RECOMMENDING CREATIONAL DESIGN PATTERNS BY DERIVING CORRESPONDING ANTI-PATTERNS

NADIA NAHAR
Master of Science in Software Engineering
Institute of Information Technology, University of Dhaka
Registration Number: 2010-113-309
Session: 2014-15

A Thesis
Submitted to the Master of Science in Software Engineering Program Office
of the Institute of Information Technology, University of Dhaka
in Partial Fulfillment of the
Requirements for the Degree

Master of Science in Software Engineering

Institute of Information Technology
University of Dhaka
DHAKA, BANGLADESH

RECOMMENDING CREATIONAL DESIGN PATTERNS BY DERIVING CORRESPONDING ANTI-PATTERNS

NADIA NAHAR

Approved:

Signature *Date*

Supervisor: Dr. Kazi Muheymin-Us-Sakib

Committee Member: Mohd. Zulfiquar Hafiz

Committee Member: Dr. Mohammad Shoyaib

Committee Member: Rayhanur Rahman

Committee Member: Dr. Muhammad Mahbub Alam

Student: Nadia Nahar

To *A. M. Nasir Ullah*, my father
who has always been there for me and motivated me

Abstract

Design patterns refer to the optimal solutions for the common recurring design problems, while anti-patterns are the bad solutions. For creating a good-quality software, appropriate design patterns are required. This selection task can be made easier with the assistance of design pattern recommendation systems. Like any design pattern, the recommendation of creational patterns is not an easy task because of the difficulties in logically defining the manual process of pattern selection. In this situation, it can be logically feasible to use anti-pattern detection, as the existence of certain anti-patterns indicates that the design can be improved by applying particular design patterns.

The first contribution of this thesis is to logically derive the characteristics of the *missing* creational design patterns. The second contribution is to recommend the creational patterns based on the detection of these derived characteristics in a faulty software design. For deriving the full anti-pattern information, that is, class structure, interactions, and linguistic relationships, three levels of analysis are performed - structural, behavioral and semantic analysis. The outcome of this analysis is encapsulated in a tool named as Anti-pattern based Creational Design Pattern Recommender (ACDPR). For generating the recommendations, it conducts three levels of matching

(structural, behavioral and semantic), and detects the analyzed anti-pattern characteristics in the software design. Each design patterns is given a score, calculated using the results of the matching levels. These scores determine which design patterns are to be recommended, and which are to be suggested for designers' further considerations.

A case study has been conducted for evaluating the applicability of the proposed approach. The case study is carried on a badly designed project requiring Abstract Factory, named as *Painter*. This case study justifies that, the recommendation process leads to the correct recommendations. For experimental analysis of ACDPR on the software designs requiring creational design patterns, the prototype of ACDPR was implemented using java. The dataset was prepared by gathering 21 projects that require any one of the creational patterns. For this dataset, ACDPR provides a precision of 1, recall of 0.95, and F-measure of 0.97.

Acknowledgments

"All praises are due to Allah"

First of all, I praise Allah, The Almighty and The Lord of The World, for giving me this opportunity and granting me the ability to continue my research work effectively.

I take this opportunity to express my significant appreciation and profound respects to my thesis supervisor Dr. Kazi Muheymin-Us-Sakib, Director and Associate Professor, Institute of Information Technology, University of Dhaka. Without his support and guidance, this research could not be successful. He has been relentless in his efforts to bring the best out of me. He has been not only a technical supervisor but also a philosophical mentor. I am truly blessed to have a leader like him.

I would like to pass on my ardent appreciation to all faculty members, Institute of Information Technology, University of Dhaka, for their support, motivation, criticism and productive feedback which has enormously reinforced my confidence during my thesis. I would also thank my thesis external Dr. Md. Saidur Rahman for his constructive feedback.

I am also thankful to the DSSE Student Research group, Institute of Information Technology, University of Dhaka, for their continuous support and valuable feedback that helped me in improving my work.

I would also like to thank the ICT Division, Ministry of Posts, Telecommunications and Information Technology, Bangladesh, for giving me research fellowship and supporting my research work.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Design patterns are the optimized solution to the common software design problems. For creating a good-quality design, appropriate design patterns are required to be applied. Designers usually apply their expertise and discretion to choose the suitable pattern for a particular problem. This manual approach is time-consuming and requires expertise. An automatic suggestion based system will always be helpful to take the right decision. Keeping this objective in mind this research intends to formalize a recommendation system for creational design patterns. This research believes that proper patterns should be identified not only from the pattern intent but also from software design problems. Hence this work first identifies the *missing* design pattern (that is, anti-pattern) from the bad software design, which ultimately leads to the actual pattern that needs to be considered.

This chapter demonstrates the issues of the recommendation task and introduces the research challenges out of these. It also briefly describes the contribution and achievement of this research. Finally the organization of this thesis is indicated for giving a reading guideline to the readers.

1.1 Motivating Example

Let us consider an example of a project named *Painter* which is needed to be designed. The usage scenario of the project is –

"The *Paint* can draw three types of *Shape*s - *Circle*, *Triangle* or *Square*. The *Shape*s can be filled with three *Color*s - *Red*, *Blue* or *Green*. *Circle*s will be *Red*, *Triangle*s will be *Blue* and *Square*s will be *Green*."

An ideal design of the project is shown in Figure 1.1. From the scenario it can be noticed that, there are three groups of classes – *Circle* and *Red*, *Triangle* and *Blue* and *Square* and *Green*. These classes are to be instantiated together to meet the usage scenario requirement. This can be maintained using individual creator classes that instantiates the classes in one particular group. The client class can get these grouped class instances through the creator class of that group. It reduces the overhead of the client class to enforce the requirement of the class groups. In Figure 1.1, the *RedCircleFactory*, *BlueTriangleFactory* and *GreenSquareFactory* are the creator classes. *RedCircleFactory* creates *Red* and *Circle*. *BlueTriangleFactory* instantiates *Blue* and *Triangle*. *Green* and *Square* are created by creator *GreenSquareFactory*. The client class *Paint* gets these grouped class instantiations by calling its required creator class.

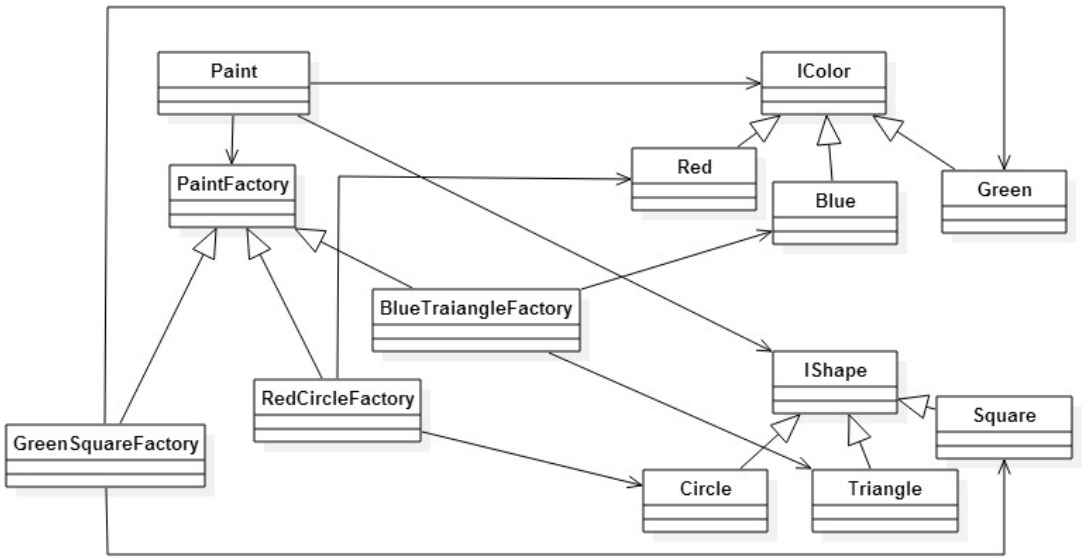
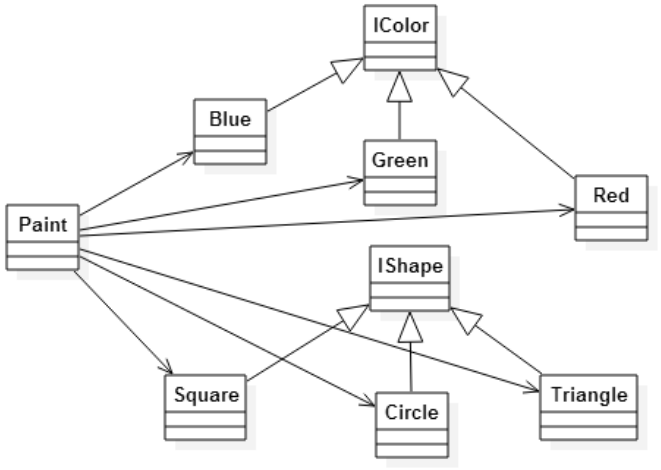


Figure 1.1: Ideal Design Example of *Painter*

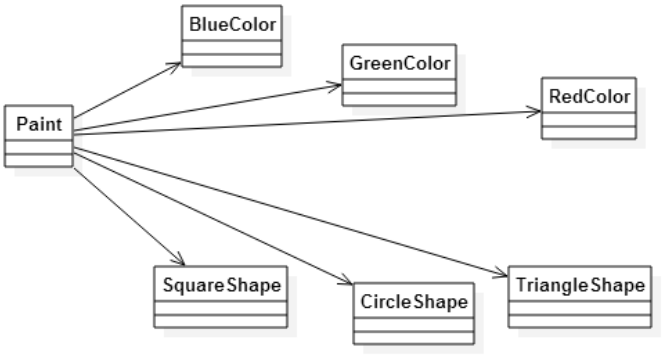
On the other hand two bad designs of the project are shown in Figure 1.2. In these designs the client class *Painter* has the responsibility to decide which classes are to be instantiated. This puts extra overhead on the client class to determine the classes to be instantiated together as per the requirements. The class groups are also unidentifiable from both these designs.

Also the design in Figure 1.1 is more maintainable and adaptable to new requirements than designs in Figure 1.2. If a requirement of a new class group comes, creating a new creator class will handle that change requirement in the ideal design. However the bad designs in Figure 1.2 cannot handle it as the client class will be responsible for this change requirement. The logic of the client class will be needed to be changed making the designs less maintainable.

This thesis aims to identify these bad designs, and help the designers to create a good design for their software by recommending appropriate designs.



(a)



(b)

Figure 1.2: Bad Designs of *Painter*

1.2 Design Pattern Recommendation

Design patterns formalize reusable solutions for some common recurring problems, while anti-patterns are the bad solutions for those which degrade the quality of software. Unlike anti-patterns, design patterns are well structured and the best proven answers for some defined scenarios; these optimize

the software construction effort as well as time, ensuring reusability and maintainability. Now design patterns are often mentioned as double-edged sword, as selecting the right pattern can produce good-quality software while selecting a wrong one (that is, anti-pattern) makes it disastrous [1]. Thus which patterns to use in which situation is a wise decision to take. Oppositely anti-patterns' existence can give an essence of which patterns to use in that situation by giving an indication of the existing design problems. The selection of a right pattern in the right part of the software is difficult. Because mapping software usage scenario or user description with pattern intent is a manual and hectic task. Detecting anti-pattern in software design and selecting a complementary pattern to replace the detected anti-pattern can make this hectic task easier.

Pattern recommendation systems give a direction on choosing proper patterns for a particular scenario by suggesting applicable patterns. Researches have been conducted for proposing pattern recommendation systems and improving their performance in offering relevant patterns. However the accuracy of correct pattern selection and recommendation is yet to be improved, since the current recommendation systems cannot provide a good precision due to the following challenges. The main challenge of automated pattern selection lies in logically defining the manual process of mapping human requirements with design pattern intent. The human requirements, be in form of usage scenario, designers' answers to questions or cases residing in knowledge base, lack connection with the design patterns. This is why the requirements have been inadequate to accurately extract the required design patterns. On the other hand establishing direct connection between design

patterns and user requirements could solve the problem, but it is not possible due to the fact that the user requirements cannot express the ideal design requirements for good software. However anti-patterns can be detected after a faulty design is created from user requirements. Now as every design pattern has its own context of design problems that it solves and every anti-pattern causes specific design problems, a relationship should exist between design patterns and anti-patterns that can be beneficial in pattern recommendation.

1.3 Issues in State-of-the-Art Approaches

Anti-pattern based design pattern recommendation is a relatively new field in the literature. However anti-pattern detection is a rich area of research [2, 3, 4, 5, 6, 7]. Fourati et. al. proposed an anti-pattern detection approach in design level using UML diagrams namely the class and sequence diagrams [2]. The detection was done based on some predefined threshold values of metrics where the values of the metrics were identified through structural, behavioral and semantic analysis. Nevertheless the approach detects only five defined anti-patterns. Another approach for design pattern detection was based on a machine learning technique, Support Vector Machines (SVM) [3], where the detection task is accomplished in three steps metric specification, SVM classifier training and detection of anti-pattern occurrences. However in this approach, the success of the detection process entirely depends on the training. Code-level design pattern recommendation based on anti-pattern detection is also available in the literature. A tool was proposed for recommending patterns dynamically during code development [8]. Anti-patterns

were identified in the code by matching structural and behavioral representations, and design patterns required to mitigate those anti-patterns were recommended. However pattern recommendation in coding phase is too late as the software is already designed and need to be changed after the recommendation, making the refactoring process costly.

On the other hand design pattern recommendation approaches that are not related to anti-patterns can be found in the literature. Most of these researches are based on text search, question-answer session or Case Based Reasoning (CBR). In the text-based search, pattern intent texts are matched with problem scenario [9, 10, 11]. However as problem scenarios are written in human language, these are often ambiguous, and also are not written from a designer's point of view. In question-answer based recommendation, designers are asked some questions about the software and the answers lead to find required patterns for that software [12, 13]. The problem is that the questions are often static or generic which makes it difficult for the designers to answer those correctly. In the CBR based approach, recommendations are given according to the previous experiences of pattern usage stored in a knowledge base [14, 15]. The major problem of this approach is that, the knowledge base needs to be very rich and accurate. A mixed approach was proposed by Sahly et. al. [16] where all three mentioned approaches were used for pattern recommendation, and thus suffered from the above stated problems. Navarro et. al. proposed a different recommendation system for suggesting additional patterns to the designer while a collection of patterns are already selected [17]. However as the approach requires some previous selected patterns, it cannot be used for new software being developed. None

of these cited researches focus on finding the design problems associated to the software and recommend design patterns with a target to solve those design problems, leading to less accuracy in pattern recommendation.

1.4 Research Questions

For recommending design patterns and improving the bad design (caused by anti-pattern), the problems of the provided software design need to be identified. Experts can easily analyze and understand design problems from requirements and select patterns according to the abilities (that is, the pattern intents) of those patterns to solve that problem. However, logically defining this manual process is challenging as human requirements is often ambiguous and conceptual that cannot be easily understood by programs. This problem can be mitigated if a reverse process (identifying design problems first and then suggesting patterns to mitigate the design problems) can be applied. For this purpose, a relationship between design patterns and anti-patterns needs to be established. Then design problems can be identified as forms of anti-patterns and the related design patterns can be suggested to remove those anti-patterns. According to GoF [18], there are 23 design patterns, and deriving all those patterns with their anti-patterns is a considerable amount of task. This is why this research focuses on the creational design patterns only. If the recommendation of creational design patterns may also be accomplished, recommending the other patterns are also possible. Hence the research question is –

- How to automate the process of creational pattern recommendation with an intention to solve design problems arose by anti-patterns?

A framework is required to be presented here, that may take software UMLs provided by Software Requirements Specification (SRS), identify anti-patterns in the softwares design, and recommend suitable design patterns for mitigating the design problems. This recommendation can be done by answering the following sub-questions –

1. How to derive the anti-patterns, to be detected from the creational design patterns?

First of all a relationship establishment between the anti- and design pattern is required. The derivation of anti-patterns related to the design patterns can lead to the design pattern recommendation after the anti-pattern detection in software design. For this the applicability of the creational design patterns needs to be analyzed to infer the alternative solutions to the patterns. These solutions are the *missing* creational patterns. The characteristics of these *missing* patterns are also needed to be identified.

2. How to detect the anti-patterns, and find the design patterns to be recommended?

Based on the identified characteristics of the anti-patterns, detection of those in the software design is to be performed. Different matching levels are needed to be defined for this. Using the outcomes of these matching levels, a score can be given to each of the design pattern. These scores can lead to the design pattern recommendation.

3. Finally how to develop a pattern recommendation framework combining the mentioned processes?

The above mentioned processes – anti-pattern derivation, anti-pattern detection and finally recommendation of the appropriate design patterns, needs to be amalgamated. The complete package needs to be provided in the form of a framework for better usage.

1.5 Contribution and Achievement

The first contribution of this research is to logically derive the characteristics of *missing* creational patterns. There are five design patterns in the creational category – Abstract Factory, Factory Method, Builder, Prototype and Singleton. These design patterns are analyzed individually to derive the alternative solutions. These solutions are marked as the anti-patterns of the design patterns. The characteristics of the anti-patterns are then described using the structural, behavioral and semantic attributes. Finally, these characteristics are formalized using a formal language, named Z [19].

The second contribution is to recommend creational design patterns using anti-pattern detection. The identified anti-pattern characteristics are used for matching with the initial software design. Three levels of matching are executed for the anti-pattern detection – structural matching, behavioral matching and semantic matching. The matching policy is used for the score calculation of the design patterns. Finally, the creational design patterns are recommended and partially suggested based on the matched attributes score.

The proposed framework has been implemented using the programming language, Java. The evaluation of its competence has been done by applying the framework on 21 sample projects and verifying the results. The prototype results are evaluated by testing the recommendation accuracy. The precision, recall and F-measure are calculated for the sample dataset which are 1, 0.95 and 0.97 respectively.

A case study has been conducted here for the assessment of the proposed approach. A project requiring Abstract Factory have been chosen as the example project. All five creational design patterns' level matchings are performed to identify which patterns are to be recommended and which are to be suggested. This case study shows ACDPR's competence in recommending design pattern. The stepwise demonstration of the approach makes it clear why ACDPR's precision, recall and F-measure are high.

1.6 Organization of the Thesis

This section gives an overview of the remaining chapters of this thesis. The chapters are organized as follows –

Chapter 2: Some preliminaries of software design patterns and anti-patterns are discussed along with their characteristical attributes. It helps to increase the understandability of the reader in this research domain.

Chapter 3: To the best of author's knowledge no existing literature incorporates anti-pattern detection in design pattern recommendation

in the software design phase. However in the literature, design patterns are recommended using different approaches. This chapter shows the existing researches in design pattern recommendation.

Chapter 4: In this chapter the anti-patterns of the creational design patterns are derived. The characteristics of these anti-patterns are also identified, and a formal specification of these characteristics are provided.

Chapter 5: An anti-pattern based design pattern recommendation framework is presented in this chapter. This framework detects the pre-defined *missing* creational patterns in the inputted software design, and recommends suitable design patterns.

Chapter 6: A case study on a sample project is shown here for the assessment of the proposed approach. This case study increases the understandability of the recommendation process.

Chapter 7: The implementation of the framework and result analysis is presented here. The accuracy of the framework is measured in terms of precision, recall and F-measure.

Chapter 8: It is the concluding chapter which contains a discussion about the framework and some future directions.

Chapter 2

Background Study

Software design allows the software engineers to model the system or product that is to be built [20]. It is the process of creating a software solution (or design solution) to one or more set of problems. A set of principles, concepts, and practices is part of the software design, that is concerned about the quality of the design. For creating optimized software design predefined design solutions exist for predefined design problems. These are called software design patterns. If the design patterns are not followed in software design, anti-patterns are raised that are the bad design solutions [21, 22]. Thus design patterns and anti-patterns are two important concepts in software design. In this chapter, the design patterns and anti-patterns are introduced along with a discussion of their characteristical attributes.

2.1 Design Pattern

Design patterns are the optimal solutions to the common recurring design problems. These provide efficient software designs, if selected and applied properly [23, 24]. Design patterns can significantly accelerate the design and development process by providing proven and tested design solutions [25, 26]. For Object-Oriented Programming (OOP), 23 patterns are defined by GoF, as the solutions of known design problems [18]. There are other patterns in different contexts as well, like Enterprise Architecture (EA) [27], security [28], Service-Oriented Architecture (SOA) [29] patterns, etc.

OOP Design patterns are classified into three categories based on the purpose of the patterns - structural, behavioral, and creational [18].

Structural Design Pattern

Structural patterns deal with the ways classes or objects are composed for creating larger structures. These describe ways to compose objects to realize new functionality. The structural patterns are – Adapter, Bridge, Composite, Decorator, Facade, Flyweight and Proxy [18].

Behavioral Design Pattern

Behavioral patterns focus on the interactions between the classes or objects. These describe not only the patterns of objects but also the patterns of communication among those. The behavioral patterns are – Observer, Strategy, Command, Interpreter, Chain of Responsibility, Iterator, Mediator, Memento, State, Template Method and Visitor [18].

Creational Design Pattern

Creational design patterns are concerned with the object instantiation process. These help making a system independent of how its objects are created, composed, and represented. The creational patterns are - Abstract Factory, Factory Method, Builder, Prototype and Singleton, etc [18]. The intents of these patterns are provided in Table 2.1.

In this thesis, only the creational patterns are considered, as considering all the design patterns is an immense task for the scope of this thesis. And thus the five creational design patterns are described here briefly [18].

Table 2.1: Intents of the Creational Patterns [18]

Design Pattern	Intent
Abstract Factory	Provide an interface for creating families of related or dependent objects without specifying their concrete classes
Factory Method	Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses
Builder	Separate the construction of a complex object from its representation so that the same construction process can create different representations
Prototype	Specify the kinds of objects to create using a prototypical and instance, create new objects by copying this prototype
Singleton	Ensure a class has only one instance, and provide a global point of access to it

2.1.1 Abstract Factory

Abstract Factory is concerned with the instantiation of objects in groups or families. It is applied when there are families of related product objects that

are used together, and one of the multiple families of products is to be used in an execution path [18].

Usage Example

Figure 2.1 shows a usage example of Abstract Factory design pattern. Consider a client who wants to build rooms having different kinds of windows and doors. A window can have wood or glass as its materials. A door can also be built using one of these two materials. The doors and windows in a room can only have one kind of material. That is, it can have wooden window and wooden door, or glass door and glass window.

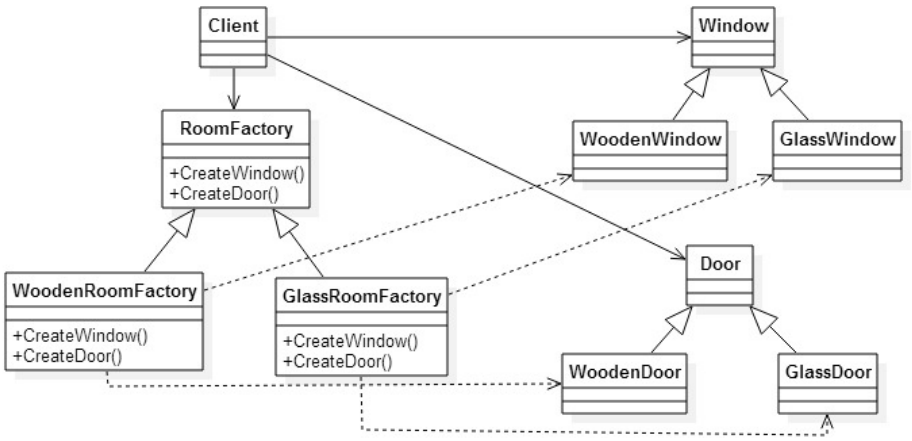


Figure 2.1: Usage Example of Abstract Factory

This problem is solved by defining an abstract factory for creating room elements, named as *RoomFactory*. There are also two abstract product classes for window and door, named as *Window* and *Door* respectively. Based on the product materials, the *Window* class have two concrete classes, *WoodenWindow* and *GlassWindow*. The *Door* class have two concrete classes, *WoodenDoor* and *GlassDoor*. The room elements are created by two concrete

factory classes that implement the abstract methods of the *RoomFactory*. The *WoodenRoomFactory* instantiates the *WoodenWindow* and the *WoodenDoor*. On the other hand, the *GlassRoomFactory* instantiates the *GlassWindow* and the *GlassDoor*. The *Client* class interacts with the abstract classes to get the concrete products. *Client* call the *CreateWindow*() and *CreateDoor*() operations of the *RooamFactory* to obtain required window and door instances. However, the *Client* class is not aware of the concrete classes being used. Thus the client stays independent of the product materials.

2.1.2 Factory Method

Factory Method is another creational pattern concerned about the object instantiations. Factory Method has a very similar intent like the Abstract Factory. Developers often become confused about the usage of these two design patterns. However, these two have a fine difference that while Abstract Factory works with the instantiation of families of classes, Factory Method focuses on the instantiation of a single class [18].

Usage Example

Usage example of Factory Method design pattern is shown in Figure 2.2. Consider a framework for applications that creates and maintains user documents [18]. The documents are application-specific. And, the application only knows when a new document should be created, not what kind of document to be created. The framework does not know, which document to create for which application.

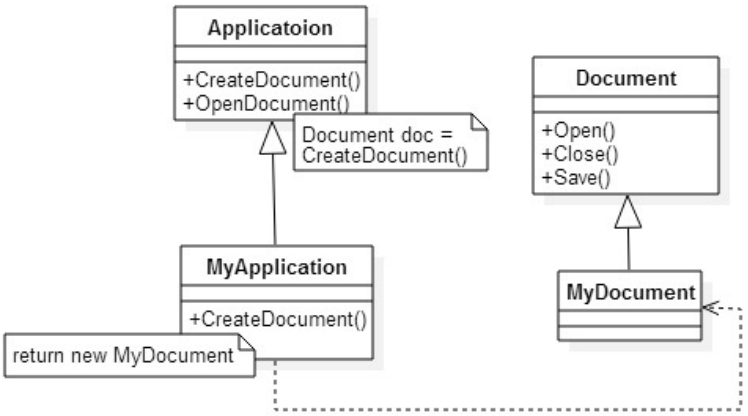


Figure 2.2: Usage Example of Factory Method

A solution to this design problem is offered by the Factory Method. The solution has two abstract classes, *Application* and *Document*. Two concrete classes extend these two classes, those are, *MyApplication* and *MyDocument*. *MyApplication* redefines the abstract *CreateDocument*() operation of *Application* to return the appropriate *Document* subclass, *MyDocument*. Once an *Application* subclass is instantiated, it can then instantiate application-specific *Document*s without knowing their class. Thus *CreateDocument*() is a factory method because of its responsibility of "manufacturing" an object.

2.1.3 Builder

Builder pattern is concerned with the creation of complex objects. Builder comes into action when a complex object having independent parts is required to be built allowing different representations of it [18].

Usage Example

Figure 2.3 shows the usage example of Builder design pattern. A reader for the Rich Text Format (RTF) document is able to convert RTF to different text formats like ASCII, TeX, etc. [18]. However, the number of possible conversions is open-ended. Thus the reader needs to add a new conversions easily and without modifying itself.

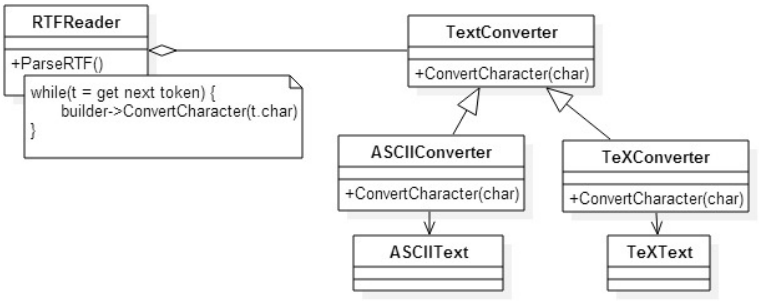


Figure 2.3: Usage Example of Builder

In the above figure, *RTFReader* is the director class. *RTFReader* parses the RTF document, and uses the *TextConverter* to perform the conversion. Subclasses of the *TextConverter* specialize in different conversions. For example, *ASCIIConverter* only converts to plain text. On the other hand, *TeXConverter* only produces TeX representations of the RTF token. In this way, each of the converter classes contribute a part of the complex object, and the abstract interface assembles the parts.

2.1.4 Prototype

Prototype is another creational design pattern, that concentrates on individual classes rather than multiple grouped ones. This pattern focuses on the creation of similar objects multiple times by cloning the existing ones [18].

Usage Example

A usage example of the Prototype design pattern is shown in Figure 2.4. Consider a drawing tool [18]. Different shapes like circle, triangle, etc., can be drawn using the tool. It also supports the feature of duplicating the drew shapes inside the tool.

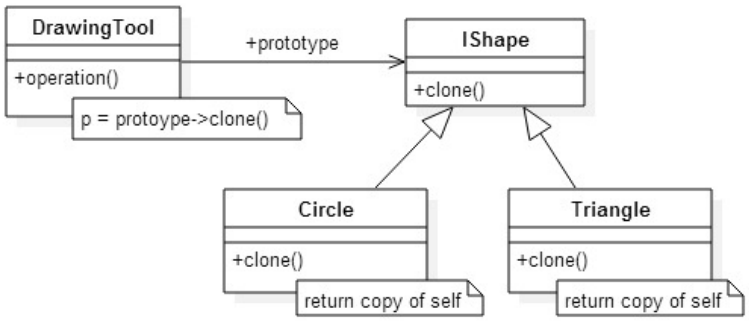


Figure 2.4: Usage Example of Prototype

DrawingTool represents the client class in the figure. It interacts with an abstract class *IShape*. Two concrete classes *Circle* and *Triangle* extend the *IShape*. This abstract class has a method *clone*() that is redefined by these two subclasses. The *clone*() method copies and returns the object of the class itself.

2.1.5 Singleton

Similar to Builder and Prototype, Singleton also concentrates on an individual class (can be mentioned as *singleton* class). It focuses on the instantiation of classes those must not be instantiated more than once [18].

Usage Example

In real life, it is very common to have exactly one instance of a system [18]. For example, an Operating System (OS) can have only one file system and one OS manager. A system can have many printers, but one printer spooler. An accounting system can only serve one company at a time.

A Singleton's solution to these scenarios is to make a class responsible for tracking its sole instance. It will ensure that no more than one instance can be created by any other classes.

If these design patterns are suitable for a software design, but are not used, then the designs are not appropriate. The designs contain anti-patterns or bad solutions. The anti-patterns are described in the next section.

2.2 Anti-pattern

Anti-pattern is a common response to a recurring problem, which is a bad solution [30]. It is the bad alternative solution to the design problem, that could be solved by applying an optimized solution (that is, design pattern). To be more specific, there can be many ways to solve a problem; among those, one is the optimized solution that is the design pattern and others are the anti-patterns.

Some of the well-known anti-patterns, for example, Blob, Lava Flow, Poltergeists, etc., are defined in the anti-pattern catalog [31], but there are many more. This thesis is concerned not only about these cataloged anti-patterns, but also the anti-patterns that are used instead of the design patterns in a badly designed software. These anti-patterns are termed as *missing* design patterns [32], as the existence of those anti-patterns express that their corresponding design patterns should have been used.

2.2.1 Some Cataloged Anti-patterns

Three categories of anti-patterns are well-known in the software industry –

- **Software Development Anti-patterns:** These anti-patterns describe situations encountered by the programmer when solving programming problems [30]. For example, The Blob, Poltergeists, Spaghetti Code, Lava Flow, Functional Decomposition, etc.

- **Software Architecture Anti-patterns:** These focus on common problems in system structure, their consequences and solutions [33]. For example, Autogenerated Stovepipe, Stovepipe Enterprise, Jumble, Cover Your Assets, Wolf Ticket, etc.

- **Software Project Management Anti-patterns:** These anti-patterns are concerned about the common problems and solutions in the software organization [34]. For example, Blowhard Jamboree, Analysis Paralysis, Death by Planning, Fear of Success, Corncob, etc.

As this thesis is concerned about the software development anti-patterns, some anti-patterns of these category are briefly described.

The Blob

When a single class contains large number of attributes and operations, it is called the Blob [30]. It is a procedural-style design even though implemented in Object Oriented Programming (OOP). It propagates all the major responsibilities to one object, while the other objects only store data or perform simple operations. It is also known as God Class. As these classes are large and complex, these are difficult to test and reuse. Also, these are unmanageable due to lack of understandability. And most importantly, Blob classes might be expensive to be loaded into the memory, as it uses excessive resources even for simple tasks.

Poltergeists

Poltergeists are the classes, that are stateless and has the only responsibility of invoking another class [30]. These are often created because the developer anticipated the need for a complex architecture. Poltergeists create unnecessary abstraction levels in the design. These cause redundant navigation paths making the software complex and relatively slow.

Spaghetti Code

Programs having a poor structure are called as Spaghetti Code [30]. It is named such because the control flow of the program is very much like a bowl of spaghetti, that is, twisted and tangled. These are caused by ignorance

and modification of the code frequently by different persons. The Spaghetti Codes become extremely unmanageable, and the OOP concepts are lost.

There are many more defined anti-patterns that can be found in the catalog [31]. These anti-patterns are needed to be avoided for achieving a good software design.

2.2.2 *missing* Design Patterns

Design patterns are originated as the solutions of common recurring design problems. If a software problem can be solved by one of the patterns, it is appropriate to be applied in the software design. However because of the lack of designers' knowledge the design pattern might not be applied in the design. In that case the software problem is solved by applying some other solution, that is not the optimal one (that is, design patterns). Since appropriate design patterns are not used, these alternative solution represents *missing* design patterns [32].

For example when a class needs to have only one single instance Singleton design pattern is required. If the Singleton pattern is not applied, an alternative solution is provided for tracking the single instance. This alternative solution is to create a conditional check whenever instantiating the class. This is the *missing* Singleton pattern or the anti-pattern of Singleton.

2.3 Characteristics of Design Patterns and Anti-patterns

For the detection of anti-patterns or the recommendation of suitable design patterns, it is required to develop a full understanding of the design patterns, anti-patterns and the software design. A system can be fully understood by analyzing its structural, behavioral and semantic characteristics [35, 2].

2.3.1 Structural Characteristics

The structural characteristic concentrates on the system classes and their relationships [2]. This characteristic represents the internal and external organization of the system components. It focuses on the static arrangement of these components.

The structure of a system can be perceived by analyzing its static architecture such as class diagram. The internal class representations and the defined relationship between the classes depict this characteristcal property. Hence a system structure is often identified from its class diagram.

2.3.2 Behavioral Characteristics

The behavioral characteristic focuses on system behaviors in various conditions (for example, class execution sequence in run-time) [2]. Unlike the structural characteristic it focuses on the dynamic activities of the system.

The behavior can be identified by analyzing the system in run-time. The dynamic interactions between the system classes can be analyzed to discover

its behavioral characteristic. Different design diagrams such as sequence, state and activity diagrams depicts the dynamic nature of a system. Hence these can be used to identify the system behavior.

2.3.3 Semantic Characteristics

The semantic characteristic tends to identify the logical relationships inside the system (for example, same types of classes in a system, classes that are always used together, etc.) [2]. Semantics basically relate the structural and behavioral aspects of the system, that is, the information of the static structure (class structure) with the dynamic behavior (class interactions).

Identification of semantic characteristic may need the incorporation of both static and dynamic properties. The knowledge gathered from structure and behavior are often combined for identifying this characteristic. Thus both the diagrams depicting static relationships (that is, class diagram) and dynamic interactions (sequence, state or activity diagrams) can be used to identify semantics.

2.4 Summary

A discussion of the software design patterns, anti-patterns and their characteristics is given in this chapter. Software design patterns are the proven best solutions to the design problems [18]. On the other hand anti-patterns are the bad solutions to those, and hence, termed as the *missing* design patterns. Both the anti- and design patterns can be completely represented using three characteristics; those are, the structure, behavior and semantic. These are

the important characteristics, which can be extremely helpful to recommend design pattern for a bad design. In the following chapter, a brief literature review of design pattern recommendation is presented.

Chapter 3

Literature Review of Design Pattern Recommendation

Design pattern recommendation researches intend to find a way of recommending the required software design patterns. It is an important field of research as selecting suitable design patterns manually is a hectic task. However it is comparatively a new research area unlike the other research trends in design pattern such as design pattern detection [21, 36, 37], design pattern instantiation [38, 23, 24, 39, 25, 40, 26, 41, 22], software refactoring [42, 43, 44], etc. This is because design pattern selection depends on human perspective, and is difficult to be perceived by an automated system. On the other hand anti-pattern detection is a well-established research trend for successfully identifying anti-patterns to check whether the software design is bad [2, 3, 6, 4, 5, 7, 45]. A relationship between these anti- and design patterns can be helpful for the design pattern recommendation, as existence of anti-patterns indicates the requirement of design patterns. However these

relationship establishment between the design pattern and anti-pattern is rare in the literature. The overview of the literature in this research is shown in Figure 3.1.

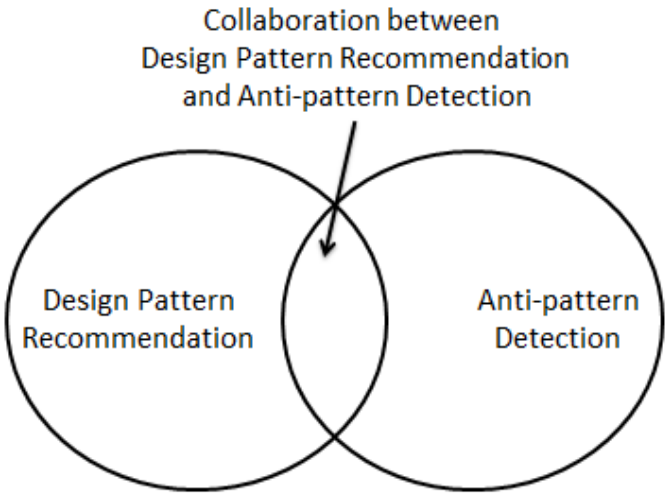


Figure 3.1: Overview of Design Pattern Recommendation Literature

In this figure three types of contributions are seen in the design pattern recommendation related research –

- **Design Pattern Recommendation:** Recommends suitable software design patterns for software.
- **Anti-pattern Detection:** Detects the existence of anti-patterns in software design.
- **Collaboration of Design Pattern Recommendation and Anti-pattern Detection:** Recommends design patterns using detected anti-patterns in software design.

In this chapter these literature contributions are discussed.

3.1 Design Pattern Recommendation

An optimized design solution can be achieved by the selection and application of suitable design patterns. Recommendation of software design patterns assists the selection of the appropriate design patterns for a software. In the literature investigations have been conducted for proposing design pattern recommendation approaches from different perspectives –

- **Text Based:** In this category of design pattern recommendation research, software problem scenario is used for matching with the design pattern intents. The matched patterns are recommended in [9, 11, 10].
- **Question-answer Based:** In this category questions are asked to the designers about their software. The answers are evaluated to suggest suitable patterns for the software [12, 13].
- **Case Based Reasoning (CBR):** In CBR category, a knowledge base or case library is maintained, storing the previous cases of design pattern usage. When a new case comes, it is matched with these previous ones, and the required pattern is identified and recommended [14, 15].

There are also a few other approaches of recommendation. For example - using some already used patterns in software for recommending new ones, using a ontology based pattern repository for retrieving required pattern list, etc. The contributions in these areas are also discussed in this section.

3.1.1 Text-based Search

Textual representations of software description such as usage scenario, user story, requirements, etc., are available. Using these texts to search the appropriate design patterns is termed as text-based search. In text-based search, pattern intents are matched with the problem scenarios for identifying the design patterns that relate mostly to the software [9, 11, 10]. This intent matching is based on set of important words [9], text classification [11] or query text search using Information Retrieval (IR) techniques [10]. However this approach is questionable for the correct recommendation of creational patterns, as the intents of these patterns possess similar texts. Moreover problem scenarios are ambiguous since those are written in human language. Those are usually not written from a designer's point of view. Hence it is impractical to identify possible design problems from problem scenarios.

In this section most of the text-based design pattern recommendation papers are described in chronological order.

Based on Set of Important Words

The concept of design pattern recommender system was first brought by Guéhéneuc et. al. [9]. It was the first paper to consider the difficulties in choosing suitable design pattern for software. It proposed a simple design pattern recommendation system for recommending from the 23 GoF design patterns [18].

First of all the textual descriptions of design patterns were analyzed manually. As a result a set of important words were extracted and connected to

those design patterns. In this way each of the design patterns were associated to a set of extracted words. These set of words and the design patterns are used to compose vectors. These are the relationship vectors that contains 1 in a cell if the important word is related to the design pattern, and 0 otherwise. After that those set of words are given to the user for selecting some of the words related to their project. Those subset of words are considered as user query. A vector of these query words was also created. Finally the design patterns were recommended by comparing and ranking those, using simple cosine distance between the vectors.

However this recommendation system does not use collaborative filtering and feedback from the users on the recommended design patterns. Hence its applicability and efficiency was questionable. Also users may not be expected to always choose the suitable important words related to the required design patterns of their software. It makes this recommendation system erroneous.

Information Retrieval (IR) based Text Search

Suresh et. at. presented an Information Retrieval (IR) based design pattern recommendation system [10]. IR is an well-known activity to retrieve relevant information from a collection.

The system worked in two steps. In the first step pattern related queries (for example, software usage scenario, problem description, etc.) were parsed to find the matching pattern intents. And the matched available intents were shown to the user. In the second step the one matching intent was to be chosen from these available ones. For this pattern intent matching questions were retrieved and user answers to those were recorded. The answers were

used to calculate a pattern weighted score, and the highest scored pattern was recommended.

However the design pattern query, that is, the problem scenario, cannot be expected to be matched with the pattern intent. Because, problem scenarios do not contain design related statements.

Text Classification Based Approach

Hasheminejad et. al. proposed a design pattern recommender using a text classification approach [11]. Text classification analyzes text documents, and assigns those to pre-defined categories. Thus, this research worked for assigning software descriptions to related design pattern categories, using the text classification.

The proposed technique worked in two steps - "Learning Design Patterns" and "Design Pattern Retrieval". In the first step classifiers were trained for each of the pattern classes, manually determined by experts. For this some pre-processing on the pattern texts were done such as stop-word removal, word stemming, term weighting and feature selection. Then a classifier was trained for each of the design pattern class using the training set corresponding to that class. In the next step the given problem description was taken as input. Preprocessing such as stemming and weighting were performed on it. Finally it was matched with the design pattern classes using cosine similarity. A threshold was applied on the gained similarity score for getting the appropriate patterns. These design patterns in the matched class were recommended.

Problem descriptions written in human language are naturally ambiguous. These should not be enough to identify the required design patterns. Thus the text-based approach may not be appropriate for the recommendation of suitable design patterns. Different approaches such as question-answer based, CBR, etc., may be used for contributing a better solution of this research problem.

3.1.2 Question-answer Based Approach

Interaction with designers is a way to understand the required design solution of software. This interaction helps to identify the design problem, and so the suitable design pattern. In question-answer based approach, designers are asked to answer some questions about the software and those answers lead to find the required patterns for that software [12, 13]. Here the mapping from question-answers to design patterns is set by formulating Goal-Question-Metric (GQM) model [12], or ontology-based techniques [13]. The problem is that the questions are often static or generic, and more related to design pattern features than software specific design problems. Also it is completely dependent on the responses and so the knowledge levels of the designers.

Goal-Question-Metric (GQM) Model

A question-based recommender named as Design Pattern Recommender (DPR) was proposed by Palma et. al. [12]. DPR is an expert system that uses Goal-Question-Metric (GQM) model to recommend patterns. GQM defines a measurement model in three levels –

- **Conceptual Level (Goal)** Defined goals such as existing design patterns
- **Operational Level (Question)** A set of questions, used to achieve a specific goal
- **Quantitative Level (Metric)** A set of metrics, associated to every question in order to answer it in a measurable way

This model was used for design pattern recommendation where the design patterns were specified as goals. The questions and metrics were defined to decide the required design pattern.

The approach was consisted of four steps. At first the circumstances in which specific design patterns were needed to be applied, were identified. Next the circumstances were refined by identifying sub-conditions of applying those design patterns. Then questions to be asked to the designers were identified. A Goal-Question-Metric (GQM) was formulated from those defined questions. As metric, a weighting scheme is used for each answers. The designers were asked the specified questions in the GQM. According to their answers, design patterns were ranked according to the weighting scheme score. This GQM was used to lead to the required patterns.

However the defined questions are generic, static and pre-defined. These are related to the design patterns, but not software specific design problems. Hence it would have been practical, if all the problems were directly mapped to a specific pattern which is not feasible.

Ontology Based Approach

Another question-based design pattern advisement approach was proposed by Pavlič et. al. [13]. It used an ontology-based design pattern repository for the recommendation. Ontology describes a subject domain using the notions of concepts, instances, attributes, relations and axioms [13]. It was used to represent design patterns and create a repository. Later design patterns were recommended using the proposed ontology and a question-based advisement approach.

The advisement procedure was performed in five phases. An ontology-based design pattern repository was created at the very first. Pattern containers were created next and question-answer pairs were connected to those. Users were asked about the pattern container questions, and the user's answers were used for getting a set of possible solutions as a subset of the containers. Then the most suitable pattern container was selected based on advice matrices and algorithms. After that the most suitable pattern of that selected pattern container was identified using groups of questions and answers for relevant patterns. Finally the selected design pattern was verified, and the related design patterns were identified for recommendation.

However similar to the previous question-based research [12], the questions are static and defined. These design pattern related questions are not suitable for finding possible software design problems and so, the required design pattern.

3.1.3 Case Based Reasoning (CBR)

Case Based Reasoning (CBR) is the process of solving new problems based on the solutions of similar past problems [46]. In CBR, recommendations are given according to the previous experiences of pattern usage stored in a knowledge base in the form of cases [14, 15]. The retrieval of cases from the knowledge base is performed either using user provided class diagrams [14] or using inputted and reformulated problem descriptions [15]. Matching cases to identify required patterns are not feasible, as the cases do not focus on the design problems a software might have.

CBR Using Class Diagrams

The first paper to automate the design pattern selection was proposed by Gomes et. al. even before the concept of design pattern recommendation system arrived [14]. A software Design Pattern Application (DPA) module was proposed here using CBR.

DPA was comprised of three phases – retrieval of applicable DPA cases, selection of most suitable DPA case and application of the selected DPA case. At first class diagrams were taken as input from the designers. Then the DPA case library was searched for DPA cases that match the problem. The matched problems were ranked, and the best matched DPA case were retrieved from the knowledge base. The DPA case was applied on the inputted diagram, and a new class diagram was returned. It was later stored in the case library.

However matching the class diagram of software cannot confirm its designing requirement for a design pattern. Many software class diagrams might have similar structures but not require the same pattern for good design.

CBR Using Problem Descriptions

Muangon et. al. used CBR and Formal Concept Analysis (FCA) to propose a design pattern searching system [15]. This research aimed to mitigate the problem of design pattern retrieval caused by difference between author keywords and user keywords. At first problem description was inputted by user. A similarity function was used to retrieve similar cases for suggesting solution patterns. If the user was not satisfied, FCA was used for reformulating problem description and retrieving related cases. First the structure of FCA enabled the discovery of related cases. These related cases were presented to the user for increasing the understandability of the problem. FCA techniques generate a more complete problem description. Then suggestions were made from the retrieved cases, matching the new description.

However CBR cases cannot express possible design problems. Matching two cases cannot ensure the suitability of previously used design patterns for the new case.

3.1.4 Other Approaches

A few researches were conducted for recommending patterns which do not fall in any of the mentioned categories. These papers are discussed in detail in this section.

Recommendation Based on Previous Selections

Navarro et al. proposed a different recommendation system for suggesting additional patterns to the designer while a collection of patterns are already selected [17]. The recommendation system was based on a heuristic function to obtain the utility or rating of all the design patterns.

First of all the applied design patterns in the initial solution were taken as input. An utility function is defined that calculates the utility for all the pattens, not included in the system. The function combines three types of information together as a weighted sum –

- previously used patterns in the solution
- relations between patterns decided by experts
- characteristical similarities between the types of patterns (defined by, structure, navigation, security, presentation, personalization and interaction)

Then a recommendation function is defined for ranking the design patterns by rating those. The rating was calculated based on three factors - number of occurrences with the initially selected design pattern set, types of relation with those and categorization of those. Finally a list of design patterns are returned as recommendation.

Now as this approach requires an initial pattern list, it cannot be used for new software. Also for a previously designed software, the designers need to know which patterns they have used in the design, which is not true always.

Recommendation Based on Ontology Formalization

Kampffmeyer et al. presented an ontology based formalization of the design patterns' intents [47]. This ontology focused on the problems rather than the solution structures. The main contribution of this paper is a Design Pattern Intent Ontology (DPIO), that is, an extensible knowledge base of design patterns.

Initially a formalisation of the intents of the GoF defined 23 patterns [18] were performed, according to the design problems. It was an ontology-based formalization, and so machine-readable and queryable. These formalizations are stored in a knowledge base named as Design Pattern Intent Ontology (DPIO). A "Design Pattern Wizard" was provided on top of DPIO that allowed design problems to be described visually. In this wizard a problem predicate and the concept constraints are taken as input. These are queried in the DPIO and a set of matching design patterns are suggested for the given design problems.

However the problem predicate and concept constraints required by this recommendation tool make its usage challenging. It is not always possible for a designer to decide these values. It requires expertise of the designers to be used effectively.

Hybrid Approach

A combination of some of the above mentioned design pattern recommendation approaches were performed by Sahly et. al. [16]. This paper incorpo-

rates the textual match, question-answer evolution and expert collaboration in one framework.

Recommendation of the patterns were performed by retrieving the matched patterns using four algorithms –

- **Query-Matching-Pattern (QMP):** This algorithm was used to parse and analyze the text of user queries (for example, problem scenario) to find matching between pattern intents and the given query. In the text analysis, stop word removal, word stemming, etc., were performed. Then Vector Space Model (VSM) [48] was used for representing pattern intents and user queries. These were represented as m-dimensional vector spaces. Similarity between the patterns and queries are determined by computing the cosine similarity between their vector representations.

- **Query-Similarity-Previous-Query(QSPQ):** This algorithm was developed to search similarity between the query and previous queries made by the user. CBR techniques were used here for the pattern selection. WordNet [49] was used for reformulation of the queries. These queries were matched using QMP like previous.

- **Question-Answer-Session (QAS):** In order to reduce the search space of design patterns this algorithm was used. Four categories of questions are asked to the designers – pattern-domain, pattern-category, pattern-intent and pattern-specific questions. The answers were used to calculate weights of the patterns.

- **Collaborative-Implicit-knowledge (CIK):** Consultancy with the experts were performed in this part. First, a description of problems

were taken as user requests. These are sent to the experts, and the experts suggest required design patterns for the problem.

As the problem descriptions do not contain design problems, the textual matching in QMP and QSPQ may not find appropriate design patterns. Hence matching with the design pattern intent is not always possible. The questions in QAS are pattern-related questions, making it impractical to consider the software specific design problems in appropriate design pattern selection. And as expert system in CIK needs manual intervention, it misses the notion of automation.

As described above, the existing approaches of design pattern recommendation use textual match with usage scenario, case match with knowledge base cases or ask design pattern related generic questions to designers. These approaches could not appropriately recommend design patterns, as design patterns are used for mitigating design problems, and these approaches do not focus on the system design problems.

3.2 Anti-pattern Detection

When the optimized solution (that is, appropriate design pattern) is not used to create a software design, anti-patterns arise. These anti-patterns represent bad software design that needs to be improved. Anti-pattern detection researches focus on finding these bad designs in software. It is a rich area of research [2, 3, 6, 4, 5, 7, 45]. Some of the important papers in this area are described in this section.

Fourati et al. proposed an anti-pattern detection approach in design level using UML diagrams [2]. The class and sequence diagrams were used here. This approach detected five well-known anti-patterns – Blob, Lava Flow, Functional Decomposition, Poltergeists and Swiss Army Knife.

First of all a set of most pertinent metrics such as Coupling Between Objects (CBO), Lack Of Cohesion in Methods (LOCM), Number Of Children (NOC) etc., are selected from [50, 51]. A set of specific metrics, useful for anti-patterns detection were then added to the metric list. Basically four categories of metrics were used –

- **Coupling:** It measures the interdependency between classes and objects. The metrics of this category are – Coupling Between Objects (CBO) and Response For Call (RFC).

- **Cohesion:** It measures the focus of class responsibilities. The metrics are – Lack Of Cohesion in Methods (LOCM), Tight Class Cohesion (TCC), Loose Class Cohesion (LCC) and Coh-Metrix.

- **Complexity:** It measures the simplicity and understandability of design. Weighted Methods per Class (WMC) is the metric of this category.

- **Inheritance:** The number of children and the tree of inheritance are measured here. The metrics of the measurement are – Depth of Inheritance of a class (DIT) and Number Of Children (NOC).

The anti-pattern detection was performed based on these metrics. This detection process has three steps – Structural, Behavioral and Semantic detec-

tions. In these levels the metric values were identified and the anti-patterns were concluded to be present for pre-defined threshold values. This prominent research assures that anti-pattern detection can be performed in the software design phase.

Another approach named as SVMDetect, for anti-pattern detection was based on Support Vector Machines (SVM) [3]. SVM is a machine learning approach that are used for classification, regression, etc. [52].

In SVMDetect, the detection task was accomplished by three steps - metric specification, SVM classifier training and detection of anti-pattern occurrences. For the metric specification, SVMDetect took the training dataset as input. Object-oriented metrics were calculated for using those as the attributes. Then the SVM classifier was trained using the dataset and the set of computed metrics. The objective of this training step is to divide the classes into two groups, Antipattern or Not-Antipattern. Then using the trained SVM classifier, the new occurrences of anti-patterns were detected.

The concept of anti-pattern training has made any defined or newly defined anti-pattern detection possible, breaking the boundary of only detection of some well-established anti-patterns (for example, Blob, Lava Flow, Poltergeists, etc.) [30].

3.3 Collaboration of Design Pattern Recommendation and Anti-pattern Detection

The conventional design pattern recommendation approaches in the literature do not consider software design problems, causing inaccurate recommendation results. On the other hand from the anti-pattern detection researches, it can be understood that anti-patterns can be detected in the software design phase successfully. Now if these anti-pattern detection approaches could be successfully incorporated in design pattern recommendation, the design problems of software could lead to the required patterns. Few researches have been conducted for recommending design patterns using anti-pattern detection.

Jebelean et. al. proposed an approach to identify *missing* Abstract Factory design pattern in object-oriented code [32]. Although the paper does not mention the detection of anti-patterns directly, it is the first paper to detect bad software designs appropriate for applying Abstract Factory.

The approach pointed the inputted code where the classes were instantiated in undesirable manner which could be improved by application of the Abstract Factory design pattern. First it identified the class/method combination of undesirable code structures (Abstract factory anti-pattern). Then these structure were detected in the inputted source codes. For this it computed the control paths of code using conditional statements (if-then-else, switch-case) and loops (for, while, do-while). These control paths were compared to the control paths of the undesired code structure. Matching between these structures conclude the detection of *missing* Abstract Factory.

In this paper, it was assumed that Abstract Factory families were instantiated in different control paths using conditional statements. However there can be more control paths generated by key listeners, button listeners, class wise instantiations, etc. This leads the process to fail to detect *missing* Abstract Factory in all cases.

An anti-pattern based design pattern recommender was proposed by Smith et. al. [8]. The tool recommended design patterns dynamically during the code development. This code-based tool was the first one to introduce the concept of pattern recommendation using anti-patten detection.

First of all it was observed how programmers try to solve a common problem in a way that could be improved using a design pattern. It is identified as the anti-pattern of that design pattern. Once the anti-patterns were defined, these were detected in the software code through structural and behavioral matching. The design patterns required to remove those detected anti-patterns were recommended. This paper has changed the concept of design pattern recommendation as it worked for improving the existing software design by identifying the design problems (anti-patterns residing in design). The matching processes of related anti-patterns of design patterns with inputted source code has given the approach more reliability as it worked on structured information rather than the unstructured or semi-structured descriptions like the textual information based approach. This work showed the recommendation of three design patterns - Singleton, Abstract Factory, and Command.

However, design pattern recommendation in the coding phase is too late as the software has already been designed. It is needed to be changed after

the recommendation. This increases the development time and cost, which is not acceptable.

An improvement of the paper of Smith et. al. [8] was proposed by Nadia et. al. [53]. It proposed a new behavioral matching approach for identifying Abstract Factory anti-pattern. And so, the accuracy of the recommendation of this pattern was also improved.

As already mentioned, Smith et. al. dynamically recommends design patterns in the software coding phase [8]. It detected anti-patterns in the source code by structural and behavioral matching, and suggested required design patterns to mitigate those. Although the approach was very promising as it worked to improve the existing software design, it was not good enough in case of Abstract Factory [54]. In the behavioral matching, it assumed code segments to have conditional if-then-else or switch-case statements for instantiating families in *missing* Abstract Factory, which is not true always. Hence for improving accuracy in Abstract Factory recommendation a refinement of the behavioral matching was proposed [53]. It considered the other cases of the existence of *missing* Abstract Factory like key listeners, button listeners, class wise instantiations etc.

Similar to [8], this paper also recommends design patterns in the coding phase, which is too late for re-designing the software.

The papers collaborating anti-pattern detection with design pattern recommendation provides the recommendations in the coding phase. Getting design pattern recommendation in this stage may not help the designers to improve the software. These recommendations are needed before the software is propagated to the development phase.

3.4 Summary

As presented in section 3.1, the existing approaches of design pattern recommendation in design phase use textual match with usage scenario, case match with knowledge base cases or ask design pattern related generic questions to the designers. These approaches may not be the proper ways to recommend design patterns. Because patterns are used for mitigating design problems, and these approaches do not focus on the software design problems. On the other hand the anti-pattern based approaches for recommendation lack robustness as the anti-pattern analysis have not been performed properly before the design pattern recommendation [8, 32]. Moreover these recommend design patterns in the coding phase, which is too late to re-design the whole software. In the following chapters these issues are addressed elaborately.

Chapter 4

Logically Deriving Creational Patterns' Anti-patterns

Design patterns are the proven best solutions of software design problems, and anti-patterns are the bad solutions to those [30]. Anti-patterns can be termed as *missing* design patterns as these are the alternative solutions to the best ones, that is, the design patterns. Thus a relationship should exist between those. This chapter aims to establish this relationship by deriving the corresponding anti-pattern of a design pattern. This relationship can lead to design pattern mapping as well as the recommendation of design patterns through the detection of the *missing* design patterns.

The relationship establishment is performed by finding the alternative solutions, that is, the anti-patterns, categorized by the design patterns. In this chapter these *missing* design patterns are derived based on the applicability of their corresponding design patterns. The identified *missing* patterns can be characterized using three properties, those are, the structural, behavioral

and semantic properties. However for utilizing the characterization, a formal representation is needed. So finally this chapter formally specifies these characteristics using Z notation [19].

As stated in the previous chapters, design patterns can be divided into three categories – creational, structural, and behavioral design patterns. Here only the *missing* creational patterns are identified and characterized as the scope of this thesis. There are five creational design patteras – Abstract Factory, Factory Method, Builder, Prototype and Singleton. These design patterns are analyzed individually to derive and characterize their anti-patterns in the following section.

4.1 Derivation and Characterization of Anti-patterns

This section is intended to provide the derivation and characterization of anti-patterns categorized by the creational design patterns. In the literature characterizations of design patterns can be found [35]. Similar to [35, 8], the characterization is performed for anti-patterns by determining the structural, behavioral and semantic properties –

- Structural property covers the static relationships between classes or components such as association, generalization, aggregation, etc.
- Behavioral property addresses the dynamic interaction between the classes or components, for example, run-time class instantiations, method invokations, etc.

- Semantic property works with the inner-meanings of the two properties, for example, identifying whether two classes are of same type.

These three attributes completely depicts the anti-pattern characteristics as discussed in Chapter 2.

4.1.1 Anti-pattern of Abstract Factory

Abstract Factory is a creational design pattern that is concerned with the instantiation of objects in groups or families [18]. In this part the anti-pattern of Abstract Factory is identified and the characteristics of this anti-pattern is described.

4.1.1.1 Derivation of Abstract Factory Anti-pattern

For deriving the anti-pattern of Abstract Factory, first the applicability of the Abstract Factory design pattern needs to be understood. A design can be said as *missing* Abstract Factory, if it demands the application of Abstract Factory but is designed differently [32].

Table 4.1 shows the applicability requirements of Abstract Factory. As GoF defines, Abstract Factory is applicable when the mentioned four requirements are present in a system [18].

For enforcing these requirements, in the design of Abstract Factory the client class does not instantiate the product objects directly. It uses factory classes to create and return the desired objects. Different factories exist for the creation of different families of products.

Table 4.1: Applicability Requirements of Abstract Factory Design Pattern

• the system is needed to be independent of the creation, composition, and representation of its products. The products are meant to be the target instantiation classes.
• the system is needed to be configured with one of the multiple families of products. Here families of products are the group of classes that are instantiated together.
• it needs to be enforced that a family of related product objects must be used together
• only the product interfaces are required to be revealed, not their implementations

The Abstract Factory structure defined by GoF is shown in Figure 4.1. In the figure there is a client class named as *Client*. There are two product types in the form of two abstract classes, *AbstractProductA* and *AbstractProductB*. There are two products of each product type, *ProductA*1 and *ProductA*2 of the *AbstractProductA*, and *ProductB*1 and *ProductB*2 of the *AbstractProductB*. An *AbstractFactory* class is defined here (Figure 4.1). The two concrete factories named as *ConcreteFactory*1 and *ConcreteFactory*2 instantiates the class families. *ConcreteFactory*1 instantiates *ProductA*1 and *ProductB*1, and *ConcreteFactory*2 instantiates *ProductA*2 and *ProductB*2. The *Client* class uses its required concrete factory to get the necessary class families, as the Abstract Factory design pattern defines.

Abstract Factory anti-pattern or *missing* Abstract Factory is originated, when a software requirements matched with these Abstract Factory applicability but is not applied. Basically the *missing* Abstract Factory is when groups of classes are required in different execution paths, but are not instantiated using factory classes [32, 8, 55].

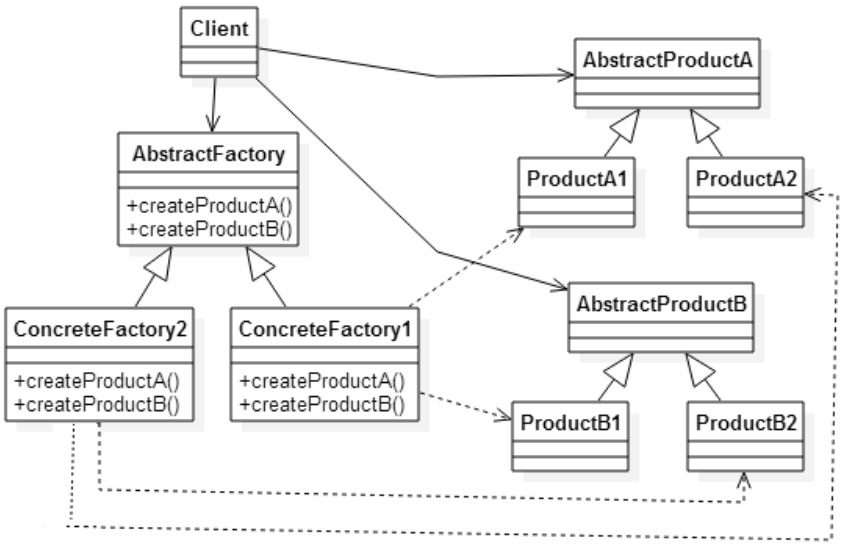


Figure 4.1: Abstract Factory Structure

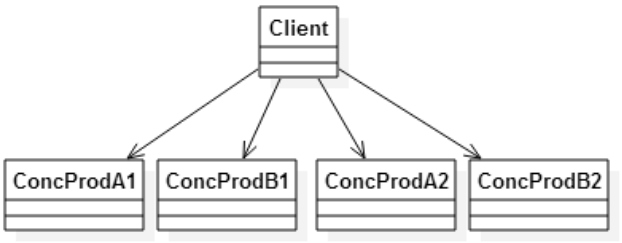
4.1.1.2 Characteristics of *missing* Abstract Factory

The characteristics of this anti-pattern (*missing* Abstract Factory) is analyzed here to identify its structural, behavioral, and semantic properties.

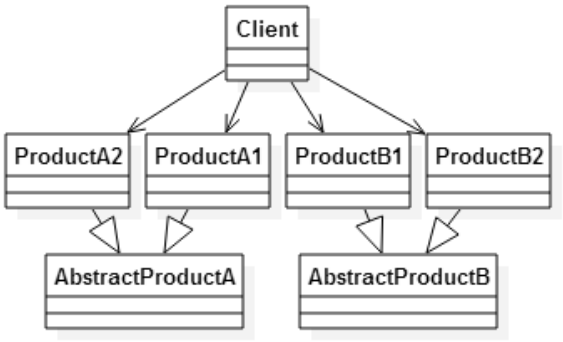
Structure of missing Abstract Factory

The structure of *missing* Abstract Factory can be derived by analyzing the structure of Abstract Factory design pattern. In the Abstract Factory structure, in spite of directly instantiating the product classes, the client class instantiates the factory classes. These factory classes create the product objects and return those to the client. In Abstract Factory anti-pattern, this structure is different; that is, the client class instantiates the product classes directly without using any factory class.

Direct client class instantiation can be done in several ways. Two of such structures are shown in Figure 4.2. In Figure 4.2(a), there are two



(a)



(b)

Figure 4.2: Structural Variants of Abstract Factory Anti-pattern [55, 32]

families of classes, *ConcreteProductA*1 (*ConcProdA*1), *ConcreteProductB*1 (*ConcProdB*1), and *ConcreteProductA*2 (*ConcProdA*2), *ConcreteProductB*2 (*ConcProdB*2). As by GoF, instead of being directly instantiated by *Client*, these families should have been instantiated using factories. Similarly in Figure 4.2(b), *ProductA*1, *ProductB*1, and *ProductA*2, *ProductB*2 are two families of classes, which should not be directly instantiated by the *Client*. Thus these two class designs represent the anti-pattern's structure of Abstract Factory [32, 55]. There are other variants of anti-pattern structure similar to these (for example, having interfaces instead of super-classes), which are uploaded on GitHub [56], and given in Appendix A.

Behavior of missing Abstract Factory

Although the design solution of the anti-pattern is different from the solution of design pattern, the behavior of the solution remains the same. Thus the behavioral feature of Abstract Factory anti-pattern is similar to Abstract Factory. The behavior is that, there are families of classes, and these families are always used together [18]. While, the classes of a family are instantiated in one execution path, classes of different families are always executed in different execution paths. An example of this behavior is the conditional instantiation of groups of classes. To be specific, there are at least two groups of classes, whose instantiations are mutually exclusive.

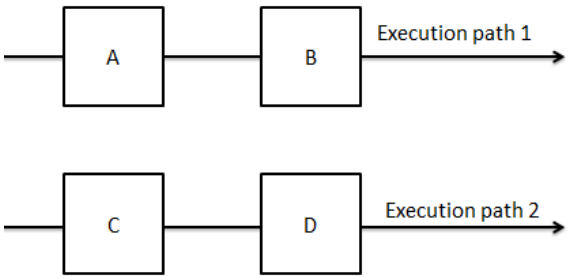


Figure 4.3: Behavior of *missing* Abstract Factory

For example, there are four classes – A, B, C, D as shown in Figure 4.3. Now, whenever class A is instantiated, class B is also instantiated in the same execution path (marked as Execution path 1). Similarly, class C and class D are always instantiated with each other in the same execution path (Execution path 2). Thus the set of classes, {A, B} can be said as a family of classes, and likewise, {C, D} is another family.

Semantics of missing Abstract Factory

Similar to the behavior, semantic of Abstract Factory and its anti-pattern are also the same; because, the semantic of the solution does not change in the *missing* Abstract Factory. In Abstract Factory, classes of similar types form different families [18], which is true for the *missing* pattern as well.

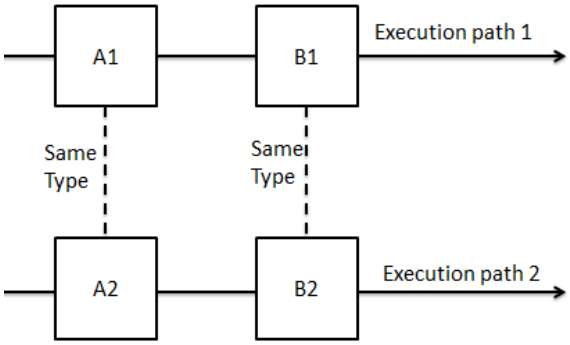


Figure 4.4: Semantic of *missing* Abstract Factory

The multiple families of the *missing* Abstract Factory are comprised with different classes of same types. Here, two classes are of same type means that, those classes possess same parent or functionality. For example, two classes extending the same class, or implementing the same interface, or providing semantically similar services are of same type. Thus the semantic property is that, if two mutually exclusive groups of classes (families) are found ({A1, B1} and {A2, B2} as shown in Figure 4.4), A1 and A2 will be of same type, and so as the B1 and B2.

4.1.2 Anti-pattern of Factory Method

The Factory Method design pattern is a creational design pattern that is concerned about the instantiation of classes of a product type. This research observed that, the anti-pattern of Factory Method is similar to the anti-pattern of Abstract Factory with some differences because of the absence of class families. The *missing* Factory Method is derived and there characteristics are analyzed here.

4.1.2.1 Derivation of Factory Method Anti-pattern

The applicability of the Factory Method design pattern can be used in inferring the anti-pattern of it. If a software requires the application of Factory Method pattern, but is designed differently, that design can be said as not applying Factory Method or 'missing' the Factory Method. Thus the design can be recognized to contain Factory Method anti-pattern or the *missing* Factory Method.

Factory Method's application is dependent on the three requirements as defined in Table 4.2 [18].

Table 4.2: Applicability Requirements of Factory Method Design Pattern

• a class cannot predict the class of objects to be created
• subclasses of the class will specify the objects to be created
• classes delegate responsibility to one of the several helper subclasses, and the knowledge of delegated helper subclass is needed to be localized

For enforcing these requirements, in the design of the Factory Method pattern, the creator class does not know the objects to be created and thus

does not instantiate the product objects directly. It declares a method that is defined by the subclasses. These creator's subclasses basically instantiate the desired objects and return those to the client. Different subclasses do the creation of different product objects.

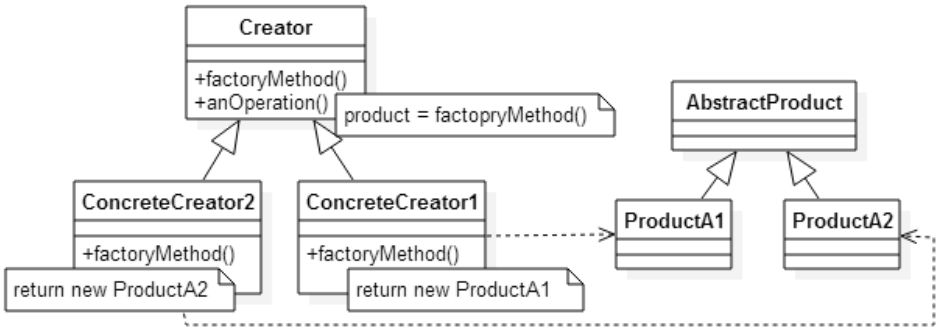


Figure 4.5: Factory Method Structure

The Factory Method design pattern's structure, defined by GoF, is shown in Figure 4.5. In this figure, there are two product classes (*ProductA*1 and *ProductA*2) of a product type (*AbstractProduct*). There is a product creator named as *Creator* that contains a factory method (*factoryMethod*()) for the instantiation of products. The two concrete creators named as *ConcreteCreator*1 and *ConcreteCreator*2 defines their own *factoryMethod*() to return the instantiations of *ProductA*1 and *ProductA*2 respectively.

Now, from the applicability of the Abstract Factory and Factory Method, it can be noticed that these two have similar applicability requirements except a single difference. In Abstract Factory, there is a concept of objects family, that distinguishes it from the Factory Method. Basically, Abstract Factory is used when there are multiple classes in a family (here, a family is a group of classes that are instantiated together), and different families are instantiated

in different scenarios. On the other hand, Factory Method is concerned about the instantiation of a single class (that is one of the multiple product objects), where different classes of same type are created in different scenarios.

Factory Method anti-pattern or *missing* Factory Method is originated when a software behavior matches with the Factory Method applicability requirements, but is not applied in the design. Here, the alternative design would be to instantiate the concrete products directly by the creator. The subclasses will not be used for specifying the required product objects.

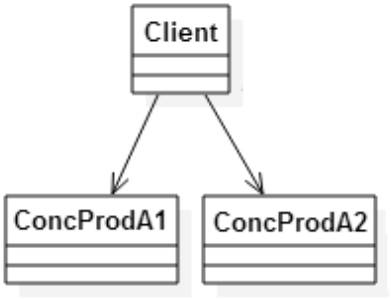
4.1.2.2 Characteristics of *missing* Factory Method

The characteristics of *missing* Factory Method is analyzed to identify its structural, behavioral, and semantic properties.

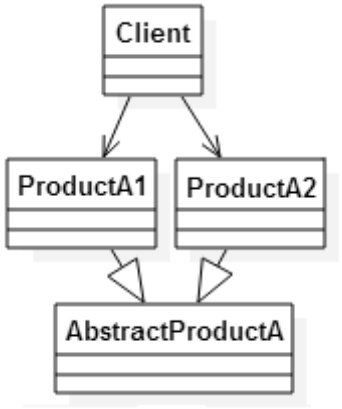
Structure of missing Factory Method

Two structural variations of the Factory Method anti-pattern are shown in Figure 4.6. There is a single type of classes instantiated directly by the *Client* (that is the Creator mentioned in Figure 4.5) in both of these variations – *ConcProdA*1 and *ConcProdA*2 in Figure 4.6(a), and *ProductA*1 and *ProductA*2 in Figure 4.6(b). However the *Client* should not have directly instantiated those, but instead, have some subclasses to take the decision of which class is to be instantiated. This is why these structures are of *missing* Factory Method. There are more variants of Factory Method anti-pattern structure, which are available in GitHub [56], and given in Appendix B.

The Abstract Factory's applicability is identified to be similar to Factory Method. A comparison between the structure of Abstract Factory and Fac-



(a)



(b)

Figure 4.6: Structural Variants of Factory Method Anti-pattern

tory Method anti-pattern is described. This comparison helps to understand why Factory Method anti-pattern has the shown structure. In Figure 4.6(a), a single type of classes are instantiated unlike Figure 4.2(a) that contains families of classes. These classes are instantiated directly by the *Client* without propagating the instantiation choice decision to any subclass. Similarly, in Figure 4.6(b), the same product type classes are instantiated singly unlike in family as shown in Figure 4.2(b). And also, the *Client* directly instantiates those classes. It does not have any helper subclass to take the instantiation decision from the choices between two product objects. Thus these two class designs represent the anti-pattern structures of Factory Method.

Behavior of missing Factory Method

As already mentioned, in Factory Method, different product objects are created by the creator class in different scenarios. As shown in Figure 4.5, the instantiation decisions are taken by the creator subclasses. Thus the identified behavior of the *missing* Factory Method is that, different classes are instantiated in different execution paths (that is, the conditional instantiation of classes). Unlike Abstract Factory, where families of classes are instantiated in different execution paths, in Factory Method only single classes are instantiated in those execution paths.

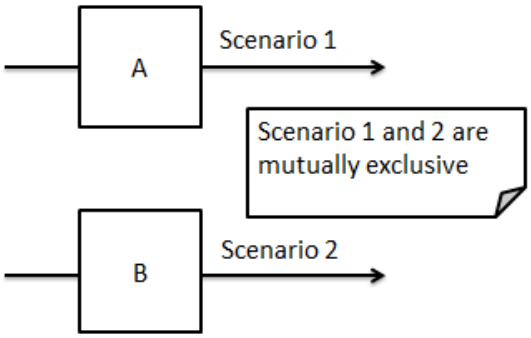


Figure 4.7: Behavior of *missing* Factory Method

For example, there are two product objects – A and B as shown in Figure 4.7. A is always used in one scenario, and B is used in another. Thus the classes A and B will always be instantiated in two different execution sequences upholding the behavior of Factory Method and so, the *missing* Factory Method as well.

Semantics of missing Factory Method

The semantics of *missing* Factory Method is that, the classes instantiated in different execution paths are of the same type. It is important to note that, when such single classes are found, Factory Method is applicable; but if there are multiple classes (that is, family of classes), Abstract Factory has the potential to be used.

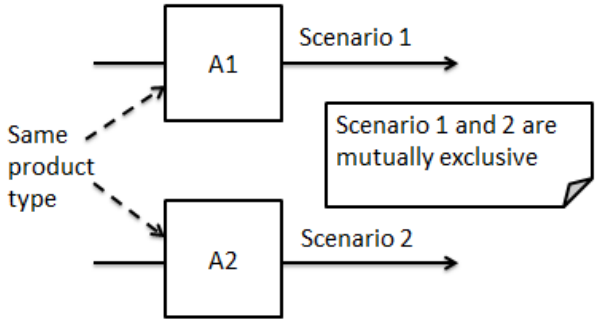


Figure 4.8: Semantic of *missing* Factory Method

Suppose, two classes A1 and A2 are instantiated in different execution paths (as in Figure 4.8). These two classes have the same parent, A. Thus A1 and A2 are the classes having the same product type. As the instantiations of these two classes are two product objects, the decision of which class needs to be instantiated is taken by the creator subclasses. These class type information is the semantic property of the *missing* Factory Method.

4.1.3 Anti-pattern of Builder

Builder design pattern deals with the construction of complex objects having different representations. The *missing* Builder is derived by analyzing the

Builder design pattern, and the characteristics of this *missing* design pattern are described.

4.1.3.1 Derivation of Builder Anti-pattern

Basically, the Builder pattern is created to find a solution to the problem of the Telescoping Constructor anti-pattern [57]. Thus the existence of this Telescoping Constructor anti-pattern represents the *missing* Builder pattern. Analysis of the Builder's applicability can justify the Telescoping Constructor's being *missing* Builder.

The two requirements, shown in Table 4.3, represent the applicability of Builder [18].

Table 4.3: Applicability Requirements of Builder Design Pattern

• creation of the complex object should be independent of the parts that build the object and the way those are assembled
• the object construction process must allow different representations of the object (for example – consider a text converter. It can have different representations like ASCII converter, Tex converter, etc.)

In Builder, for the creation of the complex object, the independent parts of the object is produced individually. All the parts are constructed by a director class one-by-one, and the final product is returned after the construction process is completed. The resulted product class is one of the different representations of the complex object.

The structure of the Builder design pattern is shown in Figure 4.9. In this figure, *Director* creates a complex object part-by-part using a method declared as *construct*(). This method calls another method named *buildPart*()

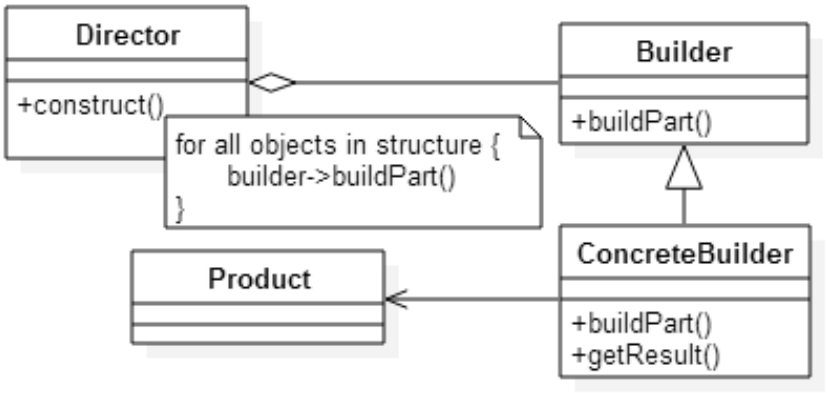


Figure 4.9: Structure of Builder Design Pattern

of the *Builder* class. A concrete builder (*ConcreteBuilder*) constructs a complex object representation named as *Product* and returns it to the *Director* when the *getResult*() method is called.

As mentioned already, Builder design pattern has basically originated for finding a solution for the anti-pattern named as the Telescoping Constructor [57]. Here, for creating the complex object with different representations, a numerous number of constructors are provided. These constructors take a different number of parameters (as different parts of the complex object) and delegate to a default constructor. An example of the Telescoping Constructor is shown in Figure 4.10. In this example, five constructors of a class is shown. Among these constructors, one is the main constructor (marked in the figure) that basically builds the object. The other constructors allow the flexibility of providing different parameters by considering the default values for the missing object parameters. Telescoping Constructor is a bad solution (that is, an anti-pattern), as this constructor parameter combination leads to an exponential list of constructors, which can be differently handled by Builder.

```java
public Constructor(int id, String name) {
    this(id, name, 0, 0, 0, "default description");
  }

  public Constructor(int id, String name, int age) {
    this(id, name, age, 0, 0, "default description");
  }

  public Constructor(int id, String name, int age, int height) {
    this(id, name, age, height, 0, "default description");
  }

  public Constructor(int id, String name, int age, int height, int weight) {
    this(id, name, age, height, weight, "default description");
  }

  /**
   * Main Constructor - actually builds the object
   */
  public Constructor(int id, String name, int age, int height, int weight,
      String description) {
    this.id = id;
    this.name = name;
    this.age = age;
    this.height = height;
    this.weight = weight;
    this.description = description;
  }
```

Figure 4.10: Telescoping Constructor Anti-pattern

When Telescoping Constructor is found in a software design, the Builder design pattern is supposed to be used. Thus the design can be said to have *missing* Builder.

4.1.3.2 Characteristics of *missing* Builder

The characteristics of *missing* Builder is analyzed here to identify its structural, behavioral, and semantic properties. This complex object construction and its representation can be fully described by the structure and behavior. Thus the semantic of *missing* Builder is not required.

Structure of *missing* Builder

Unlike Abstract Factory or Factory Method, the anti-pattern of Builder is concerned with a single class rather than multiple classes. Thus the class relations do not matter in the usage of this design pattern. And so, in spite of covering the whole class relationship structure, it requires to analyze the internal structure of single classes individually.

The structure of Telescoping Constructor anti-pattern or *missing* Builder is that, there are multiple constructors of a class with different parameters. Existence of these multiple different parameterized constructors assure that the single object has different representations revealing the potential of using Builder (referring to the Builder pattern applicability in Table 4.3).

```
public Constructor(int id) {}

public Constructor(int id, String name) {}

public Constructor(int id, String name, int type) {}

public Constructor(int id, String name, int type, int other1) {}

public Constructor(int id, String name, int type, int other1, int other2) {}
```

Figure 4.11: Telescoping Constructor Parameters

Behavior of *missing* Builder

The behavior of *missing* Builder does not include interactions with other classes, but is confined to the individual classes. The behavior lies in the parameter lists of the multi-constructor classes.

The parameter lists of Telescoping Constructor is similar to Figure 4.11, where the number of parameters is increasing with the constructors. With

inclusion of every constructor, a new parameter is introduced to that constructor. The previous constructors basically use a default value of that parameter, avoiding that parameter as input. These parameters inclusion pattern in Telescoping Constructor shows that those parameters can divide the object into individual parts as required by Builder's applicability.

Semantics of *missing* Builder

The structure and behavioral characteristics assure the complex object construction and representation, as required by the anti-pattern. The different representations of an object can be revealed by the multiple constructors' existence in the class. This is because, different constructors construct the objects differently, leading to dissimilar representations of the constructed objects. The complexity of the object can be understood by checking the parameter list of the constructors. Existence of numerous constructors with bunch of parameters indicate that the object is complicated in structure. Different parts of the object is represented by different parameters. Thus if the parameter list is increasing with the number of constructors, then the objects have that number of parts and so, complex representations.

This is why, structure and behavior of *missing* Builder completely reveal the requirements of this anti-pattern. So the semantic is not required, and not included here as *missing* Builder characteristic.

4.1.4 Anti-pattern of Prototype

Object cloning is a way to improve software performance, as creating new objects is time-consuming and costly with respect to the required resources [58]. Prototype design pattern is concerned with this object cloning concept. Here, the *missing* Prototype is identified along with its characteristics.

4.1.4.1 Derivation of Prototype Anti-pattern

Similar to the other creational patterns, the derivation of the Prototype anti-pattern is performed through analyzing the applicability of the Prototype. A design is said to contain *missing* Prototype, if it expresses the requirement of the Prototype application, but the pattern has not been applied.

The application of the Prototype design pattern is dependent on three requirements as shown in Table 4.4 [18].

Table 4.4: Applicability Requirements of Prototype Design Pattern

• the classes to be instantiated are specified at run-time such as by dynamic loading
• need to avoid building a class hierarchy of factories that parallels the class hierarchy of products
• when class objects can have one of only a few different combinations of state, instead of instantiating the class manually every time with the appropriate state, it is more convenient to install a corresponding number of prototypes and clone those

For providing a cloning mechanism, in the design of Prototype, the classes needing to be cloned contain a method that copies itself and returns the object of the same state. It declares a cloning method in an abstract class that is defined by the subclasses to be cloned.

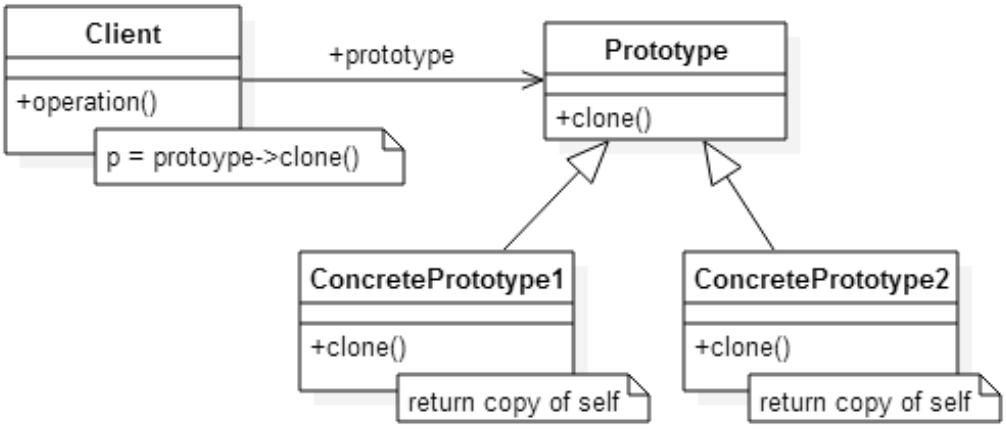


Figure 4.12: Prototype Structure

The structure of Prototype defined by GoF is shown in Figure 4.12. In this figure, *Client* class needs the cloned objects. *Prototype* class defines a cloning method (*clone*()) to clone the object by itself. There are two concrete subclasses of the class, *ConcretePrototype*1 and *ConcretePrototype*2 that implement the *clone*() method and return the cloned concrete objects.

Now, the anti-pattern of Prototype design pattern or the *missing* Prototype basically reveals the usage of multiple instances of the same class, which recommends the cloning of objects. The characteristics of *missing* Prototype is analyzed in the following section.

4.1.4.2 Characteristics of *missing* Prototype

The structural and behavioral properties of *missing* Prototype is analyzed here. Similar to Builder, semantic properties of *missing* Prototype is not required. Because, the structure and behavior is enough for identifying an object's multiple instantiation.

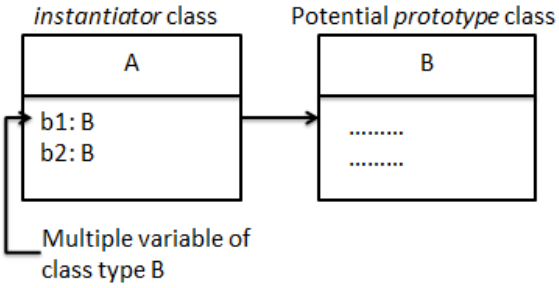


Figure 4.13: Structure of *missing* Prototype

Structure of *missing* Prototype

As stated earlier, the Prototype design pattern focuses on only individual classes (can be termed as *prototype* class). The *prototype* classes basically contain clone methods to copy instances of own-selves, that can be invoked by the other classes. A clone method is called by another class only when it requires multiple instances of the target class (that is, the *prototype* class).

The structure of *missing* Prototype focuses on these other classes, and finds whether multiple variables of the *prototype* class type are present there. To be specific, if a class contains multiple variables having the type of another class, it reveals that the first class (can be called as *instantiator* class) needs to instantiate the second class multiple times. The second class has the potential to be the *prototype* class, so in spite of being instantiated multiple times, it can be cloned. For example, suppose in a software design as shown in Figure 4.13, there is a class called A, and another named B. Class A has two variables b1 and b2. Both of these variables have the variable type of class B. This means that class A may contain multiple instantiations of class B and Thus object cloning can be useful here.

Behavior of *missing* Prototype

The behavior of *missing* Prototype is concerned with the instantiation of the *prototype* classes. In Prototype, as the structure defines, a class needs to have multiple variables having the type of *prototype* class to instantiate. Now the behavior defines that, truly the *prototype* classes are instantiated in those variables, not some other classes (in case of inheritance, sub-classes might get instantiated in a variable of super-class type).

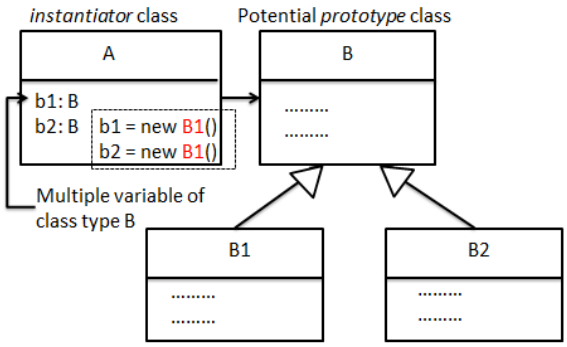


Figure 4.14: Behavior of *missing* Prototype

As mentioned in the previous example, class A has two variables, b1 and b2 of class B variable type. Now, it is not confirmed that b1 and b2 contain the instantiations of class B only. It may happen that class B has two subclasses B1 and B2 as shown in Figure 4.14. And, although variables b1 and b2 are of class B type, these contain the instantiations of class B1 and B2 respectively. In this case, object cloning cannot be done and the Prototype cannot be applied in the design. Otherwise, if both the variables contain the instantiations of B, Prototype design pattern is applicable agreeing to the behavior of *missing* Prototype.

Semantics of *missing* Prototype

The structure and behavior of *missing* Prototype entirely represents the characteristics of the Prototype anti-pattern. The semantic property is not required to identify whether a class is instantiated multiple times, and needs cloning. For revealing the multiple instance of a class, the structure detects multiple variables of that class type in the *instantiator* class. Then, the behavior confirms whether those variables truly contain the instantiation of that class. Hence the requirement of an object cloning is fully described by these two characteristics.

4.1.5 Anti-pattern of Singleton

Singleton is the last design pattern in the creational design pattern category. It deals with limiting the multiple instantiations of a class. In this section, the Singleton design pattern is analyzed to derive the *missing* Singleton, and the characteristics of the *missing* Singleton is discussed.

4.1.5.1 Derivation of Singleton Anti-pattern

Like the other creational patterns, the anti-pattern of Singleton design pattern can be derived by carefully examining the applicability of Singleton. A design contains *missing* Singleton, if it shows compliance with the Singleton's applicability requirements, but does not apply it.

The application of the Singleton design pattern is dependent on two requirements as shown in Table 4.5 [18].

Table 4.5: Applicability Requirements of Singleton Design Pattern

• exactly one instance of a class can exist in the system, and it must be accessible to clients from a well-known access point
• the instance should be extensible by using subclasses, and clients should be able to use an extended instance without modifying their code

The classes, to be forced to have only one instance, are termed as *singleton* classes. In the design of Singleton pattern, the *singleton* classes manage the existence of only one instance by itself. It provides a static method to return its instance. This method checks whether the class has already been instantiated. It contains a variable to store the instantiation, which is returned by the method once the object is stored in it. The constructor is not made public for stopping the other classes to directly instantiate it.

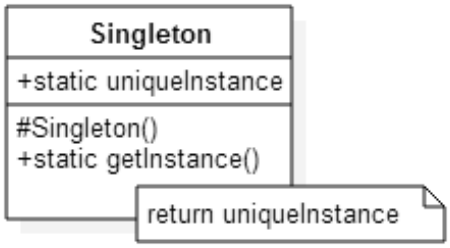


Figure 4.15: Singleton Structure

The structure of Singleton defined by GoF is shown in Figure 4.15. In this figure, there is a single class, *Singleton*, which is forced to have only one instance in the system. It contains a static variable named as *uniqueInstance* to store the single instance of the object itself. The constructor, *Singletone*(), has the protected visibility instead of public. This restricts the instantiation of the class by any outsider. It also contains a static method, *getInstance*()

that is responsible for instantiating the class and forcing the one single instantiation. The method checks whether the *uniqueInstance* variable already contains an instance first. If not, it instantiates the class and keeps it in the variable. Finally, it returns the *uniqueInstance* variable, that ensures the single instance of the class.

In the proper application of Singleton pattern, the *singleton* class itself has the responsibility of keeping one single instance. It ensures the central control of that class over the object creation. In the anti-pattern of Singleton, this single instance requirement is fulfilled in a different way. A conditional checking is performed before instantiating the *singleton* class every time by the classes that require the *singleton* object [8].

4.1.5.2 Characteristics of *missing* Singleton

The structural, behavioral and semantic characteristics of *missing* Singleton are analyzed here.

Structure of *missing* Singleton

There is no structural property of *missing* Singleton, because the class structure (individual or multi-class relation) does not affect on ensuring a single instantiation of a class. It can be seen in Figure 4.15 also, that there is a single class contributing in the design requirements of Singleton.

This is why, software design with any class structure is potential for applying Singleton based on the behavior and semantic characteristics.

Behavior of *missing* Singleton

The behavior of *missing* Singleton is to contain conditional checking before instantiating a module to verify whether it has already been instantiated or not. As mentioned in the derivation of *missing* Singleton, in the anti-pattern of Singleton the *singleton* class does not control the single instantiation by itself. The classes that use the *singleton* object need to have the validation of the instantiation.

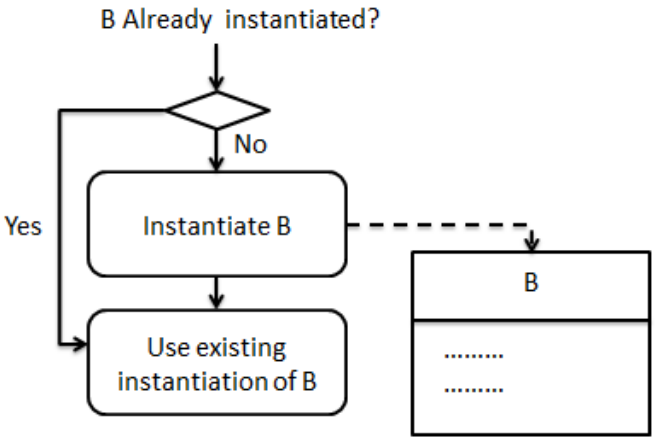


Figure 4.16: Behavior of *missing* Singleton

This is why, before instantiating the *singleton* module, the client classes need to check whether the module was instantiated before, as shown in Figure 4.16. Based on this checking, the module is instantiated or the existing instantiation is used. This conditional checking depicts the behavior of the *missing* Singleton.

Semantics of *missing* Singleton

The identified modules from the behavior receive confirmation by the semantic property of *missing* Singleton. This semantic property verifies whether the module is present in the list of classes.

The combination of the behavior and semantic expresses that, the client classes have conditional instantiation to limit a class to have one single instance. That class is identified as the *singleton* class.

4.2 Formal Specification of *missing* patterns

In this section, the characteristics of the *missing* design patterns are formalized using Z notation [19]. Formalization is done for providing a better representation of the anti-patterns than the textual description. As, Z provides a more concrete characteristics description, it is helpful for the anti-pattern detection by using these characteristics. First of all, some Z definitions are created to support the specification of the anti-patterns. Then Z schema is used for the formal specification of each of the *missing* pattern.

4.2.1 Some Definitions

Before getting into the Z formalization of the anti-patterns, some terminologies are needed for a better understanding. Thus the used Z notations in this thesis are provided in Table 4.6 along with their meaning.

Now, first of all, some basic types are declared below. Here,

- *Type* means the type of a variable (both primitive and class type),

Table 4.6: Used Z Notations

Notation	Meaning
$\mathbb{P}$	Power Set
$\Rightarrow$	Implication
$\mapsto$	Relation
$\in$	Belongs To
$\wedge$	Conjunction
$\forall$	For All
μ	Unique
$\#$	Total Count
$\bullet$	Predicate Separator
θ	Binding (Here, it is used as object of a class)
$[\ldots]$	Basic Type Declaration
$::=$	Free Type Definition

- *Value* is the variable value,
- *Constant* means unchanged value,
- and *Class* and *Interface* are declared in a generic type, *Classifier* [59].

$[Type, Value, Constant]$

$Classifier ::= Class \mid Interface$

Now, the *Variable*, *Operation*, *Class* and *Interface* are defined using Z schema. The *Variable* schema contains a string as *name*, *Type* as *type* and *Value* as *value*.

Variable

$name : string$

$type : Type$

$value : Value$

Operation schema contains a string *name*, a set of parameter *params* (*Variables*), a set of *Type* - return type *returnType*, and a boolean value *isAbstract* to check whether the operation is abstract or not.

Operation

$name : string$

$params : \mathbb{P}\ Variable$

$returnType : \mathbb{P}\ Type$

$isAbstract : boolean$

The *Class* schema has a class *name*, a set of *Variable* or attribute named as *attrs*, a set of *Operation* named as *opers*, a set of *Class* named as *children* that contains its children classes and a boolean *isAbstract* as a checking of abstract class.

Class

$name : string$

$attrs : \mathbb{P}\ Variable$

$opers : \mathbb{P}\ Operation$

$children : \mathbb{P}\ Class$

$isAbstract : boolean$

The *Interface* schema has the same properties as *Class* (except *isAbstract*), but with a condition that all its operations (*opers*) will be abstract ($\forall\ op : opes \bullet op.isAbstract()$).

Interface

$name : string$

$attrs : \mathbb{P}\ Variable$

$opers : \mathbb{P}\ Operation$

$\forall\ op : opes \bullet op.isAbstract()$

Finally, a class diagram, *ClassDiagram*, is defined using the definitions of the previously declared types. It contains a set of *Class* as *classes*, a set of *Interface* as *interfaces*, three types of relationships ($Classifier \leftrightarrow Classifier$) as *association*, *generalization*, and *aggregation*.

ClassDiagram

$classes : \mathbb{P}\, Class$

$interfaces : \mathbb{P}\, Interface$

$association : Classifier \mapsto Classifier$

$generalization : Classifier \mapsto Classifier$

$aggregation : Classifier \mapsto Classifier$

Now, *SequenceDiagram* schema is defined. It has a set of *Classifier* (both class and interface) as *lifelines* and a set of *Message*, *msgs*. The *Message* schema contains a message string *msg*, a *from Classfier*, and a *to Classfier*. The condition of *SequenceDiagram* is that, the *lifelines* are the *to* and *from* of each message m of the *msgs*.

SequenceDiagram

$lifelines : \mathbb{P}\, Classifier$

$msgs : \mathbb{P}\, Message$

$\forall\, m : msgs \bullet m.from \in lifelines \land m.to \in lifelines$

Message

$msg : string$

$from : Classifier$

$to : Classifier$

4.2.2 Formal Definition of *missing* patterns

Each *missing* patterns are represented using schemas, which will be used for anti-pattern detection. The characteristics of the anti-patterns are presented, with the support of the previously declared types.

4.2.2.1 Schema Definition of *missing* Abstract Factory

The next schema represents the characteristics of *missing* Abstract Factory. Here, *Client*, $A1$, $A2$, $B1$, $B2$, *AbsA* and *AbsB* are the *Classifier*s in the *missing* Abstract Factory structure. *Client*, $A1$, $A2$, $B1$ and $B2$ are the *Class*es, and *AbsA* and *AbsB* can be abstract classes or interfaces. These *AbsA* and *AbsB* can only exist in the system (the *ClassDiagram*) if these two are either abstract classes or interfaces (Condition 1, 2). In the system, some relationships must exist based on the structural property, which are also preserved. There are both relationships of association and generalization (Condition 3–10).

According to the behavioral aspects, some classes are bound to be in the same execution sequence. This is maintained by some conditions in a sequence of *Classifier* named as *exe* (Condition 11–14). Here, if class $A1$ is in an execution sequence, class $B1$ must be there and class $B2$ must not be in that sequence. Same goes for $A2$.

The semantic property is preserved by the last two conditions using a method to check type similarity of the intended classes. If abstract classes *AbsA* and *AbsB* are not in the system, the similarity is measured by the function, otherwise, the structural generalization is enough for the formalization.

missingAbstractFactory

$Client, A1, A2, B1, B2 : Class$

$AbsA, AbsB : Classifier$

$system : ClassDiagram$

$exe : \text{seq}\, Classifier$

$AbsA \in system \Rightarrow AbsA \in \mathbb{P}\, Interface \lor AbsA.isAbstract$

$AbsB \in system \Rightarrow AbsB \in \mathbb{P}\, Interface \lor AbsB.isAbstract$

$Client \mapsto A1 \in system.association$

$Client \mapsto A2 \in system.association$

$Client \mapsto B1 \in system.association$

$Client \mapsto B2 \in system.association$

$A1 \mapsto AbsA \in system.generalization$

$A2 \mapsto AbsA \in system.generalization$

$B1 \mapsto AbsB \in system.generalization$

$B2 \mapsto AbsB \in system.generalization$

$\theta A1 \in exe \Rightarrow \theta B1 \in exe$

$\theta A2 \in exe \Rightarrow \theta B2 \in exe$

$\theta A1 \in exe \Rightarrow \theta B2 \notin exe$

$\theta A2 \in exe \Rightarrow \theta B1 \notin exe$

$AbsA \notin system \Rightarrow ofSimilarType(A1, A2)$

$AbsB \notin system \Rightarrow ofSimilarType(B1, B2)$

4.2.2.2 Schema Definition of *missing* Factory Method

The schema representation of *missing* Factory Method is similar to the schema representation of *missing* Abstract Factory. This is the expected behavior based on the characterization of the two anti-patterns. In the schema, *Client*, $A1$, $A2$ and *AbsA* are the *Classifier*s in the *missing* Factory Method structure. *Client*, $A1$ and $A2$ are the *Class*es, and *AbsA* can be an abstract class or interface. *AbsA* only exists in the system if it is either an abstract class or interface (Condition 1). The structural property relationships of the anti-pattern are also preserved in the schema. There are both relationships of association and generalization (Condition 2–5).

missingFactoryMethod

$$Client, A1, A2 : Class$$

$$AbsA : Classifier$$

$$system : \mathbb{P}\, Classifier$$

$$exe : executionpath, \text{seq}\, Classifier$$

$$AbsA \in system \Rightarrow AbsA \in \mathbb{P}\, Interface \lor AbsA.isAbstract$$

$$Client \mapsto A1 \in system.association$$

$$Client \mapsto A2 \in system.association$$

$$A1 \mapsto AbsA \in system.generalization$$

$$A2 \mapsto AbsA \in system.generalization$$

$$A1 \in exe \Leftrightarrow A2 \notin exe$$

$$AbsA \notin system \Rightarrow ofSimilarType(A1, A2)$$

The behavioral aspect is maintained by a conditions in the sequence of *Classifier* and *exe* (Condition 6). Here, if class $A1$ is in an execution sequence, class $A2$ must not be in that sequence.

Finally, the semantic property is preserved by the last condition using a method to check the type similarity of the intended classes ($A1$ and $A2$) whether *AbsA* is not present in the system.

4.2.2.3 Schema Definition of *missing* Builder

The schema representation of *missing* Builder preserves the structural and behavioral characteristics of the anti-pattern. The *missing* Builder works on individual classes, so there is a class A and some *constructors* as a set of operations. The structural property says that there will be at least two constructors (Condition 1).

missingBuilder

$$A : Class$$

$$constructors : \mathbb{P}\, A.opers$$

$$count(constructors) \geq 2$$

$$\forall\, i : 1 \ldots \#constructors - 1 \bullet$$

$$count((constructor\ i).params) - 1 = count((constructor\ i + 1).params)$$

$$\land\ (constructor\ i).params) \subset (constructor\ i + 1).params$$

The behavioral property analyzes the parameter patterns in the multiple constructors (Condition 2). There are two predicates in the condition that a

new parameter should be introduced in the next constructor, and the previous parameters should stay intact (subset).

4.2.2.4 Schema Definition of *missing* Prototype

The following schema represents the *missing* Prototype and preserves its structural and behavioral characteristics. Here, *Instantiator* is the *instantiator* class and *Prototype* is the *prototype* class. *a* is a set of *Variable*s. This set of *Variable*s is the attributes of the *instantiator* class (Condition 1). Also, there is more than one *Variable* in the set (Condition 2). All the *Variable*s of this set are of type *Prototype* (Condition 3). This expresses the structural property of *missing* Prototype that, the *instantiator* class will contain more than one variable having type of the *prototype* class.

The remaining condition establishes the behavioral property. If the *prototype* class do not have any children ($\#Prototype.children = 0$), the value of the variables are *prototype* objects (that is, the instantiations of the *prototype* class). Now, if it has one child ($\#Prototype.children = 1$), the variable values are the objects of that child. On the other hand, if it has more than one child ($\#Prototype.children > 1$), all the variable values are the objects of any one of the children ($Prototype.children[Constant]$).

missingPrototype

$Instantiator, Prototype : Class$

$a : \mathbb{P}\ Variable$

$a \subseteq Instantiator.attrs$

$\exists\, a1 \in a, a2 \in a \bullet a1 \neq a2$

$\forall\, i : 1 \ldots \#a \bullet (a\ i).type = Prototype$

$\forall\, i : 1 \ldots \#a \bullet$

$\#Prototype.children = 0 \Rightarrow (a\ i).value = \theta Prototype \lor$

$\#Prototype.children = 1 \Rightarrow (a\ i).value = \theta Prototype \lor$

$(a\ i).value = \theta Prototype.children[0]$

$\#Prototype.children > 1 \Rightarrow (a\ i).value = \theta Prototype \lor$

$(a\ i).value = \theta Prototype.children[Constant]$

4.2.2.5 Schema Definition of *missing* Singleton

This section presents the schema of the *missing* Singleton design pattern, named as *missingSingleton*. Here, *Singleton* is a *Class* representing the *singleton* class. As, no structural attribute exists for the *missing* pattern, this schema presents the behavioral and semantic attributes by proving the uniqueness of the *singleton* class.

First of all, it establishes that there is an unique *Singleton* object in all of the classes that have an *association* relationship with the *Singleton* class (Condition 1). The *Singleton* objects inside different classes are also unique (Condition 2).

The semantic aspect is already presented by assuming *Singleton* as a *Class* type.

missingSingleton

$$Singleton : Class$$

$$\forall\, c : Class \bullet c \leftrightarrow Singleton \in R_{association} \Rightarrow$$
$$\mu\, obj : \theta Singleton \bullet obj \in c$$
$$\forall\, c1, c2 : Class \bullet c1 \leftrightarrow Singleton \in R_{association} \wedge c2 \Rightarrow$$
$$\leftrightarrow Singleton \in R_{association} \Rightarrow \mu\, obj : \theta Singleton$$
$$\bullet\ obj \in c1 \wedge obj \in c2$$

4.3 Summary

The anti-patterns of the five creational design patterns are derived from the corresponding design patterns, the characteristics of those anti-patterns are described, and formally specified in this chapter. These anti-patterns are represented using the structural, behavioral, and semantic properties of those. The next chapter will use these anti-pattern characteristics in the recommendation of the corresponding design patterns.

Chapter 5

Recommendation of the Creational Design Patterns

Existence of an anti-pattern in a software design discloses that the design is not appropriate; it can be improved by applying a suitable design pattern. If a relationship can be established between these anti- and design patterns, the detection of anti-patterns can lead to the design patterns' recommendation. In Chapter 4, anti-patterns relating to the specific design patterns are logically derived. Also, the characteristics of these anti-patterns are identified. In this chapter, these characteristics are matched with the software characteristics in a faulty software design for anti-pattern detection. According to the characteristic matchings, for recommendation, each of the pattern gets a matching score. Based on the score value, patterns are recommended or suggested. To limit the scope, in this thesis, only the creational design patterns are considered. The recommendation approach is described in the following sections.

5.1 Overview of the Recommendation Approach

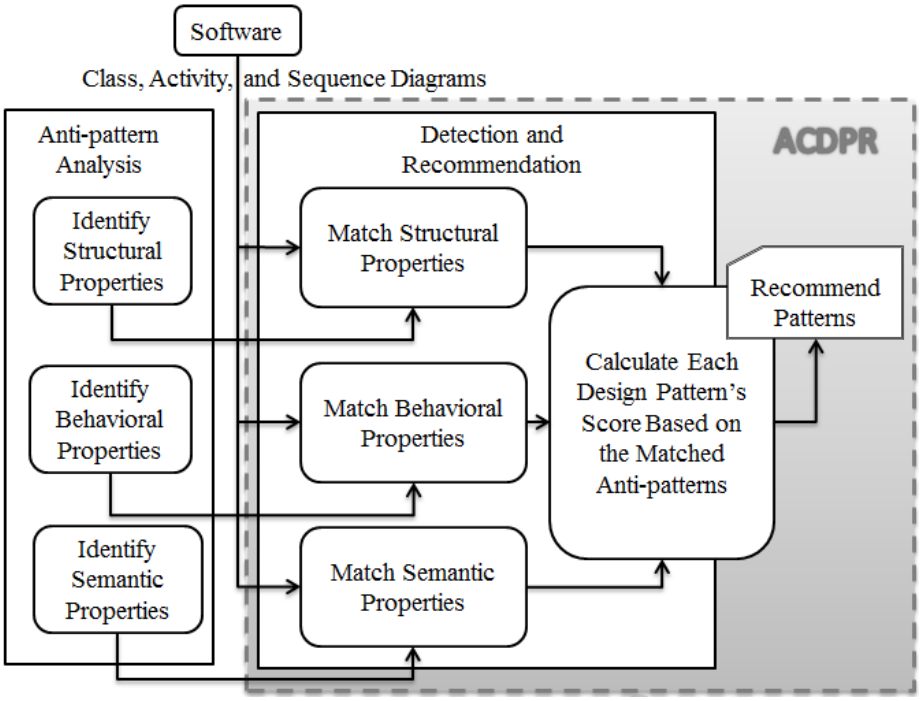


Figure 5.1: Overview of the Recommendation Approach

The overall design pattern recommendation process is shown in Figure 5.1. The process is divided into two parts. In the first part, the anti-patterns to be detected in the software are analyzed (Chapter 4). Based on this, a framework called Anti-pattern based Creational Design Pattern Recommender (ACDPR) is devised. ACDPR detects the anti-patterns in the software design, and recommends suitable design patterns. The steps of ACDPR are described below –

- **Anti-pattern Detection:** The analyzed anti-patterns' structure, behavior, and semantic are matched with software using design diagrams such as class diagram, sequence diagram and activity diagram.

- **Score Assignment:** Matching levels in the anti-pattern detection return values 0 for a mismatch and 1 for a match. A weighting factor is assigned to these levels according to their importance in the *missing* patterns' detection. Based on the matching level outcomes and their weighting factors, a score is calculated for each of the patterns.

- **Design Pattern Recommendation:** Finally, design patterns are recommended based on the obtained scores. The range of the score value is 0 to 1. For a complete match, that is a score of 1, patterns are recommended. For the partial matchings, a threshold value is introduced for providing suggestions.

It is noticeable that, predefined anti-patterns are used in ACDPR for creational pattern recommendation. However, new anti-patterns can also be incorporated in the framework, whenever available. Before including the anti-patterns, their characteristic formalizations are to be done.

In the following sections, the steps of recommendation, using ACDPR, are described in details.

5.2 Anti-pattern Detection

In Chapter 4, the anti-patterns are categorized by corresponding design patterns. Now, their detection in a faulty system design can conclude to the recommendation of those design patterns. Similar to the analysis, detection of anti-patterns needs three levels of matching - structural, behavioral and semantic matchings (as shown in Figure 5.1 "Detection and Recommendation"

phase). In this section, the detection methodology of the *missing* creational design patterns are described using the identified characteristics.

5.2.1 *missing* Abstract Factory (mAF)

In Chapter 4, the structural, behavioral and semantic characteristics of the *missing* Abstract Factory (mAF) have been identified. Here, these are matched with the software design for mAF detection.

5.2.1.1 mAF Structural Matching

For the structural matching of mAF, the Abstract Factory anti-patterns are needed to be defined into the tool first. These anti-pattern structures are matched with the software structure. Thus, mAF structural matching is done using the following steps –

- Store the anti-pattern structures
- Match the stored structures with the software structure

Figure 5.2 shows the overall procedure of mAF structural matching. In the first step, the structures of the anti-pattern (as shown in Figure 4.2) are stored. Presence of these structures represent that the Abstract Factory application is missing in the design. Here, client classes directly instantiate the product classes without using factories. These anti-pattern structures are defined by the relationships among the classes (for example, aggregation, generalization, association, etc). As class diagrams contain the different class-to-class relationships, those are used in this level [59].

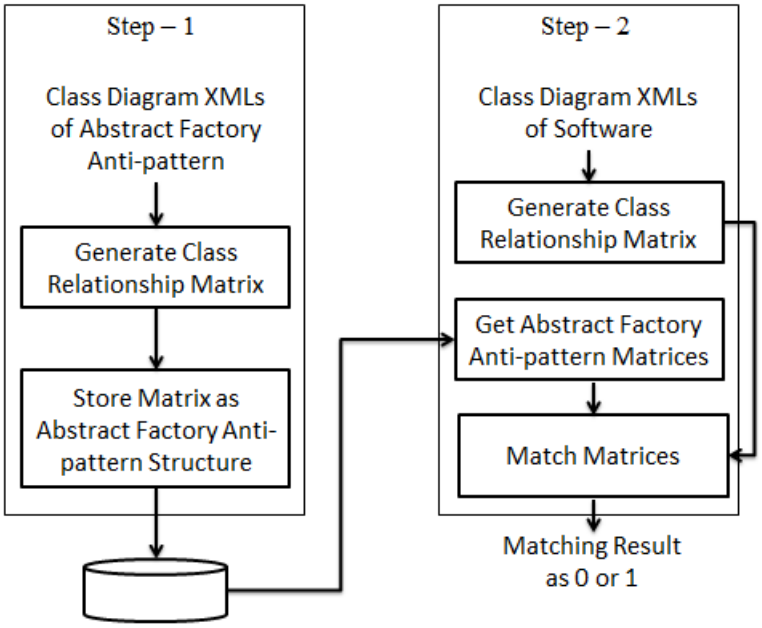
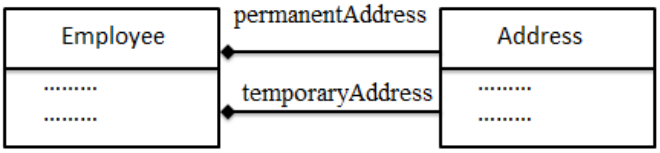


Figure 5.2: Structural Matching of the *missing* Abstract Factory

Relationship Representation

First, the class diagrams are converted to program readable format, that is XML, and inputted to the tool as shown in Figure 5.2. For keeping these relationship information (both the types of the relationships and their cardinality), a two dimensional matrix is used. This is a $n \times n$ prime numbered matrix as noted by Dong et al. [35]. Here, the usage of prime number is for tracking cardinality of the relationships when multiple relationships between two classes are present. As product of prime numbers are unique, it is possible to identify the types of the multiple relationships between classes.

For example, Figure 5.3 shows multiple relationships between *Employee* and *Address* [60]. The *Employee* class have two aggregation relationships with *Address*. It can have *Address* as permanent address (*permanentAddress*) or temporary address (*temporaryAddress*). For multiple relationships, the



(a) Classes Having Multiple Relationship

	Employee	Address
Employee	0	25
Address	0	0

(b) Matrix Representation

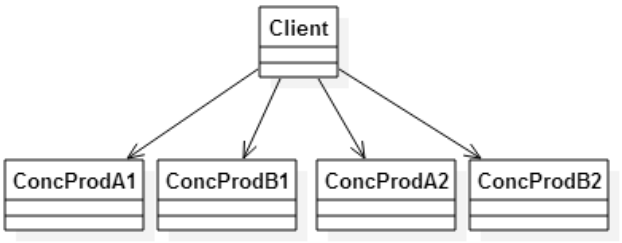
Figure 5.3: Preserving Multiple Relationships Using Prime Number

cardinality values are either added or multiplied. These operations on the simple integers cannot preserve the initial values, so fail to identify individual relationships. For example, simple integers such as 1 for *association*, 2 for *generalization* and 3 for *aggregation* were used. For *permanentAddress* and *temporaryAddress*, the addition of the relationships (that is, $(3+3) = 6$, for *aggregation* in this case) was stored. The number 6 count does not represent the type of relationship between the classes. It might represent six *association* relationships, or three *generalization* relations, or two *aggregation*s. It would have made the matrix ambiguous and failed to track the types of relationships. On the other hand, as products of prime numbers are unique, it can express the relationship types in existence of multiple relationships as well. In Figure 5.3(b), *25* is a unique number that can only be generated if two *aggregation* relationships (represented using prime number *5* according to Table 5.1) are there.

Table 5.1 shows the determined prime numbers for the specific class relations as shown in [8]. These prime values are used in representing the class-to-class relationships in class matrix.

Table 5.1: Representative Prime Numbers of the Class Relationships [8]

Class Relationship	Prime Number Representative
Association	2
Generalization	3
Aggregation	5

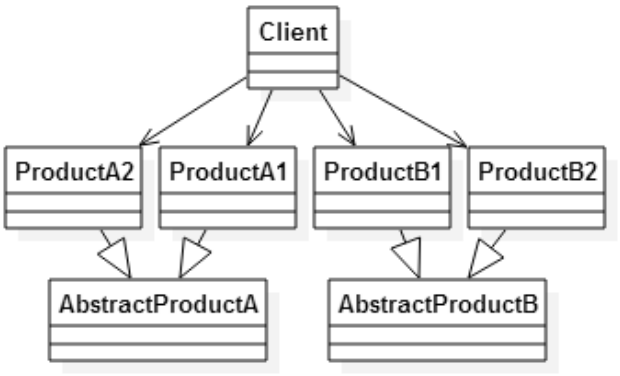


(a) The Anti-pattern Structure

	C	A1	B1	A2	B2
C	0	2	2	2	2
A1	0	0	0	0	0
B1	0	0	0	0	0
A2	0	0	0	0	0
B2	0	0	0	0	0

(b) Matrix Representation

Figure 5.4: Generated Matrix of Figure 4.2(a)



(a) The Anti-pattern Structure

	AbsA	A1	A2	AbsB	B1	B2	C
AbsA	0	0	0	0	0	0	0
A1	3	0	0	0	0	0	0
A2	3	0	0	0	0	0	0
AbsB	0	0	0	0	0	0	0
B1	0	0	0	3	0	0	0
B2	0	0	0	3	0	0	0
C	0	2	2	0	2	2	0

(b) Matrix Representation

Figure 5.5: Generated Matrix of Figure 4.2(b)

Storing mAF Structure

Figure 5.4 and Figure 5.5 show the matrices generated for the structures of Figure 4.2, these are stored as anti-pattern structures of Abstract Factory. The anti-pattern structures are repeated in Figure 5.4(a) and Figure 5.5(a) as a reminder of the structures. The matrix of Figure 5.4(b) is generated from Figure 5.4(a). Here,

- C, $A1$, $B1$, $A2$ and $B2$ represent *Client*, *ConcProdA*1, *ConcProdB*1, *ConcProdA*2 and *ConcProdB*2 respectively.

- The Association ($\xrightarrow{A}$) relations between *Client* $\xrightarrow{A}$ *ConcProdA*1, *Client* $\xrightarrow{A}$ *ConcProdB*1, *Client* $\xrightarrow{A}$ *ConcProdA*2 and *Client* $\xrightarrow{A}$ *ConcProdB*2 in Figure 5.4(a) are contained in the matrix using the prime number *2* (Table 5.1).

Similarly, the matrix of Figure 5.5(b) is generated from Figure 5.5(a), where,

- *AbsA*, *A*1, *A*2, *AbsB*, *B*1, *B*2 and *C* represent *AbstractProductA*, *ProductA*1, *ProductA*2, *AbstractProductB*, *ProductB*1, *ProductB*2 and *Client* respectively.

- Generalized ($\xrightarrow{G}$) relations (*ProductA*1 $\xrightarrow{G}$ *AbstractProductA*, *ProductA*2 $\xrightarrow{G}$ *AbstractProductA*, *ProductB*1 $\xrightarrow{G}$ *AbstractProductB* and *ProductB*2 $\xrightarrow{G}$ *AbstractProductB*) and Association relations (*Client* $\xrightarrow{A}$ *ProductA*1, *Client* $\xrightarrow{A}$ *ProductA*2, *Client* $\xrightarrow{A}$ *ProductB*1 and *Client* $\xrightarrow{A}$ *ProductB*2) are stored in the matrix using prime numbers *3* and *2* respectively (Table 5.1).

Matching mAF Structure

In the second step of structural matching, existence of the stored matrices are detected in the software design. As shown in Figure 5.2, first the XML of the software class diagram is taken as input. It is represented similarly as the matrix of Abstract Factory anti-patterns.

Finally, the stored anti-pattern structures are matched to the system's structure for finding whether any of those anti-patterns is present in the system. For this, the system matrix is matched with the Abstract Factory anti-patterns' matrices. In this matrix matching, it is identified that whether

any of the anti-pattern matrices is present in the system matrix. As the focus is on the accuracy rather the computational complexity or time, in the implementation (Chapter 7), the matrices are matched using a brute force method where every permutation of the anti-pattern matrices are taken.

5.2.1.2 mAF Behavioral Matching

The mAF behavioral matching requires to analyze the dynamic interactions between classes as the mAF behavior is defined in Chapter 4 Section 4.1.1.2. Usually, sequence diagrams represent the dynamic interactions of classes in execution [59]. So, these are used in this level. The *lifelines* of a sequence diagram are the roles or object instances, and represent the classes in the same execution sequence [61]. According to the mAF applicability, classes of same families are supposed to be in the same execution sequence, and so in the same sequence diagram *lifelines*. This is why, families of classes in Abstract Factory are identified from these *lifelines*,

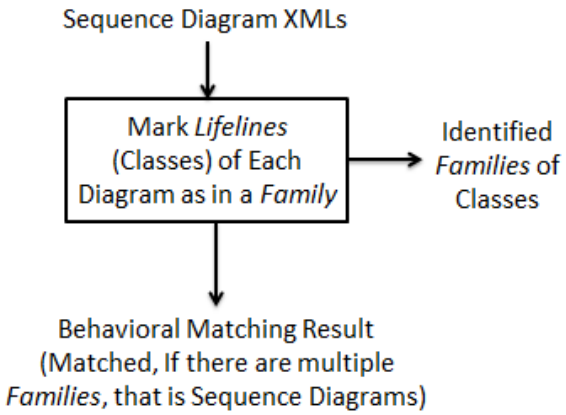


Figure 5.6: Behavioral Matching of the *missing* Abstract Factory

Figure 5.6 shows the behavioral matching process of *missing* Abstract Factory. First, the UML sequence diagrams of the system are converted to XMLs, and inputted to the tool. The XMLs are parsed to identify the *lifelines* and the corresponding classes of those. The identified classes of each sequence diagram are marked to be in the same family. These family information is maintained by preserving those as lists of classes. It is noticeable that, the existence of multiple sequence diagrams assure that there are multiple execution paths, and the classes of each execution path are considered to be in one family. So, Abstract Factory behavior is matched simply if multiple sequence diagrams are there.

5.2.1.3 mAF Semantic Matching

The semantic matching step is concerned about verifying the semantic relation of classes obtained from the behavioral matching. In mAF semantic matching, types of the classes are analyzed to validate the family information acquired from the mAF behavioral matching. The matching is executed as per the findings of semantic analysis, that is, different classes of similar types form different families.

Figure 5.7 summarizes the overall semantic matching of Abstract Factory. First, a matrix containing the similar types of class information is generated using the super-class relations. That matrix is used to analyze whether the classes in multiple families are aligned to the requirements of Abstract Factory. The family information is acquired from the behavioral step, and a family matrix is generated as shown in Figure 5.7. In Abstract Factory, there are at least two types of classes in the families [18]. If multiple same

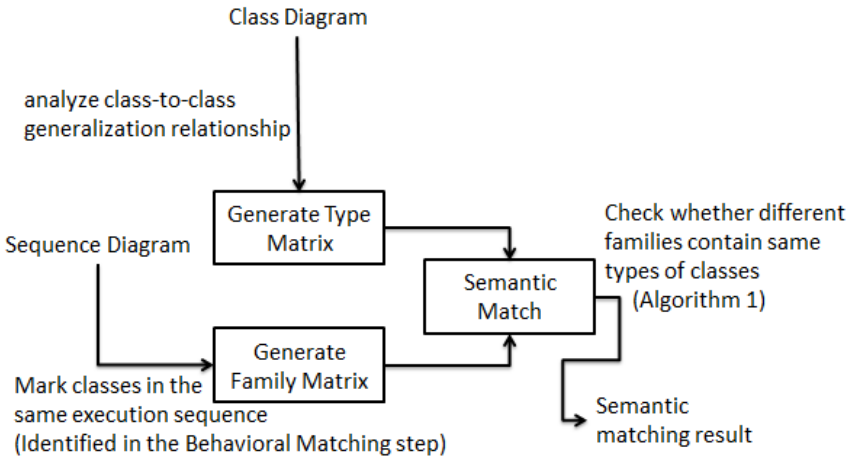


Figure 5.7: Semantic Matching Process of Abstract Factory

type classes are found in the class groups of different sequences, Abstract Factory is required.

However, sometimes the class-types could not be identified due to missing super-classes in a bad design (as in Figure 4.2(a)). For those cases, similarity in the names of the classes are analyzed to identify the same types. The class names are split based on camel case, and the parts are matched. For example, 'WoodenDoor' is split to 'Wooden' and 'Door', and 'GlassDoor' is split to 'Glass' and 'Door', and matched to each others. The percentage of similarity matching in this special case, is decided through empirical analysis. Two names are considered to be a match, if at least 50% of the split parts match with each others. It is found that, for a threshold value less than 50%, the non-similar names are also matched, and for more than 50%, similar names cannot be matched. The equation for similarity calculation is –

$$\frac{splitMatchCount(nameA, nameB)}{max(splitCount(nameA), splitCount(nameB))} >= 0.5 \qquad (5.1)$$

Algorithm 1 Semantic Matching

```
1: system: System Matrix
2: cN: System Class Names
3: seqs: Sequence Diagrams
4: procedure MatchSemantic
5:     size ← seqs.size()
6:     seq ← [size][size]
7:     type[cN.length][cN.length] ← GenTypeMatrix()
8:     for i ← 0 to size do
9:         for j ← i + 1 to size do
10:            CompareSeq(seqs.get(i), seqs.get(j), i, j)
11:        end for
12:    end for
13:    maxMatch ← 0
14:    for i ← 0 to size do
15:        for j ← 0 to size do
16:            if maxMatch < seq[i][j] then
17:                maxMatch ← seq[i][j]
18:            end if
19:        end for
20:    end for
21:    return maxMatch
22: end procedure
23: procedure CompareSeq(s1, s2, p1, p2)
24:    RemoveDuplicates(s1, s2)
25:    for i ← 0 to s1.size() do
26:        for j ← 0 to s2.size() do
27:            s ← −1, d ← −1
28:            for k ← 0 to cN.length do
29:                if s1.get(i) = cN.get(k) then
30:                    s ← k
31:                end if
32:                if s2.get(j) = cN.get(k) then
33:                    d ← k
34:                end if
35:                if s! = −1 and d! = −1 then
36:                    break
37:                end if
38:            end for
39:            if s! = −1 and d! = −1 then
40:                seq[p1][p2] ← seq[p1][p2] + type[s][d]
41:                seq[p2][p1] ← seq[p2][p1] + type[s][d]
42:            end if                    100
43:        end for
44:    end for
45: end procedure
```

If the design is too bad to neither have super-classes nor similar names for the same types of classes, the approach will fail to generate type matrix and so, match semantics. Thus, for getting recommendation, the basic design principles should be followed by the designers.

The semantic matching process is shown in Algorithm 1. First of all the type matrix is generated (Algorithm 1 Line 7). As mentioned previously, it can be generated from super-class information (that is, the generalization relationship) or similar naming of classes. The type matrix is a 0,1 matrix, where the same type classes share value 1, and others share value 0. Then, every sequences (that is, the class families) are compared to each others (Lines 8–12). The procedure CompareSeq is called for this reason. In CompareSeq, the duplicates in the sequences being compared are removed in Line 24. Then nested loops are executed for getting the sequence class positions in the *type* matrix using the class names list (cN) (Line 25–38). The value in those positions inside the *type* matrix (0 or 1) is added to the sequence (seq) matrix in Lines 40–41. After the calculation of the values in all the seq positions, the maximum match ($maxMatch$) between the sequences are identified in Lines 13–20. This $maxMatch$ is returned as the outcome of semantic matching. If the matching value is $>= 2$, there is a valid semantic match for Abstract Factory; and so, the semantic matching is successful. For other cases, semantic matching is failed.

5.2.2 *missing* Factory Method (mFM)

From the identified structural, behavioral and semantic characteristics of *missing* Factory Method, it can be noticed that these characteristics are quite similar to the characteristics of Abstract Factory. Thus, matching levels are also similar between these two patterns.

5.2.2.1 mFM Structural Matching

The structural matching phase of Factory Method is similar to that of Abstract Factory as shown in Figure 5.8. There are two phases of matching

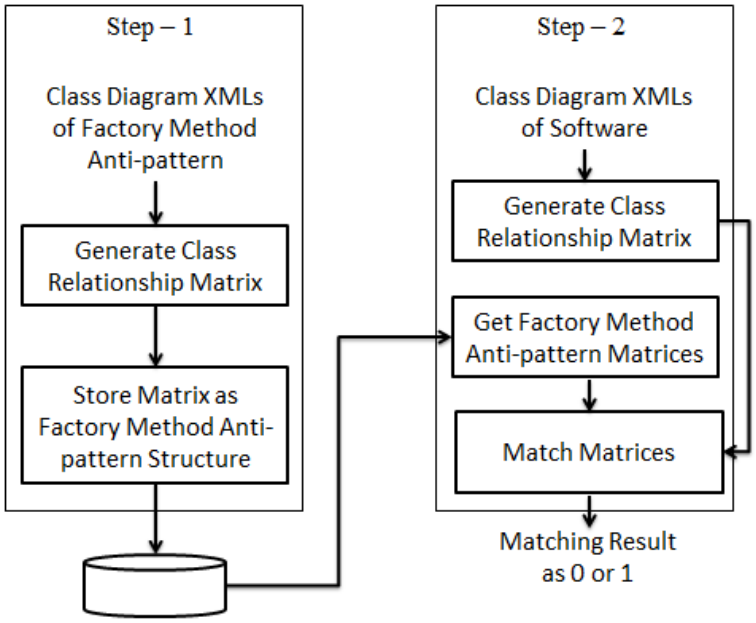
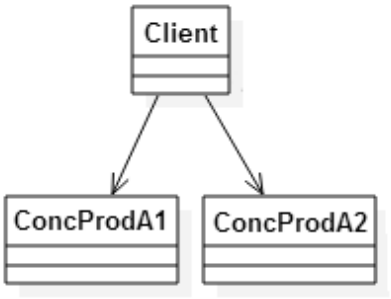


Figure 5.8: Structural Matching of the *missing* Factory Method

as well – storing anti-pattern structures and matching the stored structures with the software structure.

For storing the anti-pattern structures, $n \times n$ matrix of prime numbers is created similarly as described in mAF structural matching. For this, the structures of the Factory Method anti-pattern (as shown in Figure 4.6) is inputted in the system in XML format, and the desired matrices are created. Figure 5.9 and Figure 5.10 show the matrices generated for the structures of



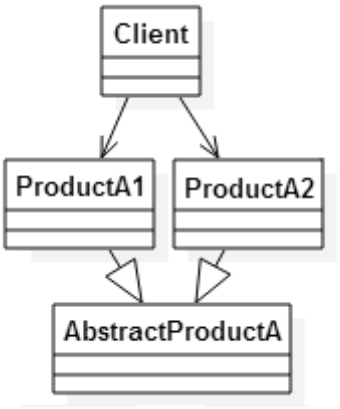
(a) Anti-pattern Structure

	C	A1	A2
C	0	2	2
A1	0	0	0
A2	0	0	0

(b) Matrix Representation

Figure 5.9: Generated Matrix of Figure 4.6(a)

Figure 4.6, stored as anti-pattern structures of Factory Method. Figure 4.6 is repeated in Figure 5.9(a) and Figure 5.10(a) as a reminder of the structures. The matrix of Figure 5.9(b) is generated from Figure 5.9(a). And the matrix of Figure 5.10(b) is generated from Figure 5.10(a). The notations are similar as the Abstract Factory notations. The matrices are created using 2, 3, and 5 for expressing *association*, *generalization*, and *aggregation* respectively as shown in Table 5.1 [8].



(a) Anti-pattern Structure

	AbsA	A1	A2	C
AbsA	0	0	0	0
A1	3	0	0	0
A2	3	0	0	0
C	0	2	2	0

(b) Matrix Representation

Figure 5.10: Generated Matrix of Figure 4.6(b)

In the second step, the system matrix is generated, and matched with these stored anti-pattern matrices using matrix matching approach. If any anti-pattern matrix is found inside the system matrix, the mFM structural matching output is 1, and otherwise 0.

5.2.2.2 mFM Behavioral Matching

The matching levels of Factory Method is similar to the matching levels of Abstract Factory, as already stated. Thus, in the behavioral phase, sequence diagrams are used and the *lifelines* are extracted. By which, the classes of different execution sequences are found from the *lifelines*.

As this level is the same for Abstract Factory and Factory Method, a comparison between these two upholds the characteristical resemblance. In this level, families are detected both for Abstract Factory and Factory Method; but as families do not exist in Factory Method, the classes of these families are conceptually individual which are executed in the same path. The existence of multiple sequence diagrams express multiple execution paths, and so, match with the behavior of Factory Method.

5.2.2.3 mFM Semantic Matching

The validation of behaviorally matched classes are performed in the semantic matching step. The semantic matching basically tests whether the classes instantiated in different execution paths are of the same type.

At first, type matrix is generated by analyzing super-class information and class name similarity. The types of classes are analyzed to validate the single class group (instantiated classes in different execution sequences are of the same type).

While, for Abstract Factory, existence of such two types of classes are mandatory, for Factory Method, only one type classes are required to be in the different execution sequences (semantic characteristic of Factory Method). If multiple such class types are found, Abstract Factory is suitable; on the other hand, for one type of such classes, Factory Method is applicable (and for no such classes, none of these two are appropriate). This is why, for semantic matching of Factory Method, Algorithm 1 is used. If the algorithm returns a matching value >= 2, it prefers Abstract Factory to be applied, for a matching value = 1, Factory Method is suitable.

5.2.3 *missing* Builder (mB)

The characteristics of the Telescoping Constructor [57] is detected in the software design for identifying the *missing* Builder (mB). As mB does not possess any semantic properties as stated in Chapter 4, structural and behavioral matching are sufficient for its detection. This detection process is described in this section.

5.2.3.1 mB Structural Matching

As the structure of Telescoping Constructor depends on an individual class, rather than a group of classes, this matching is conducted on each class in the class diagram individually.

In this level, every class is inspected one-by-one to identify existence of multiple constructors. The classes having at least two constructors are marked as the potential classes for applying Builder, and proceeds to the next level of matching.

5.2.3.2 mB Behavioral Matching

The classes having multiple constructors participate in the behavioral matching of Telescoping Constructor. The constructors of each class are examined to identify a parameter list pattern like Figure 4.11. Here, it is examined, whether with every constructor, new parameters are introduced, but the parameters of the previous constructors are preserved.

Therefore, at first the constructors are sorted based on the increasing number of parameters. Then each constructor is compared with its previous

constructor to check whether the constructor has introduced new parameters. The other parameters are common between these two constructors and are unchanged in the current one.

Algorithm 2 mB Behavioral Matching

```
1: constructors: List of Class Constructors List (2D List)
2: builderClasses: Empty List to Store Builder Classes
3: procedure MatchBehavior(constructors, builderClasses)
4:     maxInClass ← 0
5:     leastTelescope ← MAX
6:     for i ← 0 to constructors.size() do
7:         for j ← 0 to constructors.get(i).size() do
8:             for k ← 0 to constructors.get(i).size() do
9:                 if j==k then
10:                    continue
11:                end if
12:                variables1 ← constructors.get(i).get(j).parameters
13:                variables2 ← constructors.get(i).get(k).parameters
14:                commonSet ← variables1 ∩ variables2
15:                if commonSet.size() < leastTelescope
           ∧ commonSet.size()! = 0 then
16:                    leastTelescope ← commonSet.size()
17:                end if
18:                if commonSet.size() > 1 then
19:                    builderClasses.add(constructors.get(i).get(j).name);
20:                end if
21:            end for
22:        end for
23:        if leastTelescope > maxInClass ∧ leastTelescope! = MAX then
24:            maxInClass ← leastTelescope
25:        end if
26:    end for
27:    return maxInClass == MAX?0 : maxInClass
28: end procedure
```

Algorithm 2 shows the comparison between the constructors to match the mB behavior. As input, a list of constructor lists are taken. In this

list of lists, constructors of individual classes are stored. Also, an empty list, *builderClasses* is taken to store the potential *builder* classes. In Line 4 and 5, two variables are initialized. Variable *maxInClass* represents the maximum number of constructor parameter match in all classes, and *leastTelescope* tracks the minimum number of telescoping constructors in a class. Its initial value is selected to be a *MAX* value, and the least number is calculated gradually through iterations. Loop in Line 6 iterates through all the individual classes to analyze its constructor list. Then comparisons between the constructors of that class are performed in nested loops (Line 7–21). Comparison between the same constructors are avoided in Line 9–11. For the other cases, constructor parameters are taken and intersected to find the common variable set (Line 12–14). The set size is stored in the *leastTelescope* variable (Line 15–17), and for size > 1, the constructor class is stored as a *builder* class. Also, the maximum number of telescoping constructors is saved in *maxInClass* (Line 23–25). Finally, the *maxInClass* is returned as the count of telescoping constructors. For $maxInClass > 0$, mB behavior is considered to be matched.

5.2.4 *missing* Prototype (mP)

Similar to the *missing* Builder, *missing* Prototype also focuses on individual classes. However, this focus is not on the *prototype* classes, but on the other classes that instantiates those. The detection of the *missing* Prototype by examining these classes are described. The structural and behavioral matchings are demonstrated, as these two characteristics exist for *missing* Prototype.

5.2.4.1 mP Structural Matching

The structural matching of *missing* Prototype is done by checking the variables of the individual classes. If a class contains multiple variables having the type of another class, that second class is a candidate of *prototype* class. If any such classes are found, the structural matching is successful, and the behavioral matching is proceeded.

Hence, in this step, class diagram XML is taken as input. Each class of the diagram is analyzed individually to detect the existence of certain multiple variables. These variables, in that particular class, are of another class type. If any such structural characteristic is detected, the framework proceeds to the behavioral matching step of Prototype.

Algorithm 3 shows the mP structural matching. Here, *classDiagram* stores the class diagram of the software. *prototypeClasses* and *callClasses* are two empty lists for storing the *prototype* and *instantiator* classes respectively. In Line 5 and 6, two variables are initialized. For storing the number of matches, *matchCount* is declared with an initial value 0. Variable n stores the number of classes in the class diagram. For each class in the class diagram, variable counts of that class type are stored in an array, *classVarCount* (Line 7–16). If a class have more than one variable of another class type ($classVarCount > 0$), the classes are added in *prototypeClasses* and *callClasses* respectively (Line 17–25). In Line 27, the *matchCount* is returned. For $matchCount > 1$, mP structural matching is successful.

Algorithm 3 mP Structural Matching

```
1: classDiagram: Class Diagram
2: prototypeClasses: Empty String ArrayList for storing prototype classes
3: callClasses: Empty String ArrayList for storing instantiator classes
4: procedure matchStructure
5:     matchCount ← 0
6:     n ← classDiagram.classes.size()
7:     for i ← 0 to n do
8:         class ← classDiagram.classes.get(i)
9:         classVarCount ← [n]
10:        for j ← 0 to classs.variables.size() do
11:            for k ← 0 to n do
12:                if classs.variables.get(j).type! = null
           ∧ classs.variables.get(j).type
           .matches(classDiagram.classes.get(k).name) then
13:                    classVarCount[k] ← classVarCount[k] + 1
14:                end if
15:            end for
16:        end for
17:        for j ← 0 to classVarCount.length do
18:            if classVarCount[j] > 1 then
19:                callClasses.add(classDiagram.classes.get(i).name)
20:                prototypeClasses.add(classDiagram.classes.get(j).name)
21:            end if
22:            if matchCount < classVarCount[j] then
23:                matchCount ← classVarCount[j]
24:            end if
25:        end for
26:    end for
27:    return matchCount > 1?matchCount : 0
28: end procedure
```

5.2.4.2 mP Behavioral Matching

From the anti-pattern analysis section it can be identified that, the behavior of the *missing* Prototype can be two-fold. The overall process is shown in Figure 5.11. The process description is as follows.

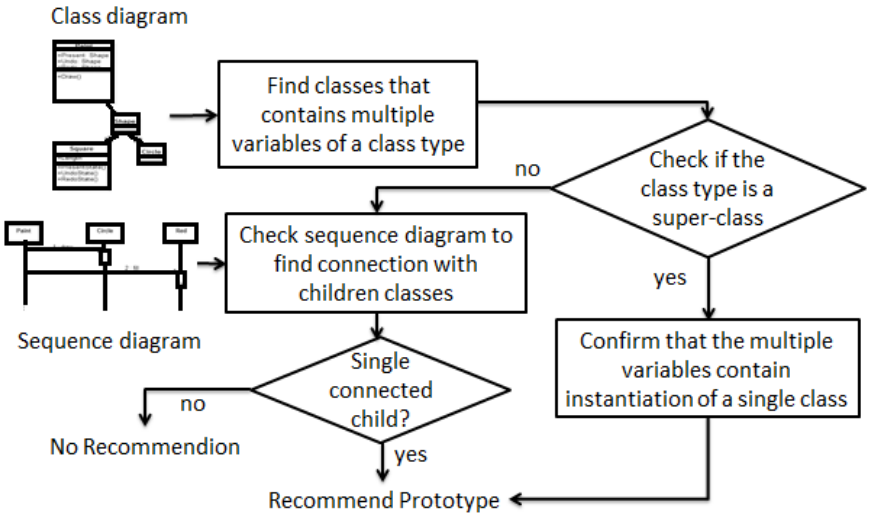


Figure 5.11: Behavioral Matching Process of Prototype

In the first case, the identified class variable type (the class that has multiple occurrences) is not of a super-class (does not have a generalization relation from any other classes). In this case, it is confirmed that the variables will not contain instantiations of any other classes (children classes to be specific); it will contain the instantiations of that class type only. And so, there is no confusion that the multiple instantiations are of a single class only. That single class is the *prototype* class.

In the second case, if the target class is a generalized class, it may happen that the variables contain the instantiations of its children classes. Thus, it is possible that the multiple instantiations are not of a single class. Those can

be of multiple classes that are stored in a generalized variable type. Now, for the verification of the instantiated classes in the second case, sequence diagrams are used. The sequence diagrams having the *instantiater* class as a *lifeline* are analyzed. It is checked, whether multiple children of the target class are connected to the *instantiater* class in the diagram. If multiple children are not connected, it is confirmed that the instantiations are of only one class (the single connected child). So, peresence of multiple instantiations of a single class makes it the *prototype* class. Otherwise, the instantiations are of different children. So, the multiple instantiations of one class cannot be confirmed, accepting the behavior of *missing* Prototype to be absent.

5.2.5 *missing* Singleton (mS)

From chapter 4 Section 4.1.5.2, it is identified that *missing* Singleton (mS) only posses the behavioral and semantic characteristics. Thus, matching the software design characteristics with mS is performed through behavioral and semantic matchings. The procedure of detecting mS is described below.

5.2.5.1 mS Behavioral Matching

As stated in the anti-pattern analysis of Singleton (Chapter 4 Section 4.1.5), it contains conditional checking before instantiating *singleton* classes to verify whether it has already been instantiated. This behavior can be found in the activity diagrams as it represents the program flow. The activity diagram describing such a behavior has a structure like Figure 5.12, so these are analyzed in this level for finding such structures.

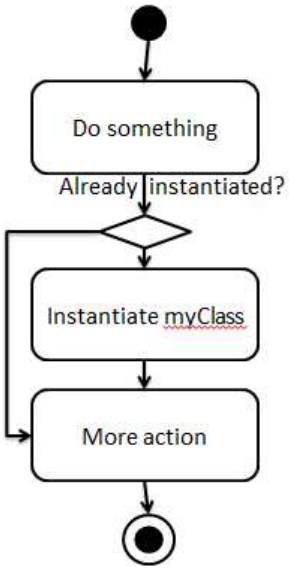


Figure 5.12: Activity Diagram Describing Singleton Behavior

Here, WordNet[1] is used to find any terms as 'instantiate', 'create', 'initiate', 'load', etc. in action nodes of the diagrams. If found, its connectivities are checked to identify whether any decision node is connected to that action node. The connectivities of the decision nodes are also examined by analyzing the activity diagram XML, to find the other action nodes. Finally, if the full structure of conditional checking is found (as shown in Figure 5.12), the behavior of *missing* Singleton is matched.

5.2.5.2 mS Semantic Matching

This is the confirmation step of the *missing* Singleton's existence. If the conditional checking, described in the behavioral matching, is found in the activity diagram, the diagram nodes participating in the checking are stored. In this semantic matching step, those nodes are searched in the class list of

[1]A large lexical database of English, for details - https://wordnet.princeton.edu/

the software class diagram. If a matched class is found, the class is concluded to be the *singleton* class.

If the activity diagram does not mention the class name (for example, point the class as 'instantiate the module' where 'module' is not the class name), this step will fail to detect *missing* Singleton.

5.3 Score Assignment

Based on the match or mismatch of the structural, behavioral, and semantic properties, each design pattern obtains a score between 0 to 1. If a system design is matched with an anti-pattern completely (structurally, behaviorally and semantically), only then the corresponding design pattern obtains a full score of 1. The formula of the score calculation is –

$$score_i = \frac{\alpha \cdot str_i + \beta \cdot beh_i + \gamma \cdot sem_i}{\alpha + \beta + \gamma} \tag{5.2}$$

where,

- str, beh, sem are the outcome of the structural, behavioral and semantic match respectively, and possesses a boolean value of 0 or 1 (0 for mismatch, 1 for match);
- α, β, γ are the weights of the structural, behavioral and semantic properties of the anti-patterns respectively;
- and, $i \in \{1, ..., n\}$.

For some *missing* patterns, all these three properties are not present (for example, semantic in Builder and Prototype, structure in Singleton). In those cases, the weight of that property is set to 0 ($\gamma = 0$ for Builder, Prototype; $\alpha = 0$ for Singleton). For other cases, the weight is set based on the significance of that property in the anti-pattern detection. As it can be seen from the matchings, the structural match provides a general level of matching; the behavioral match refines the anti-pattern's presence by specifying it; and finally the semantic match concludes the existence of the anti-pattern by considering the semantic details. Thus, the relation between the weights of these levels are - $\alpha < \beta < \gamma$.

As the matching levels are interrelated to each others (for example, semantic matching uses the outcome of the behavioral matching for refining the anti-pattern's existence), the next level of matching is only successful if the previous level is matched. If the next level is matched, it is an evidence that the previous levels are also matched to some extent; making the level more significant than the previous ones. This is why, the weight of a level is given a value, twice the value of the previous level's weight; $\alpha = 1, \beta = 2, \gamma = 4$.

These selected weights are justified by the experimentation in Chapter 7.

5.4 Design Pattern Recommendation

After the scores of each design pattern are calculated, those can be recommended based on the obtained scores. If a design pattern gets the highest score that is 1, it perfectly fits in that software design, and thus is recommended. For the other cases, mismatch occurred in some level of matching.

This mismatch can be a reason of failure to extract the software design properly, or because of incomplete design diagrams, or may be the anti-pattern is truly not present in the design. Thus, if two of the properties match, and one mismatch, their is a chance that the anti-pattern might exist in the software, but could not be detected.

For a partial matching (score < 1), the design patterns are also suggested based on a threshold value. As already stated, design patterns can be suggested if multiple levels match; making an appropriate threshold value of 0.43, as shown below –

$$\frac{\alpha + \beta}{\alpha + \beta + \gamma} = \frac{3}{7} = 0.43$$

However, it can be noticed that if only the semantic is matched and the other two levels mismatched, it will still have a value greater than the threshold –

$$\frac{\gamma}{\alpha + \beta + \gamma} = \frac{4}{7} = 0.57 > 0.43$$

However, it can never be happened, because the semantic match refines the behavioral match; if the behavior matching is failed, the semantic matching cannot get a positive response.

This threshold value is only appropriate for Abstract Factory and Factory Method as these two have all the three levels of matching, while Builder, Prototype and Singleton only have two levels of matching. The suggestions of Builder, Prototype and Singleton should be given if one of the levels is

matched successfully, making an appropriate threshold value of 0.3 –

$$\frac{\beta}{\beta+\gamma} = \frac{2}{6} = 0.33 \approx 0.3, \text{for Builder, Prototype}$$

$$\frac{\alpha}{\alpha+\beta} = \frac{1}{3} = 0.33 \approx 0.3, \text{for Singleton}$$

This value also preserves the requirements of the patterns having the three levels of matching that the matching response of at least two levels need to be positive.

$$\frac{\alpha}{\alpha+\beta+\gamma} = \frac{1}{7} = 0.14 < 0.3$$

$$\frac{\beta}{\alpha+\beta+\gamma} = \frac{2}{7} = 0.29 < 0.3$$

And so, it is appropriate for all the design patterns. Thus, for a score value greater than the threshold value, 0.3, design pattern is suggested for the software design.

This threshold value of 0.3 is also justified by the experimentation in Chapter 7.

5.5 Summary

In this chapter, an approach to recommend creational design patterns using anti-patterns in the software design phase is introduced. For the recommen-

dation, it uses the derivation of the anti-pattern characteristics in the previous chapter. A tool is proposed named ACDPR, where those anti-patterns are detected in a software design by structural, behavioral and semantic matching. The design patterns are recommended based on their obtained scores (for a score of 1) in these matching levels. Design patterns are also suggested for designers' further consideration based on a threshold value on the scores. The next chapter shows a case study for one of the sample projects to improve the understandability of the framework.

Chapter 6

Case Study on a Sample Project: "Painter"

In this chapter, a detail walk through on the proposed recommendation approach is given. This will give an insight to the pattern recommendation process. The design pattern recommender, named as ACDPR, is proposed in the previous chapter. The process of recommendation includes detection of different anti-patterns by characteristic matchings with software design, calculating scores of the design patterns, and finally providing suitable recommendations. This chapter provides a step-by-step analysis of the recommendation process. All five creational design patterns' level matchings are performed, to identify which patterns are to be recommended and which are to be suggested. A project requiring Abstract Factory have been chosen here as the example project. This is because, *missing* Abstract Factory (mAF) detection possesses the matching of all three characteristics as mentioned in Chapter 5.

6.1 About Project *Painter*

The project, *Painter* [62] is a well-known example of Abstract Factory usage. The scenario of the project is as follows:

"The *Paint* can draw three types of *Shape*s - *Circle*, *Triangle*, or *Square*. The *Shape*s can be filled with three *Color*s - *Red*, *Blue*, or *Green*. *Circle*s will be *Red*, *Triangle*s will be *Blue*, and *Square*s will be *Green*."

It can be seen that, the scenario has three family of classes – *Circle* and *Red*, *Triangle* and *Blue*, and *Square* and *Green*. This existence of families are well-handled by Abstract Factory. Also, the classes of different families are of same types as required by Abstract factory applicability. *Circle*, *Triangle* and *Square* are of type *Shape*, and *Red*, *Blue* and *Green* are of type *Color*. So, the optimized design of *Painter* needs Abstract Factory. The design, applying Abstract Factory, is shown in Figure 6.1.

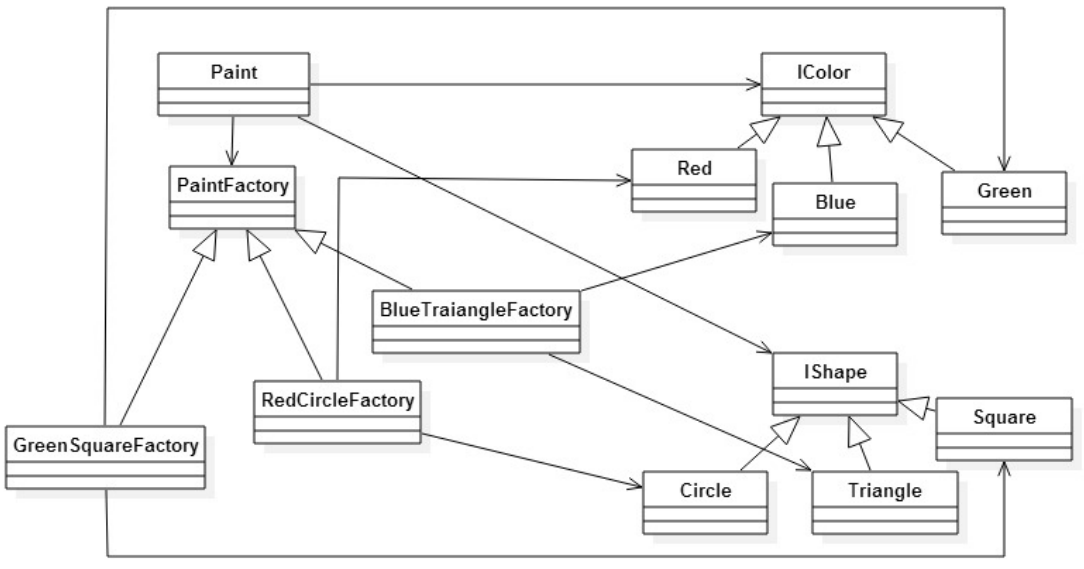
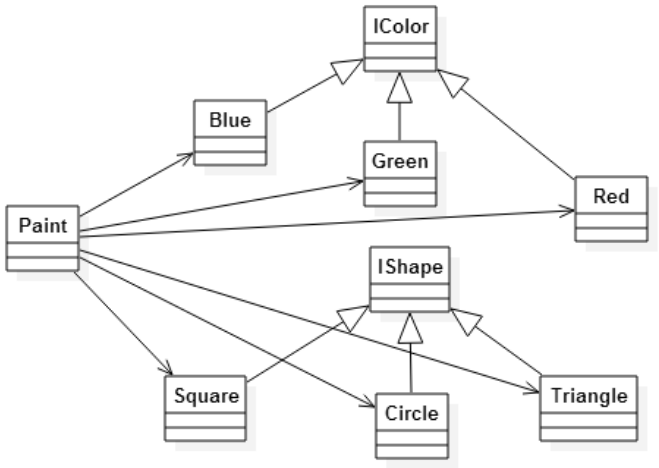
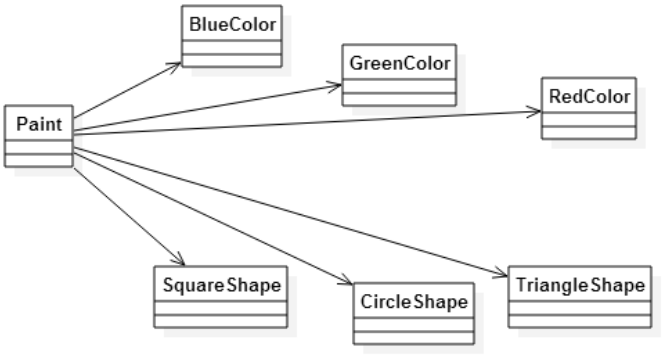


Figure 6.1: Design of *Painter*, Implementing Abstract Factory

In this figure, there are different factories for creating different families - *RedCircleFactory*, *BlueTriangleFactory* and *GreenSquareFactory*. These factories instantiate the *Client*'s required classes and returns those. *RedCircleFactory* instantiates *Red* and *Circle*. *BlueTriangleFactory* creates *Blue* and *Triangle*. *Green* and *Square* are instantiated by *GreenSquareFactory*. These factories are the concrete classes of the abstract class *PaintFactory*. And, *IShape* and *IColor* are the super-classes of *Circle*, *Triangle* and *Square*, and *Red*, *Blue* and *Green* respectively.



(a)



(b)

Figure 6.2: Bad Designs of *Painter*

For testing the recommendation tool, the project is intentionally badly designed without implementing Abstract Factory design pattern. This badly designed project will not have any factory class to instantiate the product classes. Two badly designed structures are shown in Figure 6.2. Figure 6.2(a) shows a structure with abstract product classes, and Figure 6.2(b) shows the design with no abstract class. Figure 6.2(a) is considered the case of bad design for performing this case study. And, Figure 6.2(b) is used as an example of a special case.

6.2 Detection of *missing* creational patterns for *Painter*

In this section, for every creational design pattern, the matching levels are executed on project *Painter*. The matching results are used to calculate the score for each pattern in the next section.

It is assumed that, the analysis of anti-patterns have already been performed. And thus, the required anti-pattern information are already stored in the tool. These are used for detecting anti-patterns in the inputted system design and recommending the corresponding design patterns.

6.2.1 *missing* Abstract Factory Detection for *Painter*

The following levels perform Abstract Factory anti-pattern matching with the *Painter* project.

6.2.1.1 Structural Matching

As mentioned in 'Structural Matching' in Chapter 5, the system structure is to be matched with the *missing* Abstract Factory (mAF) structure. For this, the initial class diagram of *Painter*, shown in Figure 6.2(a), is converted to XML format. The XML is shown in Figure 6.3.

Here, classes are found in the "*pacakegeElement*" tags in attribute "*xmi* : *type*", "*uml* : *Class*". The association relationships between classes are found inside the classes in *ownedMember* tags with "*xmi* : *type*" attribute "*uml* : *Association*". Similarly, the generalization relationships can be seen inside the *generalization* tags.

This inputted XML is converted into a matrix of prime numbers for preserving the relationships between the classes (instructed in [35]), as shown in Figure 6.2(a). There are six association ($Paint \xrightarrow{A} Blue$, $Paint \xrightarrow{A} Green$, $Paint \xrightarrow{A} Red$, $Paint \xrightarrow{A} Square$, $Paint \xrightarrow{A} Triangle$, $Paint \xrightarrow{A} Circle$) and six generalization (($Blue \xrightarrow{G} IColor$, $Green \xrightarrow{G} IColor$, $Red \xrightarrow{G} IColor$, $Square \xrightarrow{G} IShape$, $Triangle \xrightarrow{G} IShape$, $Circle \xrightarrow{G} IShape$)) relationships in the diagram. These are completely preserved by putting value '2' in places of association and '3' in places of generalization (Table 5.1).

```
<xmi:XMI xmlns:uml="http://schema.omg.org/spec/UML/2.0" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1" xmi:version="2.1">
  <xmi:Documentation exporter="StarUML" exporterVersion="2.0"/>
  <uml:Model xmi:id="AAAAAAFNpelNWzaj06w=" xmi:type="uml:Model" name="RootModel">
    <packagedElement xmi:id="AAAAAAFF+qBWK6M3Z8Y=" name="Model" visibility="public" xmi:type="uml:Model">
      <packagedElement xmi:id="AAAAAAFNpeQrMit1gRc=" name="Blue" visibility="public" isAbstract="false" isFinalSpecialization="false" isLeaf="false" xmi:type="uml:Class"
      isActive="false">
        <generalization xmi:id="AAAAAAFNpeYsnCzQW24=" visibility="public" xmi:type="uml:Generalization" specific="AAAAAAFNpeQrMit1gRc=" general="AAAAAAFNpeRumivk4B0="/>
      </packagedElement>
      <packagedElement xmi:id="AAAAAAFNpeRa0iuaqVY=" name="Green" visibility="public" isAbstract="false" isFinalSpecialization="false" isLeaf="false" xmi:type="uml:Class"
      isActive="false">
        <generalization xmi:id="AAAAAAFNpeZAFizh6Yw=" visibility="public" xmi:type="uml:Generalization" specific="AAAAAAFNpeRa0iuaqVY=" general="AAAAAAFNpeRumivk4B0="/>
      </packagedElement>
      <packagedElement xmi:id="AAAAAAFNpeRlKCu/nig=" name="Red" visibility="public" isAbstract="false" isFinalSpecialization="false" isLeaf="false" xmi:type="uml:Class" isActive="false">
        <generalization xmi:id="AAAAAAFNpeZWrCzybg0=" visibility="public" xmi:type="uml:Generalization" specific="AAAAAAFNpeRlKCu/nig=" general="AAAAAAFNpeRumivk4B0="/>
      </packagedElement>
      <packagedElement xmi:id="AAAAAAFNpeRumivk4B0=" name="IColor" visibility="public" isAbstract="false" isFinalSpecialization="false" isLeaf="false" xmi:type="uml:Class"
      isActive="false"/>
      <packagedElement xmi:id="AAAAAAFNpeR5UiwJN/0=" name="IShape" visibility="public" isAbstract="false" isFinalSpecialization="false" isLeaf="false" xmi:type="uml:Class"
      isActive="false"/>
      <packagedElement xmi:id="AAAAAAFNpeSDBiwu5oA=" name="Square" visibility="public" isAbstract="false" isFinalSpecialization="false" isLeaf="false" xmi:type="uml:Class"
      isActive="false">
        <generalization xmi:id="AAAAAAFNpeb69i0It38=" visibility="public" xmi:type="uml:Generalization" specific="AAAAAAFNpeSDBiwu5oA=" general="AAAAAAFNpeR5UiwJN/0="/>
      </packagedElement>
      <packagedElement xmi:id="AAAAAAFNpeSMICxT/wM=" name="Circle" visibility="public" isAbstract="false" isFinalSpecialization="false" isLeaf="false" xmi:type="uml:Class"
      isActive="false">
        <generalization xmi:id="AAAAAAFNpecKaS0ZDJI=" visibility="public" xmi:type="uml:Generalization" specific="AAAAAAFNpeSMICxT/wM=" general="AAAAAAFNpeR5UiwJN/0="/>
      </packagedElement>
      <packagedElement xmi:id="AAAAAAFNpeSVkCx45wg=" name="Triangle" visibility="public" isAbstract="false" isFinalSpecialization="false" isLeaf="false" xmi:type="uml:Class"
      isActive="false">
        <generalization xmi:id="AAAAAAFNpecZ4y0qAlU=" visibility="public" xmi:type="uml:Generalization" specific="AAAAAAFNpeSVkCx45wg=" general="AAAAAAFNpeR5UiwJN/0="/>
      </packagedElement>
      <packagedElement xmi:id="AAAAAAFNpeSfhCyd3VU=" name="Paint" visibility="public" isAbstract="false" isFinalSpecialization="false" isLeaf="false" xmi:type="uml:Class"
      isActive="false">
        <ownedMember xmi:id="AAAAAAFNpeec0C0+XHs=" visibility="public" xmi:type="uml:Association" isDerived="false">...</ownedMember>
        <ownedMember xmi:id="AAAAAAFNpeer0C2HxME=" visibility="public" xmi:type="uml:Association" isDerived="false">...</ownedMember>
        <ownedMember xmi:id="AAAAAAFNpefBwS3lUqU=" visibility="public" xmi:type="uml:Association" isDerived="false">...</ownedMember>
        <ownedMember xmi:id="AAAAAAFNpefU0i5YMr0=" visibility="public" xmi:type="uml:Association" isDerived="false">...</ownedMember>
        <ownedMember xmi:id="AAAAAAFNpeflLi7gJnw=" visibility="public" xmi:type="uml:Association" isDerived="false">...</ownedMember>
        <ownedMember xmi:id="AAAAAAFNpeiFwzOKHZE=" visibility="public" xmi:type="uml:Association" isDerived="false">...</ownedMember>
      </packagedElement>
    </packagedElement>
  </uml:Model>
</xmi:XMI>
```


Figure 6.3: Class XML of *Painter*

	Blue	Green	Red	IColor	IShape	Square	Circle	Triangle	Paint
Blue	0	0	0	3	0	0	0	0	0
Green	0	0	0	3	0	0	0	0	0
Red	0	0	0	3	0	0	0	0	0
IColor	0	0	0	0	0	0	0	0	0
IShape	0	0	0	0	0	0	0	0	0
Square	0	0	0	0	3	0	0	0	0
Circle	0	0	0	0	3	0	0	0	0
Triangle	0	0	0	0	3	0	0	0	0
Paint	2	2	2	0	0	2	2	2	0

Figure 6.4: Class Relation Matrix of *Painter*

The stored structures of anti-patterns are matched with *Painter* matrix using naive matrix matching. From Figure 6.2(a) and Figure 4.2(a), a match is encountered, which is shown by coloring the matched regions (that is, classes and relationship connections) in Figure 6.5. The structural matching is accomplished, and the tool will proceed to the next level of matching.

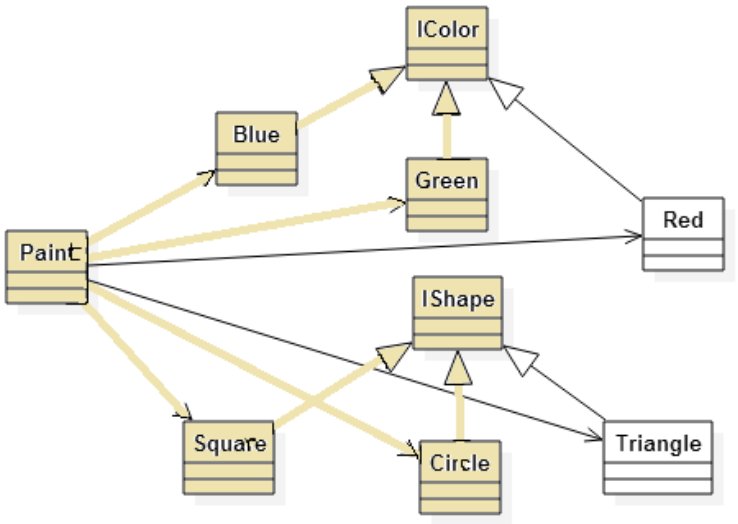


Figure 6.5: Matched Regions of *Painter* Structure

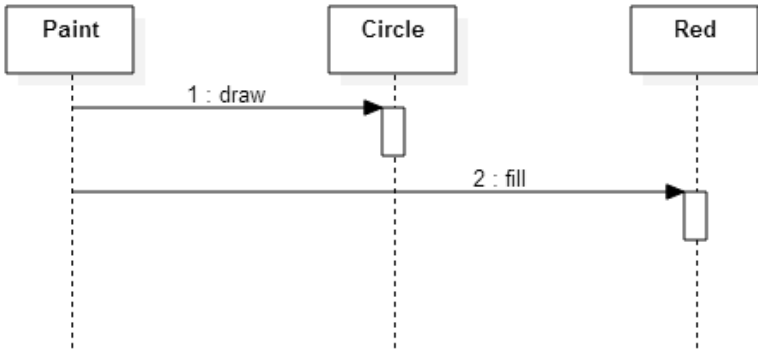
6.2.1.2 Behavioral Matching

The behavior of mAF is that, there are multiple families of classes in the software. For behavioral matching, the information about the interactions between classes in execution is required. This information is extracted from the sequence diagrams. From the scenario of *Painter*, three sequence diagrams can be drawn (Figure 6.6).

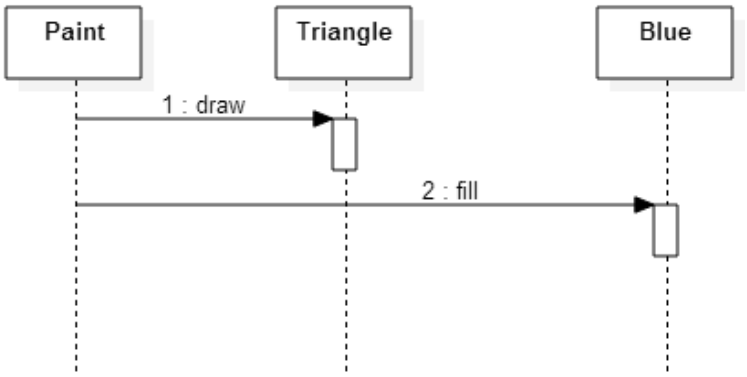
The class families are identified from the *lifelines* of these sequence diagrams. As, three sequence diagrams are inputted, three families are identified from those. The first family consists of *Paint*, *Circle* and *Red*; the second family has the classes *Paint*, *Triangle* and *Blue*; and the third family is comprised of *Paint*, *Square* and *Green*. As multiple families are found, the behavior of mAF is matched.

6.2.1.3 Semantic Matching

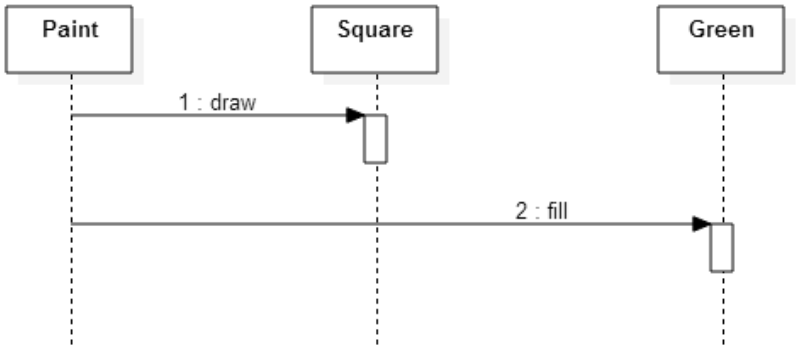
The mAF semantic matching identifies whether different families of classes contain classes with same types. The three families identified in the behavioral matching is validated in this level. First of all, the type matrix (as mentioned in subsection 5.2.1.3 "Semantic Matching") is generated using the super-class information from the class relation matrix (Figure 6.4). The type matrix is shown in Figure 6.7. Situations can occur that the super-class information is missing. For example, another variation of a badly designed class diagram can be created by the designer as shown in Figure 6.2(b). It is noticeable that, though the super-classes are missing, type matrix will still be generated from the similarity in the names of the same types of



(a) Circle Is Red



(b) Triangle Is Blue



(c) Square Is Green

Figure 6.6: Sequence Diagrams of *Painter*

classes. Red*Color*, Blue*Color*, Green*Color*; and Circle*Shape*, Triangle*Shape*, Square*Shape* are identified as same types.

However, if the names of same types are not similar, the approach will fail to generate the type matrix. For example - if the names of the classes are similar as Figure 6.2(b), but the super-classes *IShape* and *IColor* are missing, the approach will fail.

	Blue	Green	Red	IColor	IShape	Square	Circle	Triangle	Paint
Blue	0	1	1	0	0	0	0	0	0
Green	1	0	1	0	0	0	0	0	0
Red	1	1	0	0	0	0	0	0	0
IColor	0	0	0	0	0	0	0	0	0
IShape	0	0	0	0	0	0	0	0	0
Square	0	0	0	0	0	0	1	1	0
Circle	0	0	0	0	0	1	0	1	0
Triangle	0	0	0	0	0	1	1	0	0
Paint	0	0	0	0	0	0	0	0	0

Figure 6.7: Type Matrix of *Painter*

After the type matrix is generated, the class families are analyzed to test whether different classes having the same types are situated in different families. The three identified families are analyzed here, and found that all three families contain classes of same types. *Circle* (family-1), *Traiangle* (family-2) and *Square* (family-3) are of the same type, and similarly *Red* (family-1), *Blue* (family-2) and *Green* (family-3) are also same typed. So, the semantic matching ensures that the identified families from the behavioral matching are valid families.

6.2.2 *missing* Factory Method Detection for *Painter*

After Abstract Factory, the matching levels of Factory Method anti-pattern is as follows.

6.2.2.1 Structural Matching

As declared in Chapter 5, *missing* Factory Method matching levels are similar to *missing* Abstract Factory. And so, the tasks are also similar. First, the system structure is to be matched with the anti-patterns' structure of Factory Method. For this, the *Painter* project matrix (generated in the previous section) shown in Figure 6.4 is matched with the anti-pattern structures.

The Factory Method anti-pattern structures are assumed to be stored in the tool, similar as Abstract Factory. The structures of those stored anti-patterns (shown in Appendix B) are matched with the *Painter* matrix using naive matrix matching. However, no matching is found for the project matrix in Figure 6.4. Thus, the structural matching fails, yet the tool proceeds to the next level of matching.

6.2.2.2 Behavioral Matching

For behavioral matching of *missing* Factory Method, sequence diagrams are used to extract the classes in the same execution path. Sequence diagrams in Figure 6.6 are used for this purpose.

The classes in the same execution path are identified from the *lifelines* of these sequence diagrams. As, three sequence diagrams are inputted, three classes in each execution path are identified. The first group has *Paint*, *Circle*, and *Red*; the second group contains the classes *Paint*, *Triangle*, and *Blue*; and the third group is consisted of *Paint*, *Square*, and *Green*.

The identification is similar to Abstract Factory. The level is accomplished as there are multiple sequence diagrams, so, multiple execution paths.

6.2.2.3 Semantic Matching

In this semantic matching level, the three groups of classes identified in the behavioral matching is verified to find the conditionally instantiated class group. Now, as mentioned in Chapter 5, this level is the distinguisher in the matchings of *missing* Abstract Factory and Factory Method. While for Abstract Factory more than two groups of classes need to fulfill the semantic criteria, for Factory Method only one single group is required.

The tasks are similar as before, first the type matrix is generated (Figure 6.7). Then it is compared with the class group information. The class groups are analyzed to test whether different classes having the same types are situated in different groups. Three such groups are identified, *Circle* (group-1), *Traiangle* (group-2) and *Square* (group-3) are of the same type, and similarly *Red* (group-1), *Blue* (group-2) and *Green* (group-3) are also same typed. So, the semantic matching returns three groups, violating the existance of only one such group. This is why, the semantic matching of *missing* Factory Method fails.

6.2.3 *missing* Builder Detection for *Painter*

The detection of *missing* Builder has two matching levels (structural, behavioral) that are described as follows.

6.2.3.1 Structural Matching

The *missing* Builder concentrates on individual classes instead of grouped ones. In this level, individual class information is extracted from class dia-

gram of *Painter*. Thus, the full information of classes with attributes and operations are needed, as shown in Figure 6.8.

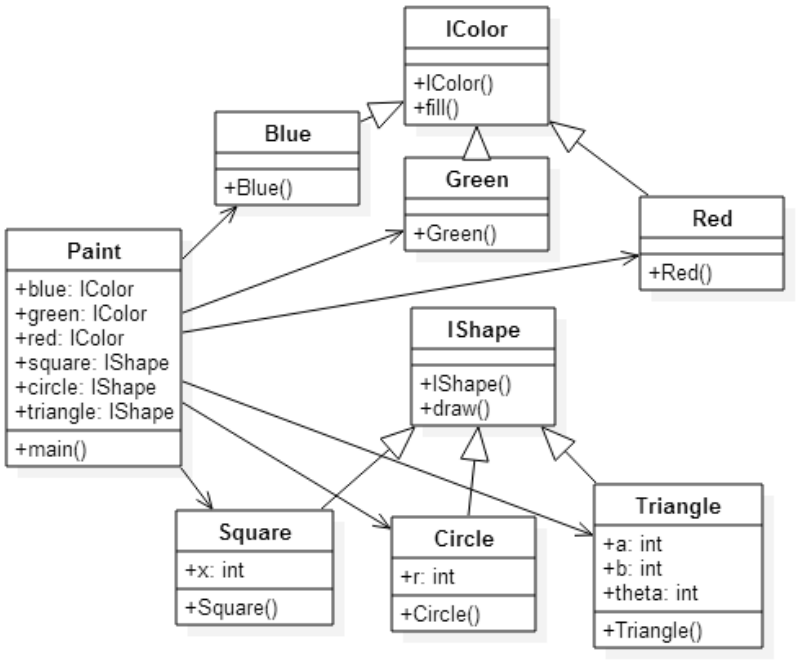


Figure 6.8: Class Diagram of *Painter* with Attributes and Operations

The Telescoping Constructor anti-pattern (*missing* Builder) has the structural property of containing multiple constructors in a class. From Figure 6.8, it can be seen that none of the classes contain multiple constructors. Thus, the structural matching fails for *missing* Builder.

6.2.3.2 Behavioral Matching

For behavioral matching of *missing* Builder, the multiple constructors are examined to find a specific parameter list pattern. New parameters are included with each new constructor here, by preserving the parameter list of the previous constructors in it.

However, the structural property of having multiple constructor is failed for *Painter*. So, as more than one constructor does not exist, the parameter list cannot be found. So, the behavior of *missing* Builder is also not matched.

6.2.4 *missing* Prototype Detection for *Painter*

The matching levels of *missing* Prototype are also the structural and behavioral, similar as *missing* Builder. The levels are described below.

6.2.4.1 Structural Matching

Structural matching of *missing* Prototype determines whether any of the individual classes have multiple variables of same class type. Here, information of classes with attributes are needed that can be found in Figure 6.8.

From the figure, it can be noticed that, *Paint* class contains multiple variables of same class types. It has three variables *red*, *blue*, and *green* of class type *IColor*; and three variables *square*, *circle*, and *triangle* of class type *IShape*. Thus, the structural matching of *missing* Builder has a positive response for *Painter* project, where class *Paint* is the *instantiator*, and *IColor* and *IShape* are potential *prototype* classes.

6.2.4.2 Behavioral Matching

There are two cases for the behavioral matching of *missing* Prototype. In the first case, the potential *prototype* class are not a super-class, and in the second case, it is super-class.

Here, the second case is applicable, as both *IColor* and *IShape* are superclasses (as shown in Figure 6.2(a)). In this case, the sequence diagrams are checked whether multiple children of these classes are connected to the *instantiator*. For *IColor*, *Paint* is connected with all of its children classes, *Red*, *Blue*, and *Green*. It cannot be confirmed that, which class will be initialized in the variables, failing in matching the behavior. Similarly for *IShape*, all three children classes *Square*, *Circle*, and *Triangle* are connected to the *instantiator* class *Paint*. This is why, *IShape* cannot be considered as a *prototype* class either. So, the behavioral matching level is not successful for any of the classes.

6.2.5 *missing* Singleton Detection for *Painter*

The structural matching does not exist in the *missing* Singleton detection. The behavioral and semantic matching with the *Painter* project is described in following sections.

6.2.5.1 Behavioral Matching

For the behavioral matching of *missing* Singleton, activity diagram is required. The activity diagram of *Painter* project is shown in Figure 6.9.

The activity diagram of a project requiring Singleton needs to have a conditional checking before the class it requires to be the *singleton* class. In the activity diagram in Figure 6.9, no such conditional checking can be found. So, the behavioral matching is failed for *missing* Singleton.

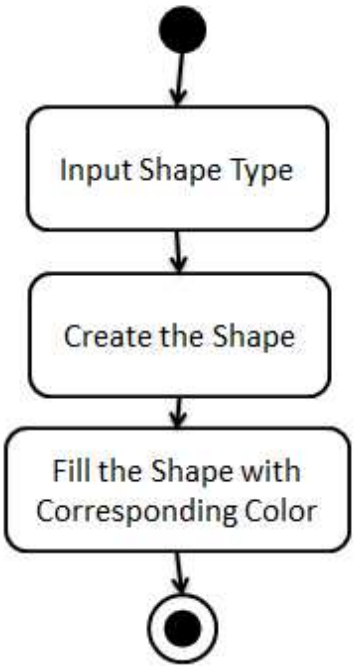


Figure 6.9: Activity Diagram of *Painter*

6.2.5.2 Semantic Matching

The semantic matching of *missing* Singleton verifies the found potential *singleton* classes, by checking their existence is the class list. In this case, the semantic matching fails, because, no potential *singleton* class is found in the behavioral matching step.

6.2.6 Score Calculation of Project *Painter*

From the above matching descriptions, the successful and failed matching values (0/1) are found. Table 6.1 shows the matching values for each level. From these values, a score for each pattern is calculated with $\alpha = 1$, $\beta = 2$, and $\gamma = 4$. The final scores are shown in Table 6.2.

Table 6.1: Outcome of the Matching Levels for *Painter*

	Abstract Factory			Factory Method			Builder		Prototype		Singleton	
	str	beh	sem	str	beh	sem	str	beh	str	beh	beh	sem
Painter	1	1	1	0	1	0	0	0	1	0	0	0

Table 6.2: Scores of the Design Patterns for $\alpha = 1, \beta = 2$ and $\gamma = 4$

	Abstract Factory	Factory Method	Builder	Prototype	Singleton
Painter	1	0.29	0	0.33	0

6.2.7 Design Pattern Recommendation and Suggestion for *Painter*

All the three matching levels of *missing* Abstract Factory indicate that the Abstract Factory design pattern is required to improve the project design. This pattern gets the score value 1, and so, Abstract Factory is recommended for this project. Most importantly, this recommendation is obtained in the design phase of the project, making it possible to re-design it.

As suggestion to the designers, Prototype pattern is selected. Using a threshold of 0.3, it is suggested for further consideration as a partial match. The project *Painter* has a good potential to use Prototype, as the shapes can contain cloning methods to clone itself (as required by Prototype). Often the shape related projects provide this cloning feature of objects. This suggestion also seems to be helpful to the designers.

6.3 Summary

This case study shows ACDPR's competence in recommending design pattern. The stepwise demonstration of the approach makes it clear why ACDPR is effective in design pattern recommendation. The next chapter describes the experimental analysis of ACDPR.

Chapter 7

Implementation and Result Analysis

This chapter aims to experimentally evaluate the performance of ACDPR by applying it on sample projects. A prototype of ACDPR have been implemented using Java programming language for assessing the performance of this proposed approach. First of all, the values of the weighting factors (those are, α, β and γ) were determined experimentally. For all the sample projects, each design patterns are assigned a score based on their matching values and these weighting factor values of the matching levels. Design pattern recommendations are provided for complete matchings with the pattern characteristics. The accuracy of these ACDPR recommendations was calculated using the precision, recall and F-measure metrics. Suggestions of design patterns were provided for a partial score using a threshold value. The threshold value is also determined in this section, for getting the most relevant suggestions. Both the recommendations and suggestions were ana-

lyzed over the expected recommendations according to GoF. To summarize, the effectiveness of ACDPR recommendations and suggestions are depicted in this chapter.

7.1 Environmental Setup

This section discusses the tools used to develop the ACDPR prototype and experimental procedures for the evaluation task. As mentioned earlier, the prototype was developed using Java programming language. In order to develop the prototype, following tools were used:

- Eclipse Luna (4.4.1) [63]: Java IDE for ADPR implementation
- StarUML Version-2.1.4 [64]: UML editor and XML converter

The implemented prototype is uploaded in GitHub [65] for supporting the result reproduction.

The experiments were performed on the following Desktop configuration:

- 2.20 GHz Intel Core 2 Duo Processor
- 6GB of RAM
- Windows 8 OS
- Java 7

For the assessment of the approach, 21 projects requiring different creational patterns have been used as dataset. These projects have been collected from the students of Institute of Information Technology, University of Dhaka.

These are the examples of bad designs, when the required design patterns were not applied. The project design diagrams are available on GitHub [66]. Table 7.1 shows the projects with its attributes – name, number of classes, and expected recommendation based on GoF. For example, the first row of Table 7.1 presents a project, Project_01. The name of this project is *CarOil*. It is a small project with eight classes. According to GoF, this project requires the implementation of Abstract Factory design pattern, but the pattern was not applied in the design of *CarOil*.

Table 7.1: Experimented Projects

Id	Project Name	No. of Classes	Expected Recommendation (According to GoF)
Project_01	CarOil	8	Abstract Factory
Project_02	Paint	9	Abstract Factory
Project_03	GameScene	10	Abstract Factory
Project_04	MazeGame	12	Abstract Factory
Project_05	Builder	6	Factory Method
Project_06	Dog	6	Factory Method
Project_07	Product	7	Factory Method
Project_08	Toy	8	Factory Method
Project_09	HouserBuilder	6	Builder
Project_10	KitBuilder	7	Builder
Project_11	FoodTelescoping	2	Builder
Project_12	RFTToASCII	6	Builder
Project_13	CarBuilder	5	Builder
Project_14	Analysis	4	Prototype
Project_15	Car	5	Prototype
Project_16	Scenary	4	Prototype
Project_17	Square	4	Prototype
Project_18	Configuration	7	Singleton
Project_19	FactoryTransaction	5	Singleton
Project_20	Logger	3	Singleton
Project_21	PDFReader	5	Singleton

Some prearrangements need to be done before running ACDPR on the sample project set. The design diagram UMLs are needed to be converted to XMLs using StarUML for providing as input. Also, the class structures for matching the anti-pattern matrices of Abstract Factory (Figure 4.2) and Factory Method (Figure 4.6) are needed to be stored in the framework, before running on the sample projects. The required prior arrangements are described next.

ACDPR Input Pre-processing

First of all, ACDPR stores the anti-pattern structures of Abstract Factory and Factory Method into the tool. The structures of these patterns, those are, the class diagrams, are depicted in Appendices A and B respectively. These diagrams are converted to XML format using StarUML, and stored in individual folders for Abstract Factory and Factory Method. The folder paths are inputted into the tool as pre-configuration. It is noticeable that, if new structures of anti-patterns are found, simply storing the XMLs of the diagrams in the pointed folders, will incorporate those into the tool.

Two sample XML structures of Abstract Factory and Factory Method are shown in Figure 7.1. Figure 7.1(a) shows the generated XML of Figure 4.2(a). It represents an anti-pattern structure of Abstract Factory. Figure 7.1(b) depicts the generated XML of Figure 4.6(a), showing the generated XML of Factory Method anti-pattern structure. In the figures, the classes are marked with underlines. The classes can be found by parsing the *packageElement* tags to find the "$xmi : type = uml : Class$". The relationships between classes can be found inside these class elements in the

```
▼<xmi:XMI xmlns:uml="http://schema.omg.org/spec/UML/2.0"
 xmlns:xmi="http://schema.omg.org/spec/XMI/2.1" xmi:version="2.1">
   <xmi:Documentation exporter="StarUML" exporterVersion="2.0"/>
  ▼<uml:Model xmi:id="AAAAAAFNQyOehxfoij4=" xmi:type="uml:Model" name="RootModel">
    ▼<packagedElement xmi:id="AAAAAAFF+qBWK6M3Z8Y=" name="Model" visibility="public"
     xmi:type="uml:Model">
      ▼<packagedElement xmi:id="AAAAAAFNQx/0ORUaVU4=" name="Client"
       visibility="public" isAbstract="false" isFinalSpecialization="false"
       isLeaf="false" xmi:type="uml:Class" isActive="false">
        ▶<ownedMember xmi:id="AAAAAAFNQyJqmxXmBKs=" visibility="public"
         xmi:type="uml:Association" isDerived="false">...</ownedMember>
        ▶<ownedMember xmi:id="AAAAAAFNQyJ6bxYv6Do=" visibility="public"
         xmi:type="uml:Association" isDerived="false">...</ownedMember>
        ▶<ownedMember xmi:id="AAAAAAFNQyKH+xaN0i0=" visibility="public"
         xmi:type="uml:Association" isDerived="false">...</ownedMember>
        ▶<ownedMember xmi:id="AAAAAAFNQyKXvBcARfk=" visibility="public"
         xmi:type="uml:Association" isDerived="false">...</ownedMember>
       </packagedElement>
       <packagedElement xmi:id="AAAAAAFNQyAAzRU/du8=" name="ConcProdA1"
       visibility="public" isAbstract="false" isFinalSpecialization="false"
       isLeaf="false" xmi:type="uml:Class" isActive="false"/>
       <packagedElement xmi:id="AAAAAAFNQyAKVBVkHzQ=" name="ConcProdB1"
       visibility="public" isAbstract="false" isFinalSpecialization="false"
       isLeaf="false" xmi:type="uml:Class" isActive="false"/>
       <packagedElement xmi:id="AAAAAAFNQyAWrBWJGGU=" name="ConcProdA2"
       visibility="public" isAbstract="false" isFinalSpecialization="false"
       isLeaf="false" xmi:type="uml:Class" isActive="false"/>
       <packagedElement xmi:id="AAAAAAFNQyAkKxWuziY=" name="ConcProdB2"
       visibility="public" isAbstract="false" isFinalSpecialization="false"
       isLeaf="false" xmi:type="uml:Class" isActive="false"/>
     </packagedElement>
   </uml:Model>
 </xmi:XMI>
```

(a) Abstract Factory Anti-pattern XML

```
▼<xmi:XMI xmlns:uml="http://schema.omg.org/spec/UML/2.0"
 xmlns:xmi="http://schema.omg.org/spec/XMI/2.1" xmi:version="2.1">
   <xmi:Documentation exporter="StarUML" exporterVersion="2.0"/>
  ▼<uml:Model xmi:id="AAAAAAFUOZBPmP5fcxg=" xmi:type="uml:Model" name="RootModel">
    ▼<packagedElement xmi:id="AAAAAAFF+qBWK6M3Z8Y=" name="Model" visibility="public"
     xmi:type="uml:Model">
      ▼<packagedElement xmi:id="AAAAAAFNQx/0ORUaVU4=" name="Client" visibility="public"
       isAbstract="false" isFinalSpecialization="false" isLeaf="false"
       xmi:type="uml:Class" isActive="false">
        ▶<ownedMember xmi:id="AAAAAAFNQyJqmxXmBKs=" visibility="public"
         xmi:type="uml:Association" isDerived="false">...</ownedMember>
        ▶<ownedMember xmi:id="AAAAAAFNQyKH+xaN0i0=" visibility="public"
         xmi:type="uml:Association" isDerived="false">...</ownedMember>
       </packagedElement>
       <packagedElement xmi:id="AAAAAAFNQyAAzRU/du8=" name="ConcProdA1"
       visibility="public" isAbstract="false" isFinalSpecialization="false" isLeaf="false"
       xmi:type="uml:Class" isActive="false"/>
       <packagedElement xmi:id="AAAAAAFNQyAWrBWJGGU=" name="ConcProdA2"
       visibility="public" isAbstract="false" isFinalSpecialization="false" isLeaf="false"
       xmi:type="uml:Class" isActive="false"/>
     </packagedElement>
   </uml:Model>
 </xmi:XMI>
```

(b) Factory Method Anti-pattern XML

Figure 7.1: Sample XML of Anti-pattern Structure

ownedMember attribute. For example, four "*uml* : *Association*", that is, association relationships, can be seen in Figure 7.1(a) and two associations are seen in Figure 7.1(b). These XMLs are parsed to generate prime number matrices as described in Chapter 5.

As input, ACDPR takes the design diagrams of software, to be tested for design pattern recommendation. The inputted design diagrams are –

- **Class Diagram:** For the structural matching of Abstract Factory, Factory Method, Builder and Prototype, class diagram is used. Also, semantic matching of Abstract Factory, Factory Method and Singleton, and behavioral matching of Builder and Prototype also rely on class diagram of the software.

- **Sequence Diagram:** Sequence diagrams are used for the behavioral matching of Abstract Factory, Factory Method and Prototype.

- **Activity Diagram:** In the behavioral matching of the Singleton design pattern, activity diagrams are analyzed.

These three types of diagrams are inputted into the tool after converting to the XML format. The sample XMLs of a class, sequence, and activity diagrams are shown in Figure 7.2. In Figure 7.2(a), a software class diagram XML is shown. Here, three classes can be identified by parsing the *packageElement*, *uml* : *Class* type. The classes are – *User*, *Logger* and *MainClass*. The relationship between these classes can be identified through *ownedMember* attribute, as mentioned before. *ownedOperation* attributes are also seen in the figure. These attributes store the class methods. Thus,

this XML contains all the individual class information as well as relationships among those.

(a) XML of a Class Diagram

(b) XML of a Sequence Diagram

```
▼<xmi:XMI xmlns:uml="http://schema.omg.org/spec/UML/2.0" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
 xmi:version="2.1">
   <xmi:Documentation exporter="StarUML" exporterVersion="2.0"/>
 ▼<uml:Model xmi:id="AAAAAAFQdv6cUVP/Nqo=" xmi:type="uml:Model" name="RootModel">
   ▼<packagedElement xmi:id="AAAAAAFF+qBWK6M3Z8Y=" name="Model" visibility="public" xmi:type="uml:Model">
     ▼<packagedElement xmi:id="AAAAAAFQdKq+t5wsjYc=" name="Activity1" visibility="public"
      isReentrant="true" xmi:type="uml:Activity" isReadOnly="false" isSingleExecution="false">
        <node xmi:id="AAAAAAFQdKsWJpwxaIA=" name="InitialNode1" visibility="public"
        xmi:type="uml:InitialNode"/>
        <node xmi:id="AAAAAAFQdKsyPpw20JY=" name="Enter username and password" visibility="public"
        xmi:type="uml:OpaqueAction" isLocallyReentrant="false" isSynchronous="true"/>
        <node xmi:id="AAAAAAFQdKuMTJxRug8=" name="DecisionNode1" visibility="public"
        xmi:type="uml:DecisionNode"/>
        <node xmi:id="AAAAAAFQdKxMPJxfF3A=" name="create new logger" visibility="public"
        xmi:type="uml:OpaqueAction" isLocallyReentrant="false" isSynchronous="true"/>
        <node xmi:id="AAAAAAFQdKx6/Jx5PCU=" name="use logger" visibility="public"
        xmi:type="uml:OpaqueAction" isLocallyReentrant="false" isSynchronous="true"/>
        <node xmi:id="AAAAAAFQdKykrJyTSVw=" name="ActivityFinalNode1" visibility="public"
        xmi:type="uml:ActivityFinalNode"/>
        <edge xmi:id="AAAAAAFQdK0SjJya890=" visibility="public" source="AAAAAAFQdKsWJpwxaIA="
        target="AAAAAAFQdKsyPpw20JY=" xmi:type="uml:ControlFlow"/>
        <edge xmi:id="AAAAAAFQdK1oTZyrJuQ=" visibility="public" source="AAAAAAFQdKsyPpw20JY="
        target="AAAAAAFQdKuMTJxRug8=" xmi:type="uml:ControlFlow"/>
        <edge xmi:id="AAAAAAFQdK6fhZy9Nrg=" name="no" visibility="public" source="AAAAAAFQdKuMTJxRug8="
        target="AAAAAAFQdKxMPJxfF3A=" xmi:type="uml:ControlFlow"/>
        <edge xmi:id="AAAAAAFQdK67BZzO8z8=" visibility="public" source="AAAAAAFQdKxMPJxfF3A="
        target="AAAAAAFQdKx6/Jx5PCU=" xmi:type="uml:ControlFlow"/>
        <edge xmi:id="AAAAAAFQdK7V9Zzfpoc=" visibility="public" source="AAAAAAFQdKx6/Jx5PCU="
        target="AAAAAAFQdKykrJyTSVw=" xmi:type="uml:ControlFlow"/>
        <edge xmi:id="AAAAAAFQdK/abpzxugs=" name="yes" visibility="public" source="AAAAAAFQdKuMTJxRug8="
        target="AAAAAAFQdKx6/Jx5PCU=" xmi:type="uml:ControlFlow"/>
      </packagedElement>
    </packagedElement>
  </uml:Model>
</xmi:XMI>
```


(c) XML of a Activity Diagram

Figure 7.2: Sample Input XMLs

Figure 7.2(b) depicts a sequence diagram XML. The *lifelines* of the sequence diagrams can be found in the *lifeline* tags. There are two *lifelines* in the diagram, *Paint* and *Square*. The messages between these diagrams are found inside the *message* tags. *Draw*, *Undo* and *Redo* are the passed messages between the *lifelines*.

A sample activity diagram XML is shown in Figure 7.2(c). The activity diagram nodes can be found in *node* tags and the edges between the nodes in the *edge* tags. In this diagram, there are six nodes and six edges between those. In this node list, one is the initial node, one is the final node, one is a decision node and the others are action nodes.

The mentioned three design diagrams represent the initial design of a software along with its showed behavior and semantics. These diagrams are used for characteristics matching with the anti-patterns and recommending the design patterns. The results are depicted in the next section.

7.2 Result Analysis

First of all, the values of the weighting factors in Equation 5.2 are needed to be defined. Then, recommendations are provided based on the scores, calculated using the equation. These recommendations are compared to the GoF suggested patterns. However, the recommendations might miss some potential design patterns due to incomplete design diagrams. For reducing its effect, more suggestions are given to the designers based on a threshold value on the scores.

Next, for getting more suggestions from ACDPR the threshold value is determined, and the suggestions are justified. So, the steps are –

- Determination of the weighting factor values
- Calculation of the recommendations accuracy using the weighting factor values
- Determination of score threshold value for suggestion

These steps are described in detail in the following sections.

7.2.1 Determining the Weighting Factor Values

Score of the design patterns are calculated using the matching level values. These matching levels are not of equal importance. This is why, weighting factors are used in the score calculation Equation 5.2. The values of these weighting factors are decided in this section.

The dataset projects were run using ACDPR, for determining the values of the weighting factors. For each project, the values of the structural, behavioral and semantic matchings corresponding to each design pattern are identified. The values are depicted in Table 7.2. Here, the column values correspond to the matching values of the characteristics (*str*, *beh* and *sem*) of each pattern (listed in the top header), and the row values correspond to each of the projects (named in the side header).

The cell values of '1' represent that the pattern characteristic could be detected in the project. Similarly, a cell value of '0' expresses a mismatch of characteristics between project and pattern. For example, the structural matching value of Project_01, for Abstract Factory, is 1. Similarly, for behavioral and semantic matching of Abstract Factory and behavioral matching of Factory Method, the value is 1. For the other matching levels of the different patterns, the matching value is 0. On the other hand, Project_19 only has a matching value 1, for behavioral matching of Singleton. All other matching levels are mismatched for this project, and so, possess values 0.

Based on these values, the scores of each design pattern are calculated for the projects using Equation 5.2 with different values of α, β and γ. The best fitted values of α, β and γ are determined.

Table 7.2: Outcome of the Matching Levels

	Abstract Factory			Factory Method			Builder		Prototype		Singleton	
	str	beh	sem	str	beh	sem	str	beh	str	beh	beh	sem
Project_01	1	1	1	0	1	0	0	0	0	0	0	0
Project_02	1	1	1	0	1	0	0	0	0	0	0	0
Project_03	1	1	1	0	1	0	0	0	0	0	0	0
Project_04	1	1	1	0	1	0	0	0	0	0	0	0
Project_05	0	1	0	1	1	1	0	0	0	0	0	0
Project_06	0	1	0	1	1	1	0	0	0	0	0	0
Project_07	1	1	0	1	1	1	0	0	0	0	0	0
Project_08	1	1	0	1	1	1	0	0	0	0	0	0
Project_09	0	0	0	0	0	0	1	1	0	0	0	0
Project_10	1	0	0	0	0	0	1	1	0	0	0	0
Project_11	1	1	0	0	1	0	1	1	0	0	0	0
Project_12	0	0	0	0	0	0	1	1	0	0	0	0
Project_13	0	0	0	0	0	0	1	1	0	0	0	0
Project_14	0	0	0	0	0	0	0	0	1	1	0	0
Project_15	0	0	0	0	0	0	0	0	1	1	0	0
Project_16	0	0	0	0	0	0	0	0	1	1	0	0
Project_17	0	0	0	0	0	0	0	0	1	1	0	0
Project_18	0	0	0	0	0	0	0	0	0	0	1	1
Project_19	0	0	0	0	0	0	0	0	0	0	1	0
Project_20	0	0	0	0	0	0	0	0	0	0	1	1
Project_21	0	0	0	0	0	0	0	0	0	0	1	1

7.2.1.1 Scores with $\alpha = 1$, $\beta = 1$ and $\gamma = 1$

The matching values of the attributes from Table 7.2 are used in Equation 5.2 with $\alpha = 1$, $\beta = 1$ and $\gamma = 1$. The scores are shown in Table 7.3.

Here, it can be seen that, as the weights of the levels are same, the matching levels does not have any effect on the scores. For example, Project_01, Project_02, Project_03 and Project_04 have provided a score of 0.33 from Factory Method. From Table 7.2, it can be seen that, the matching level is

Table 7.3: Scores of the Design Patterns with $\alpha = 1, \beta = 1$ and $\gamma = 1$

	Abstract Factory	Factory Method	Builder	Prototype	Singleton
Project_01	1	0.33	0	0	0
Project_02	1	0.33	0	0	0
Project_03	1	0.33	0	0	0
Project_04	1	0.33	0	0	0
Project_05	0.33	1	0	0	0
Project_06	0.33	1	0	0	0
Project_07	0.67	1	0	0	0
Project_08	0.67	1	0	0	0
Project_09	0	0	1	0	0
Project_10	0.33	0	1	0	0
Project_11	0.67	0.33	1	0	0
Project_12	0	0	1	0	0
Project_13	0	0	1	0	0
Project_14	0	0	0	1	0
Project_15	0	0	0	1	0
Project_16	0	0	0	1	0
Project_17	0	0	0	1	0
Project_18	0	0	0	0	1
Project_19	0	0	0	0	0.5
Project_20	0	0	0	0	1
Project_21	0	0	0	0	1

behavioral matching for these projects. On the other hand, Project_10 also has a score of 0.33 for Abstract Factory, where the matching level is structural matching (Table 7.2). However, from these scores no significance of the levels can be understood, as both the scores are same for different levels. Thus, these weighting factor values cannot express the difference between design pattern scores, calculated using distinct levels of matching.

Table 7.4: Scores of the Design Patterns with $\alpha = 1, \beta = 2$ and $\gamma = 3$

	Abstract Factory	Factory Method	Builder	Prototype	Singleton
Project_01	1	0.33	0	0	0
Project_02	1	0.33	0	0	0
Project_03	1	0.33	0	0	0
Project_04	1	0.33	0	0	0
Project_05	0.33	1	0	0	0
Project_06	0.33	1	0	0	0
Project_07	0.5	1	0	0	0
Project_08	0.5	1	0	0	0
Project_09	0	0	1	0	0
Project_10	0.17	0	1	0	0
Project_11	0.5	0.33	1	0	0
Project_12	0	0	1	0	0
Project_13	0	0	1	0	0
Project_14	0	0	0	1	0
Project_15	0	0	0	1	0
Project_16	0	0	0	1	0
Project_17	0	0	0	1	0
Project_18	0	0	0	0	1
Project_19	0	0	0	0	0.33
Project_20	0	0	0	0	1
Project_21	0	0	0	0	1

7.2.1.2 Scores with $\alpha = 1, \beta = 2$ and $\gamma = 3$

As the score calculation with $\alpha = 1, \beta = 1$ and $\gamma = 1$ does not uphold matching level significance, and $\alpha < \beta < \gamma$ (Section 5.3, Chapter 5), new α, β and γ values are needed to be determined. Thus, in this section, the matching values of the attributes from Table 7.2 are used in Equation 5.2 with $\alpha = 1$, $\beta = 2$ and $\gamma = 3$. The scores are shown in Table 7.4.

It can be seen that, the level significance controls the score value. However, the score values of different design patterns still raise some issues. For

example, Project_19 has score of 0.33 for Singleton pattern. Other projects like Project_05, Project_06, etc. also have 0.33 for Abstract Factory, and Project_01 and Project_02 for Factory Method. However, the 0.33 for Singleton and 0.33 for Abstract Factory and Factory have different significance. While Singleton score is calculated using two levels, these other two patterns have three levels of matching. Yet the scores are same for two levels of matchings of Abstract Factory, Factory Method (structural and behavioral, as depicted in Table 7.2), and one level of matching of Singleton (structural as seen in Table 7.2). Thus, the number of levels does not play any significance using these weighting factor values.

7.2.1.3 Scores with $\alpha = 1, \beta = 2$ and $\gamma = 4$

For considering both the number of levels and the levels' significance, the final experimentation is done with the selected weights in Section 5.3, Chapter 5. The weight values are, $\alpha = 1, \beta = 2$ and $\gamma = 4$. These scores of the design patterns for each project is shown in Table 7.5.

Here, both number of levels and the levels' significance have their impacts on the score. The issue with $\alpha = 1, \beta = 1$ and $\gamma = 1$ was that, for different levels of match the scores were the same. In this new value assignment ($\alpha = 1, \beta = 2$ and $\gamma = 4$), this problem is solved. For Example, score of Project_01, Project_02, Project_03 and Project_04 is 0.29 for behavioral match. On the other hand, Project_10 has a score of 0.14 for structural matching. Thus, the significance of different matching levels are preserved by these weighting factor values.

Table 7.5: Scores of the Design Patterns for $\alpha = 1, \beta = 2$ and $\gamma = 4$

	Abstract Factory	Factory Method	Builder	Prototype	Singleton
Project_01	1	0.29	0	0	0
Project_02	1	0.29	0	0	0
Project_03	1	0.29	0	0	0
Project_04	1	0.29	0	0	0
Project_05	0.29	1	0	0	0
Project_06	0.29	1	0	0	0
Project_07	0.43	1	0	0	0
Project_08	0.43	1	0	0	0
Project_09	0	0	1	0	0
Project_10	0.14	0	1	0	0
Project_11	0.43	0.29	1	0	0
Project_12	0	0	1	0	0
Project_13	0	0	1	0	0
Project_14	0	0	0	1	0
Project_15	0	0	0	1	0
Project_16	0	0	0	1	0
Project_17	0	0	0	1	0
Project_18	0	0	0	0	1
Project_19	0	0	0	0	0.33
Project_20	0	0	0	0	1
Project_21	0	0	0	0	1

Also, the issue with $\alpha = 1, \beta = 2$ and $\gamma = 3$ is also considered here. The number of levels did not play any significance in that weight assignment. However, for $\alpha = 1, \beta = 2$ and $\gamma = 4$, the number of levels contribute in the scoring mechanism. For example, in Project_01, the score is 0.29 for one (behavioral) of the three levels of matching. On the other hand, in Project_19, the score is 0.33 for one (behavioral) of the two levels (behavioral and semantic) of matching. So, the difference of scores for different number of levels is explicit.

Thus, $\alpha = 1, \beta = 2$ and $\gamma = 4$ can be selected as the preferable weight values for ACDPR.

7.2.2 Recommendation of Design Patterns

For a score value of 1, the design pattern is recommended to be applied in the project, as all the characteristics of the *missing* design pattern could be detected from the project design diagrams. Table 7.6 shows the recommended design patterns along with the expected recommendations for the projects. The expected recommendations were identified from manual analysis of the projects following the design pattern application requirements of GoF. Here, as anti-patterns of Builder, Prototype and Singleton patterns are concerned with individual classes, the class names are also mentioned along with the recommendations of these patterns (Table 7.6).

Now, from the given recommendations, the precision and recall of ACDPR can be measured. The quality of a recommendation system is typically described using these precision and recall metrics [67].

Precision

Precision basically measures how well the recommender filters out the irrelevant results. It is the fraction of the returned relevant results in the overall result set. Thus, the fraction of true positive results in total returned results (true positive + false positive) is the precision.

Table 7.6: Recommendations and Suggestions

	Expected Recommendation	Recommendation
Project_01	Abstract Factory	Abstract Factory
Project_02	Abstract Factory	Abstract Factory
Project_03	Abstract Factory	Abstract Factory
Project_04	Abstract Factory	Abstract Factory
Project_05	Factory Method	Factory Method
Project_06	Factory Method	Factory Method
Project_07	Factory Method	Factory Method
Project_08	Factory Method	Factory Method
Project_09	Builder	Builder (for class Car)
Project_10	Builder	Builder (for class Housebuilder)
Project_11	Builder	Builder (for class PromoKit)
Project_12	Builder	Builder (for class FoodTelescopingDemo)
Project_13	Builder	Builder (for class Client)
Project_14	Prototype	Prototype (for class Analysis)
Project_15	Prototype	Prototype (for class Wheel, Sit, Door)
Project_16	Prototype	Prototype (for class Tree, House)
Project_17	Prototype	Prototype (for class Square)
Project_18	Singleton	Prototype (for class EnglishWriter,BengaliWrite)
Project_19	Singleton	-
Project_20	Singleton	Singleton (for class Logger)
Project_21	Singleton	Singleton (for class PdfReader)

Let, tp = true positive, fp = false positive, fn = false negative. From Table 7.6, $tp = 20$, $fp = 0$, $fn = 1$. Thus,

$$Precision = \frac{tp}{tp + fp} = \frac{20}{20 + 0} = 1$$

As, ACDPR provides no false positive recommendation, it possesses the maximum precision of 1.

Recall

Recall measures how well the recommender finds relevant results. It is the fraction of the relevant returned results in the overall collection of relevant results. Thus, recall takes the fraction of the true positive results with the total relevant results (true positive + false negative).

$$Recall = \frac{tp}{tp + fn} = \frac{20}{20 + 1} = 0.95$$

Thus, the possibility to identify the required design patterns using ACDPR, is 95%.

Balanced F-score

F-measure is a single measure that trades off precision versus recall. It is the weighted harmonic mean of the precision and recall. If precision and

recall are equally weighted, balanced F-score (F_1 score) is found which is the default F-measure. F_1 score or F-measure is calculated as the accuracy of the recommendation.

Using the calculated precision and recall, F_1 score can be calculated for ACDPR.

$$F_1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} = 2 \cdot \frac{1 \cdot 0.95}{1 + 0.95} = 0.97$$

So, the accuracy of ACDPR recommendation for the sample dataset, is pretty good, that is, 0.97.

7.2.3 Determining Suggestion Threshold

For a score < 1, the design patterns are suggested to be considered by the designers. This suggestions are given based on a threshold value which is selected to be 0.3, as justified in Section 5.4. Yet, in this section, different threshold values are compared to check its compatibility with the suggestions.

Table 7.7 and Table 7.8 shows the suggestions of patterns with different threshold values. These threshold values are decided from the possible partial scores of design patterns –

- 0.14: match structure only, when all three levels of matching are present in the *missing* pattern detection –

$$\frac{\alpha \cdot str_i + \beta \cdot beh_i + \gamma \cdot sem_i}{\alpha + \beta + \gamma} = \frac{1 \cdot 1 + 2 \cdot 0 + 4 \cdot 0}{1 + 2 + 4} = 0.14$$

- 0.29: match behavior only, when all three levels of matching are present

  –

$$\frac{\alpha \cdot str_i + \beta \cdot beh_i + \gamma \cdot sem_i}{\alpha + \beta + \gamma} = \frac{1 \cdot 0 + 2 \cdot 1 + 4 \cdot 0}{1 + 2 + 4} = 0.29$$

- 0.33: match structure, when two levels of matching (that is, structure and behavior) are present; or match behavior, when behavior and semantic is present.

$$\frac{\alpha \cdot str_i + \beta \cdot beh_i + \gamma \cdot sem_i}{\alpha + \beta + \gamma} = \frac{1 \cdot 1 + 2 \cdot 0 + 0 \cdot 0}{1 + 2 + 0} = \frac{0 \cdot 0 + 2 \cdot 1 + 4 \cdot 0}{0 + 2 + 4} = 0.33$$

- 0.43: match structure and behavior, when all three levels of matching are present

$$\frac{\alpha \cdot str_i + \beta \cdot beh_i + \gamma \cdot sem_i}{\alpha + \beta + \gamma} = \frac{1 \cdot 1 + 2 \cdot 1 + 4 \cdot 0}{1 + 2 + 4} = 0.43$$

- 0.66: match behavior, when two levels of matching (that is, structure and behavior) are present.

$$\frac{\alpha \cdot str_i + \beta \cdot beh_i + \gamma \cdot sem_i}{\alpha + \beta + \gamma} = \frac{1 \cdot 0 + 2 \cdot 1 + 0 \cdot 0}{1 + 2 + 0} = 0.66$$

Matching only semantic is not considered here, as, if behavior does not match, semantic will not be matched.

As already stated in Section 5.4 Chapter 5, recommendations are given for the score value of 1, and suggestions are given for a value greater than

Table 7.7: Suggestions with Different Threshold Values–1 $(0.14, 0.29, 0.33)$

	Suggestion with threshold 0.14	Suggestion with threshold 0.29	Suggestion with threshold 0.33
Project_01	Factory Method	Factory Method	-
Project_02	Factory Method	Factory Method	-
Project_03	Factory Method	Factory Method	-
Project_04	Factory Method	Factory Method	-
Project_05	Abstract Factory	Abstract Factory	-
Project_06	Abstract Factory	Abstract Factory	-
Project_07	Abstract Factory	Abstract Factory	Abstract Factory
Project_08	Abstract Factory	Abstract Factory	Abstract Factory
Project_09	-	-	-
Project_10	Abstract Factory	-	-
Project_11	Abstract Factory, Factory Method	Abstract Factory	Abstract Factory
Project_12	-	-	-
Project_13	-	-	-
Project_14	-	-	-
Project_15	-	-	-
Project_16	-	-	-
Project_17	-	-	-
Project_18	-	-	-
Project_19	Singleton	Singleton	Singleton
Project_20	-	-	-
Project_21	-	-	-

0.3 (0.33). According to this threshold value, for the failed case of recommendation (Project_19), suggestion is given to the designers to consider Singleton which was the expected recommendation. The other suggestions (Project_07, Project_08 and Project_11) have been given for Abstract Factory, which have a partial matching score of 0.43 (actual recommendations are Factory Method, Factory Method and Builder respectively as shown in the second row in Table 7.6). Although these three suggestions are not applicable for the projects, these might be helpful for the designers. This is

Table 7.8: Suggestions with Different Threshold Values–2 $(0.43, 0, 66)$

	Suggestion with threshold 0.43	Suggestion with threshold 0.66
Project_01	-	-
Project_02	-	-
Project_03	-	-
Project_04	-	-
Project_05	-	-
Project_06	-	-
Project_07	Abstract Factory	-
Project_08	Abstract Factory	-
Project_09	-	-
Project_10	-	-
Project_11	Abstract Factory	-
Project_12	-	-
Project_13	-	-
Project_14	-	-
Project_15	-	-
Project_16	-	-
Project_17	-	-
Project_18	-	-
Project_19	-	-
Project_20	-	-
Project_21	-	-

because, Factory Method can lead to Abstract Factory, as these two patterns are interrelated (referring to GoF Related Patterns-Abstract Factory/Factory Method in [18]). Similarly, Builder and Abstract Factory are also similar to some extent (referring to GoF Related Patterns-Builder in [18]).

On the other hand, for a threshold value of 0.14 and 0.29, for even one level of successful matching (structural and behavioral respectively), Abstract Factory and Factory Method are suggested. This can lead to designers' confusion about using these patterns. Now, for threshold value of 0.43,

the required suggestion of Singleton for Project_19 has not been provided. And for threshold value, 0.66, no suggestion is provided.

From this analysis, it can be seen that the most effective suggestions are acquired from threshold value 0.33 (≈ 0.3). Thus, 0.3 is the most appropriate threshold for ACDPR.

7.3 Summary

This chapter intends to demonstrate the implementation and result analysis of ACDPR. A prototype of ACDPR is developed using Java. The prototype results are analyzed to test the recommendation accuracy. The precision, recall, and F-measure are measured for the sample dataset which are 1, 0.95, and 0.97 respectively, for weighing factors $\alpha = 1, \beta = 2$ and $\gamma = 4$. Also, for the false positive results in the recommendation (missed recall 0.05), the patterns were suggested, for a threshold value of 0.3. The next chapter concludes this thesis after providing a proper future direction.

Chapter 8

Conclusion

Unlike the conventional design pattern recommendation approaches, this research proposes to use anti-pattern detection in the software design for selecting the patterns to be recommended. For this, identification of the *missing* creational patterns and their characterization are done. A framework named ACDPR is proposed for recommending the creational design patterns. ACDPR uses initial software design for detecting the *missing* patterns through structural, behavioral and semantic matching. Outcomes of the matching levels are used to calculate design patterns' score and recommend the patterns. In this chapter the document is concluded by a profound but brief discussion of the research along with the future direction.

8.1 Discussion

This research introduces an approach to recommend creational design patterns using anti-patterns in the software design phase. A tool is proposed

named ACDPR where anti-pattern detection is utilized for recommendation of appropriate design patterns in the software design phase.

Without defining anti-pattern characteristics it is not possible to detect the anti-patterns in software. Hence this research first derives the characteristics of the *missing* creational patterns (that is, anti-patterns of creational pattern) through structural, behavioral and semantic analysis. After the definition a tool is proposed named ACDPR, where those anti-patterns are detected in a software design by structural, behavioral and semantic matchings. These three levels of matching are required for identifying the three mentioned characteristics in software design. The matched levels are used to calculate scores of the design patterns. Then the patterns are recommended for a complete matching (that is, score of 1), as it expresses the existence of the *missing* pattern in the software design. For partial matchings design patterns are also suggested for designers' further consideration based on a threshold value. This suggestions increase the probability of getting appropriate recommendations even for incomplete design diagrams.

A case study on a sample project design evaluated the applicability of the approach. Also experiments have been conducted on software projects containing anti-patterns of the five creational design patterns. For this purpose a prototype of ACDPR was implemented in Java. Twenty-one projects requiring different creational patterns were used as dataset. The precision, recall and F-measure of the recommender were calculated. ACDPR possesses a precision of 1, recall of 0.95 and F-measure of 0.97 for its recommendation decisions on this dataset. The reason behind this high accuracy is that, ACDPR completely incorporates the anti-pattern characteristics be-

fore matching, reducing the probability of false matchings (that is, false negatives). Also for the false positive results in the recommendation (missed recall 0.05), the required patterns were suggested to the designers. These patterns were accurately suggested for a threshold value of 0.3.

8.2 Threats to Validity

An *internal validity* threat of this research is that ACDPR requires proper design diagrams that reflect the software accurately. As design diagrams are the only means from which the anti-patterns can be identified before the coding phase, these need to reveal those anti-patterns by upholding the true characteristics of the software design. For example, the class diagrams should have properly defined class types (through super-class) in case of Abstract Factory and Factory Method. For Builder the existence of multiple constructors in a class is needed to be mentioned in the class diagrams along with the parameter lists. In Singleton the activity diagrams are needed to be related to the class names to some extent, for identifying the singleton class name in the semantic matching step. Small deviations from the ideal cases have been handled by the tool. For example, measuring similarity in class names for identifying classes of same type in absence of super-classes, use of *WordNet* on terms of activity diagrams, etc. Yet for getting correct recommendations proper design diagrams should be provided as input.

An *external validity* threat is that ACDPR has been applied on in-house classroom projects. Experimentation on industrial projects could not be performed due to the unavailability of design documents.

Regarding *reliability validity* the experimental projects have been uploaded on GitHub [66], and thus are publicly available on the Internet.

8.3 Future Work

In this thesis the characteristics of the *missing* creational patterns are analyzed only. The future direction related to this research lies in –

- Analyzing the other *missing* design patterns (those are, the structural and behavioral patterns), and extending the tool to recommend those patterns also.
- Generalizing the research to recommend patterns in different areas like Enterprise Architecture (EA) patterns [27], security patterns [28], Service-Oriented Architecture (SOA) patterns [29], etc.

The idea of incorporating anti-pattern detection in design pattern recommendation opens a number of directions for future research –

- Anti-pattern detection and removal researches [7, 45] detect existence of anti-patterns in software and remove those. The proposed anti-pattern detection technique can be used in this field, for better detection. Usage of the recommended design patterns can also be performed for removing the anti-patterns.
- Design pattern instantiation [38, 39] applies the required design patterns in the software design. Design pattern recommendation is quite

related to this field. As this research successfully recommends the required design patterns for software, those can be automatically applied in the design.

- Design pattern detection [21, 36] is another established research field for detecting the existence of design patterns in software. The proposed mechanism of anti-pattern detection can be used in design pattern detection.

- Similar to the relation of anti- and design patterns, relationships between anti-patterns and code smells can also be discovered [68]. It can play a significant role in anti-pattern detection leading to code smell refactoring [42, 44].

Appendix A

Abstract Factory Anti-pattern Structures

The followings are the eight identified structures of the Abstract Factory anti-pattern. These anti-pattern structures are stored in ACDPR for detection.

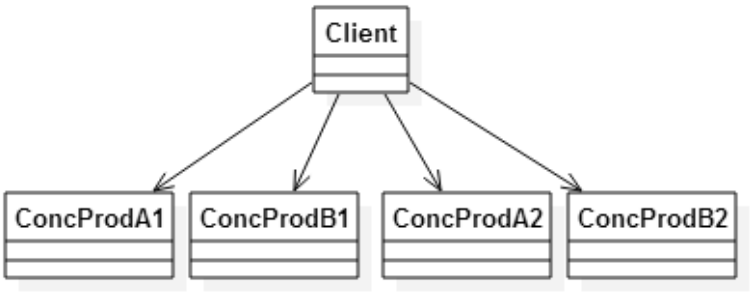


Figure A.1: Abstract Factory Anti-pattern Structure A

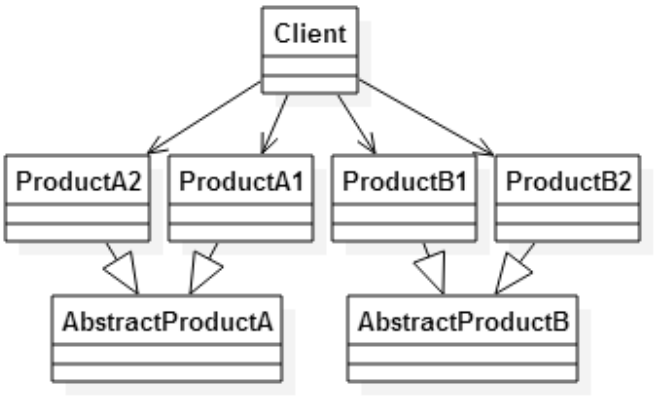


Figure A.2: Abstract Factory Anti-pattern Structure B

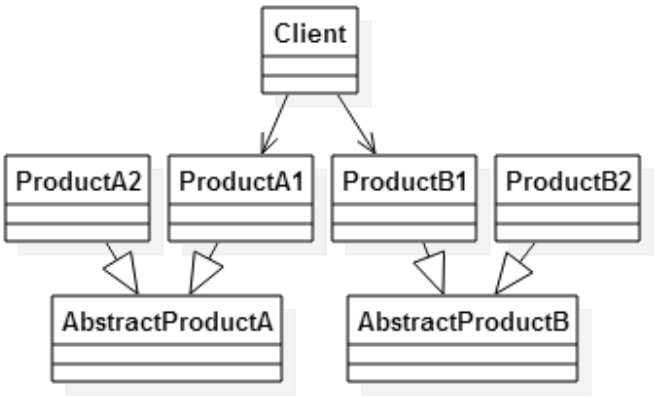


Figure A.3: Abstract Factory Anti-pattern Structure C

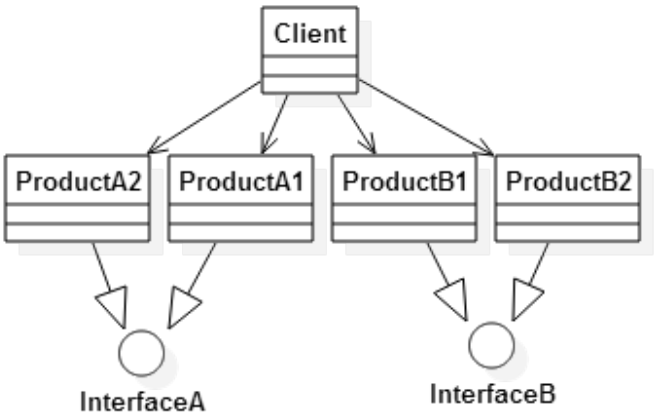


Figure A.4: Abstract Factory Anti-pattern Structure D

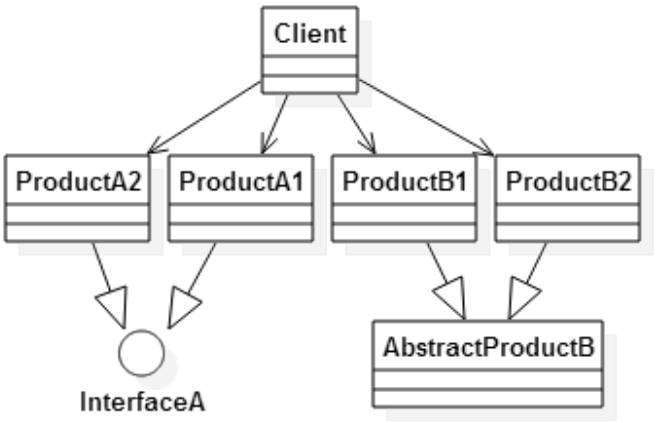


Figure A.5: Abstract Factory Anti-pattern Structure E

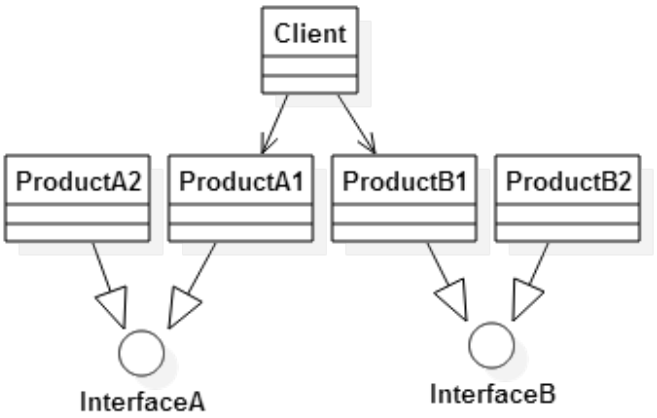


Figure A.6: Abstract Factory Anti-pattern Structure F

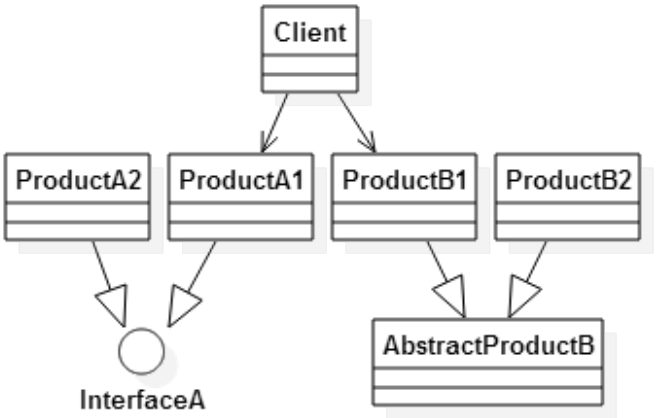


Figure A.7: Abstract Factory Anti-pattern Structure G

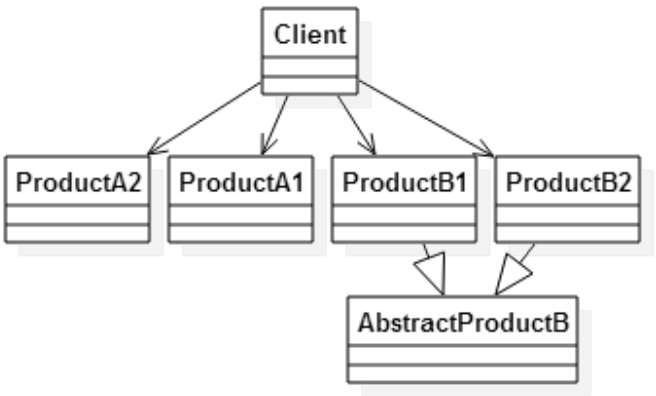


Figure A.8: Abstract Factory Anti-pattern Structure H

Appendix B

Factory Method Anti-pattern Structures

The followings are the identified structures of the Factory Method anti-pattern. These anti-pattern structures are stored in ACDPR for detection.

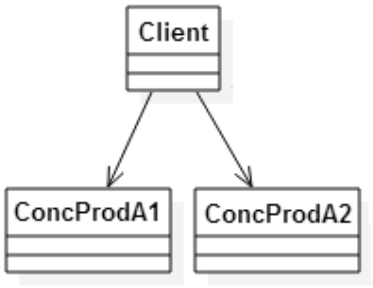


Figure B.1: Factory Method Anti-pattern Structure A

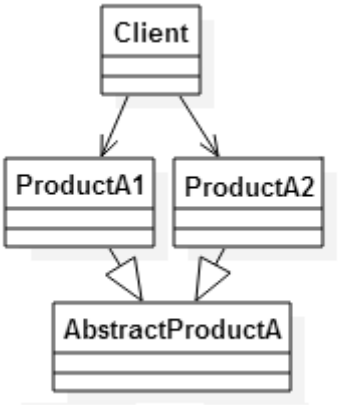


Figure B.2: Factory Method Anti-pattern Structure B

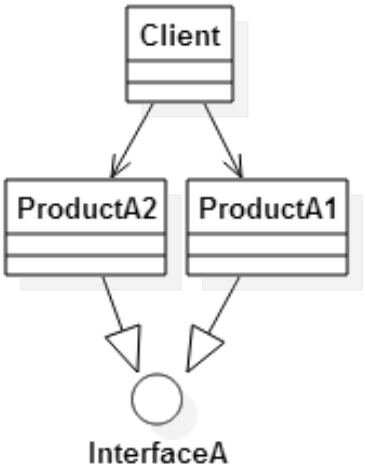


Figure B.3: Factory Method Anti-pattern Structure C

List of Publications

1. "Automatic Recommendation of Software Design Patterns Using Anti-patterns in the Design Phase: A Case Study on Abstract Factory," *in Proceedings of the 3rd International Workshop on Quantitative Approaches to Software Quality co-located with the 22nd Asia-Pacific Software Engineering Conference (APSEC)*, pp. 9-16, New Delhi, India, December 1, 2015.

2. "ACDPR: A Recommendation System for the Creational Software Design Patterns Using Anti-patterns in the Design Phase," *In Proceedings of the 3rd International Workshop on Patterns Promotion and Anti-patterns Prevention (PPAP) co-located with the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Osaka, Japan, March 15, 2016.

3. "An Improved Behavioral Matching for Anti-pattern Based Abstract Factory Recommendation," *In Proceedings of the International Conference on Electronics, Information and Vision (ICIEV)*, Dhaka, Bangladesh, 2016.

---

* These papers are produced from the two contributing chapters, Chapter 4 and Chapter 5

Bibliography

[1] N. Bautista, "A Beginners Guide to Design Patterns." http://code.tutsplus.com/articles/a-beginners-guide-to-design-patterns--net-12752. Accessed: 2015-01-01.

[2] R. Fourati, N. Bouassida, and H. B. Abdallah, "A Metric-Based Approach for Anti-pattern Detection in UML Designs," *Studies in Computational Intelligence, Springer Berlin Heidelberg*, vol. 364, pp. 17–33, 2011.

[3] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y.-G. Guéhéneuc, G. Antoniol, and E. Aïmeur, "Support Vector Machines for Anti-pattern Detection," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 278–281, 2012.

[4] T. Feng, J. Zhang, H. Wang, and X. Wang, "Software Design Improvement through Anti-patterns Identification," in *Proceedings of the 20th IEEE International Conference on Software Maintenance*, p. 524, IEEE, 2004.

[5] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y.-G. Guéhéneuc, and E. Aimeur, "SMURF: A SVM-based Incremental Anti-pattern Detection Approach," in *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE)*, pp. 466–475, IEEE, 2012.

[6] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "Decor: A Method for the Specification and Detection of Code and Design Smells," *IEEE Transactions on Software Engineering, IEEE*, vol. 36, no. 1, pp. 20–36, 2010.

[7] V. Cortellessa, A. Di Marco, R. Eramo, A. Pierantonio, and C. Trubiani, "Digging into UML Models to Remove Performance Antipatterns," in *Proceedings of the 32th ICSE Workshop on Quantitative Stochastic Models in the Verification and Design of Software Systems*, pp. 9–16, ACM, 2010.

[8] S. Smith and D. R. Plante, "Dynamically Recommending Design Patterns," in *Proceedings of the 24th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pp. 499–504, 2012.

[9] Y.-G. Guéhéneuc and R. Mustapha, "A Simple Recommender System for Design Patterns," in *Proceedings of the 1st EuroPLoP Focus Group on Pattern Repositories*, 2007.

[10] S. Suresh, M. Naidu, S. A. Kiran, and P. Tathawade, "Design Pattern Recommendation System: a Methodology, Data Model and Algorithms," in *Proceedings of the International Conference on Computational Techniques and Artificial Intelligence (ICCTAI)*, 2011.

[11] S. M. H. Hasheminejad and S. Jalili, "Design Patterns Selection: An Automatic Two-phase Method," *Journal of Systems and Software, Elsevier*, vol. 85, no. 2, pp. 408–424, 2012.

[12] F. Palma, H. Farzin, Y.-G. Guéhéneuc, and N. Moha, "Recommendation System for Design Patterns in Software Development: An DPR Overview," in *Proceedings of the 3rd International Workshop on Recommendation Systems for Software Engineering*, pp. 1–5, IEEE, 2012.

[13] L. Pavlič, V. Podgorelec, and M. Heričko, "A Question-based Design Pattern Advisement Approach," *Computer Science and Information Systems*, vol. 11, no. 2, pp. 645–664, 2014.

[14] P. Gomes, F. C. Pereira, P. Paiva, N. Seco, P. Carreiro, J. L. Ferreira, and C. Bento, "Using CBR for Automation of Software Design Patterns," *Advances in Case-Based Reasoning, Springer Berlin Heidelberg*, vol. 2416, pp. 534–548, 2002.

[15] W. Muangon and S. Intakosum, "Case-based Reasoning for Design Patterns Searching System," *International Journal of Computer Applications*, vol. 70, no. 26, pp. 16–24, 2013.

[16] E. M. Sahly and O. M. Sallabi, "Design Pattern Selection: A Solution Strategy Method," in *Proceedings of the 2012 International Conference on Computer Systems and Industrial Informatics (ICCSII)*, pp. 1–6, IEEE, 2012.

[17] I. Navarro, P. Díaz, and A. Malizia, "A Recommendation System to Support Design Patterns Selection," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 269–270, IEEE, 2010.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994.

[19] J. Woodcock and J. Davies, "Using Z. Specification, Refinement, and Proof," 1996.

[20] R. S. Pressman, *Software Engineering: A Practitioner's Approach*. Palgrave Macmillan, 2005.

[21] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design Pattern Detection Using Similarity Scoring," *IEEE Transactions on Software Engineering*, vol. 32, no. 11, pp. 896–909, 2006.

[22] F. Bergenti and A. Poggi, "Improving UML Designs Using Automatic Design Pattern Detection," in *Proceedings of the 12th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pp. 336–343, 2000.

[23] J. Hannemann and G. Kiczales, "Design Pattern Implementation in Java and AspectJ," in *ACM Sigplan Notices*, vol. 37, pp. 161–173, ACM, 2002.

[24] G. El Boussaidi and H. Mili, "A MNodel-driven Framework for Representing and Applying Design Patterns," in *Proceedings of the 31st Annual International on Computer Software and Applications Conference, (COMPSAC)*, vol. 1, pp. 97–100, IEEE, 2007.

[25] H. Mili and G. El-Boussaidi, "Representing and Applying Design Patterns: What Is the Problem?," in *Model Driven Engineering Languages and Systems*, pp. 186–200, Springer, 2005.

[26] D. Heuzeroth, T. Holl, G. Högström, and W. Löwe, "Automatic Design Pattern Detection," in *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, pp. 94–103, IEEE, 2003.

[27] M. Fowler, *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[28] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad, *Security Patterns: Integrating security and systems engineering*. John Wiley & Sons, 2013.

[29] M. Endrei, J. Ang, A. Arsanjani, S. Chua, P. Comte, P. Krogdahl, M. Luo, and T. Newling, *Patterns: service-oriented architecture and*

web services. IBM Corporation, International Technical Support Organization, 2004.

[30] W. J. Brown, H. W. McCormick, T. J. Mowbray, and R. C. Malveau, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley New York, 1998.

[31] "Anti Patterns Catalog." http://c2.com/cgi/wiki?AntiPatternsCatalog. Accessed: 2015-02-06.

[32] C. Jebelean, "Automatic Detection of Missing Abstract-Factory Design Pattern in Object-Oriented Code," in *Proceedings of the International Conference on Technical Informatics*, 2004.

[33] W. J. Brown, H. W. McCormick III, S. H. Thomas, *et al.*, *AntiPatterns and Patterns in Software Configuration Management*. John Wiley & Sons, Inc., 1999.

[34] W. J. Brown, H. W. McCormick, and S. W. Thomas, *Anti-Patterns Project Management*. John Wiley & Sons, Inc., 2000.

[35] J. Dong, D. S. Lad, and Y. Zhao, "DP-Miner: Design Pattern Discovery Using Matrix," in *Proceedings of the 14th Annual IEEE International Conference and Workshops on Engineering of Computer-Based Systems (ECBS)*, pp. 371–380, IEEE, 2007.

[36] A. De Lucia, V. Deufemia, C. Gravino, and M. Risi, "An Eclipse Plug-in for the Detection of Design Pattern Instances through Static and Dynamic Analysis," in *Proceesdings of the 26th IEEE International Conference on Software Maintenance (ICSM)*, pp. 1–6, IEEE, 2010.

[37] S. Alhusain, S. Coupland, R. John, and M. Kavanagh, "Design Pattern Recognition by Using Adaptive Neuro Fuzzy Inference System," in *Proceesdings of the IEEE 25th International Conference on Tools with Artificial Intelligence (ICTAI)*, pp. 581–587, IEEE, 2013.

[38] H. Albin-Amio, P. Cointe, Y.-G. Guéhéneuc, and N. Jussien, "Instantiating and Detecting Design Patterns: Putting Bits and Pieces Together," in *Proceedings of the 16th Annual International Conference on Automated Software Engineering (ASE)*, pp. 166–173, IEEE, 2001.

[39] D. Mapelsden, J. Hosking, and J. Grundy, "Design Pattern Modelling and Instantiation Using DPML," in *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*, pp. 3–11, Australian Computer Society, Inc., 2002.

[40] J. Dong, Y. Sun, and Y. Zhao, "Design Pattern Detection by Template Matching," in *Proceedings of the 2008 ACM symposium on Applied computing*, pp. 765–769, ACM, 2008.

[41] K. Brown, "Design Reverse-engineering and Automated Design-pattern Detection in Smalltalk," tech. rep., North Carolina State University at Raleigh, January 1996.

[42] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Pearson Education India, 2009.

[43] E. Van Emden and L. Moonen, "Java Quality Assurance by Detecting Code Smells," in *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE)*, pp. 97–106, IEEE, 2002.

[44] T. Mens and T. Tourwé, "A Survey of Software Refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[45] C. Trubiani and A. Koziolek, "Detection and Solution of Software Performance Antipatterns in Palladio Architectural Models," *ACM SIGSOFT Software Engineering Notes*, vol. 36, no. 5, pp. 19–30, 2011.

[46] R. L. de Mantaras, "Case-based Reasoning," *Machine Learning and Its Applications*, pp. 127–145, 2001.

[47] H. Kampffmeyer and S. Zschaler, "Finding the Pattern You Need: The Design Pattern Intent Ontology," *Model Driven Engineering Languages and Systems, Springer Berlin Heidelberg*, vol. 4735, pp. 211–225, 2007.

[48] G. Salton, A. Wong, and C.-S. Yang, "A Vector Space Model for Automatic Indexing," *Communications of the ACM*, vol. 18, no. 11, pp. 613–620, 1975.

[49] P. University, "WordNet, A Lexical Database for English." https://wordnet.princeton.edu/. Accessed: 2015-08-15.

[50] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[51] J. M. Bieman and B.-K. Kang, "Cohesion and Reuse in an Object-Oriented System," *ACM SIGSOFT Software Engineering Notes*, vol. 20, no. SI, pp. 259–262, 1995.

[52] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[53] N. Nahar and K. Sakib, "Automatic Recommendation of Software Design Patterns Using Anti-patterns in the Design Phase: A Case Study on Abstract Factory," in *Proceedings of the 3rd International Workshop on Quantitative Approaches to Software Quality (QuASoQ)*, p. 11, 2015.

[54] S. Richardson, "Dynamically Recommending Design Patterns," bachelor's thesis, Stetson University, 2011.

[55] A. Jarvi, "Abstract Factory: 2005." http://staff.cs.utu.fi/kurssit/Programming-III/AbstractFactory(10).pdf. Accessed: 2015-01-03.

[56] N. Nahar, "NadiaIT/ACDPRAntiPatterns: 2015." https://github.com/NadiaIT/ACDPRAntiPatterns. Accessed: 2015-11-10.

[57] J. Hughes, "The Builder Pattern." https://yobriefca.se/blog/2015/05/13/the-builder-pattern/, May 13, 2015. Accessed: 2015-10-13.

[58] "Prototype Pattern." http://www.oodesign.com/prototype-pattern.html. Accessed: 2015-10-15.

[59] H. Zhu and I. Bayley, "An Algebra of Design Patterns," *ACM Transactions on Software Engineering and Methodology (TOSEM), ACM*, vol. 22, no. 3, p. 23, 2013.

[60] J. Moore, "How would you use UML to model multiple relationships between the same two classes?." http://www.jguru.com/faq/view.jsp?EID=124085, May 14, 2012. Accessed: 2016-04-12.

[61] R. Perera, "The Basics & the Purpose of Sequence Diagrams - Part 1." http://creately.com/blog/diagrams/the-basics-the-purpose-of-sequence-diagrams-part-1/. Accessed: 2015-03-02.

[62] "Design Pattern - Abstract Factory Pattern." http://www.tutorialspoint.com/design_pattern/abstract_factory_pattern.htm. Accessed: 2015-10-19.

[63] "Eclipse Luna." http://eclipse.org/luna/. Accessed: 2015-03-02.

[64] "StarUML." http://staruml.io/. Accessed: 2015-03-02.

[65] N. Nahar, "NadiaIT/ACDPR: Anti-pattern based Creational Design Pattern Recomender." https://github.com/NadiaIT/ACDPR. Accessed: 2016-04-19.

[66] N. Nahar, "NadiaIT/ACDPR-dataset: 2015." https://github.com/NadiaIT/ACDPR-dataset. Accessed: 2015-11-10.

[67] C. D. Manning, P. Raghavan, H. Schütze, *et al.*, *An Introduction to Information Retrieval*, vol. 1. Cambridge University Press, Cambridge, 2008.

[68] Y. Luo, A. Hoss, and D. L. Carver, "An Ontological Identification of Relationships between Anti-patterns and Code Smells," in *Proceedings of the 2010 Aerospace Conference*, pp. 1–10, IEEE, 2010.