategy in order to implement the appropriate solution for the context, while context composes the strategy classes to decide among different solutions.

Strategy UML integrated Representation

The UML integrated metamodel representation of Strategy design pattern, See Appendix B, contains all the three main view features in concrete file.

❖ Mediator

We will consider one use case for Mediator design pattern. Mediator design pattern is used to reduce the communication complexity between multiple objects. Mediator provides a mediator object which normally handles all the communications between different objects. Mediator design pattern can be considered as a communication center for the objects when an object needs to communicate with another object it does not call the other object directly. Instead, it calls the mediator object whose main duty is to route the messages to the destination object. . The three views representation is as follows:

Functional View Using Use Case Diagram

Similarly, as what we have done with previous design patterns, Mediator design pattern we will have use case diagram that describes its functionality, the use case will describe the main functionality of strategy design pattern.

The use case diagram of a chain of responsibility design pattern as follows:

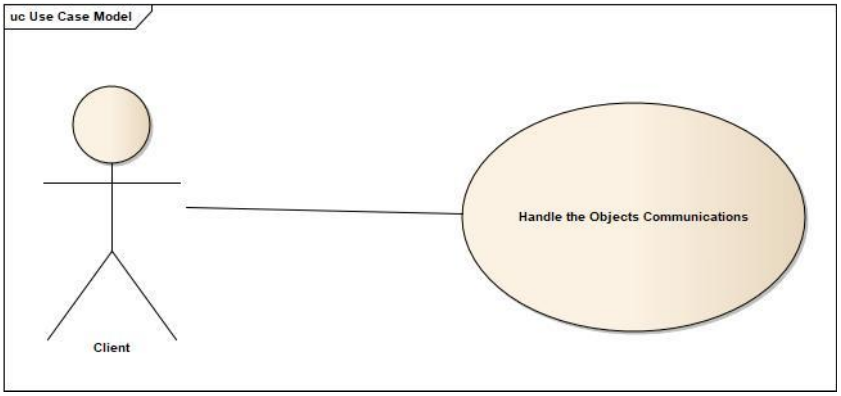


Figure 55 Mediator Design Pattern Use Case Diagram

Figure 55 shows that Mediator design pattern has one main use case: Handle the Objects Communications. This main use case represents the main flow of strategy. The following are the description of each use case:

Table 45 The Description of Handle the Objects Communications Use Case

Use Case Name	Handle the Objects Communications
Actor	Client
Main Flow	1- The colleagues send a request to the mediator. 2- The mediator handles the request. 3- The mediator implements the cooperative behavior and routing the request to the appropriate colleagues. 4- The colleagues receive the results from the mediator.

Table 45 shows the description of Mediator main uses case: Handle the Objects Communications. The description showed the different steps of the use case in the main flow.

Behavioral View Using Sequence Diagram

Each use case has its separate sequence diagram that showing the flows of scenario steps from each and to each object and actors of the use case.

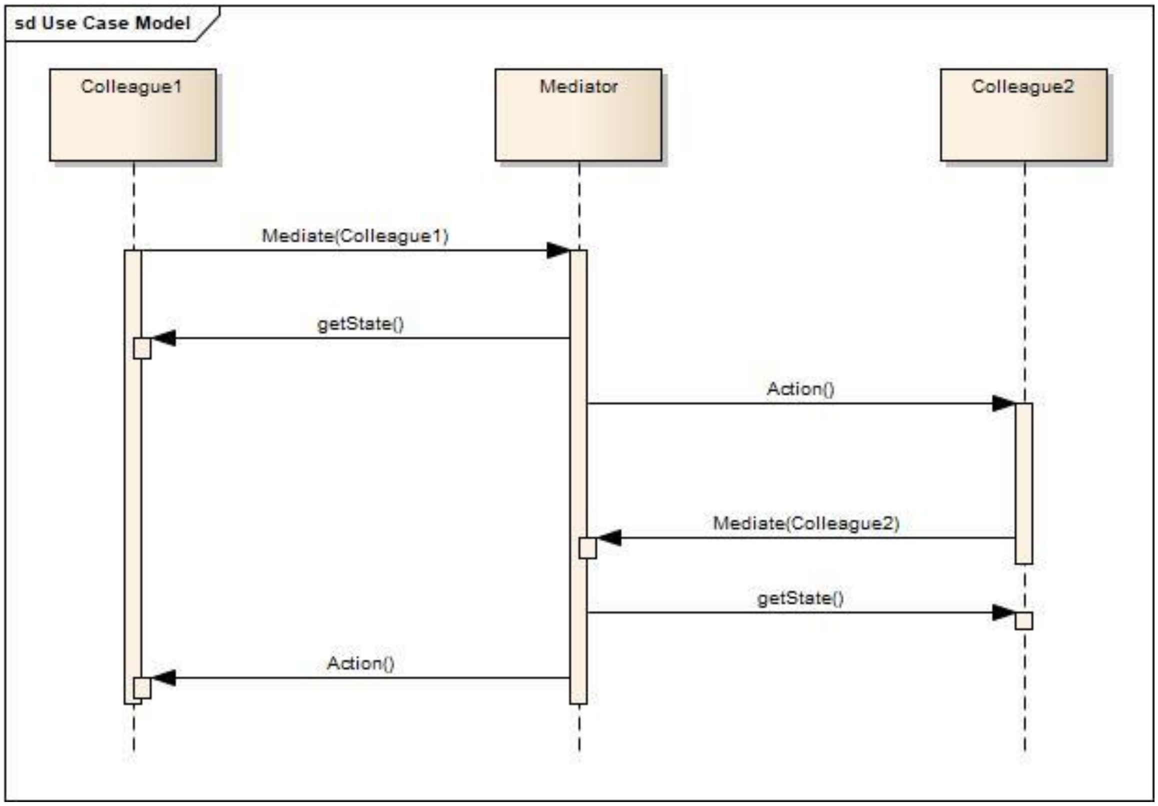


Figure 56 Mediator Design Pattern Sequence Diagram

Figure 56 shows the sequence flow of Mediator main uses case: Handle the Objects Communications. The sequence behavior between the Mediator and the two different concrete classes Colleague1 and 2, shows how the flow of action happens where the mediator handles and maintain the communications between different colleagues.

Structural View Using Class

In the structural view of Mediator design pattern, the class diagram similar to the traditional class diagram. Where we have the client and the context classes that have many different colleagues classes.

The following is the class diagram of Mediator:

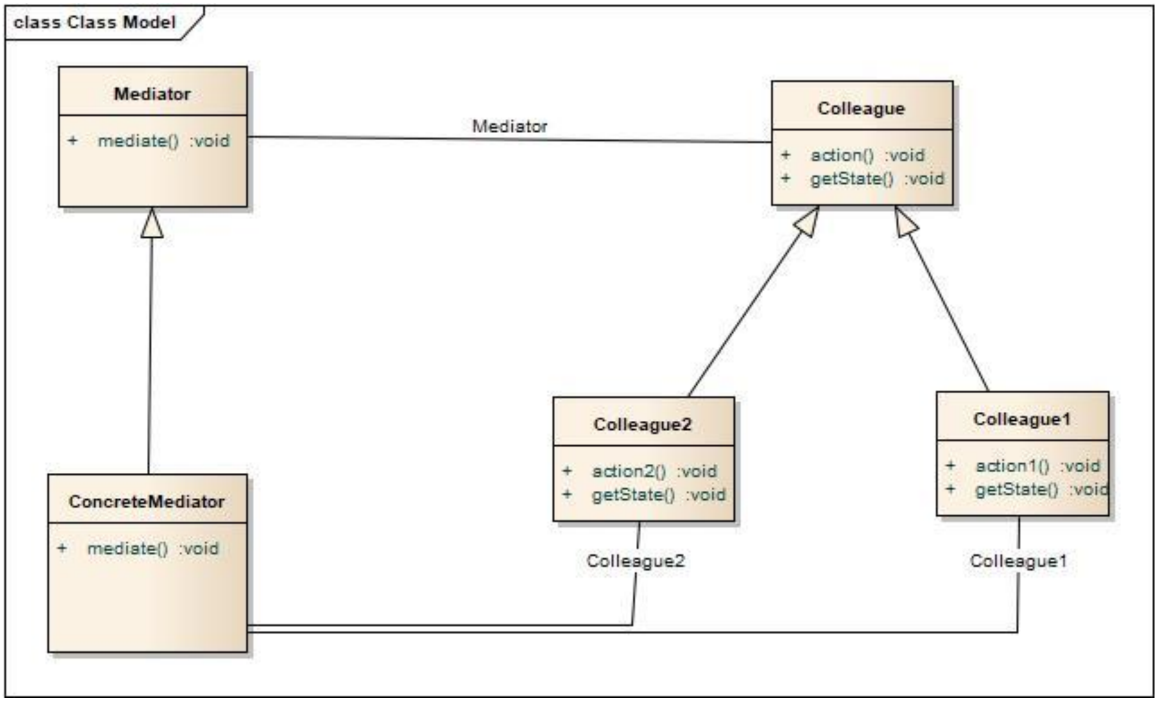


Figure 57 Mediator Design Pattern Class Diagram

Figure 57 shows the class diagram of Mediator design pattern. The class diagram illustrates all the different structural view features. The class diagram contains five classes: Mediator, ConcreteMediator, Colleague, Colleague1, and Colleague2. Colleague1 and Colleague2 inherit Colleague, it defines the interface for communication with other Colleagues, while ConcreteMediator implements the Mediator interface and coordinates communication between Colleague objects. It is aware of all of the Colleagues and their purposes with regards to inter-communication. It defines the interface for communication between Colleague objects

Strategy UML integrated Representation

The UML integrated metamodel representation of Mediator design pattern, See Appendix B, contains all the three main view features in the concrete file.

❖ State

We will consider one use case for State design pattern. State design pattern allows an object to alter its behavior when its internal state changes. Mediator allows an object to completely change its behavior depending upon its current internal state.

The three views representation is as follows:

Functional View Using Use Case Diagram

Similarly, as what we have done with previous design patterns, State design pattern we will have use case diagram that describes its functionality, the use case will describe the main functionality of strategy design pattern.

The use case diagram of state design pattern as follows:

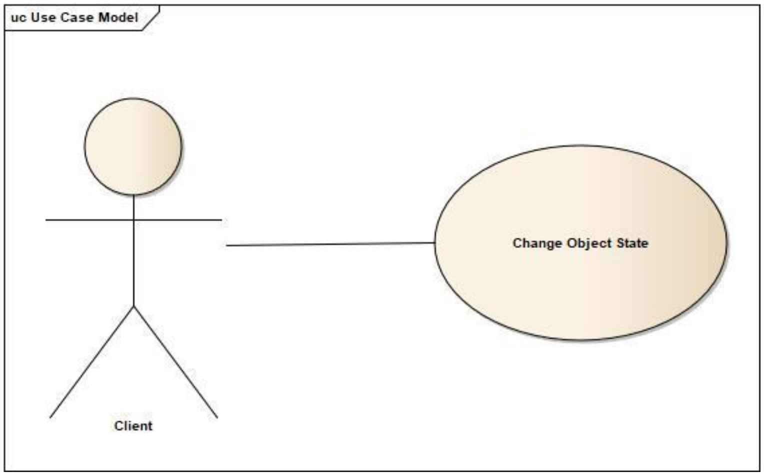


Figure 58 State Design Pattern Use Case Diagram

Figure 58 shows that State design pattern has one main use case: Change Object State. This main use case represents the main flow of strategy. The following are the description of each use case:

Table 46 the Description of Change Object State Use Case

Use Case Name	The Description of Change Object State Use Case
Actor	Client
Main Flow	1- The Context keeps the state of the object not changed. 2- If the internal state of the object changed then the context requests to change the state of the object based on the current state. 3- The context returns to the default state.

Table 46 shows the description of State main uses case: Change Object State. The description showed the different steps of the use case in the main flow.

Behavioral View Using Sequence Diagram

Each use case has its separate sequence diagram that showing the flows of scenario steps from each and to each object and actors of the use case.

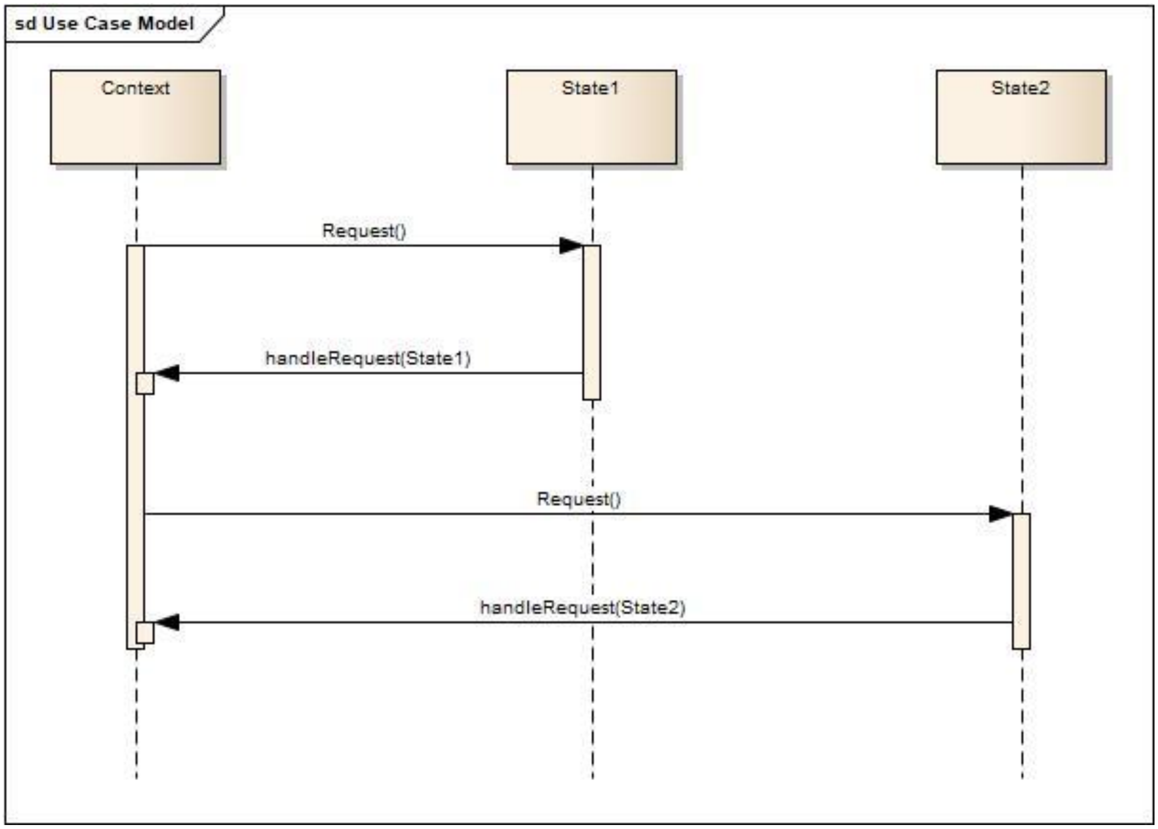


Figure 59 State Design Pattern Sequence Diagram

Figure 59 shows the sequence flow of State main use case: Change the object state. The sequence behavior between the Context and the two different concrete classes State1 and 2, shows how the flow of action happens if the object state changed at runtime.

Structural View Using Class

In the structural view of State design pattern, the class diagram similar to the traditional class diagram. Where we have the context class and the different state classes.

The following is the class diagram of State:

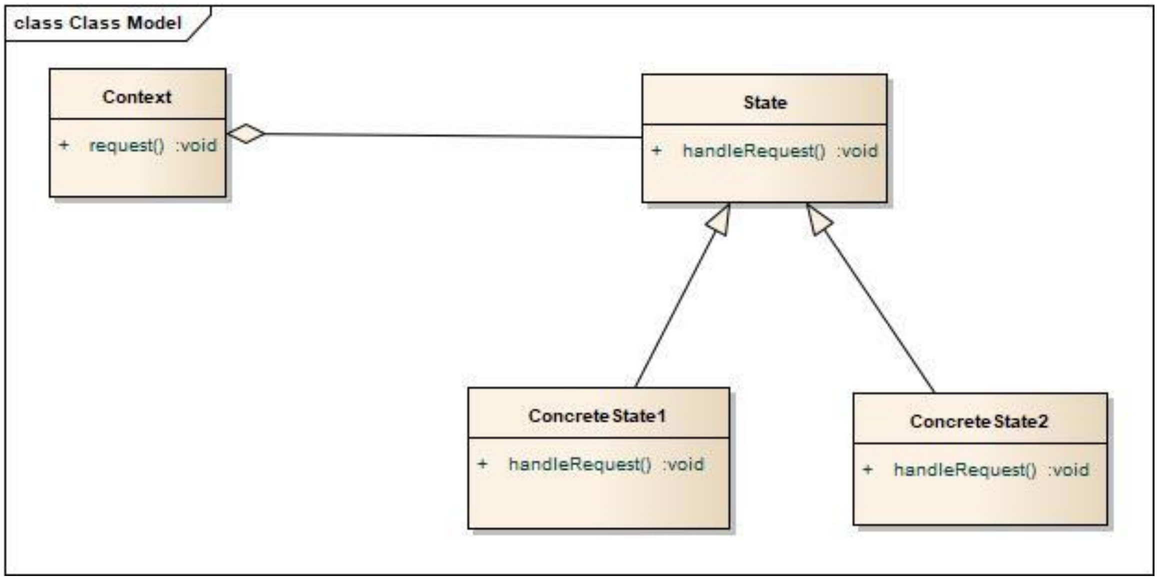


Figure 60 State Design Pattern Class Diagram

Figure 60 shows the class diagram of State design pattern. The class diagram illustrates all the different structural view features. The class diagram contains four classes: Context, State, and two concrete State classes. Context class maintains an instance of a ConcreteState subclass that defines the current state. State class defines an interface for encapsulating the behavior associated with particular state of the Context. ConcreteState subclasses each subclass implements a behavior associated with a state of the Context.

State UML integrated Representation

The UML integrated metamodel representation of State design pattern, See Appendix B, contains all the three main view features in on concrete file.

❖ Visitor

We will consider one use case for Visitor design pattern. Visitor is used to represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

The three views representation is as following:

Functional View Using Use Case Diagram

Similarly, as what we have done with previous design patterns, Visitor design pattern we will have use case diagram that describes its functionality, the use case will describe the main functionality of strategy design pattern.

The use case diagram of Visitor design pattern as following:

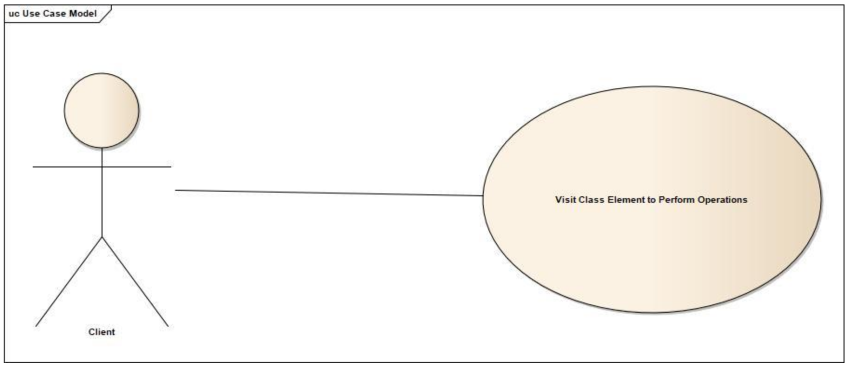


Figure 61 Visitor Design Pattern Use Case Diagram

Figure 61 shows that Visitor design pattern has one main use case: Visit Class Elements to Perform Operations. This main use case represents the main flow of Visitor. The following are the description of the use case:

Table 47 the Description of Change Object State Use Case

Use Case Name	Visit Class Elements to Perform Operations
Actor	Client
Main Flow	1- The Visitor sends a request to visit class elements. 2- The Elements should approve the Visitor. 3- The Visitor traces each class elements to perform specific operations. 4- End.

Table 47 shows the description of Visitor main uses case: Visit Class Elements to Perform Operations. The description showed the different steps of the use case in the main flow.

Behavioral View Using Sequence Diagram

Each use case has its separate sequence diagram that showing the flows of scenario steps from each and to each object and actors of the use case.

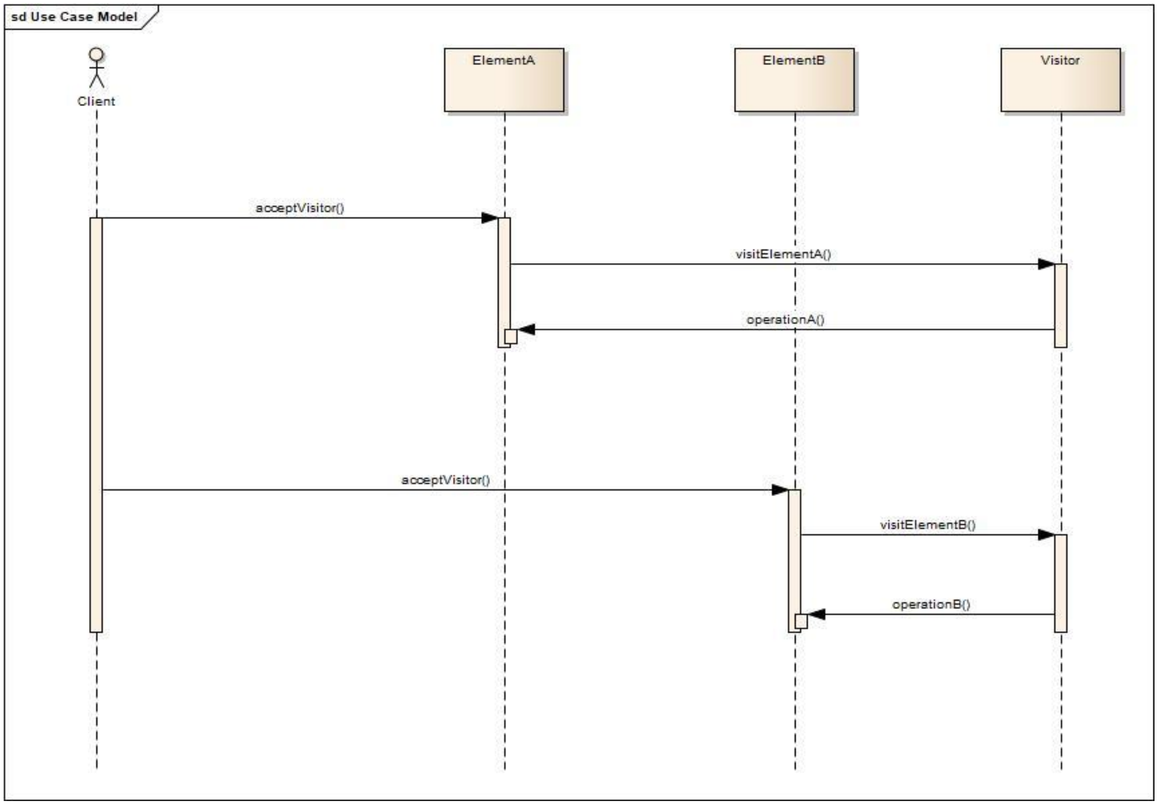


Figure 62 Visitor Design Pattern Sequence Diagram

Figure 62 shows the sequence flow of Visitor main use case: Visit Class Elements to Perform Operations. The sequence behavior between the Visitor and the two different concrete Elements A and B shows how the flow of action happens if the Visitor object wants to perform operations with class elements without altering it is definition.

Structural View Using Class

In the structural view of Visitor design pattern, the class diagram similar to the traditional class diagram. We will have a class diagram with two deferent class elements and one concrete visitor.

The following is the class diagram of visitor:

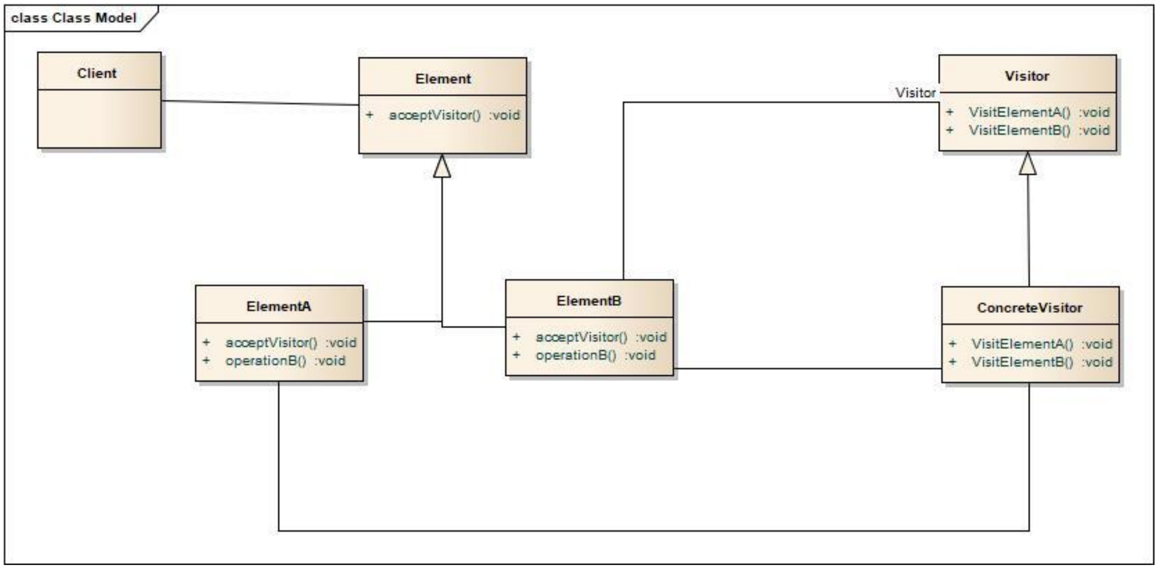


Figure 63 Visitor Design Pattern Class Diagram

Figure 63 shows the class diagram of Visitor design pattern. The class diagram illustrates all the different structural view features. The class diagram contains four classes: Visitor declares a Visit operation for each class of ConcreteElement in the object structure. The operation's name and signature identify the class that sends the Visit request to the visitor. That lets the visitor determine the concrete class of the element being visited.

Then the visitor can access the element directly through its particular interface. ConcreteVisitor implements each operation declared by Visitor. Each operation implements a fragment of the algorithm defined for the corresponding class of object in the structure. ConcreteVisitor provides the context for the algorithm and stores its local state. This state often accumulates results during the traversal of the structure. Element defines an Accept operation that takes a visitor as an argument. ConcreteElement implements an Accept operation that takes a visitor as an argument.

Visitor UML integrated Representation

The UML integrated metamodel representation of Visitor design pattern, See Appendix B, contains all the three main view features in the concrete file.

❖ Template Method

We will consider one use case for the Template Method design pattern. Template Method design pattern Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

The three views representation is as follows:

Functional View Using Use Case Diagram

Similarly, as what we have done with previous design patterns, Template Method design pattern we will have use case diagram that describes its functionality, the use case will describe the main functionality of Template Method design pattern.

The use case diagram of Template Method design pattern as follows:

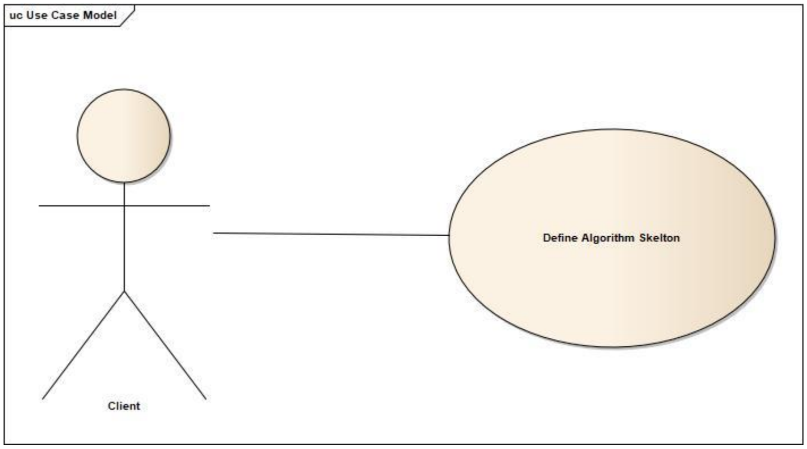


Figure 64 Template Method Design Pattern Use Case Diagram

Figure 64 shows that Template Method design pattern has one main use case: Define Algorithm Skelton. This main use case represents the main flow of Template Method. The following are the description of each use case:

Table 48 the Description of Define Algorithm Skelton Use Case

Use Case Name	Define Algorithm Skelton
Actor	Client
Main Flow	1- The AbstractClass defines the sequence of the ConcreteClass operations in the TemplateMethod. 2- The ConcreteClass follows the sequences of performing the operations as per defined in the TemplateMethod. 3- Ends.

Table 48 shows the description of Template Method main uses case: Define Algorithm Skelton. The description showed the different steps of the use case in the main flow.

Behavioral View Using Sequence Diagram

Each use case has its separate sequence diagram that showing the flows of scenario steps from each and to each object and actors of the use case.

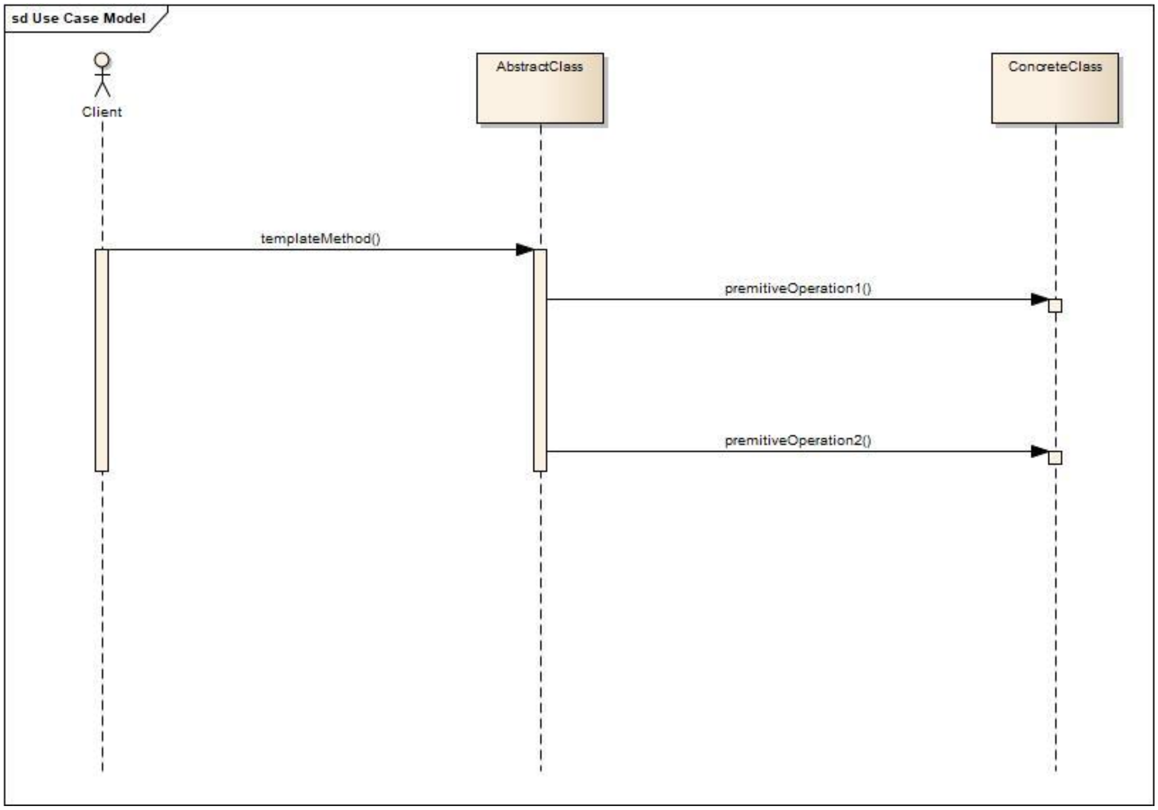


Figure 65 Template Method Design Pattern Sequence Diagram

Figure 65 shows the sequence flow of Template Method main use case: Define Algorithm Skelton. The sequence behavior between the AbstractClass and the ConcreteClass, where the ConcreteClass performing the operations based on the sequence identified in the TemplateMethod.

Structural View Using Class

In the structural view of Template Method design pattern, the class diagram similar to the traditional class diagram. Where we have the AbstractClass and the ConcreteClasses.

The following is the class diagram of Template Method:

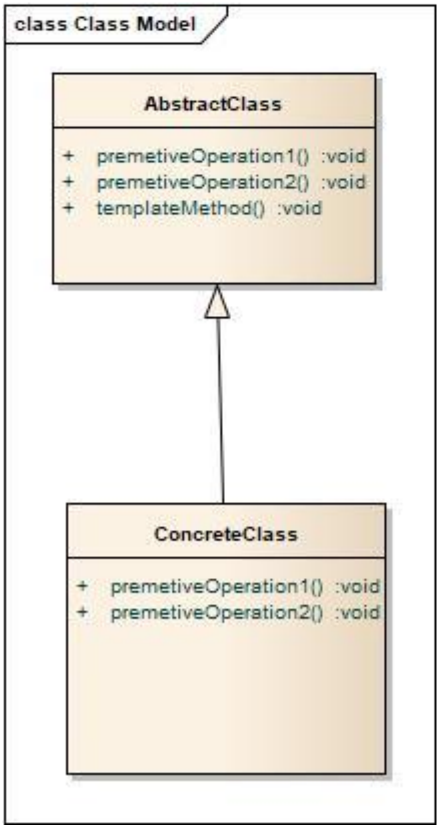

class Class Model
AbstractClass
+ premetiveOperation1() :void
+ premetiveOperation2() :void
+ templateMethod() :void
ConcreteClass
+ premetiveOperation1() :void
+ premetiveOperation2() :void

Figure 66 Template Method Design Pattern Class Diagram

Figure 66 shows the class diagram of the Template Method design pattern. The class diagram illustrates all the different structural view features. AbstractClass defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm. It implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects. ConcreteClass implements the primitive operations to carry out subclass specific steps of the algorithm.

Template Method UML integrated Representation

The UML integrated metamodel representation of Template Method design pattern, See Appendix B, contains all the three main view features in the concrete file.

❖ Command

We will consider one use case for Command design pattern. Command design pattern encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

The three views representation is as follows:

Functional View Using Use Case Diagram

Similarly, as what we have done with previous design patterns, Command design pattern we will have use case diagram that describes its functionality, the use case will describe the main functionality of Command design pattern.

The use case diagram of Command design pattern as follows:

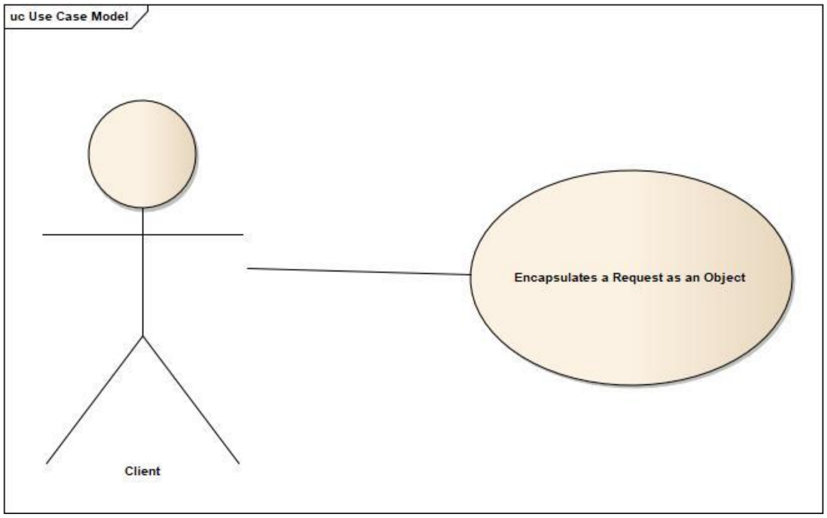


Figure 67 Command Design Pattern Use Case Diagram

Figure 67 shows that Command design pattern has one main use case: Encapsulate a Request as an Object. This main use case represents the main flow of Command. The following are the description of each use case:

Table 49 the Description of Encapsulate a Request as an Object Use Case

Use Case Name	Encapsulate a Request as an Object
Actor	Client
Main Flow	1- The client creates a ConcreteCommand object and specifies its receiver. 2- An Invoker object stores the ConcreteCommand object. 3- The invoker issues a request by calling Execute on the command. When commands are undoable, ConcreteCommand stores state for undoing the command prior to invoking Execute. 4- The ConcreteCommand object invokes operations on its receiver to carry out the request. 5- End.

Table 49 shows the description of Command main uses case: Encapsulate a Request as an Object. The description showed the different steps of the use case in the main flow.

Behavioral View Using Sequence Diagram

Each use case has its separate sequence diagram that showing the flows of scenario steps from each and to each object and actors of the use case.

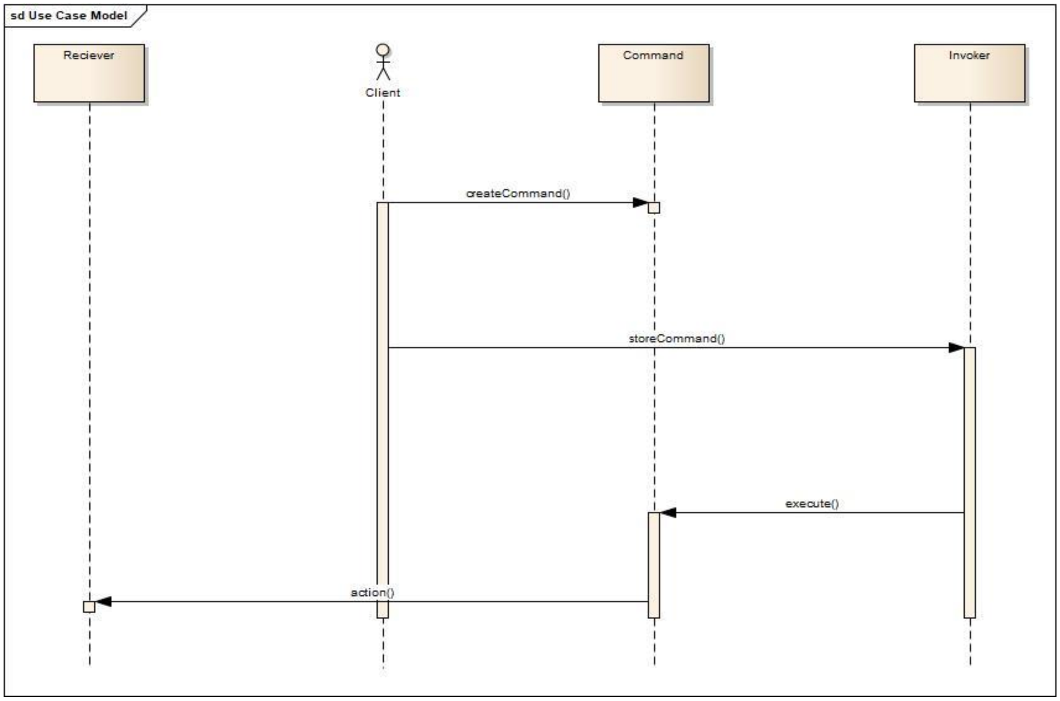


Figure 68 Command Design Pattern Sequence Diagram

Figure 68 shows the sequence flow of Command main use case: Encapsulate a Request as an Object. The sequence behavior between the Command, Invoker and receiver when the command encapsulates a request the invoker and then the request executed by the receiver.

Structural View Using Class

In the structural view of Command design pattern, the class diagram similar to the traditional class diagram. Where we have the three main classes Command, Invoker, and A Receiver.

The following is the class diagram of Command:

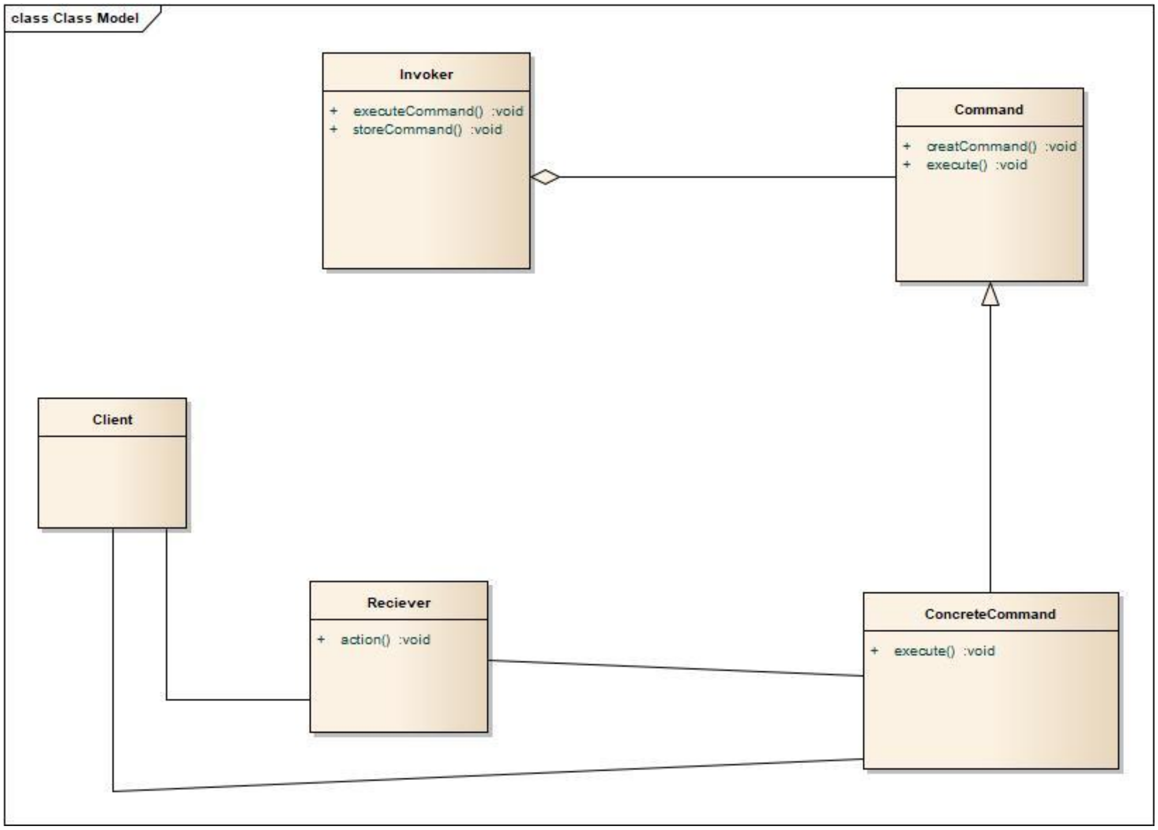


Figure 69 Command Method Design Pattern Class Diagram

Figure 69 shows the class diagram of Command design pattern. The class diagram illustrates all the different structural view features. Command declares an interface for executing an operation. ConcreteCommand defines a binding between a Receiver object and an action. It implements Execute by invoking the corresponding operation(s) on Receiver. The Client creates a ConcreteCommand object and sets its receiver. Invoker asks the command to carry out the request. The Receiver knows how to perform the operations associated with carrying out a request. Any class may serve as a Receiver.

Command UML integrated Representation

The UML integrated metamodel representation of Command design pattern, See Appendix B, contains all the three main view features in concrete file.

CHAPTER 5

VALIDATION

5.1 Visual Validation

In this section, we are going to validate our proposed technique by two main ways, the visual validation, and the automatic validation to see if the integrated metamodel gives more information and better accuracy to the design pattern detection process. Before starting discussing the validation process, we are going to elaborate on how to use the integrated UML metamodel to represent design pattern.

Choosing a good representation model of design patterns is the crucial part of any design pattern detection technique or methodology since the representation of design patterns will decide how accurate and beneficial the detection technique is. Different representation forms have been used in the literature review like XML, Ontology-Based or text. In our proposed technique we are going to use the integrated metamodel to represent the design patterns, we chose two design patterns (Chain of Responsibility, and Observer).

To show a clear picture of how the design patterns were defined and represented and compare it with our study using the UML integrated metamodel. We will discuss the four design patterns in two cases the traditional case as the design pattern have been defined and represented in the literature and how we defined and represented the design patterns using the integrated metamodel representations. We will also show the XML representation of both the traditional design pattern representation and the representation using the UML integrated metamodel. For the traditional representation and definition of the design patterns, we are going to consider Gang of Four (GoF) [9] definition of Design patterns. The traditional representation and definition of design patterns are considering only the behavioral and Structural views. Mostly design patterns were represented and defined in the literature using the class diagram and sequence diagram. While in our technique we are representing the design patterns with the three main views the Functional, Behavioral and Structural using the Use case diagram, sequence diagram and class diagram.

The XML representations of the traditional form of design patterns will be considered separately for each view using the Enterprise Architect framework. In our representation form, we used the IntegraUML tool proposed by [3, 24] to produce the design pattern XML representation.

The visual validation process will be established by comparing the separated XML source code of each design pattern class and sequence diagrams in the traditional definition of the XML source code of the integrated metamodel of the design pattern use case, sequence and class diagram to show that the UML integrated metamodel gives more information for the design pattern to be detected and discovered, as well as the integrated metamodel gathers all the design pattern functional, behavioral, and structural features in one concrete XML file which will help to reduce the false-positive and true-negative.

We also developed a tool that detects the design patterns from a UML integrated metamodel file. The tool detects design patterns based on its specification since each design pattern has different functional, behavioral, and structural features that distinguish it from the other design patterns. The design pattern specifications were taken from the design pattern definition and representation.

We developed also a Pseudocode for each design pattern instances and then we converted it to a hard code using Java programming language.

We conducted the manual validation of our proposed technique by comparing two different cases, the traditional representation case with our proposed case using the integrated metamodel. Where in the traditional case when we look at a separate view of a design such as a behavioral view, it might look like a design pattern but when we check the other view such as structural view; we realized that it is not actually a design pattern.

The reason behind that is the traditional representation each view is considered separately from other views, they are not integrated or connected by any case. In the other way around, using the UML integrated metamodel, since all the three views are integrated and connected, gives more information about the design pattern Functional, Behavioral, and Structural features in one integrated file.

Converting each view separately to its metamodel will reduce the other features of the other views. For instance, converting the behavioral view (sequence diagram) to the integrated metamodel will only give the behavioral features of the design pattern, and it will give no clue about what are the structural features of that design pattern. Similarly converting the structural view (Class Diagram) will only give the structural features of the design pattern without mention anything about the behavioral features.

Using the integrated metamodel to represent the design pattern will concrete all the different views features in one single place. All the functional, Behavioral and Structural feature are connected and integrated into one single file, which will reduce the false-positive and true-negative of design pattern detection.

We will give examples for each studied design patterns were looking to a single view might look like a design pattern but checking the other view will show that it is not a design pattern.

5.1.1 Case 1

In this case, we will look at a sequence diagram and its integrated metamodel that looks like a chain of responsibility design pattern, but when we investigated the class diagram, it shows that this is not actually a design pattern.

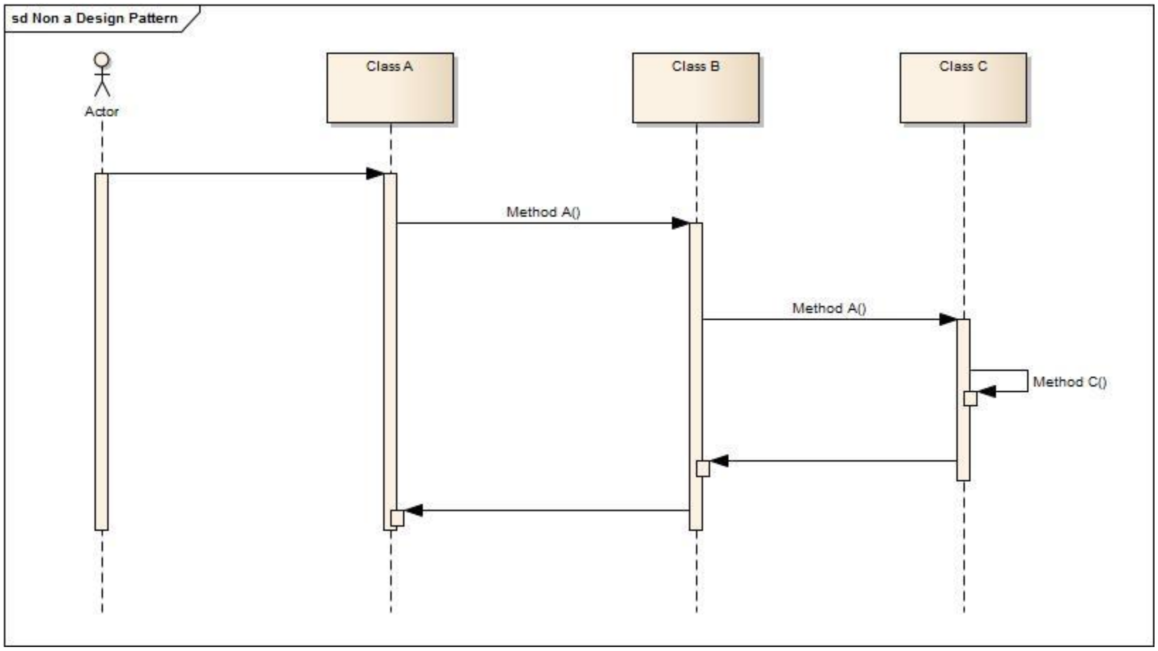


Figure 70 A Sequence Diagram of a program that looks like a Chain of Responsibility Design Pattern

Looking to sequence diagram Figure 70, the sequence of the methods between classes are like the sequence diagram of a chain of responsibility, and it satisfied all the behavioral features of the design pattern. When we look at its class diagram in figure 19, the class diagram does not satisfy the structural features of a chain of responsibility design pattern.

Class C in class diagram Figure 71, does not implement Class A, instead it has a 0...* dependency relationship with Class A. While to satisfy the structural features of a chain of responsibility design pattern both Classes B and C should implement Class A.

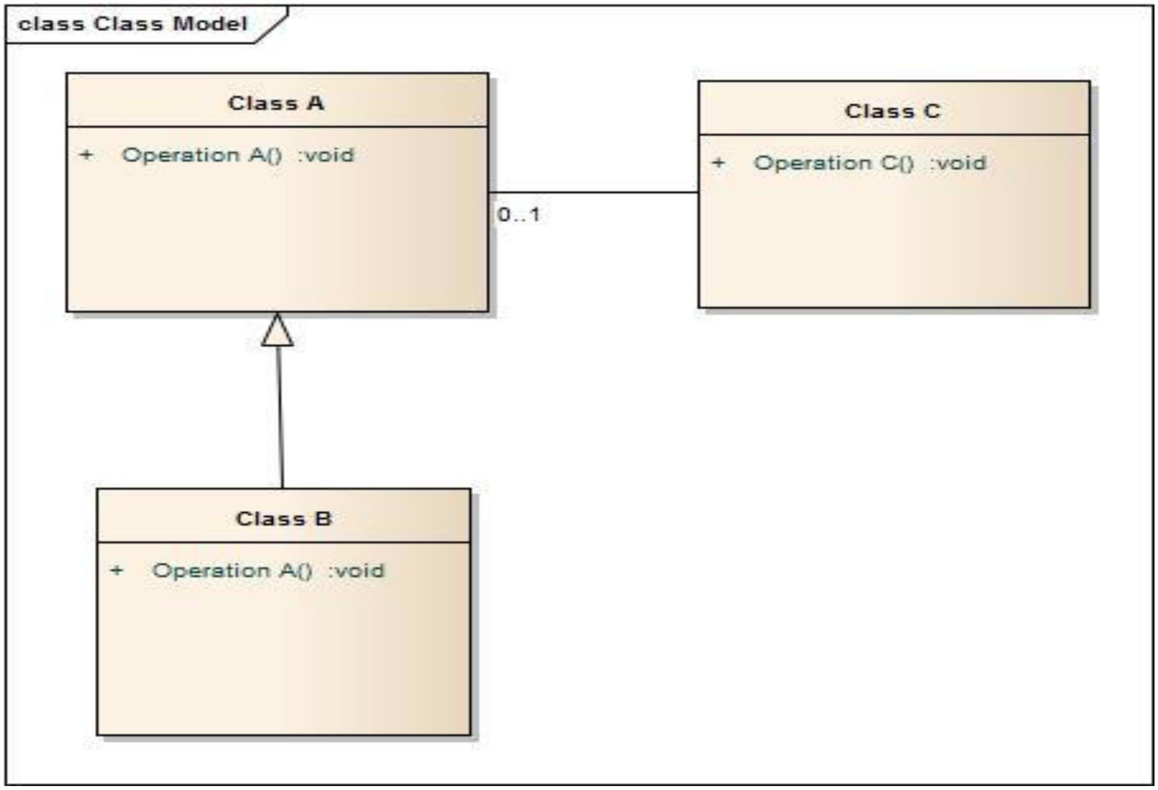


Figure 71 A Class Diagram of a Program that looks like a Chain of Responsibility Design Pattern

5.1.2 Case 2

Like the case 1, we have another case study where we have a sequence diagram in figure 20, that looks like an Observer design pattern but when we checked its class diagram figure 21, the class diagram does not satisfy the structural features of Observer design pattern.

As we mentioned before looking only to one view of a design pattern might cause a wrong detection of a design pattern. We need to look at all the different views to detect the design pattern accurately.

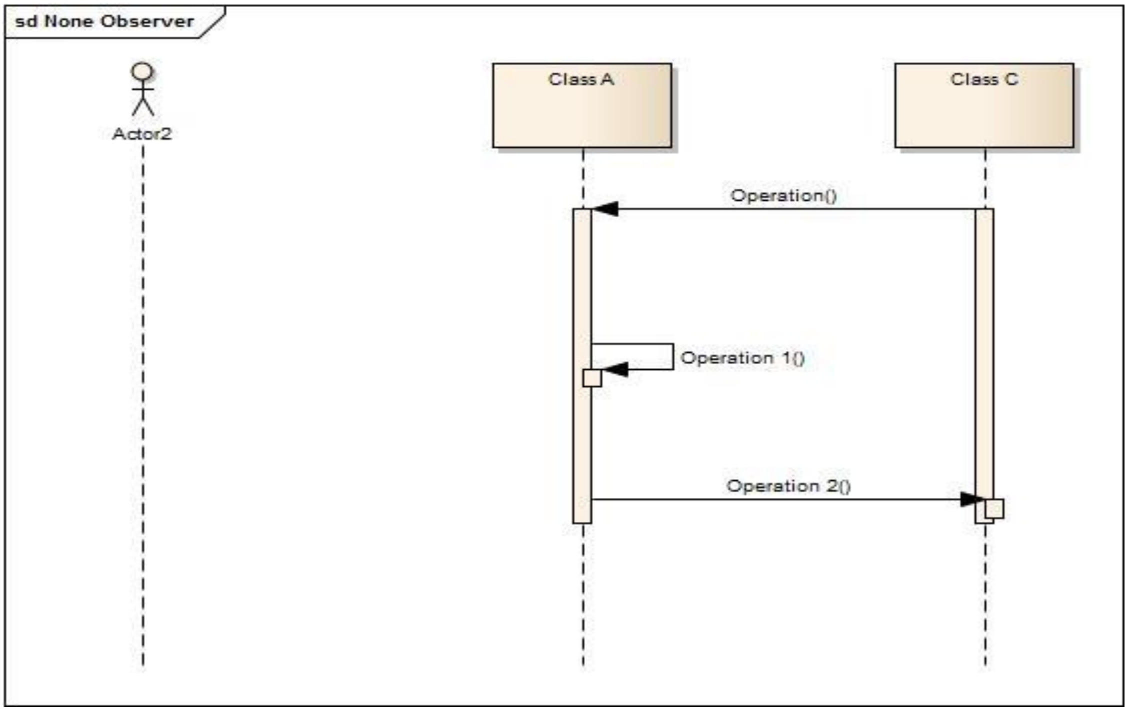


Figure 72 A Sequence Diagram of a Program that looks like an Observer Design Pattern

The sequence diagram in Figure 72 looks like an Observer design pattern since all the different behavioral features and specifications have maintained and satisfied. However, figure 22 is the class diagram of the sequence diagram represented in figure 31, the class diagram does not actually satisfy all the structural features of Observer design pattern. Class A uses Class B with 0…1 multiplicity, while in the Observer class diagram the multiplicity should be 1…*. In addition, Class A is concrete in Figure 73, while the true case of Observer there should be another class implements Class A.

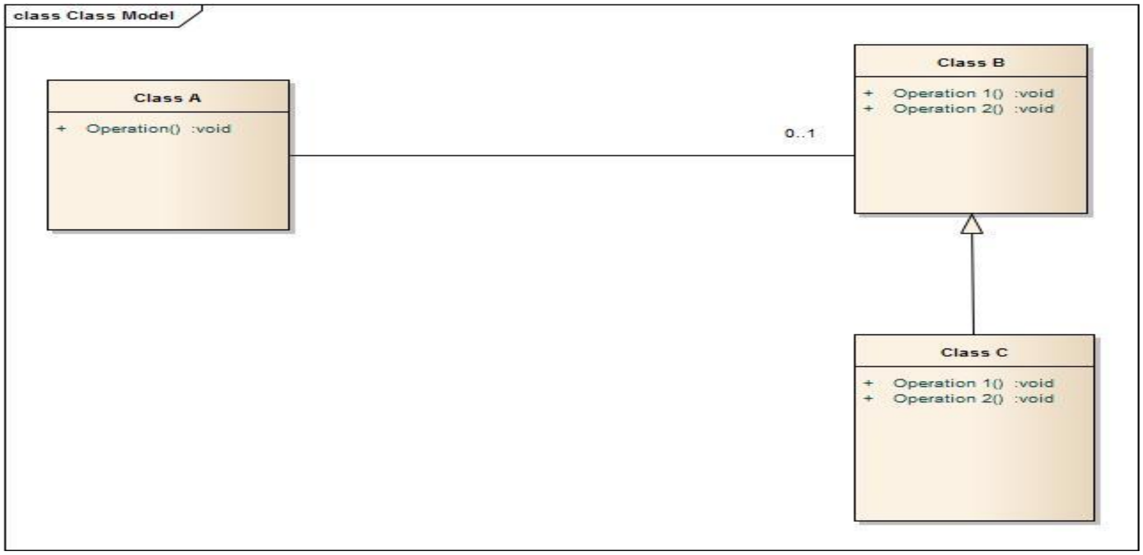


Figure 73 a Class Diagram of a Program that looks like an Observer Design Pattern

5.2 Design Pattern Detection Tool

Based on the design pattern views specifications, we have implemented a design pattern detection tool that detects the design patterns from an XML file. The tool reads an XML file that is based on the integrated metamodel with a number of use cases. Some of these use cases are real design patterns where some of these use cases are not. The tool is based on Python programming language. The detecting parser checks the file use case by use case with comparing each use case with the design pattern specification. The parser claims that the use case is a design pattern whenever it meets the specifications of a specific design pattern, otherwise, the detection parser claims that the use case is not a design pattern.

The tool consists of two parts, the first part is the design pattern specification where the tool contains all the GoF design patterns specifications, and the second part is the design pattern detection parser. The parser compares each use case in the XML file with all the GoF design patterns specification.

5.2.1 Design Pattern Detection Algorithm

In this section we are going to explain our proposed design pattern algorithm. Our proposed design pattern algorithm consists of two main parts:

The Design Pattern Specifications Dictionary: Which is represented by text paths. Each design pattern is represented by number of XML tags path. For instance this is one path of the chain of responsibility design pattern: *("Interaction/InteractionFragment/MultiOperand/InteractionOperand/Alt/Interaction/class/Message",1).* All the design patterns paths are stored towards it design pattern.

The Detection Code: This is the actual design pattern detector. The detector traces all the different use cases in the integrated metamodel XML file. The detector compares each use case with the stored design pattern XML tags Paths. Whenever a use case matches a specific design pattern XML tags paths it reports that it found a design pattern, otherwise it escapes to the next use case until it reaches the end of the XML file. The following is the python pseudocode of our design pattern detector:

Table 50 Design Pattern Detection Algorithm Pseudocode

```
Design Pattern Detection Algorithm Pseudocode
Require :
            1-XML file contains one or more use cases
Procedure :
//Create a dictionary for all the 16 design patterns

1.   Define DPdictionary
//inside the dictionary define all the design patterns corresponding to their features :

2.   Define chain of responsibly implementation1 :
     A.  ("Interaction/InteractionFragment/MultiOperand/InteractionOperand/Alt/Interaction/class/Message",1),
     B.  ("Interaction/InteractionFragment/MultiOperand/InteractionOperand/Alt/Interaction/class/implements",1),
         ("Interaction/InteractionFragment/MultiOperand/InteractionOperand/Alt/Interaction/class/Message",1),
         ("Interaction/InteractionFragment/MultiOperand/InteractionOperand/Alt/Interaction/class/implements",1),
                 ("Interaction/class/Message",2),
                 ("Interaction/class/Message",1),
                 ("Interaction/class/implements",1),
                 ("Interaction/class/implements",1)

//Read an XML file that contains one or more use case , the use cases might be real
design pattern use cases see Appendix B, or use cases that looks like a design pattern
see Figure 74:
     3-  Read (File.xml)
     4-  Start from the file root.
     5-  Set Patterncounter to 0
     6-  Set UseCase Counter to 0

 #for all the design patterns in the design pattern dictionary check all the design
 pattern features (tag paths) against each use case in the XML file.

     7-  For each use case in the XML file
               a-  For each design pattern in the dictionary do
                   UseCase+1
  #if the design pattern features met the traced use case then the specific-pattern
  counter updated otherwise it goes to the next design pattern.
                   For each feature in the design pattern do
                              If use case feature == design pattern features then
                                          Patterncounter+1
                              Otherwise
                                          Read next use case

     8-  Display UseCase

     9-  Display Patterncounter
```

5.3 Empirical Experiment

The objective of this section is to analyze our design pattern detection technique based on the UML integrated metamodel with respect to design pattern detection techniques that use only one or two views separately from the point of view of the researcher, designer, and programmer.

This experiment will discuss the results of using the three views representation and implementing the design patterns using the integrated metamodel compared to using individual and non-integrated views design pattern representation and detection techniques.

5.3.1 The Contest

The context of this experiment is the two design patterns selected from GoF design patterns. Each design pattern was defined and represented with the three main views (Functional, Behavioral, and Structural) and compared to the definition and representation of design patterns proposed in the literature review using one and two views.

The validation and the comparison between the two views were conducted in two stages: visual validation, and automatic validation. In visual validation, we have two case studies of designs where it looks like a design pattern when we consider each view separately, but when we consider both views we realize that the design does not satisfy the different design pattern view features. In the automatic validation, we developed a java parser that parses an XML design pattern representation using the UML integrated metamodel of a design pattern and an XML file that uses one view representation.

5.3.2 Research Questions and Hypotheses

We need first to know all the previous detection techniques that based on one or two views to detect design patterns, we also need to know how design patterns we implemented, defined and presented. Based on the objectives of our case study, we designed the following research questions:

1. Does the design pattern detection technique using the UML integrated metamodel which is based on the integration of the three UML main views detects design pattern in a more accurate manner than using each view separately?
2. Does defining and presenting design patterns using the three main views (Functional, Behavioral, and Structural) gives more information to detect the design patterns than the traditional definition and representing using one or two views?

Based on the previous research questions we formulated the following Null Hypotheses of this study:

$H1_0$: The design pattern detection technique using the UML integrated metamodel which is based on the integration of the three UML main views does not detect design patterns in a more accurate manner than using each view separately.

$H2_0$: Defining and presenting design patterns using the three main views (Functional, Behavioral, and Structural) does not give more information to detect the design patterns than the traditional definition and representing using one or two views.

The alternative hypotheses of this study are as the following:

H1: The design pattern detection technique using the UML integrated metamodel which is based on the integration of the three UML main views detects design patterns in a more accurate manner than using each view separately.

H2: Defining and presenting design patterns using the three main views (Functional, Behavioral, and Structural) gives more information to detect the design patterns than the traditional definition and representing using one or two views.

To reject the null hypotheses and accept the hypotheses, we need to discuss the results of three main stages of the study: the functional representation of design patterns, the visual validation of design patterns, and the automatic validation using the Java XML parser we developed.

5.3.3 Objects

To investigate and validate our proposed technique, we must select the appropriate design patterns. The design patterns we selected for this study are GoF design patterns proposed by [9]. The reasons behind selected the GoF design pattern above other design patterns like web design patterns are as follows:

1. GoF design patterns are well known and widely used design patterns in the empirical studies and the literature review. From the table of comparison in section 3, we can realize that the majority of design pattern detection techniques are using GoF design patterns.
2. The GoF design patterns are well presented and defined in the literature review, structural and behavioral features specifically.
3. Many of GoF design patterns have more than one instance and different scenarios.

5.3.4 Subjects

The subjects of this case study are the tools that we used to build up the design patterns UML diagrams of the main three view (use case, sequence, and class), as well as the tools that we used to represent the design patterns using the UML integrated metamodel. We also used other tools that convert and builds up the XML code for the integrated metamodel and UML diagrams.

5.3.5 Variable Selection

The independent variables in this study are the GoF design patterns. Whereas, the dependent variables are the different representation forms of design patterns.

5.3.6 Instrumentation

This case study uses different instrumentations to help to conduct the experiment; we used Python Editor, Eclipse, XML Spy, and Enterprise Architecture.

5.3.7 Experimental Procedure

First, we have conducted two main literature reviews: the traditional literature review and the systematic literature review to collect all the necessary data of the previous design patterns detection techniques, in order to conduct and compare our proposed technique with the techniques we reviewed and surveyed.

Second, we defined the selected GoF design patterns with the three main views (structural, behavioral, and functional).

Third, we represented the selected GoF design patterns using the UML integrated metamodel.

Fourth, we defined and specified the design patterns specifications of the three main views.

Fifth, we conducted a visual validation of two case studies of a software design that look like design patterns if we look at each view individually. However, when we consider the other views it showed that they are not actually designed patterns.

Finally, we developed java parser tool that parses the XML file and checks for all design patterns three main views specifications. The tool reads different XML files and checks for all the design pattern specification when it all the specification of a design pattern is satisfied then the tool prompt that it found a design pattern, otherwise it prompts that there are no design patterns found.

5.3.8 Results Discussion

In this section, we are going to discuss the results of experiment procedure individually, and then we are going to either accept or reject the null hypotheses based on the results:

Visual Validation results:

In the visual validation procedure in section 4, we have discussed two different case studies of different software designs. In the visual validation, we checked each view of the case studies individually. The results showed that it is not accurate to look at only one view of a design pattern without looking to the views of the design pattern because each view gives specific features and information than the other view cannot provide. So from the visual validation results, we have found that if we look only at a single view without considering the other views might give a wrong decision in detecting design patterns as what happened in case studies 1, and 2. The UML metamodel of these case studies showed that each view provides specific information about the design patterns, for instance, class diagram will provide only the structural information about the design pattern without mentioning anything about how the sequential procedural is going between the objects and class. Likely wise, sequence diagram provides the behavioral aspects of the different design patterns without mentioning anything about how the design patterns have been structurally defined.

Based on the results, considering more and integrated three main (structural, behavioral, and functional) views of a design pattern will provide a full specification of the main different views features. The UML integrated metamodel integrates all the three main views features in one concrete XML file, so this will increase the level of accuracy in detecting the design patterns.

In this case, we are going to reject the first two null hypotheses and accept the alternative ones.

Evaluation of Precision and Recall

To validate out proposed technique we have developed a Python code Parser that detects the design patterns occurrences in the XML file. We have conducted two precision and recall tests. The tool reads the XML file and checks all the structural, behavioral, and functional features of the design pattern. If the XML file contains all the features then the tool claims that it found a design pattern otherwise, it prompts that it didn't find any design patterns. For the calculation of precision and recall, we need the following definitions:

- **True Positive (TP):** a real design pattern exists in the XML file and it is detected by the design pattern detection tool.

- **False Positive (FP):** there is no real design pattern exists in the XML file but the design pattern detection tool reports that there is a design pattern exists.

- **False Negative (FN):** a real design pattern exists in the XML file but the design pattern detection tool did not detect it.

The tool reads an XML file that based on the integrated metamodel contains different use cases some of these use cases are design patterns while others are not:

1. **Precision and recall test no.1**: In this test, we have only real design patterns use cases, we have one integrated metamodel file that contains only GoF real use cases. These use cases are based on the UML integrated metamodel that contains all the three main view features of a design pattern. When the tool reads this type of use cases, it reports that it found a design pattern and then it lists the number of occurrences. We created an XML file that contains all 16 use cases for the design patterns implemented in our study, each use case represents a real design pattern, the flowing table representing the results:

Table 51 Precise and Recall test No.1

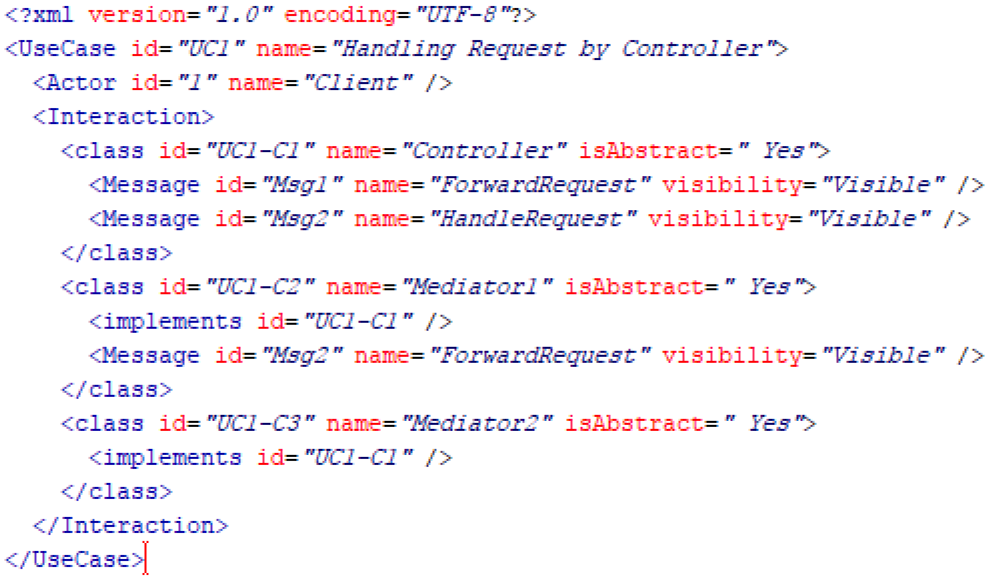
Design Pattern	Category	TP	FN	FP	Precision (%)	Recall (%)
Builder	Creational	1	0	0	100%	100%
Prototype	Creational	1	0	0	100%	100%
Singleton	Creational	1	0	0	100%	100%
Adapter	Structural	1	0	0	100%	100%

Bridge	Structural	1	0	0	100%	100%
Decorator	Structural	1	0	0	100%	100%
Flyweight	Structural	1	0	0	100%	100%
Proxy	Structural	1	0	0	100%	100%
Chain of Resp.	Behavioral	1	0	0	100%	100%
Observer	Behavioral	1	0	0	100%	100%
Strategy	Behavioral	1	0	0	100%	100%
Mediator	Behavioral	1	0	0	100%	100%
State	Behavioral	1	0	0	100%	100%
Visitor	Behavioral	1	0	0	100%	100%
Template Method	Behavioral	1	0	0	100%	100%
Command	Behavioral	1	0	0	100%	100%

The results in Table 51 showed that the detection tool detected one instance of each design pattern, the number of expected design pattern instance use cases equals to the number of the detected design patterns. The results implies that representing the design patterns using the integrated metamodel showed a high accurate results than representing design patterns with individual views.

2. **Precision and recall test no.2**: In this test, we created two use cases that looks like a real design pattern for each design pattern of the 16 design patterns we are reviewing in this study. This two fake design patterns where added to the XML file,

this XML files ends up with three use cases for each design pattern one real design pattern use case as well as the two fake design patterns. The following is an example of an XML file for a fake chain of responsibility design pattern:



```xml
<?xml version="1.0" encoding="UTF-8"?>
<UseCase id="UC1" name="Handling Request by Controller">
  <Actor id="1" name="Client" />
  <Interaction>
    <class id="UC1-C1" name="Controller" isAbstract=" Yes">
      <Message id="Msg1" name="ForwardRequest" visibility="Visible" />
      <Message id="Msg2" name="HandleRequest" visibility="Visible" />
    </class>
    <class id="UC1-C2" name="Mediator1" isAbstract=" Yes">
      <implements id="UC1-C1" />
      <Message id="Msg2" name="ForwardRequest" visibility="Visible" />
    </class>
    <class id="UC1-C3" name="Mediator2" isAbstract=" Yes">
      <implements id="UC1-C1" />
    </class>
  </Interaction>
</UseCase>
```

Figure 74 A Use Case That Looks Like a Design Pattern

This use case looks like a chain of responsibility design pattern, but it not a real chain of responsibility design pattern since it does not satisfies the three main views specification of chain of responsibility that we defined before. When the tool reads this type of XML use cases, it reports that it did not find any design patterns due to lacking the design pattern specifications. We have created an XML file that contains all the real design patterns uses with two fake use cases for each, the results were as following:

Table 52 Precise and Recall Test No.2

Design Pattern	TP	No. Of Non real design patterns	FN	FP	Precision (%)	Recall (%)
Builder	1	2	0	0	100%	100%

Prototype	1	2	0	0	100%	100%
Singleton	1	2	0	0	100%	100%
Adapter	1	2	0	0	100%	100%
Bridge	1	2	0	0	100%	100%
Decorator	1	2	0	0	100%	100%
Flyweight	1	2	0	0	100%	100%
Proxy	1	2	0	0	100%	100%
Chain of Resp.	1	2	0	0	100%	100%
Observer	1	2	0	0	100%	100%
Strategy	1	2	0	0	100%	100%
Mediator	1	2	0	0	100%	100%
State	1	2	0	0	100%	100%
Visitor	1	2	0	0	100%	100%
Template Method	1	2	0	0	100%	100%
Command	1	2	0	0	100%	100%

From Table 52 we can see that the tools detects only the use cases that satisfies all the design pattern specifications, while it ignores all the other fake use cases. In order to ensure that the tool detected the real use case of the design patterns we removed the real use case of the design pattern from the same XML file and then we re-conducted the same test, thus we got the following results:

Table 53 Re-Conducted Test No.2

Design Pattern	TP	No. Of Non real design patterns	FN	FP	Precision (%)	Recall (%)
Builder	0	2	0	0	100%	100%
Prototype	0	2	0	0	100%	100%
Singleton	0	2	0	0	100%	100%
Adapter	0	2	0	0	100%	100%
Bridge	0	2	0	0	100%	100%
Decorator	0	2	0	0	100%	100%
Flyweight	0	2	0	0	100%	100%
Proxy	0	2	0	0	100%	100%
Chain of Resp.	0	2	0	0	100%	100%
Observer	0	2	0	0	100%	100%
Strategy	0	2	0	0	100%	100%
Mediator	0	2	0	0	100%	100%
State	0	2	0	0	100%	100%
Visitor	0	2	0	0	100%	100%

Template Method	0	2	0	0	100%	100%
Command	0	2	0	0	100%	100%

The results in Table 53 showed that the detection tool did not report any occurrence of design pattern in the modified XML file. This implies that the results in table 35 were accurate and precise.

So, from the results above, using the UML integrated metamodel is more accurate in detecting design patterns than using individual views. Based on these results, we are going to reject the first two hypotheses and accept the alternative ones.

5.3.9 Empirical Experimental Conclusion

Based on these results and analysis we are going to reject all the null hypotheses and accept the alternative ones. Defining and representing design patterns using the three views give more information about the design pattern so it can be detected in more accurate way.

From the documentation point of view, representing design patterns using the three main views (Functional, Behavioral, and Structural) increase the level of documentation of the design patterns for designers, and researchers. Furthermore, documenting the three views features help in discussing the all possible implementations and instances of the design pattern in one concrete document.

5.3.10 Threats to Validity

Internal Validity: in these study, there might be some confounding factors that may affect the results of the experiment. The design patterns instances and scenarios were implemented based on our understanding, thus the results might be biased. There is also a manual intervention in our proposed technique, so that may effects the results.

External Validity: this study is limited to 16 GoF design patterns, the results are only biased by the design patterns that we selected for this study. However, we might not get the same results if we conducted the proposed technique on all GoF design patterns. Thus, the results cannot be generalized

Construct Validity: this study has used the class, sequence, and use case diagram to represent the structural, behavioral, and functional views respectively. There are other diagrams might represent the three views in a more efficient way than the selected diagrams. We also chose GoF design patterns, although there are other types of design patterns like web design patterns, and elemental design patterns.

6 CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

In this study, we proposed a design pattern detection method and representation using UML integrated Metamodel. The integrated metamodel integrates the three main views (Functional, Behavioral, and Structural). The representation using UML integrated metamodel gives more information about the design patterns than taking each view separately.

The proposed technique provided better accuracy to detect the design patterns and decreased the false-positive and true-negative of the detection process because one XML file of the UML integrated metamodel contains the whole concrete Functional, Behavioral, and Structural features of the design pattern in one self-contained file.

The systematic literature review and the traditional literature review showed that most the previous design pattern detection techniques were based only on one view. Some of the proposed detection techniques were using two views, but they are considering each view separately, which will decrease the detection process.

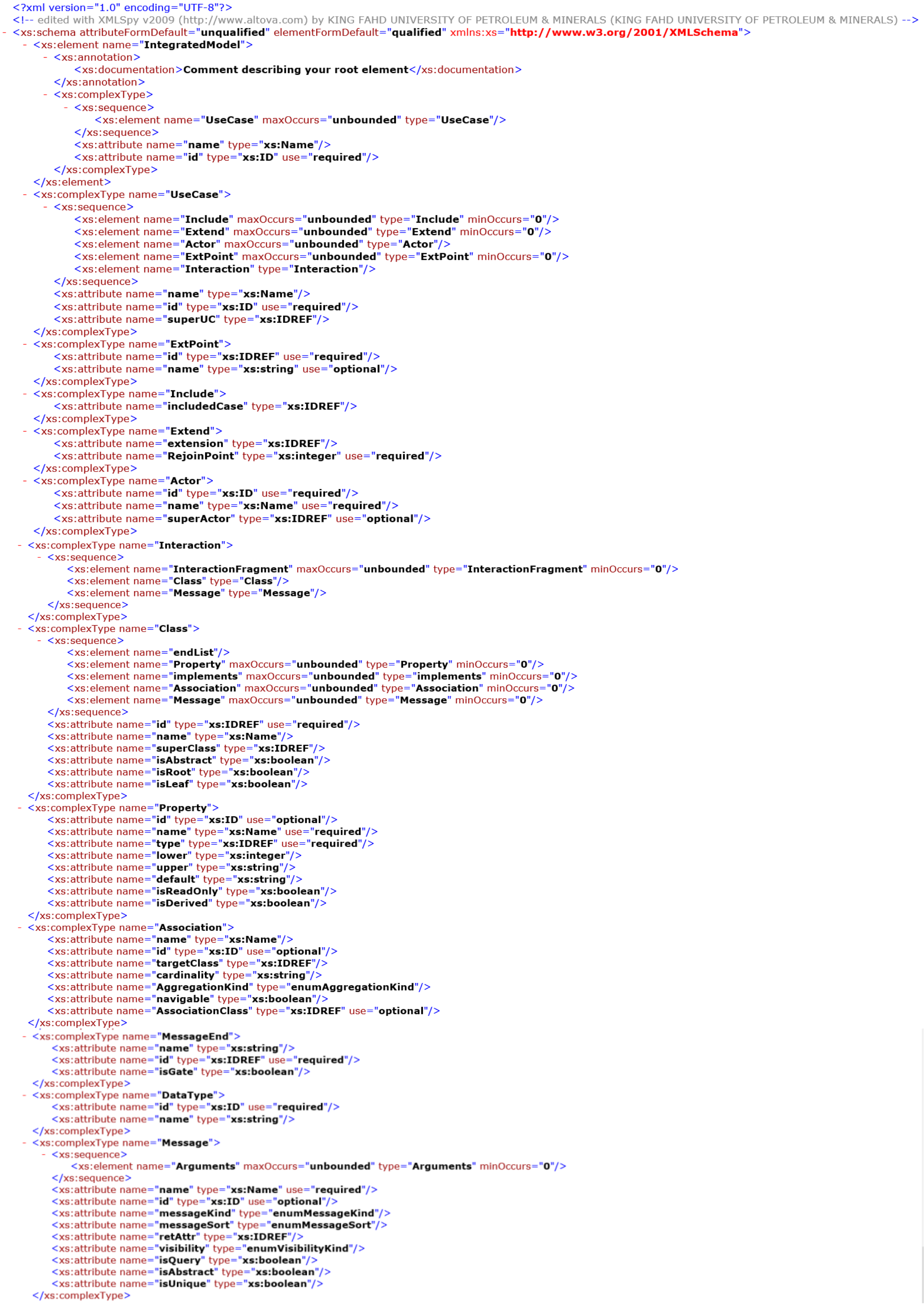
6.2 FUTURE WORK

In this study, we applied our design pattern detection technique to all GoF design patterns to prove and validate our concept. Since we have approved the concept of detection design patterns using the UML integrated metamodel we are planning to apply our proposed technique to other design pattern types.

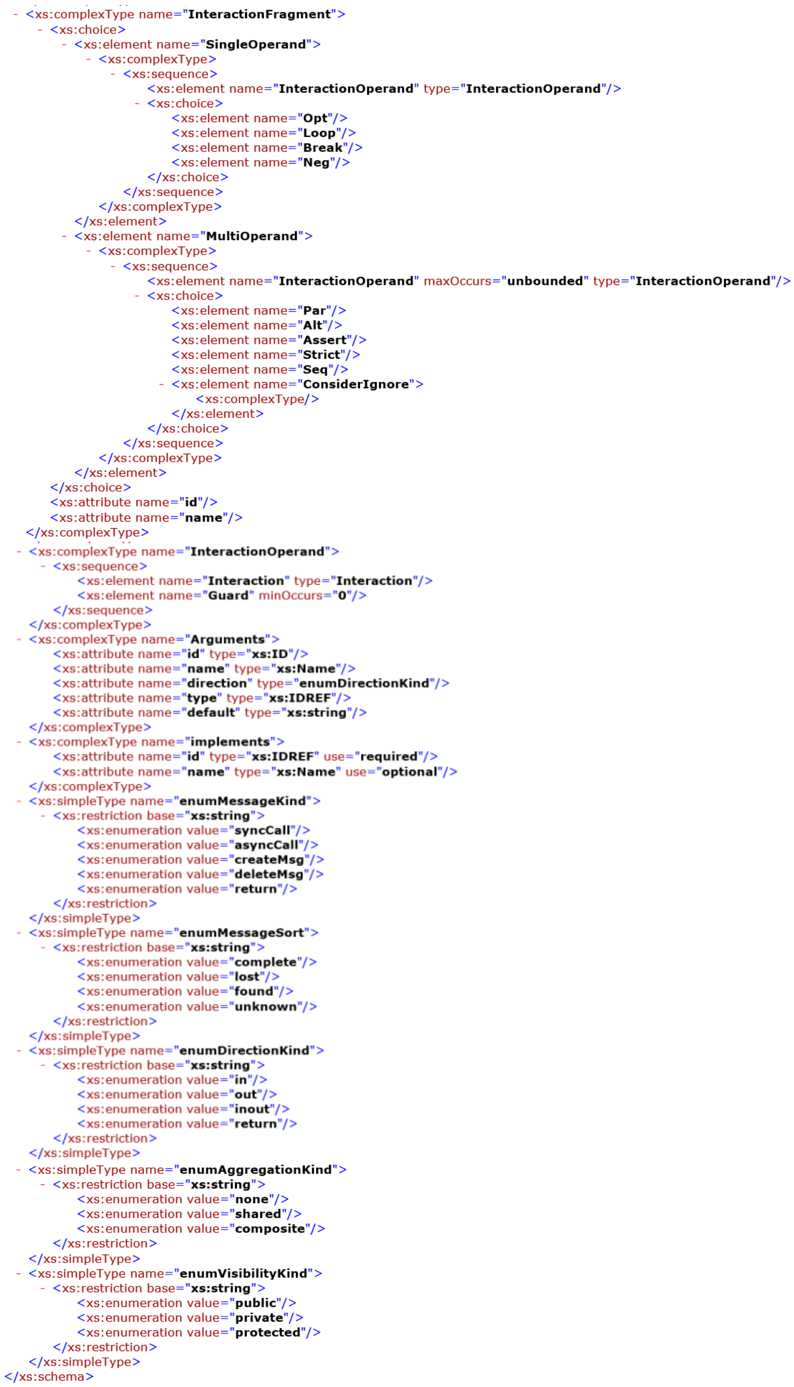
A fully automated procedure will be followed to detect design patterns, from the first stage of the automated methodology.

Using the UML integrated metamodel will open a future research opportunity to discover design patterns semantically since it provides a name for each design pattern components and features.

Appendix A: XMI Schema for Integrated Metamodel



```
  <?xml version="1.0" encoding="UTF-8"?>
  <!-- edited with XMLSpy v2009 (http://www.altova.com) by KING FAHD UNIVERSITY OF PETROLEUM & MINERALS (KING FAHD UNIVERSITY OF PETROLEUM & MINERALS) -->
- <xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  - <xs:element name="IntegratedModel">
    - <xs:annotation>
        <xs:documentation>Comment describing your root element</xs:documentation>
      </xs:annotation>
    - <xs:complexType>
      - <xs:sequence>
          <xs:element name="UseCase" maxOccurs="unbounded" type="UseCase"/>
        </xs:sequence>
        <xs:attribute name="name" type="xs:Name"/>
        <xs:attribute name="id" type="xs:ID" use="required"/>
      </xs:complexType>
    </xs:element>
  - <xs:complexType name="UseCase">
    - <xs:sequence>
        <xs:element name="Include" maxOccurs="unbounded" type="Include" minOccurs="0"/>
        <xs:element name="Extend" maxOccurs="unbounded" type="Extend" minOccurs="0"/>
        <xs:element name="Actor" maxOccurs="unbounded" type="Actor"/>
        <xs:element name="ExtPoint" maxOccurs="unbounded" type="ExtPoint" minOccurs="0"/>
        <xs:element name="Interaction" type="Interaction"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:Name"/>
      <xs:attribute name="id" type="xs:ID" use="required"/>
      <xs:attribute name="superUC" type="xs:IDREF"/>
    </xs:complexType>
  - <xs:complexType name="ExtPoint">
      <xs:attribute name="id" type="xs:IDREF" use="required"/>
      <xs:attribute name="name" type="xs:string" use="optional"/>
    </xs:complexType>
  - <xs:complexType name="Include">
      <xs:attribute name="includedCase" type="xs:IDREF"/>
    </xs:complexType>
  - <xs:complexType name="Extend">
      <xs:attribute name="extension" type="xs:IDREF"/>
      <xs:attribute name="RejoinPoint" type="xs:integer" use="required"/>
    </xs:complexType>
  - <xs:complexType name="Actor">
      <xs:attribute name="id" type="xs:ID" use="required"/>
      <xs:attribute name="name" type="xs:Name" use="required"/>
      <xs:attribute name="superActor" type="xs:IDREF" use="optional"/>
    </xs:complexType>
 - <xs:complexType name="Interaction">
   - <xs:sequence>
       <xs:element name="InteractionFragment" maxOccurs="unbounded" type="InteractionFragment" minOccurs="0"/>
       <xs:element name="Class" type="Class"/>
       <xs:element name="Message" type="Message"/>
     </xs:sequence>
   </xs:complexType>
 - <xs:complexType name="Class">
   - <xs:sequence>
       <xs:element name="endList"/>
       <xs:element name="Property" maxOccurs="unbounded" type="Property" minOccurs="0"/>
       <xs:element name="implements" maxOccurs="unbounded" type="implements" minOccurs="0"/>
       <xs:element name="Association" maxOccurs="unbounded" type="Association" minOccurs="0"/>
       <xs:element name="Message" maxOccurs="unbounded" type="Message" minOccurs="0"/>
     </xs:sequence>
     <xs:attribute name="id" type="xs:IDREF" use="required"/>
     <xs:attribute name="name" type="xs:Name"/>
     <xs:attribute name="superClass" type="xs:IDREF"/>
     <xs:attribute name="isAbstract" type="xs:boolean"/>
     <xs:attribute name="isRoot" type="xs:boolean"/>
     <xs:attribute name="isLeaf" type="xs:boolean"/>
   </xs:complexType>
 - <xs:complexType name="Property">
     <xs:attribute name="id" type="xs:ID" use="optional"/>
     <xs:attribute name="name" type="xs:Name" use="required"/>
     <xs:attribute name="type" type="xs:IDREF" use="required"/>
     <xs:attribute name="lower" type="xs:integer"/>
     <xs:attribute name="upper" type="xs:string"/>
     <xs:attribute name="default" type="xs:string"/>
     <xs:attribute name="isReadOnly" type="xs:boolean"/>
     <xs:attribute name="isDerived" type="xs:boolean"/>
   </xs:complexType>
 - <xs:complexType name="Association">
     <xs:attribute name="name" type="xs:Name"/>
     <xs:attribute name="id" type="xs:ID" use="optional"/>
     <xs:attribute name="targetClass" type="xs:IDREF"/>
     <xs:attribute name="cardinality" type="xs:string"/>
     <xs:attribute name="AggregationKind" type="enumAggregationKind"/>
     <xs:attribute name="navigable" type="xs:boolean"/>
     <xs:attribute name="AssociationClass" type="xs:IDREF" use="optional"/>
   </xs:complexType>
  - <xs:complexType name="MessageEnd">
      <xs:attribute name="name" type="xs:string"/>
      <xs:attribute name="id" type="xs:IDREF" use="required"/>
      <xs:attribute name="isGate" type="xs:boolean"/>
    </xs:complexType>
  - <xs:complexType name="DataType">
      <xs:attribute name="id" type="xs:ID" use="required"/>
      <xs:attribute name="name" type="xs:string"/>
    </xs:complexType>
  - <xs:complexType name="Message">
    - <xs:sequence>
        <xs:element name="Arguments" maxOccurs="unbounded" type="Arguments" minOccurs="0"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:Name" use="required"/>
      <xs:attribute name="id" type="xs:ID" use="optional"/>
      <xs:attribute name="messageKind" type="enumMessageKind"/>
      <xs:attribute name="messageSort" type="enumMessageSort"/>
      <xs:attribute name="retAttr" type="xs:IDREF"/>
      <xs:attribute name="visibility" type="enumVisibilityKind"/>
      <xs:attribute name="isQuery" type="xs:boolean"/>
      <xs:attribute name="isAbstract" type="xs:boolean"/>
      <xs:attribute name="isUnique" type="xs:boolean"/>
    </xs:complexType>
```



```
 - <xs:complexType name="InteractionFragment">
   - <xs:choice>
     - <xs:element name="SingleOperand">
       - <xs:complexType>
         - <xs:sequence>
             <xs:element name="InteractionOperand" type="InteractionOperand"/>
           - <xs:choice>
               <xs:element name="Opt"/>
               <xs:element name="Loop"/>
               <xs:element name="Break"/>
               <xs:element name="Neg"/>
             </xs:choice>
           </xs:sequence>
         </xs:complexType>
       </xs:element>
     - <xs:element name="MultiOperand">
       - <xs:complexType>
         - <xs:sequence>
             <xs:element name="InteractionOperand" maxOccurs="unbounded" type="InteractionOperand"/>
           - <xs:choice>
               <xs:element name="Par"/>
               <xs:element name="Alt"/>
               <xs:element name="Assert"/>
               <xs:element name="Strict"/>
               <xs:element name="Seq"/>
             - <xs:element name="ConsiderIgnore">
                 <xs:complexType/>
               </xs:element>
             </xs:choice>
           </xs:sequence>
         </xs:complexType>
       </xs:element>
     </xs:choice>
     <xs:attribute name="id"/>
     <xs:attribute name="name"/>
   </xs:complexType>
 - <xs:complexType name="InteractionOperand">
   - <xs:sequence>
       <xs:element name="Interaction" type="Interaction"/>
       <xs:element name="Guard" minOccurs="0"/>
     </xs:sequence>
   </xs:complexType>
 - <xs:complexType name="Arguments">
     <xs:attribute name="id" type="xs:ID"/>
     <xs:attribute name="name" type="xs:Name"/>
     <xs:attribute name="direction" type="enumDirectionKind"/>
     <xs:attribute name="type" type="xs:IDREF"/>
     <xs:attribute name="default" type="xs:string"/>
   </xs:complexType>
 - <xs:complexType name="implements">
     <xs:attribute name="id" type="xs:IDREF" use="required"/>
     <xs:attribute name="name" type="xs:Name" use="optional"/>
   </xs:complexType>
 - <xs:simpleType name="enumMessageKind">
   - <xs:restriction base="xs:string">
       <xs:enumeration value="syncCall"/>
       <xs:enumeration value="asyncCall"/>
       <xs:enumeration value="createMsg"/>
       <xs:enumeration value="deleteMsg"/>
       <xs:enumeration value="return"/>
     </xs:restriction>
   </xs:simpleType>
 - <xs:simpleType name="enumMessageSort">
   - <xs:restriction base="xs:string">
       <xs:enumeration value="complete"/>
       <xs:enumeration value="lost"/>
       <xs:enumeration value="found"/>
       <xs:enumeration value="unknown"/>
     </xs:restriction>
   </xs:simpleType>
 - <xs:simpleType name="enumDirectionKind">
   - <xs:restriction base="xs:string">
       <xs:enumeration value="in"/>
       <xs:enumeration value="out"/>
       <xs:enumeration value="inout"/>
       <xs:enumeration value="return"/>
     </xs:restriction>
   </xs:simpleType>
 - <xs:simpleType name="enumAggregationKind">
   - <xs:restriction base="xs:string">
       <xs:enumeration value="none"/>
       <xs:enumeration value="shared"/>
       <xs:enumeration value="composite"/>
     </xs:restriction>
   </xs:simpleType>
 - <xs:simpleType name="enumVisibilityKind">
   - <xs:restriction base="xs:string">
       <xs:enumeration value="public"/>
       <xs:enumeration value="private"/>
       <xs:enumeration value="protected"/>
     </xs:restriction>
   </xs:simpleType>
</xs:schema>
```

Appendix B: UML INTEGRATED NETAMODEL REPRESENTATION OF GoF DESIGN PATTERNS

Builder

```
1-  <?xml version="1.0" encoding="UTF-8"?>
2-  <IntegratedModel id="1" name="Builder">
3-  //UseCase 1
4-  <UseCase id="UC1" name="Build the Complex Object">
5-  <Actor id="1" name="Client" />
6-  <Interaction>
7-  <class id="UC1-C1" name="Director" isAbstract=" No">
8-  <Message id="Msg1" name="Director " visibility="Visible" />
9-  <Message id="Msg2" name="Construct " visibility="Visible" />
10- <Association name="Compose" targetClass="C1-Builder" AggregationKind="
    Compossite" />
11- </class>
12- <class id="UC1-C2" name="ConcreteBuilder" isAbstract=" No">
13- <implements id="UC1-Builder" />
14- <Message id="Msg3" name="buildPartA" visibility="Visible" />
15- <Message id="Msg4" name="buildPartB" visibility="Visible" />
16- <Message id="Msg5" name="buildPartC" visibility="Visible" />
17- <Message id="Msg6" name="getResult" visibility="Visible" />
18- </class>
19- </Interaction>
20- </UseCase>
21- </IntegratedModel>
```

The previous XML code represented the UML integrated metamodel for Builder design pattern. Line 4 and 6 represented the three main views of the design pattern. Each use case in the integrated metamodel each tag represents a feature of the design pattern. Each use case has its name, actor and the extended or included use case. The tags in 6 to 19 represented the interaction of the different objects of the design pattern. Each tag contains all the different classes with its structural features and methods as well as the sequence of the classes. Hence, all the different views features are integrated into one XML file.
This will add a lot of benefits to the detection approach since all the different features and collaboration sequences are maintained and integrated into one XML file.

Prototype

```
2-  <?xml version="1.0" encoding="UTF-8"?>
3-  <IntegratedModel id="1" name="Prototype">
4-  //UseCase 1
5-  <UseCase id="UC1" name="Clone Object">
6-  <Actor id="1" name="Client" />
```

```
7-  <Interaction>
8-  <class id="UC1-C1" name="ConcretePrototype1" isAbstract=" No">
9-  <Message id="Msg1" name="doneItSelf() " visibility="Visible" />
10- <implements id="Prototype" />
11- </class>
12- <class id="UC1-C2" name="ConcretePrototype2" isAbstract=" No">
13- <Message id="Msg2" name="doneItSelf() " visibility="Visible" />
14- <implements id="Prototype" />
15- </class>
16- </Interaction>
17- </UseCase>
18- </IntegratedModel>
```

The previous XML code represented the UML integrated metamodel for Prototype design pattern. Line 5 and 7 represented the three main views of the design pattern. Each use case in the integrated metamodel each tag represents a feature of the design pattern. Each use case has its name, actor and the extended or included use case. The tags in 7 to 16 represented the interaction of the different objects of the design pattern. Each tag contains all the different classes with its structural features and methods as well as the sequence of the classes. Hence, all the different views features are integrated into one XML file.
This will add a lot of benefits to the detection approach since all the different features and collaboration sequences are maintained and integrated into one XML file.

Singleton

```
1-  <?xml version="1.0" encoding="UTF-8"?>
2-  <IntegratedModel id="1" name="Singleton">
3-  //UseCase 1
4-  <UseCase id="UC1" name="Get Instance">
5-  <Actor id="1" name="Client" />
6-  <Interaction>
7-  <class id="UC1-C1" name="Singelton" isAbstract=" No">
8-  <Message id="Msg1" name="getInstance" visibility="Visible" />
9-  <Message id="Msg2" name="singelton" visibility="Visible" />
10- <Association name="ComposeItself" targetClass="C1-Singelton"
    AggregationKind=" Compossite" />
11- </class>
12- </Interaction>
13- </UseCase>
14- </IntegratedModel>
```

The previous XML code represented the UML integrated metamodel for Singleton design pattern. Line 4 and 6 represented the three main views of the design pattern. Each use case in the integrated metamodel each tag represents a feature of the design pattern. Each use case has its name, actor and the extended or included use case. The tags in 6 to 13 represented the interaction of the different objects of the design pattern. Each tag contains

all the different classes with its structural features and methods as well as the sequence of the classes. Hence, all the different views features are integrated into one XML file.
This will add a lot of benefits to the detection approach since all the different features and collaboration sequences are maintained and integrated into one XML file.

Decorator

```
1-  <?xml version="1.0" encoding="UTF-8"?>
2-  <IntegratedModel id="1" name="Decorator">
3-  //UseCase 1
4-  <UseCase id="UC1" name="Add Behaivor">
5-  <Actor id="1" name="Client" />
6-  <Interaction>
7-  <class id="UC1-C1" name="Decorator1" isAbstract=" No">
8-  <Message id="Msg1" name="operation " visibility="Visible" />
9-  <Message id="Msg2" name="addBehaivor " visibility="Visible" />
10- <implements id="Decorator" />
11- <Association name="Compose" targetClass="C1-Builder" AggregationKind="
    Compossite" />
12- </class>
13- <class id="UC1-C1" name="Decorator2" isAbstract=" No">
14- <Message id="Msg1" name="operation " visibility="Visible" />
15- <Message id="Msg2" name="addBehaivor " visibility="Visible" />
16- <implements id="Decorator" />
17- </class>
18- <class id="UC1-C1" name="ConcreteComponenet" isAbstract=" No">
19- <Message id="Msg1" name="operation " visibility="Visible" />
20- <implements id="Componenet" />
21- <Association name="Compose" targetClass="Decorator" AggregationKind="
    Compossite" />
22- </class>
23- </Interaction>
24- </UseCase>
25- </IntegratedModel>
```

The previous XML code represented the UML integrated metamodel for Decorator design pattern. Line 4 and 6 represented the three main views of the design pattern. Each use case in the integrated metamodel each tag represents a feature of the design pattern. Each use case has its name, actor and the extended or included use case. The tags in 6 to 23 represented the interaction of the different objects of the design pattern. Each tag contains all the different classes with its structural features and methods as well as the sequence of the classes. Hence, all the different views features are integrated into one XML file.
This will add a lot of benefits to the detection approach since all the different features and collaboration sequences are maintained and integrated into one XML file.

Proxy

```
1-  <?xml version="1.0" encoding="UTF-8"?>
2-  <IntegratedModel id="1" name="Proxy">
3-  //UseCase 1
4-  <UseCase id="UC1" name="Representing the Functionality of Subject">
5-  <Actor id="1" name="Client" />
6-  <Interaction>
7-  <class id="UC1-C1" name="Proxy" isAbstract=" No">
8-  <Message id="Msg1" name="operation " visibility="Visible" />
9-  <Association name="Compose" targetClass="UC1-C2" AggregationKind=" share" />
10- <implements id="Subject" />
11- </class>
12- <class id="UC1-C2" name="RealSubject" isAbstract=" No">
13- <implements id="Subject" />
14- <Message id="Msg2" name="opertion" visibility="Visible" />
15- </class>
16- </Interaction>
17- </UseCase>
18- </IntegratedModel>
```

The previous XML code represented the UML integrated metamodel for Proxy design pattern. Line 4 and 6 represented the three main views of the design pattern. Each use case in the integrated metamodel each tag represents a feature of the design pattern. Each use case has its name, actor and the extended or included use case. The tags in 6 to 16 represented the interaction of the different objects of the design pattern. Each tag contains all the different classes with its structural features and methods as well as the sequence of the classes. Hence, all the different views features are integrated into one XML file.
This will add a lot of benefits to the detection approach since all the different features and collaboration sequences are maintained and integrated into one XML file.

Adapter

```
1-  <?xml version="1.0" encoding="UTF-8"?>
2-  <IntegratedModel id="1" name="Adapter">
3-  //UseCase 1
4-  <UseCase id="UC1" name="Adapting the Complex Object">
5-  <Actor id="1" name="Client" />
6-  <Interaction>
7-  <class id="UC1-C1" name="Adapter" isAbstract=" No">
8-  <Message id="Msg1" name="requiredMethod " visibility="Visible" />
9-  <implements id="Target" />
10- <Association name="Compose" targetClass="UC1-C2" AggregationKind="
    Compossite" />
11- </class>
12- <class id="UC1-C2" name="Adaptee" isAbstract=" No">
```

```
13- <Message id="Msg3" name="specifiedMethod" visibility="Visible" />
14- </class>
15- </Interaction>
16- </UseCase>
17- </IntegratedModel>
```

The previous XML code represented the UML integrated metamodel for Adapter design pattern. Line 4 and 5 represented the three main views of the design pattern. Each use case in the integrated metamodel each tag represents a feature of the design pattern. Each use case has its name, actor and the extended or included use case. The tags in 6 to 15 represented the interaction of the different objects of the design pattern. Each tag contains all the different classes with its structural features and methods as well as the sequence of the classes. Hence, all the different views features are integrated into one XML file.
This will add a lot of benefits to the detection approach since all the different features and collaboration sequences are maintained and integrated into one XML file.

Bridge

```
1-  <?xml version="1.0" encoding="UTF-8"?>
2-  <IntegratedModel id="1" name="Bridge">
3-  //UseCase 1
4-  <UseCase id="UC1" name="Decouple the Abstraction from The Implementation">
5-  <Actor id="1" name="Client" />
6-  <Interaction>
7-  <class id="UC1-C1" name="ConcreteAbstractrion" isAbstract=" No">
8-  <Message id="Msg1" name="operation " visibility="Visible" />
9-  <Association name="Compose" targetClass="Implementor" AggregationKind="
    Compossite" />
10- </class>
11- <class id="UC1-C2" name="ConcreteImplementor1" isAbstract=" No">
12- <implements id="Implementor" />
13- <Message id="Msg3" name="operationImp" visibility="Visible" />
14- </class>
15- <class id="UC1-C2" name="ConcreteImplementor2" isAbstract=" No">
16- <implements id="Implementor" />
17- <Message id="Msg3" name="operationImp" visibility="Visible" />
18- </class>
19- </Interaction>
20- </UseCase>
21- </IntegratedModel>
```

The previous XML code represented the UML integrated metamodel for Bridge design pattern. Line 4 and 6 represented the three main views of the design pattern. Each use case in the integrated metamodel each tag represents a feature of the design pattern. Each use case has its name, actor and the extended or included use case. The tags in 6 to 19 represented the interaction of the different objects of the design pattern. Each tag contains

all the different classes with its structural features and methods as well as the sequence of the classes. Hence, all the different views features are integrated into one XML file.
This will add a lot of benefits to the detection approach since all the different features and collaboration sequences are maintained and integrated into one XML file.

Flyweight

```
1-  <?xml version="1.0" encoding="UTF-8"?>
2-  <IntegratedModel id="1" name="Flyweight">
3-  //UseCase 1
4-  <UseCase id="UC1" name="Reduce Memory Load">
5-  <Actor id="1" name="Client" />
6-  <Interaction>
7-  <class id="UC1-C1" name="FlyweightFactory" isAbstract=" No">
8-  <Message id="Msg1" name="getFlyweight " visibility="Visible" />
9-  <Message id="Msg1" name="findFlyweight " visibility="Visible" />
10- <Association name="Compose" targetClass="Flyweight" AggregationKind="
    Compossite" />
11- </class>
12- <class id="UC1-C2" name="Flyweight" isAbstract=" No">
13- <Message id="Msg3" name="create" visibility="Visisble" />
14- <Message id="Msg4" name="operationExtrinsicState" visibility="Visible" />
15- </class>
16- </Interaction>
17- </UseCase>
18- </IntegratedModel>
```

The previous XML code represented the UML integrated metamodel for Flywright design pattern. Line 4 and 6 represented the three main views of the design pattern. Each use case in the integrated metamodel each tag represents a feature of the design pattern. Each use case has its name, actor and the extended or included use case. The tags in 6 to 16 represented the interaction of the different objects of the design pattern. Each tag contains all the different classes with its structural features and methods as well as the sequence of the classes. Hence, all the different views features are integrated into one XML file.
This will add a lot of benefits to the detection approach since all the different features and collaboration sequences are maintained and integrated into one XML file.

Chain of Responsibility

```
1-      <?xml version="1.0" encoding="UTF-8"?>
2-      <IntegratedModel id="1" name="Chain of Responsibility">
3-      //UseCase 1
4-      <UseCase id="UC1" name="Handling Request by Controller">
5-      <Actor id="1" name="Client">
6-      </Actor>
7-      <Interaction>
```

```
<InteractionFragment id="1" name="IF the Request Can't be handled by the Controller">
<MultiOperand id="1" name="UC1-Alt">
<InteractionOperand>
<Alt>
</Alt>
<Interaction>
<class id="UC1-C1" name ="Controller" isAbstract= " Yes" >
<Message id="Msg1" name="ForwardRequest" visibility="Visible">
</Message>
</class>
<class id="UC1-C2" name ="Mediator1" isAbstract= " Yes" >
<implements id="UC1-C1">
</implements>
<Message id="Msg2" name="ForwardRequest" visibility="Visible">
</Message>
</class>
<class id="UC1-C3" name ="Mediator2" isAbstract= " Yes" >
<implements id="UC1-C1">
</implements>
</class>
</Interaction>
</InteractionOperand>
</MultiOperand>
</InteractionFragment>
<class id="UC1-C1" name ="Controller" isAbstract= " Yes" >
<Message id="Msg1" name="ForwardRequest" visibility="Visible">
</Message>
<Message id="Msg2" name="HandleRequest" visibility="Visible">
</Message>
</class>
<class id="UC1-C2" name ="Mediator1" isAbstract= " Yes" >
<implements id="UC1-C1">
</implements>
<Message id="Msg2" name="ForwardRequest" visibility="Visible">
</Message>
</class>
<class id="UC1-C3" name ="Mediator2" isAbstract= " Yes" >
<implements id="UC1-C1">
</implements>
</class>
</Interaction>
</UseCase>
//UseCase 2
<UseCase id="UC2" name="Handling Request by a Mediator">
<Actor id="1" name="Client">
```

```
</Actor>
<Extend extension="UC1">
</Extend>
<Interaction>
<InteractionFragment id="2" name="IF the Request Can't Be handled then forward it to the next Mediator">
<MultiOperand id="1" name="UC2-Alt">
<InteractionOperand>
<Alt>
</Alt>
<Interaction>
<class id="UC1-C1" name ="Controller" isAbstract= " Yes" >
<Message id="Msg1" name="ForwardRequest" visibility="Visible">
</Message>
</class>
<class id="UC1-C2" name ="Mediator1" isAbstract= " Yes" >
<implements id="UC1-C1">
</implements>
<Message id="Msg2" name="ForwardRequest" visibility="Visible">
</Message>
</class>
<class id="UC1-C3" name ="Mediator2" isAbstract= " Yes" >
<implements id="UC1-C1">
</implements>
<Message id="Msg2" name="HandleRequest" visibility="Visible">
</Message>
</class>
</Interaction>
</InteractionOperand>
</MultiOperand>
</InteractionFragment>
<class id="UC2-C1" name ="Controller" isAbstract= " Yes" >
<Message id="Msg1" name="ForwardRequest" visibility="Visible">
</Message>
</class>
<class id="UC2-C2" name ="Mediator1" isAbstract= " Yes" >
<implements id="UC2-C1">
</implements>
<Message id="Msg2" name="ForwardRequest" visibility="Visisble">
</Message>
<Message id="Msg2" name="HandleRequest" visibility="Visisble">
</Message>
</class>
<class id="UC2-C3" name ="Mediator2" isAbstract= " Yes" >
<implements id="UC1-C1">
</implements>
```

```
98-     </class>
99-     </Interaction>
100-    </UseCase>
101-    //UseCase 3
102-    <UseCase id="UC3" name="Handling Request by a Mediator">
103-    <Actor id="1" name="Client">
104-    </Actor>
105-    <Extend extension="UC1">
106-    </Extend>
107-    <Interaction>
108-    <InteractionFragment id="3" name="IF the Request Can't Be handled then
forward it to the next Mediator">
109-    <MultiOperand id="1" name="UC3-Alt">
110-    <InteractionOperand>
111-    <Alt>
112-    </Alt>
113-    </InteractionOperand>
114-    </MultiOperand>
115-    </InteractionFragment>
116-    <class id="UC3-C1" name ="Controller" isAbstract= " Yes" >
117-    <Message id="Msg1" name="ForwardRequest" visibility="Visible">
118-    </Message>
119-    <Message id="Msg2" name="HandlePartially"  visibility="Visible">
120-    </Message>
121-    </class>
122-    <class id="UC3-C2" name ="Mediator1" isAbstract= " Yes" >
123-    <implements id="UC2-C1">
124-    </implements>
125-    <Message id="Msg3" name="ForwardPartialy" visibility="Visible">
126-    </Message>
127-    <Message id="Msg4" name="HandlePartially" visibility="Visible">
128-    </Message>
129-    </class>
130-    <class id="UC3-C3" name ="Mediator2" isAbstract= " Yes" >
131-    <implements id="UC1-C1">
132-    </implements>
133-    <Message id="Msg5" name="HandlePartially" visibility="Visible">
134-    </Message>
135-    </class>
136-    </Interaction>
137-    </UseCase>
138-    </IntegratedModel>
```

The previous XML code represented the UML integrated metamodel for a chain of responsibility design pattern. Line 4, 36, and 62 represented the three main views of the design pattern. In each use case in the UML integrated metamodel, every tag represents a feature of the design pattern. Each use case has its name, actor and extended or included use case. The tags in 7, 13, and 46 represents the interaction of the different objects of the design pattern. Each tag contains all the different classes with its structural features and methods as well as the sequence of the classes. Hence, all the different views features are integrated into one XML file.

This will add a lot of benefits to the design patterns detection approach since all the different features and collaboration sequences are maintained and integrated into one XML file.

Observer

```
1-      <?xml version="1.0" encoding="UTF-8"?>
2-      <IntegratedModel id="1" name="Observer">
3-      //UseCase 1
4-      <UseCase id="UC1" name="Watching Item">
5-      <Actor id="1" name="Client">
6-      </Actor>
7-      <Include includeCase="2">
8-      </Include>
9-      <Interaction>
10-     <class id="UC1-C1" name ="ConcreteSubject" isAbstract= " Yes" >
11-     <implements id="UC1-C00">
12-     </implements>
13-     <Message id="Msg1" name="RegisterObserver" visibility="Visible">
14-     </Message>
15-     <Message id="Msg2" name="RemoveObserver" visibility="Visible">
16-     </Message>
17-     <Message id="Msg3" name="CheckItem" visibility="Visible">
18-     </Message>
19-     </class>
20-     <class id="UC1-C2" name ="Obserever 1" isAbstract= " Yes" >
21-     <implements id="UC1-C01">
22-     </implements>
23-     </class>
24-     <class id="UC1-C3" name ="Obserever 2" isAbstract= " Yes" >
25-     <implements id="UC1-C01">
26-     </implements>
27-     </class>
28-     </Interaction>
29-     </UseCase>
30-     //UseCase 2
31-     <UseCase id="UC1" name="Item State Changed">
32-     <Actor id="1" name="Client">
33-     </Actor>
34-     <Interaction>
35-     <class id="UC1-C1" name ="ConcreteSubject" isAbstract= " Yes" >
36-     <implements id="UC1-C00">
```

```
37-     </implements>
38-     <Message id="Msg1" name="NotifyObserver" visibility="Visible">
39-     </Message>
40-     </class>
41-     <class id="UC1-C2" name ="Obserever 1" isAbstract= " Yes" >
42-     <implements id="UC1-C01">
43-     </implements>
44-     <Message id="Msg1" name="Update" visibility="Visible">
45-     </Message>
46-     </class>
47-     <class id="UC1-C3" name ="Obserever 2" isAbstract= " Yes" >
48-     <implements id="UC1-C01">
49-     </implements>
50-     <Message id="Msg1" name="Update" visibility="Visible">
51-     </Message>
52-     </class>
53-     </Interaction>
54-     </UseCase>
55-     </IntegratedModel>
```

The previous XML code represented the UML integrated metamodel for Observer design pattern. Line 4 and 31 represented the two main use cases of the design pattern. Each use case in the integrated metamodel each tag represents a feature of the design pattern. Each use case has its name, actor and the extended or included use case. The tags in 9, and 34 represented the interaction of the different objects of the design pattern. Each tag contains all the different classes with its structural features and methods as well as the sequence of the classes. Hence, all the different views features are integrated into one XML file.
This will add a lot of benefits to the detection approach since all the different features and collaboration sequences are maintained and integrated into one XML file

Strategy

```
1-  <?xml version="1.0" encoding="UTF-8"?>
2-  <IntegratedModel id="1" name="Strategy">
3-  //UseCase 1
4-  <UseCase id="UC1" name="Select the appropriate solution">
5-  <Actor id="1" name="Client" />
6-  <Interaction>
7-  <class id="UC1-C1" name="Context" isAbstract=" No">
8-  <Message id="Msg3" name="selectAlgorithm" visibility="Visible" />
9-  <Association name="Compose" targetClass="Strategy" AggregationKind="
    Compossite" />
10- </class>
11- <class id="UC1-C2" name="StrategyA" isAbstract=" No">
12- <Message id="Msg1" name="AlgorithmA " visibility="Visible" />
13- <implements id="Strategy" />
```

```
14- </class>
15- <class id="UC1-C3" name="StrategyB" isAbstract=" No">
16- <Message id="Msg1" name="AlgorithmB " visibility="Visible" />
17- <implements id="Strategy" />
18- </class>
19- </Interaction>
20- </UseCase>
21- </IntegratedModel>
```

The previous XML code represented the UML integrated metamodel for Strategy design pattern. Line 4 and 6 represented the three main views of the design pattern. Each use case in the integrated metamodel each tag represents a feature of the design pattern. Each use case has its name, actor and the extended or included use case. The tags in 6 to 16 represented the interaction of the different objects of the design pattern. Each tag contains all the different classes with its structural features and methods as well as the sequence of the classes. Hence, all the different views features are integrated into one XML file.
This will add a lot of benefits to the detection approach since all the different features and collaboration sequences are maintained and integrated into one XML file.

Mediator

```
1-  <?xml version="1.0" encoding="UTF-8"?>
2-  <IntegratedModel id="1" name="Mediator">
3-  //UseCase 1
4-  <UseCase id="UC1" name="Handle the Objects Communications">
5-  <Actor id="1" name="Client" />
6-  <Interaction>
7-  <class id="UC1-C1" name="ConcreteMediator" isAbstract=" No">
8-  <Message id="Msg3" name="mediate" visibility="Visible" />
9-  <implements id="Mediator" />
10- <Association name="use" targetClass="Colleague" AggregationKind=" none" />
11- </class>
12- <class id="UC1-C2" name="Colleague1" isAbstract=" No">
13- <Message id="Msg1" name="action1 " visibility="Visible" />
14- <Message id="Msg1" name="getState " visibility="Visible" />
15- <implements id="Colleague" />
16- </class>
17- <class id="UC1-C2" name="Colleague2" isAbstract=" No">
18- <Message id="Msg1" name="action2" visibility="Visible" />
19- <Message id="Msg1" name="getState " visibility="Visible" />
20- <implements id="Colleague" />
21- </class>
22- <class id="UC1-C3" name="StrategyB" isAbstract=" No">
23- <Message id="Msg1" name="AlgorithmB " visibility="Visible" />
24- <implements id="Strategy" />
25- </class>
```

```
26- </Interaction>
27- </UseCase>
28- </IntegratedModel>
```

The previous XML code represented the UML integrated metamodel for Mediator design pattern. Line 4 and 6 represented the three main views of the design pattern. Each use case in the integrated metamodel each tag represents a feature of the design pattern. Each use case has its name, actor and the extended or included use case. The tags in 6 to 26 represented the interaction of the different objects of the design pattern. Each tag contains all the different classes with its structural features and methods as well as the sequence of the classes. Hence, all the different views features are integrated into one XML file.
This will add a lot of benefits to the detection approach since all the different features and collaboration sequences are maintained and integrated into one XML file.

State

```
1-   <?xml version="1.0" encoding="UTF-8"?>
2-   <IntegratedModel id="1" name="State">
3-   //UseCase 1
4-   <UseCase id="UC1" name="Change Object State">
5-   <Actor id="1" name="Client" />
6-   <Interaction>
7-   <class id="UC1-C1" name="Context" isAbstract=" No">
8-   <Message id="Msg1" name="request " visibility="Visible" />
9-   <Association name="Compose" targetClass="State" AggregationKind="
     Compossite" />
10- </class>
11- <class id="UC1-C2" name="State1" isAbstract=" No">
12- <implements id="State" />
13- <Message id="Msg3" name="handleRequest" visibility="Visible" />
14- </class>
15- <class id="UC1-C3" name="State3" isAbstract=" No">
16- <implements id="State" />
17- <Message id="Msg3" name="handleRequest" visibility="Visible" />
18- </class>
19- </Interaction>
20- </UseCase>
21- </IntegratedModel>
```

The previous XML code represented the UML integrated metamodel for State design pattern. Line 4 and 6 represented the three main views of the design pattern. Each use case in the integrated metamodel each tag represents a feature of the design pattern. Each use case has its name, actor and the extended or included use case. The tags in 6 to 19 represented the interaction of the different objects of the design pattern. Each tag contains all the different classes with its structural features and methods as well as the sequence of the classes. Hence, all the different views features are integrated into one XML file.

This will add a lot of benefits to the detection approach since all the different features and collaboration sequences are maintained and integrated into one XML file.

Visitor

```
1-  <?xml version="1.0" encoding="UTF-8"?>
2-  <IntegratedModel id="1" name="Visitor:">
3-  //UseCase 1
4-  <UseCase id="UC1" name="Visit Class Elements to Perform Operations">
5-  <Actor id="1" name="Client" />
6-  <Interaction>
7-  <class id="UC1-C1" name="ElementA" isAbstract=" No">
8-  <Message id="Msg1" name="acceptVisitor " visibility="Visible" />
9-  <Message id="Msg2" name="operation " visibility="Visible" />
10- <implements id="Element" />
11- </class>
12- <class id="UC1-C1" name="ElementB" isAbstract=" No">
13- <Message id="Msg1" name="acceptVisitor " visibility="Visible" />
14- <Message id="Msg2" name="operation " visibility="Visible" />
15- <implements id="Element" />
16- </class>
17- <class id="UC1-C2" name="ConcreteVisitor" isAbstract=" No">
18- <implements id="Visitor" />
19- <Message id="Msg3" name="visitElementA" visibility="Visible" />
20- <Message id="Msg3" name="visitElementB" visibility="Visible" />
21- <implements id="Visitor" />
22- <Association name="use" targetClass="Element" AggregationKind=" none" />
23- </class>
24- </Interaction>
25- </UseCase>
26- </IntegratedModel>
```

The previous XML code represented the UML integrated metamodel for Visitor design pattern. Line 4 and 6 represented the three main views of the design pattern. Each use case in the integrated metamodel each tag represents a feature of the design pattern. Each use case has its name, actor and the extended or included use case. The tags in 6 to 24 represented the interaction of the different objects of the design pattern. Each tag contains all the different classes with its structural features and methods as well as the sequence of the classes. Hence, all the different views features are integrated into one XML file.

This will add a lot of benefits to the detection approach since all the different features and collaboration sequences are maintained and integrated into one XML file.

Template Method

```
1-  <?xml version="1.0" encoding="UTF-8"?>
2-  <IntegratedModel id="1" name="Template Method">
3-  //UseCase 1
4-  <UseCase id="UC1" name="Define Algorithm Skelton">
5-  <Actor id="1" name="Client" />
6-  <Interaction>
7-  <class id="UC1-C1" name="AbstracClass" isAbstract=" No">
8-  <Message id="Msg1" name="templateMethod  " visibility="Visible" />
9-  </class>
10- <class id="UC1-C2" name="ConcreteClass" isAbstract=" No">
11- <implements id="AbstracClass" />
12- <Message id="Msg2" name="premitiveOperation1" visibility="Visible" />
13- <Message id="Msg3" name="premitiveOperation2" visibility="Visible" />
14- </class>
15- </Interaction>
16- </UseCase>
17- </IntegratedModel>
```

The previous XML code represented the UML integrated metamodel for Template Method design pattern. Line 4 and 6 represented the three main views of the design pattern. Each use case in the integrated metamodel each tag represents a feature of the design pattern. Each use case has its name, actor and the extended or included use case. The tags in 6 to 16 represented the interaction of the different objects of the design pattern. Each tag contains all the different classes with its structural features and methods as well as the sequence of the classes. Hence, all the different views features are integrated into one XML file.

This will add a lot of benefits to the detection approach since all the different features and collaboration sequences are maintained and integrated into one XML file.

Command

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <IntegratedModel id="1" name="Command">
3.  //UseCase 1
4.  <UseCase id="UC1" name="Encapsulate a Request as an Object">
5.  <Actor id="1" name="Client" />
6.  <Interaction>
7.  <class id="UC1-C1" name="ConcreteCommand" isAbstract=" No">
8.  <Message id="Msg1" name="createCommand " visibility="Visible" />
9.  <Message id="Msg2" name="action " visibility="Visible" />
10. <implements id="Command" />
11. <Association name="Compose" targetClass="Reciever" AggregationKind="
    Composite" />
12. </class>
13. <class id="UC1-C2" name="Invoker" isAbstract=" No">
14. <implements id="" />
```

```
15. <Message id="Msg3" name="storeComman" visibility="Visible" />
16. <Message id="Msg4" name="executeCommand" visibility="Visible" />
17. </class>
18. <class id="UC1-C3" name="Reciever" isAbstract=" No">
19. <Message id="Msg3" name="action" visibility="Visible" />
20. </class>
21. </Interaction>
22. </UseCase>
23. </IntegratedModel>
```

The previous XML code represented the UML integrated metamodel for Command design pattern. Line 4 and 6 represented the three main views of the design pattern. Each use case in the integrated metamodel each tag represents a feature of the design pattern. Each use case has its name, actor and the extended or included use case. The tags in 6 to 21 represented the interaction of the different objects of the design pattern. Each tag contains all the different classes with its structural features and methods as well as the sequence of the classes. Hence, all the different views features are integrated into one XML file.
This will add a lot of benefits to the detection approach since all the different features and collaboration sequences are maintained and integrated into one XML file.

References:

1. http://www.omg.org/. *OMG Group*. Website.
2. Chihada, A., et al., *Source code and design conformance, design pattern detection from source code by classification approach.* Applied Soft Computing, 2015. **26**: p. 357-367.
3. Misbhauddin, M., *Towards an integrated metamodel based approach to software refactoring*. 2012.
4. De Lucia, A., et al. *Behavioral pattern identification through visual language parsing and code instrumentation*. in *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*. 2009. IEEE.
5. Tsantalis, N., et al., *Design pattern detection using similarity scoring.* IEEE Transactions on Software Engineering, 2006. **32**(11): p. 896-909.
6. Dong, J., Y. Zhao, and Y. Sun, *A matrix-based approach to recovering design patterns.* IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans, 2009. **39**(6): p. 1271-1282.
7. Meyer, M. and L. Wendehals, *Selective tracing for dynamic analyses.* Comprehension through Dynamic Analysis, 2005. **33**.
8. Lee, H., H. Youn, and E. Lee. *Automatic detection of design pattern for reverse engineering*. in *5th ACIS International Conference on Software Engineering Research, Management & Applications (SERA 2007)*. 2007. IEEE.
9. Gamma, E., *Design patterns: elements of reusable object-oriented software*. 1995: Pearson Education India.
10. Jacobson, I., *Object-oriented software engineering: a use case driven approach*. 1993: Pearson Education India.
11. Kaczor, O., Y.-G. Guéhéneuc, and S. Hamel. *Efficient identification of design patterns with bit-vector algorithm*. in *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*. 2006. IEEE.
12. Rumbaugh, J., et al., *Object-oriented modeling and design*. Vol. 199. 1991: Prentice-hall Englewood Cliffs, NJ.
13. Grady, B., *Object-oriented analysis and design with applications*. 1994, Addison Wesley Longman.
14. Arief, L.B. and N.A. Speirs. *A UML tool for an automatic generation of simulation programs*. in *Proceedings of the 2nd international workshop on Software and performance*. 2000. ACM.
15. Meyer, B., *UML: the positive spin.* American Programmer, 1997. **10**: p. 37-41.
16. Kruchten, P.B., *The 4+ 1 view model of architecture.* IEEE software, 1995. **12**(6): p. 42-50.
17. Iivari, J., *Object-orientation as structural, functional and behavioural modelling: a comparison of six methods for object-oriented analysis.* Information and Software Technology, 1995. **37**(3): p. 155-163.
18. http://www.uml-diagrams.org/. *UML diagrams*.
19. Misbhauddin, M. and M. Alshayeb, *Extending the UML use case metamodel with behavioral information to facilitate model analysis and interchange.* Software & Systems Modeling, 2015. **14**(2): p. 813-838.
20. Boronat, A., et al., *Formal model merging applied to class diagram integration.* Electronic Notes in Theoretical Computer Science, 2007. **166**: p. 5-26.

21. Da Silva, P.P. and N.W. Paton, *User interface modeling in UMLi.* IEEE software, 2003. **20**(4): p. 62-69.
22. Egyed, A. and N. Medvidovic. *Extending architectural representation in UML with view integration*. in *International Conference on the Unified Modeling Language*. 1999. Springer.
23. Tchertchago, A., *Formal Semantics for a UML fragment using UML/OCL Metamodeling.* Project, Dresden University of Technology, Department of Computer Science, Germany, February, 2002.
24. Misbhauddin, M. and M. Alshayeb, *An integrated metamodel-based approach to software model refactoring.* Software & Systems Modeling, 2017: p. 1-38.
25. Keller, R.K., et al. *Pattern-based reverse-engineering of design components*. in *Proceedings of the 21st international conference on Software engineering*. 1999. ACM.
26. Rasool, G., I. Philippow, and P. Mäder, *Design pattern recovery based on annotations.* Advances in Engineering Software, 2010. **41**(4): p. 519-526.
27. Stencel, K. and P. Wegrzynowicz. *Detection of diverse design pattern variants*. in *Software Engineering Conference, 2008. APSEC'08. 15th Asia-Pacific*. 2008. IEEE.
28. Vokáč, M. and J.M. Glattetre. *Using a domain-specific language and custom tools to model a multi-tier service-oriented application—experiences and challenges*. in *International Conference on Model Driven Engineering Languages and Systems*. 2005. Springer.
29. Paakki, J., et al. *Software metrics by architectural pattern mining*. in *Proceedings of the International Conference on Software: Theory and Practice (16th IFIP World Computer Congress)*. 2000.
30. *https://www.uni-paderborn.de/.*
31. Antoniol, G., R. Fiutem, and L. Cristoforetti. *Design pattern recovery in object-oriented software*. in *Program Comprehension, 1998. IWPC'98. Proceedings., 6th International Workshop on*. 1998. IEEE.
32. Tsantalis, N., et al., *Design pattern detection using similarity scoring.* IEEE transactions on software engineering, 2006. **32**(11).
33. *http://www.ptidej.net/.*
34. Niere, J., et al. *Towards pattern-based design recovery*. in *Proceedings of the 24th international conference on Software engineering*. 2002. ACM.
35. Shi, N. and R.A. Olsson. *Reverse engineering of design patterns from java source code*. in *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*. 2006. IEEE.
36. Beyer, D. and C. Lewerentz. *CrocoPat: Efficient pattern analysis in object-oriented programs*. in *Program Comprehension, 2003. 11th IEEE International Workshop on*. 2003. IEEE.
37. Heuzeroth, D., et al. *Automatic design pattern detection*. in *Program Comprehension, 2003. 11th IEEE International Workshop on*. 2003. IEEE.
38. Balanyi, Z. and R. Ferenc. *Mining design patterns from C++ source code*. in *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*. 2003. IEEE.

39. Wang, W. and V. Tzerpos. *Design pattern detection in Eiffel systems*. in *Reverse Engineering, 12th Working Conference on*. 2005. IEEE.

40. Ferenc, R., et al. *Design pattern mining enhanced by machine learning*. in *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*. 2005. IEEE.

41. Huang, H., et al., *A practical pattern recovery approach based on both structural and behavioral analysis.* Journal of Systems and Software, 2005. **75**(1): p. 69-87.

42. Arcelli, F. and L. Christina. *Enhancing software evolution through design pattern detection*. in *Software Evolvability, 2007 Third International IEEE Workshop on*. 2007. IEEE.

43. Rasool, G. and D. Streitfdert, *A survey on design pattern recovery techniques.* IJCSI International Journal of Computer Science Issues, 2011. **8**(2): p. 251-260.

44. Priya, R.K. *A survey: Design pattern detection approaches with metrics*. in *Emerging Trends In New & Renewable Energy Sources And Energy Management (NCET NRES EM), 2014 IEEE National Conference On*. 2014. IEEE.

45. Iacob, C. *A design pattern mining method for interaction design*. in *Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems*. 2011. ACM.

46. Arnold, M. and H. Corporaal. *Automatic detection of recurring operation patterns*. in *Hardware/Software Codesign, 1999.(CODES'99) Proceedings of the Seventh International Workshop on*. 1999. IEEE.

47. Alnusair, A., T. Zhao, and G. Yan. *Automatic recognition of design motifs using semantic conditions*. in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. 2013. ACM.

48. Pappalardo, G. and E. Tramontana. *Automatically discovering design patterns and assessing concern separations for applications*. in *Proceedings of the 2006 ACM symposium on Applied computing*. 2006. ACM.

49. Dong, J., Y. Sun, and Y. Zhao. *Design pattern detection by template matching*. in *Proceedings of the 2008 ACM symposium on Applied computing*. 2008. ACM.

50. Dabain, H., A. Manzer, and V. Tzerpos. *Design pattern detection using FINDER*. in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. 2015. ACM.

51. Stephan, M. and J.R. Cordy. *Identifying instances of model design patterns and antipatterns using model clone detection*. in *Modeling in Software Engineering (MiSE), 2015 IEEE/ACM 7th International Workshop on*. 2015. IEEE.

52. Seemann, J. and J.W. von Gudenberg. *Pattern-based design recovery of Java software*. in *ACM SIGSOFT Software Engineering Notes*. 1998. ACM.

53. Wendehals, L. and A. Orso. *Recognizing behavioral patterns atruntime using finite automata*. in *Proceedings of the 2006 international workshop on Dynamic systems analysis*. 2006. ACM.

54. Sudhakar, N. and J. Gyani. *Tecdp: a tool for extracting creational design patterns*. in *Proceedings of the International Conference and Workshop on Emerging Trends in Technology*. 2010. ACM.

55. Bernardi, M.L., M. Cimitile, and G. Di Lucca, *Design pattern detection using a DSL‐ driven graph matching approach.* Journal of Software: Evolution and Process, 2014. **26**(12): p. 1233-1266.

56. Di Martino, B. and A. Esposito, *A rule‐ based procedure for automatic recognition of design patterns in UML diagrams.* Software: Practice and Experience, 2016. **46**(7): p. 983-1007.

57. Ng, J.K.Y., Y.G. Guéhéneuc, and G. Antoniol, *Identification of behavioural and creational design motifs through dynamic analysis.* Journal of Software Maintenance and Evolution: Research and Practice, 2010. **22**(8): p. 597-627.

58. Ng, J.K.Y., Y.G. Guéhéneuc, and G. Antoniol, *Identification of behavioural and creational design motifs through dynamic analysis.* Journal of Software: Evolution and Process, 2010. **22**(8): p. 597-627.

59. Tekin, U. and F. Buzluca, *A graph mining approach for detecting identical design structures in object-oriented design models.* Science of Computer Programming, 2014. **95**: p. 406-425.

60. Fontana, F.A. and M. Zanoni, *A tool for design pattern detection and software architecture reconstruction.* Information sciences, 2011. **181**(7): p. 1306-1324.

61. Gaitani, M.A.G., et al., *Automated refactoring to the null object design pattern.* Information and Software Technology, 2015. **59**: p. 33-52.

62. Christopoulou, A., et al., *Automated refactoring to the Strategy design pattern.* Information and Software Technology, 2012. **54**(11): p. 1202-1214.

63. De Lucia, A., et al., *Design pattern recovery through visual language parsing and source code analysis.* Journal of Systems and Software, 2009. **82**(7): p. 1177-1193.

64. Wang, W. and V. Tzerpos, *DPVK-an eclipse plug-in to detect design patterns in Eiffel systems.* Electronic Notes in Theoretical Computer Science, 2004. **107**: p. 71-86.

65. Kaczor, O., Y.-G. Guéhéneuc, and S. Hamel, *Identification of design motifs with pattern matching algorithms.* Information and Software Technology, 2010. **52**(2): p. 152-168.

66. Fabry, J. and T. Mens, *Language-independent detection of object-oriented design patterns.* Computer Languages, Systems & Structures, 2004. **30**(1): p. 21-33.

67. Antoniol, G., et al., *Object-oriented design patterns recovery.* Journal of Systems and Software, 2001. **59**(2): p. 181-196.

68. Zanoni, M., F.A. Fontana, and F. Stella, *On applying machine learning techniques for design pattern detection.* Journal of Systems and Software, 2015. **103**: p. 102-117.

69. Haitzer, T. and U. Zdun, *Semi-automatic architectural pattern identification and documentation using architectural primitives.* Journal of Systems and Software, 2015. **102**: p. 35-57.

70. Fontana, F.A., S. Maggioni, and C. Raibulet, *Understanding the relevance of micro-structures for design patterns detection.* Journal of Systems and Software, 2011. **84**(12): p. 2334-2347.

71. Philippow, I., et al., *An approach for reverse engineering of design patterns.* Software and Systems Modeling, 2005. **4**(1): p. 55-70.

72. Qing-hua, L., Z. Zhi-xiang, and B. Ke-rong, *Design pattern mining using graph matching.* Wuhan University Journal of Natural Sciences, 2004. **9**(4): p. 444-448.

73. Alnusair, A., T. Zhao, and G. Yan, *Rule-based detection of design patterns in program code.* International Journal on Software Tools for Technology Transfer, 2014. **16**(3): p. 315-334.

74. Issaoui, I., N. Bouassida, and H. Ben-Abdallah, *Using metric-based filtering to improve design pattern detection approaches.* Innovations in Systems and Software Engineering, 2015. **11**(1): p. 39-53.

75. Issaoui, I., N. Bouassida, and H. Ben-Abdallah, *A design pattern detection approach based on semantics.* Software Engineering Research, Management and Applications 2012, 2012: p. 49-63.

76. Kim, H. and C. Boldyreff. *A method to recover design patterns using software product metrics*. in *International Conference on Software Reuse*. 2000. Springer.

77. Pande, A., M. Gupta, and A.K. Tripathi, *A new approach for detecting design patterns by graph decomposition and graph isomorphism.* Contemporary Computing, 2010: p. 108-119.

78. Bouassida, N. and H. Ben-Abdallah. *A new approach for pattern problem detection*. in *Advanced Information Systems Engineering*. 2010. Springer.

79. Kirasić, D. and D. Basch. *Ontology-based design pattern recognition*. in *International Conference on Knowledge-Based and Intelligent Information and Engineering Systems*. 2008. Springer.

80. Bouassida, N. and H. Ben-Abdallah, *Structural and behavioral detection of design patterns.* Advances in Software Engineering, 2009: p. 16-24.

81. Bernardi, M.L., M. Cimitile, and G.A. Di Lucca. *A model-driven graph-matching approach for design pattern detection*. in *Reverse Engineering (WCRE), 2013 20th Working Conference on*. 2013. IEEE.

82. Zhang, Z.-X., Q.-H. Li, and K.-R. Ben. *A new method for design pattern mining*. in *Machine Learning and Cybernetics, 2004. Proceedings of 2004 International Conference on*. 2004. IEEE.

83. Paydar, S. and M. Kahani. *A semantic web based approach for design pattern detection from source code*. in *Computer and Knowledge Engineering (ICCKE), 2012 2nd International eConference on*. 2012. IEEE.

84. De Lucia, A., et al. *A two phase approach to design pattern recovery*. in *Software Maintenance and Reengineering, 2007. CSMR'07. 11th European Conference on*. 2007. IEEE.

85. Muangon, W. and S. Intakosum. *Adaptation of design pattern retrieval using CBR and FCA*. in *Computer Sciences and Convergence Information Technology, 2009. ICCIT'09. Fourth International Conference on*. 2009. IEEE.

86. Chen, L. and M. Qiu. *An algorithm for automatic mining design pattern*. in *Computer Science and Education (ICCSE), 2010 5th International Conference on*. 2010. IEEE.

87. Ren, W. and W. Zhao. *An observer design-pattern detection technique*. in *Computer Science and Automation Engineering (CSAE), 2012 IEEE International Conference on*. 2012. IEEE.

88. Gupta, M., et al. *Design pattern detection by normalized cross correlation*. in *Methods and Models in Computer Science (ICM2CS), 2010 International Conference on*. 2010. IEEE.

89. Gupta, M., R.S. Rao, and A.K. Tripathi. *Design pattern detection using inexact graph matching*. in *Communication and Computational Intelligence (INCOCCI), 2010 International Conference on*. 2010. IEEE.

90. Ferenc, R., et al. *Design pattern mining enhanced by machine learning*. in *21st IEEE International Conference on Software Maintenance (ICSM'05)*. 2005. IEEE.

91. Pande, A., M. Gupta, and A. Tripathi. *Design pattern mining for GIS application using graph matching techniques*. in *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on*. 2010. IEEE.

92. Basu, N., S. Chatterjee, and N. Chaki. *Notice of Violation of IEEE Publication Principles Design Pattern Mining from Source Code for Reverse Engineering*. in *TENCON 2005 2005 IEEE Region 10*. 2005. IEEE.

93. Alhusain, S., et al. *Towards machine learning based design pattern recognition*. in *Computational Intelligence (UKCI), 2013 13th UK Workshop on*. 2013. IEEE.

94. Kramer, C. and L. Prechelt. *Design recovery by automated search for structural design patterns in object-oriented software*. in *Reverse Engineering, 1996., Proceedings of the Third Working Conference on*. 1996. IEEE.

95. Binun, A. and G. Kniesel. *DPJF-design pattern detection with high accuracy*. in *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*. 2012. IEEE.

96. Washizaki, H., et al. *Detecting design patterns using source code of before applying design patterns*. in *Computer and Information Science, 2009. ICIS 2009. Eighth IEEE/ACIS International Conference on*. 2009. IEEE.

97. Stoianov, A. and I. Şora. *Detecting patterns and antipatterns in software using Prolog rules*. in *Computational Cybernetics and Technical Informatics (ICCC-CONTI), 2010 International Joint Conference on*. 2010. IEEE.

98. Thongrak, M. and W. Vatanawood. *Detection of design pattern in class diagram using ontology*. in *Computer Science and Engineering Conference (ICSEC), 2014 International*. 2014. IEEE.

99. Dongjin, Y., J. Ge, and W. Wu. *Detection of design pattern instances based on graph isomorphism*. in *Software Engineering and Service Science (ICSESS), 2013 4th IEEE International Conference on*. 2013. IEEE.

100. Pradhan, P., A.K. Dwivedi, and S.K. Rath. *Detection of design pattern using graph isomorphism and normalized cross correlation*. in *Contemporary Computing (IC3), 2015 Eighth International Conference on*. 2015. IEEE.

101. Thankappan, J. and V. Patil. *Detection of Web Design Patterns Using Reverse Engineering*. in *Advances in Computing and Communication Engineering (ICACCE), 2015 Second International Conference on*. 2015. IEEE.

102. Pandel, A., M. Gupta, and A. Tripathi. *DNIT—A new approach for design pattern detection*. in *Computer and Communication Technology (ICCCT), 2010 International Conference on*. 2010. IEEE.

103. Dong, J., D.S. Lad, and Y. Zhao. *DP-Miner: Design pattern discovery using matrix*. in *Engineering of Computer-Based Systems, 2007. ECBS'07. 14th Annual IEEE International Conference and Workshops on the*. 2007. IEEE.

104. Nguyen, T. and R. Pooley. *Effective Recognition of Patterns in Object-Oriented Designs*. in *Software Engineering Advances, 2009. ICSEA'09. Fourth International Conference on*. 2009. IEEE.

105. Arcelli, F., S. Masiero, and C. Raibulet. *Elemental design patterns recognition in Java*. in *Software Technology and Engineering Practice, 2005. 13th IEEE International Workshop on*. 2005. IEEE.

106. De Lucia, A., et al. *ePadEvo: A tool for the detection of behavioral design patterns*. in *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*. 2015. IEEE.

107. Sandhu, P.S., P.P. Singh, and A.K. Verma. *Evaluating quality of software systems by design patterns detection*. in *Advanced Computer Theory and Engineering, 2008. ICACTE'08. International Conference on*. 2008. IEEE.

108. Lebon, M. and V. Tzerpos. *Fine-grained design pattern detection*. in *Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual*. 2012. IEEE.

109. Rasool, G. and P. Mäder. *Flexible design pattern detection based on feature types*. in *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*. 2011. IEEE.

110. Yu, D., et al. *From sub-patterns to patterns: an approach to the detection of structural design pattern instances by subgraph mining and merging*. in *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*. 2013. IEEE.

111. Heuzeroth, D., S. Mandel, and W. Lowe. *Generating design pattern detectors from pattern specifications*. in *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*. 2003. IEEE.

112. He, C., Z. Li, and K. He. *Identification and Extraction of Design Pattern Information in Java Program*. in *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2008. SNPD'08. Ninth ACIS International Conference on*. 2008. IEEE.

113. De Lucia, A., et al. *Improving behavioral design pattern detection through model checking*. in *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*. 2010. IEEE.

114. Albin-Amiot, H., et al. *Instantiating and detecting design patterns: Putting bits and pieces together*. in *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*. 2001. IEEE.

115. Haqqie, S. and A.A. Shahid. *Mining Design Patterns for Architecture Reconstruction using an Expert System*. in *9th International Multitopic Conference, IEEE INMIC 2005*. 2005. IEEE.

116. Bernardi, M.L. and G.A. Di Lucca. *Model-driven detection of Design Patterns*. in *Software Maintenance (ICSM), 2010 IEEE International Conference on*. 2010. IEEE.

117. Nagy, A. and B. Kovari. *Programming language neutral design pattern detection*. in *Computational Intelligence and Informatics (CINTI), 2015 16th IEEE International Symposium on*. 2015. IEEE.

118. Fayad, M., J. Rajagopalan, and H. Hamza. *Recovery design pattern*. in *Information Reuse and Integration, 2003. IRI 2003. IEEE International Conference on*. 2003. IEEE.

119. Wegrzynowicz, P. and K. Stencel. *Relaxing queries to detect variants of design patterns*. in *Computer Science and Information Systems (FedCSIS), 2013 Federated Conference on*. 2013. IEEE.

120. Li, W., G. Chen, and J. Pan. *Research on detecting and validating design pattern instances from source code*. in *Computer Science & Service System (CSSS), 2012 International Conference on*. 2012. IEEE.

121. Miao, K., et al. *Run-time discovery of Java design patterns*. in *Artificial Intelligence, Management Science and Electronic Commerce (AIMSEC), 2011 2nd International Conference on*. 2011. IEEE.

122. Smith, J.M. and D. Stotts. *SPQR: Flexible automated design pattern extraction from source code*. in *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*. 2003. IEEE.

123. Zhu, H., et al. *Tool support for design pattern recognition at model level*. in *Computer Software and Applications Conference, 2009. COMPSAC'09. 33rd Annual IEEE International*. 2009. IEEE.

124. Guéhéneuc, Y.-G. and G. Antoniol, *Demima: A multilayered approach for design pattern identification.* IEEE Transactions on Software Engineering, 2008. **34**(5): p. 667-684.

125. Antoniol, G., R. Fiutem, and L. Cristoforetti. *Using metrics to identify design patterns in object-oriented software*. in *Software Metrics Symposium, 1998. Metrics 1998. Proceedings. Fifth International*. 1998. IEEE.

Vitae

Name : Abdullah Alwi Hussien Al-Baity

Nationality : Yemeni

Date of Birth : 7/28/1985

Email : mr.albaity@gmail.com

Address : Dhahran Saudi Arabia

Academic Background : Abdullah Al-Baity earned his Bachelor of Computer Science degree from Al-Ahgaff University Hadramout, Yemen, in June 2010. He completed his Master in King Fahd University of Petroleum and Minerals (KFUPM) Technology, Dhahran, Saudi Arabia in January 2017. His research interests include empirical studies in software engineering, Enterprise Resource Planning (SAP), Design Patterns, Project Management, Programming Languages, Requirements engineering, and Programming Languages.

Publications : **Abdullah A. Al-Baity**, Kanaan Faisal, and Moataz Ahmed, **Software Reuse: The State of Art Software Reuse**, worldcomp-proceedings p2013