Verification of Architectural Rules and Design Patterns

Shravan Shetty and Vinod Menezes

August 23, 2011

Master Thesis
Eindhoven University of Technology
Department of Mathematics and Computer Science

In cooperation with:
Philips Healthcare
Best, The Netherlands

Supervisor at TU/e:
Prof. Dr. M. v. d. Brand
Professor and Chair of SET Group
Department of CS & E, TU/e, Eindhoven
M.G.J.v.d.Brand@tue.nl

Supervisors at Philips Healthcare:
Kasper van Wouw
Software Architect, iXR Department
kasper.van.wouw@philips.com

Supervisor at MIT:
Dr. Preetham Kumar
Associate Professor
Department of I & CT, MIT, Manipal
preetham.kumar@manipal.edu

Albert Faber
Software Architect, iXR Department
albert.faber@philips.com
Acknowledgements

This master thesis is carried out in the Software Engineering Technology (SET) group at Technische Universiteit Eindhoven (TU/e), in cooperation with Philips Healthcare, Best, the Netherlands. A Master in Science degree will be honored to us upon the successful completion of this thesis. We would like to take this opportunity to thank all who has directly or indirectly supported us.

We would like to convey our sincere thanks to Prof. Dr. Mark van den Brand, our supervisor at TU/e for his immense support and guidance throughout our work.

We are grateful for Dr. Manohara. Pai. M. M for providing us with an opportunity to pursue our study at the TU/e. We also thank Dr. Preetham Kumar for being our supervisor at Manipal Institute of Technology (MIT).

We are grateful to Dr. Alexander Serebrenik Asst. Professor TU/e, Dr. Serguei Roubtsov and Dr. Tom Verhoeff for their valuable comments and suggestion regarding our thesis.

Our sincere thanks to Albert Faber and Kasper van Wouw, Software Architects at Philips Healthcare for giving us the opportunity to perform this research at Philips Healthcare and guiding us in our work. We would extend our gratitude to all folks at Philips Healthcare for a friendly working environment.

Our sincere thanks to Alex Skorkin, Developer at DevExpress, for his support and valuable inputs regarding implementation in DXCore.

We would also like to thank our friend Ajith Kumar for supporting us through out the thesis. We are grateful to our family for their support and encouragement when it was most needed.
Abstract

Software design is the process of translating specifications into an effective software solution. This problem solving approach will provide a foundation from which concrete functionality can be developed and tested. It follows that, design issues have direct implications on the implementation and testability of the software. A recent study by US National Institute of Standards and Technology claim an astounding $60 billion annual loss owing to software defects. Software defects are mainly functional, so they can be detected by employing effective testing techniques. However, software design flaws are not evident immediately, but will be obvious as the software degrades with subsequent updates. An example of a design flaw is the inability of the current software design to adapt a new requirement. In the worst case, it might even require redesigning the entire software again. This makes design issues more dangerous than functional defects. This demands the need of automatic verification techniques to curb maintenance costs. Verification tools must be able to detect design violations at the source, and not let it propagate into the subsequent development phases.

A design pattern provides an abstract reusable solution for a commonly occurring design problem. Improper implementation of the design patterns might lead to degradation of the software quality and maintainability. Design pattern detection techniques would give a good understanding of how the software is built, however it would not detect any design violations.

Our approach uses static code analysis to verify the presence of any design pattern violations. A static representation of the codebase is stored as an object-model. The object-model allows us to extract and manipulate specific details about different elements in the software. Design guidelines are specified in order to identify which specific part of the object-model has to be parsed. Traditionally design guidelines are outlined in an informal way, giving rise to lot of ambiguities in understanding. This ambiguity will be more obvious when a new design pattern is defined. The Unified Modeling Language (UML) can be used to graphically represent a design pattern, but they are not expressive enough to precisely define all the design patterns. We have adapted a domain specific modeling language derived from UML to formally specify the design guidelines. This language allows us to have an equally readable language, but much more expressive
than UML. This language also allows us to build an easy translation from the formal specification to the implementation of the tool. We also explain the testing methodology employed to eliminate false positive and false negative violations being generated from our tool.
# Contents

1 Introduction .......................... 1
   1.1 Problem statement .................. 2
   1.2 Research contribution .............. 3
   1.3 Thesis outline ..................... 3

2 Approach ............................. 5

3 Background ........................... 7
   3.1 Philips medical imaging application . 7
   3.2 Literature overview .................. 8

4 Design patterns ....................... 11
   4.1 Introduction ........................ 11
   4.2 Structure of design patterns ........ 13
   4.3 Examples ............................ 14
   4.4 Standard design patterns .......... 15
      4.4.1 Creational ...................... 16
      4.4.2 Structural ...................... 20
      4.4.3 Behavioral ...................... 23

5 Design patterns formalization .... 26
   5.1 Introduction ........................ 26
   5.2 Adapting the GEBNF syntax .......... 27
      5.2.1 Introduction .................... 27
      5.2.2 Translation of a class diagram 28
      5.2.3 Building predicates ............. 32
      5.2.4 Formalization of standard patterns 33

6 Philips specific design patterns ... 36
   6.1 Dispose pattern .................... 36
      6.1.1 Dispose pattern guidelines .... 38
   6.2 MVVM Pattern ....................... 55
      6.2.1 MVVM design guidelines ....... 58
Chapter 1

Introduction

This chapter is an introduction to software design patterns and the motivation behind providing an infrastructure to verify design pattern violations. We also discuss the problem statement and the research contribution in the domain of design pattern verification techniques.

Classes and objects form the fundamentals of any object-oriented software. As the software evolves due to new requirements, it becomes complex and bulky. Complex software must be easily manageable for new requirements to be easily adapted without significant costs of software redesign. A design pattern solves this issue by defining relationships between software components, which are simple as well as flexible. Once a design pattern proves to be an effective solution for a particular problem, it forms a reusable asset in software design. Design patterns are defined by Gamma et al in [1] as: “descriptions of objects and classes that are customized to solve a generic design problem in a particular context”. The design patterns illustrated in the book are called as the Gang-of-Four (GoF) patterns. Few techniques adopted from this book are explained when they are first introduced.

The process of verifying design patterns in existing software will protect the software against degradation. This reverse engineering approach will help an architect improve the existing design or even ensure whether the desired patterns are not violated. Design issues are quite different from development issues. The latter can be rectified during testing or maintenance phases. Contrary to development flaws, design flaws may not be noticed even after the software is released. They are detected when the existing design does not support, the implementation of new requirements. Critical changes in software design might result in large scale code refactoring, making it very expensive. The London Ambulance fiasco in 1992 [2] incurred a loss of over £ 7M due to design failures.
This thesis is concerned with verifying if the implementation adheres to the design pattern guidelines. Throughout this thesis we maintain a similar structure to clearly describe each design pattern stating the benefits of using the specific design pattern. Although developing a tool to validate design patterns is our objective, precisely specifying design patterns help the designer identify any loopholes. A Domain Specific Modeling language (DSML) can be used by a designer to abstract the functionality, constraints and inter-relation of model elements. The level of abstraction can also be decided by the designer based on the complexity of the software or his/her domain expertise. Not only does this language allow designers to communicate precisely, but it also helps building a tool. We define the problems involved in developing a tool for validating design patterns in Section 1.1.

1.1 Problem statement

Re-engineering of design patterns from the source code is always a challenge. This involves static analysis of the entire code base to capture the interrelation between elements of the source code. The implicit problem with static analysis is to capture an object-model. An object-model describes the static relationship between different software elements. We broadly divide the problem into the following parts:

- **Precise specification of design patterns:**
  Design patterns must be clearly specified to alleviate any ambiguous understanding. The objective is to employ a language that can precisely specify any design pattern. The adopted language must also be convenient for a novice to understand and specify design patterns.

- **Capturing the object-model:**
  We perform static analysis on source code in C# language and record its structure as an object-model. The object-model can be used to extract structural relationship between different elements in the source code. This object-model must also provide a simple framework to validate the custom design patterns considered in this thesis.

- **Automatic verification of design patterns:**
  None of the software revisions must violate any existing designs considered. This avoids any degradation of the software product, when it goes through maintenance. It is most effective when the design violations are detected as soon as possible to eliminate any redesign costs.

  If a tool is developed to detect design pattern violations in a large codebase, false-positive violation reports must be handled effectively. It would be frustrating for the end user to witness false-positive design violations from the tool. False-
negatives are even more dangerous, as it would completely miss the error.

1.2 Research contribution

The purpose of this research is to provide a reverse engineering approach that will enable verification of custom design patterns in C# source code. We illustrate how the design patterns considered are specified in a modeling language. The design pattern specified in this language is verified against the object model of the source code by performing static code analysis. The design violations are validated against any false positive or false negative by following an iterative verification methodology.

We also provide a tool to develop guidelines for verifying custom design patterns. The implemented framework supports on-the-fly as well as offline verification mechanisms. On-the-fly verification enables the developer to correct the implementation, as soon as the issue is seen in Visual Studio IDE. Offline verification generates a violation report for the entire code base. We also compare the two approaches.

One of the important contribution of this research is to provide a generic framework that is not restricted to just the design pattern checks that are currently useful. The tool is implemented on a simple and flexible interface, making new extensions possible.

1.3 Thesis outline

Chapter 1 presents an introduction to design patterns and the need for verification of design patterns in the source code. We state the problem and the research contribution of our work in this chapter.

Chapter 2 presents the approach of our work and also focuses on the development model we have used to meet the requirements.

Chapter 3 includes the details of the Philips application for which the verification of design patterns is conducted. Also the background information about the techniques in design pattern specification and verification.

Chapter 4 includes a short introduction to design patterns and the need for design patterns. This chapter also includes few examples of standard design pattern.

Chapter 5 presents the importance of formal specification and introduces the formalization we use to specify the design pattern guidelines. As an example to the formalization, we have formalized three standard design patterns described in Chapter 4.
Chapter 6 focuses on three design patterns followed at Philips. It includes the need for these guidelines and their formalization. Each pattern is described in detail by formalizing each guideline separately.

Chapter 7 has a background on existing tools that can be used for our work. This chapter also includes the tool we developed for automatic verification of design pattern guidelines in Philips code-base. This chapter gives the co-relation between the implementation and the formalization.

Chapter 8 presents the verification of the tool and different testing techniques used on each component of the tool.

Chapter 9 presents the test results of the guidelines implemented using the tool. The implemented guidelines are tested on few test cases and the Philips code-base.

Chapter 10 has the conclusion of our work and the recommendations for future work.

The Appendix includes details of the Windows Presentation Foundation and its key features, followed by few user interface patterns like Model View Controller and Model View Presenter. This part also includes few additional design pattern guidelines and their formalization.
Chapter 2

Approach

In this section, we discuss the methodology adopted in solving the problems described in Section 1.1.

We follow an iterative and incremental development approach for implementing our tool for design pattern validation. Figure 2.1 shows a classical iterative and incremental development model.

We initially consider a basic set of design patterns and later consider complex cases by adopting an incremental development. The approach considered to tackle the problems listed in Section 1.1 is explained below:

- **Precise specification of design patterns:** We prepare a catalog of all the design patterns considered in this research. These design patterns exist in the current code base and are also expected to hold in all the future revisions of the software. We can relate this step to the Requirements phase in Figure 2.1. The technique of describing a design pattern is discussed in Section 4.2. Although this gives the reader a firm understanding of the complexity
and the kind of design patterns dealt with, it does not specify any design pattern precisely. We use a modeling language specific to the design patterns considered. This step can be mapped to the Analysis and Design phase in Figure 2.1. We discuss about this approach in Chapter 5.

- **Capturing the object-model:** We extract the static properties of the source code and record it as an object model. This allows us to access and extract the code structure by traversing through it. The implemented tool validates the design pattern implementations in this object-model. This step can be mapped to the Implementation phase in Figure 2.1. We discuss about this approach in Chapter 7.

- **Automatic verification of design patterns:** The implemented framework is utilized to perform an on-the-fly design pattern detection together with the traditional offline checking. An on-the-fly technique ensures that design pattern validation is always active in the background while coding is done. Hence any new functionality being implemented adheres to the design patterns defined. This step also belongs to the Implementation phase in Figure 2.1.

- **Prevention of false-positives and false-negatives:** It is very hard to ensure that a design pattern is correctly implemented. Strict design guidelines will limit the freedom of the developer and hence may result in lot of false-positives. Contrary to this, a slack in design guidelines, might result in false-negatives. A trade-off between false-positives and false-negatives is considered. Manual testing is performed to verify if the design violations are true-positives or not. This step can be mapped to the Testing phase in Figure 2.1. The results from the tool are evaluated based on the feedback of designers/architects at Philips Healthcare. We discuss about this phase in Chapter 9.

Figure 2.2 shows the development model considered for this research.
Chapter 3

Background

In this chapter, we will discuss the background of design pattern verification and its use in Philips Healthcare. We describe few of the related approaches in design pattern verification and explain our approach from the standpoint of Philips Healthcare. We also give a brief introduction to the technology behind the applications developed at Philips Healthcare.

3.1 Philips medical imaging application

This research has been carried out with the Viewing architecture team at interventional XRay (iXR) department of Philips Healthcare. In this subsection, we give an idea about the objectives of this team and how our research (partly) assists them.

The iXR department is involved with developing state-of-the-art medical imaging products for cardio, vascular and neuro X-ray procedures. Figure 3.1 shows a typical setup of such a product. The prime objective is to create software products that assist a doctor to perform an intervention, as quickly as possible with maximum comfort to the patient. To ensure this, a lot of 3D rendered graphical visuals are presented to the doctor. The software should also allow easy extensions for sustaining a high innovation speed and to have quicker stable products. Testability must also be considered as an integral part of software evolution, so that high test coverage can be achieved.

These objectives can be met if the software is built on well defined design patterns. A strictly layered architecture is adopted where specific functionalities are grouped together into one layer. The domain layer having the actual data (like patient images and file streams) is segregated from the view layer, which has knowledge only about the way to visualize it. This will ensure maximum decoupling of code by enabling each component
layer to be developed independent of the other.

Figure 3.2 shows a three-layered architecture being used at the iXR department. This technique along with associated advantages and design guidelines are discussed in Section 6.2.1.

Preserving the design guidelines in all future revisions of the software product is of prime importance. This will ensure minimum code degradation and also limit the expenses incurred due to code refactoring. This research helps in validating the design patterns implemented, hinting the developer to make changes as soon as any design violations are detected.

3.2 Literature overview

Research on design pattern specification and verification techniques has been active for several years now, with a whole range of ideas being proposed. In this section, we will discuss some of the popular techniques for design pattern specification and verification. This will also include the technique that has inspired us to form the basis of our research. These approaches [3] can be broadly classified into:

- Developing new formalization technique (requires steep learning curve and time consuming) vs. adapting existing techniques (quicker and build on available knowledge).

- Developing graphical models (easier to visualize and interpret) vs. textual models (easier to manipulate and maintain) vs. hybrid model (having both features).
3.2. Literature overview

- Specifying patterns at the model level (convenient for developers) vs. meta-model level (convenient for building tools).

- Developing new tools for design pattern verification (longer development time) vs. existing tools (shorter development time) vs. hybrid approach (building a specific functions on top of existing framework).

Many techniques for design pattern specification and detection are based on logic calculus. Temporal logic is the basis for capturing the behavior of the software in [4]. A combination of predicate logic for static aspects and temporal logic for dynamic aspects of the software, was proposed in [5]. In these approaches, each design pattern must be mapped onto a particular domain specific modeling language. This makes it very difficult for a new member to specify and validate design rules.

Other approaches used techniques to analyze source code details and translate it to representations that are more convenient for detecting design patterns. For instance, conversion of source code into a graph representation is discussed in [6] and [7] proposes the use of Binary Decision Diagrams (BDD) for representation of source code. [8] proposes the use of SQL-like query to represent relationship between entities (like classes).
for design pattern detection. One of the prime reasons for not adopting these approaches is the issues with interpreting the translated representation. This would involve developing a domain specific language, making it non-trivial for a developer to understand and build new design rules for validation.

There are many commercial tools that perform static and dynamic analysis of the source code to detect design patterns. Pattern Detection Engine (PDE) [9] extracts design pattern candidates in the Java source code and finds design patterns by using the roles and collaboration information existing in the software. Fujaba [10] is an open source tool supporting model-based software engineering and re-engineering. Although PDE and Fujaba are tested for all the 23 original design patterns in [1], it is restricted to design pattern detection in Java source code. They are also intended for the detection of design pattern rather than verifying wrong design patterns. These approaches cannot be adopted in our research as they do not provide an easy framework to verify violations of design patterns nor work with C# source code.

Considering these problems, it would be ideal if an existing formalization technique is adopted, that would still be relatively easy for a designer to specify design patterns. It should also be extendible into a toolset for verifying the design patterns and validate them. For this reason, we adopt the following scheme from [11] and [12]:

Dong et al in [12] proposes an approach based on first-order logics for specifying structural aspects of the design patterns. The dynamic aspects are modeled by Temporal Logic of Action (TLA). A very similar approach is considered in [11], defining a new meta-modeling language based on first order predicate logic. The notations in this approach are based on a graphical extension of Backus Naur Form (BNF). Design patterns are specified by formulae derived from this formalism. This formalism also aids in having an easy mapping from the formal specification to a tool implementation.

In our research, we extend the approach used in [11], by defining a syntax built specifically for capturing the structure of object-oriented languages. This also allows us to define predicates on different levels of abstraction, making it an ideal specification technique for both novice or expert users. This feature can be used by architects to define software at system level and even a developer at the unit level. Due to which reason, this formalism enhances communication in an effective way across all members of a team.
Chapter 4

Design patterns

An introduction to design patterns was given in Chapter 1. In this chapter, we consider few examples from the original Gang of four patterns and explain how design patterns are described. Using this as a basis, we also explain few modern patterns specified by Microsoft. Section 4.1 introduces how design patterns originated and its advantages. This chapter forms a basis for creating and managing custom design patterns discussed in Chapter 6.

4.1 Introduction

Design and Architecture were originally associated with buildings and towns only. Oxford dictionary describes Architecture as a discipline dealing with the art or practice of designing and constructing buildings. The plan of how different elements of the architecture (like room) are connected is dictated by a Design. Architecture and design can save a lot of human effort by reusing it as a solution in any other project. A famous quote by Christopher Alexander, a civil engineer says that “Each architectural pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”.

The concept of architecture and design have been carry forwarded into the modern age of Object-Oriented Programming. An object in the software world represents an entity, like a building of the civil world. A software design dictates how different elements of the software (like classes) are related. When such designs form a generic solution to a reoccurring problem, it is called as a Design Pattern. As explained in, modern software products are bound to evolve over time owing to new features being added. A large program cycle makes the development increasingly unpredictable, leading to
increased maintenance costs. This can controlled by using designs which are flexible enough to adapt to new functionalities easily rather than redesigning the system. The use of design patterns gives a template for creating flexible designs. A list of such software design patterns are explained by the Gang of Four (GoF) [1]. We discuss some merits and demerits of using design patterns by the software architects or developers.

Merits:

- **Reusability:**
  Design patterns define a reusable programming paradigm to a reoccurring problem. It also helps to prevent subtle issues that can cause major problems when the software evolves [1].

- **Flexibility:**
  Adherences to design patterns enable the growth of the software in a smooth way. In OO languages, a design pattern establishes high-level abstractions implementing generic solution rather concrete ones. This will implicitly mean that software product can be easily maintained over its lifetime [1].

- **Reduce code pollution:**
  One of the common problems seen with any large scale software product, similar to our case at Philips Healthcare, is *code pollution*. Code pollution, is the degradation of software as a result of developing source code not adhering to standards. Such software might produce the current required outputs, but will be very difficult to understand and extend over time. A design pattern gives an accepted coding standard.

- **Common vocabulary:**
  Design patterns give a common vocabulary corresponding to a specific design pattern, providing for architects and designers to picture the design problem immediately by referring to the pattern name [16].

Although design patterns bring in the merits listed above, there are few challenges in adopting one. The following points explain these challenges.

Challenges:

- **Complexity:**
  Applying the correct design pattern for a specific problem is always a challenge. By choosing a pattern that does not solve the design problem efficiently might significantly increase the development cost of the software [16]. This design pattern is termed as *Anti-Patterns* in literature [17].

- **Memory consumption:**
  Due to a generalized structure of design patterns, the source code might often
consume more memory than not applying any design pattern [18].

4.2 Structure of design patterns

In Section 4.1, we explained an advantage of developing a common vocabulary by using design patterns. Such vocabulary can only be developed if each design pattern is precisely stated. In this section, we discuss the following four components for building a good vocabulary [1]:

1. **Pattern name:**
   An apt name is very important while developing a design pattern. The name must also indicate the kind of problem being tackled. The name should be decided as soon as there is any idea of design problem, but can also be altered based on needs later on. This enables the designers to communicate better and formulate a solution quickly.

2. **Problem:**
   This describes the context of the design problem and under what circumstances must the designers use the design pattern. Usually a list of preconditions is drawn which indicates when a particular design pattern must be applied.

3. **Solution:**
   Given a problem, a suitable design pattern will form the generic solution. Although before deciding on any design pattern, it must be communicated well among the designers and extensively be tested for optimal reuse. Solution must have details about what relationships exist among different classes. It should be taken care that a solution must be abstract and never be concrete, as it should enable reuse across many similar problems.

4. **Consequences:**
   This should carry information about the result and trade-offs of applying the pattern. Consequences are important to judge if there are any design alternatives. Consequences should also highlight the cost benefits of applying as well as not applying the design pattern.

We consider the above four details corresponding to a design pattern. However there are other details like Alias name, Alternatives, Structure and Sample code can documented for a design pattern. The exhaustive list of such details can be seen in [1].
4.3 Examples

In Sections 4.1 and 4.2 we explained the advantages and details that must be associated with each design pattern, at a very high level. In this section, we take will have a closer look at describing a design pattern used commonly in object-oriented languages with an example.

The use of design patterns can really depend on the designer, as some patterns can be considered as a normal way of solving a problem. On this basis, we present a simple design principle to help understand how design patterns impact software development.

*Program to an interface, not an implementation* [1]

**Pattern name:**
Although this principle is not stated as a software design pattern, it forms the basis for all the creational patterns discussed in [1]. We discuss creational patterns in Section 4.4.1. For convenience we refer to this design principle as Interface Pattern.

**Problem:**
Object instances created by implementing concrete classes result in implementation dependencies between different classes of the software. Such an implementation can be termed as tight coupling.

Consider the example [19] in Listing 4.1 showing the design problem.

```csharp
public class PepperoniPizza
{
    public string Eat()
    {
        return "pepperoni";
    }
}
public static void Main()
{
    PepperoniPizza myPizza = new PepperoniPizza();
    myPizza.Eat(); // "outputs pepperoni"
}
```

Listing 4.1: Programming to an implementation

**Solution:**
Inheritance in object-oriented languages extends the functionality of a parent class. Each function in the child class either adds its own functions or overrides the existing functions in the parent class. By inheriting from an abstract class, we can have all subclass functions declared in its parent abstract class. The abstract class provides a common interface for accessing all its subtypes. By adopting this approach, we can define a loosely
coupled interface that can be referred by its decedents dynamically. Listing 4.2 shows an implementation scheme to tackle the problem of tight coupling.

```java
public interface IPizza
{
    string Eat();
}

public class PepperoniPizza : IPizza
{
    public string Eat()
    {
        return "pepperoni";
    }
}

public static void Main()
{
    IPizza myPizza = new PepperoniPizza();
    myPizza.Eat(); // outputs "pepperoni"
}
```

Listing 4.2: Programming to an interface

**Consequences:**
All subclasses implicitly become subtypes of the abstract class. This technique has the following advantages:

1. All objects adhere to the abstract class, making an abstraction over the specific types of objects used.

2. The concrete class that inherits the abstract class, will be hidden. This enables encapsulation of details by a single class type.

### 4.4 Standard design patterns

In Section 4.1 we discussed the basics of design patterns. Using this background, in this section we discuss some of the common design patterns found in literature [1]. These design patterns are broadly organized into Creational, Structural and Behavioral patterns on the basis of its intent. These are explained in Sections 4.4.1, 4.4.2 and 4.4.3. This categorization helps in finding families of related patterns and even forming new design patterns in the future. Defining new design patterns can be done by comparing its purpose to that of an existing one. As this section serves as an introduction to some of the standard design pattern, we pick one design pattern from each category and explain it same detail as in Section 4.2.
4.4.1 Creational

The design patterns falling in this category concerns the way objects are created in the software. Creational patterns give a high level view of the object instantiation and composition. A class creational pattern uses inheritance to extend the functionality of an instantiated class, whereas an object creational pattern delegates the instantiation to another object [1]. In general these patterns make class instantiation and object creation generic to ensure maximum reuse.

Many commonly known standard creational patterns are explained in [1]. As an introduction we will discuss one of the creational patterns known as Builder pattern.

**Pattern name:**
Builder Pattern

**Problem:**
The problem is to provide an abstract representation objects by encapsulating the way the objects are created.

Consider a program in Listing 4.3 to assemble different types of computer monitors. In this example, BuildA and BuildB are objects created specifically to call its respective functions (DoIt) to build a Monitor. This is another instance of a tightly coupled implementation. The problem now is to create a common interface for accessing the individual methods in the individual objects created (BuildA and BuildB).

```java
class BuilderA
{
    Monitor m = new Monitor();
    public Monitor DoIt()
    {
        // Necessary code for building the computer type A
        m.type = "mono";
        return m;
    }
}
class BuilderB
{
    Monitor m = new Monitor();
    public Monitor DoIt()
    {
        // Necessary code for building the computer type B
        m.type = "color";
        return m;
    }
}
```
4.4. Standard design patterns

```java
class Monitor {
    String type;
}
class MyClient {
    public static void Main() {
        BuilderA BuildA = new BuilderA();
        BuilderB BuildB = new BuilderB();
        Monitor m1 = BuildA.DoIt();
        Monitor m2 = BuildB.DoIt();
    }
}
```

List 4.3: Design Problem without Builder pattern

**Solution:**
The aim of the builder pattern is to isolate the object representation from its construction. The Builder pattern is an example where the design principle discussed in Section 4.3 is used. The object representation can be isolated by introducing a class to encapsulate both the Builder classes (BuilderA and BuilderB). Consider the structure shown in Figure 4.1.

![Builder pattern structure](image)

- **Builder** provides an interface for implementing different concrete products.
• **Director** will construct product objects using the Builder interface.

• **BuilderA** and **BuilderB** implements the Builder interface and provides a specific implementation for creating the product.

• **Product** is a final object that is built by the Director.

Figure 4.1 shows the relationship between classes, but the interaction between objects is captured by a sequence diagram as shown in Figure 4.2.

![Figure 4.2: Sequence diagram in a Builder pattern](image)

Initially the client creates a Director object and a desired abstract Builder object. The ConcreteBuilder object is passed to the Construct function of the Director object. This is illustrated in the example snippet in Listing 4.4.

**Example:**
Consider a Builder pattern being applied to a Computer Assembly program as seen in Code Snippet 4.4 [9]. The process of assembling a Computer is generic to what specific components are installed. Hence it is simple if the process of assembling the computer is kept independent of the components of the actual computer. Listing 4.4 is a code sample in C# explaining this approach.

```csharp
class Director
{
    public Monitor Construct(IBuilder builder)
    {
        builder.DoIt();
    }
}
```
interface IBuilder
{
    Monitor DoIt();
}
class BuilderA : IBuilder
{
    Monitor m = new Monitor();
    public Monitor DoIt()
    {
        // Necessary code for building the computer type A
        m.type = "mono";
        return m;
    }
}
class BuilderB : IBuilder
{
    Monitor m = new Monitor();
    public Monitor DoIt()
    {
        // Necessary code for building the computer type B
        m.type = "color";
        return m;
    }
}
class Monitor
{
    String type;
}
class MyClient
{
    public static void Main()
    {
        Director d = new Director();
        Monitor m1 = d.Construct(new BuilderA());
        Monitor m2 = d.Construct(new BuilderB());
    }
}

Listing 4.4: Example for a Builder pattern

Consequences:
The Builder pattern will provide flexibility to the developer, in using a similar object
collection step but represent it in different forms. For instance, in Listing 4.4, any
new Monitor type can be constructed by adding a new Builder class implementing the
IBuilder interface. The representation scheme is completely isolated from the Client, who needs to access only the generic Director functions (like Construct) to construct specific Monitor types.

4.4.2 Structural

Structural aspects deal with identifying a simple way to realize how classes and objects can be related. A structural class pattern can be a composition of many finer classes and objects to form a larger class. Multiple inheritance is a simple example of a structural class pattern. A structural object pattern composes different objects rather than classes to add more functionality. As a result, the structural class patterns involve inheritance, whereas the structural object patterns involve object delegation. A distinct advantage with this approach is that the composition of objects can be changed at compile time. Many types of Creational patterns can be found in [1], we pick the Adapter pattern as it is one of the most commonly used pattern in object-oriented software.

Many standard structural patterns are listed in the GoF patterns [6]. We will consider a structural pattern known as Adapter pattern.

Pattern name:
Adapter Pattern.

Problem:
There are situations when the interface of a particular class is not in the same format as expected by the client. In such situations there has to be a mechanism of bridging this gap. A similar day-to-day example is the client expecting to use an electric plug not complying to the present standards.

Solution:
The Adapter pattern addresses the problem by introducing another class that will wrap the incompatible class so that it adheres to client expectations.

Figure 4.3 is a UML class diagram representing the Adapter class pattern. The Client class calls the generic method in the ITarget interface implemented in the Adapter class. The Adapter class will adapt all other Adaptee classes.

In the Adapter class pattern, the Adapter inherits the properties of the Adaptee, where as in the object pattern the Adapter contains an Adaptee object. The structure for the Adapter object pattern is shown in Figure 4.4.

- ITarget provides the required interface based on Client needs.
- The SpecificRequest method in Adaptee has to be adapted so that Clients access is limited to the Adapter only.
4.4. Standard design patterns

Figure 4.3: Adapter Class Pattern Structure

Figure 4.4: Adapter Object pattern structure

Sequence Diagram:
Figure 4.5 shows a simple sequence of how classes interact in an Adapter pattern.

Initially Client creates an Adapter object for accessing the adapter method. This method will abstract calls to the Adaptee specific methods and returns the control back to the Client.

Example:
Consider the example shown in Listing 4.5. The Client (MyMain) calls the generic method in the ITarget interface. The Adapter class implementing this interface adapts the definition of the ForeignMethod from a float to an int value.
// the class which is acceptable by the client
interface ITarget
{
    int Method();
}

// Adaptor class which makes the Adaptee class accessible by the client
// inherit from ITarget interface
class Adaptor : ITarget
{
    private Adaptee ad = new Adaptee();
    public int Method()
    {
        int a;
        a = (int) ad.FOREIGNMETHOD();
        return a;
    }
}

// adaptee class which we have to be accepted by the client
class Adaptee
{
    static float a = 8.3f;
    static float b = 3.3f;

    public float FOREIGNMETHOD()
    {
        Console.WriteLine("Foreign Adapted Method");
        return a * b;
    }
}
class MyClient
{
    public static void Main()
    {
        ITarget t = new Adaptor();
        int intValue = t.Method();
    }
}

Listing 4.5: Example for an Adapter class pattern
4.4. Standard design patterns

**Consequences:**
The example shown in Listing 4.5 is of an Adapter object pattern. This approach allows the developer to have multiple Adaptee types within the same Adapter class. In an Adapter class pattern, extending multiple Adaptee’s would not be possible as multiple inheritance is restricted in Java and C#.

### 4.4.3 Behavioral

Behavioral patterns are concerned with the communication scheme between objects and classes. It provides a flexible representation of a complex interaction scheme, which is often difficult to trace during runtime. Inheritance to distribute behavior among classes is explained as a simple example in [1]. There are many commonly known standard behavioral patterns listed in [1].

We will introduce one of the behavioral patterns known as Template Method pattern.

**Pattern name:**
Template Method Pattern.

**Problem:**
There are situations when the purpose of an algorithm would be the same but the steps in the algorithm are performed differently in each of the class instance. Creating two algorithms in each of the class and the client having to create class instances for calling each algorithm would result in a very inefficient and heavy framework.

Consider a simple example of accessing two different email accounts. If the function for accessing is separate for each account, the client has to explicitly know which function targets which account.

**Solution:**
The problem can be solved by having an abstract class providing a common interface for calling any of its child function calls. Consider the UML diagram shown in Figure 4.6 representing the Template Method pattern. The common interface is provided by the Algorithm method in the abstract class, AbstractTemplate.

- **AbstractTemplate** has the function signature for Algorithm(s).
- **ConcreteClass(es)** has specific steps implemented as part of the method, Algorithm.
- **Algorithm** forms the Template method, providing a generic interface for the client.

**Sequence Diagram:**
Figure 4.7 shows a simple sequence of how classes interact in a Template Method pattern.
Initially Client creates an instance of the Concrete class object via the AbstractTemplate class. A call to the generic method, Algorithm will internally invoke the methods, Step1 and Step2.

Example:
Consider the Template pattern implementation example shown in Listing 4.6. The template method Login, provides a signature for implementing the methods Step1 and Step2 in its concrete child classes, YahooMail and GMail.

```java
abstract class AbstractLogin {
    public abstract void Step1();
    public abstract void Step2();
    public void Login() // 'Template method'
    {
        Step1();
        Step2();
    }
}
```
// 'ConcreteClass1' class
class YahooMail : AbstractLogin
{
    public override void Step1()
    {
        //ConcreteClassA implementation for Step1 of Login
    }
    public override void Step2()
    {
        //ConcreteClassA implementation for Step2 of Login
    }
}

// 'ConcreteClass2' class
class GMail : AbstractLogin
{
    public override void Step1()
    {
        //ConcreteClassB implementation for Step1 of Login
    }
    public override void Step2()
    {
        //ConcreteClassB implementation for Step2 of Login
    }
}
class MainApp
{
    static void Main()
    {
        AbstractLogin aYahoo = new YahooMail();
        aYahoo.LogIn();

        AbstractLogin aGMail = new GMail();
        aGMail.LogIn();
    }
}

Listing 4.6: Example for a Template Method pattern

Consequences:
One distinct advantage of this pattern is any new implementation of an algorithm in
the concrete classes will only require the addition of this method call in the Template
Method. The call in the client class (MainApp in Listing 4.6) need not be altered,
allowing a simple and flexible interface. This communication scheme is termed as inverted
control structure, where the parent class calls the operations of its child classes and not
vice versa.
Chapter 5

Design patterns formalization

In this chapter, we discuss the importance of a specification technique that can formally describe a design pattern. We define a modeling language whose syntax is based on the Extended Backus Naur Form (EBNF). Structural aspects of a design pattern can be abstracted by the use of predicates. We explain a formalism based on first-order predicate logic to specify a design pattern precisely.

5.1 Introduction

All the design patterns considered in Section 4.2 are specified in an informal way. By informal, we mean that the design patterns are specified with explanations in English text along with few examples. A better way of representing a design pattern is by using Class Diagrams (CDs). CDs are meta-notations for depicting the relationship between different entities in the system. Although the design specifications using CD is very readable, it is not expressive enough to describe all the facts relevant to a design pattern. Explicit specification for absence of a relationship can never be expressed by a CD alone. Although informal specification of design patterns like in Section 4.2 might give clear description of the motive, a CD can only show the presence of inter-class relationships but not its absence. This is the most important reason in adapting a technique to formally specify the design patterns.

If we could cover all the design guidelines precisely by using just CDs, we could think of a direct translation of a CD to a design pattern guideline that can be verified by a tool. Building a formal syntax for specifying design patterns would alleviate the problems of using informal descriptions. In general, formal specification has the following advantages:

1. Eliminates any ambiguous interpretations by the reader.
5.2. Adapting the GEBNF syntax

2. Can result into a consistent generic language for specifying design patterns, so that automated tools can be built for validating the guidelines.

3. A common syntax for designers and architects to communicate.

5.2 Adapting the GEBNF syntax

In this section, we describe the scheme used to formally specify the design patterns considered in this thesis. In Section 3.2, we discussed some of techniques for design pattern specification. We concluded in same section that the technique used in [11] is best suited for our research. In this section we explain this extension to support our problem.

5.2.1 Introduction

Extended Backus-Naur form (EBNF) originally proposed in [20], improves the basic BNF notation by expressing context-free languages using a regular expression syntax. Similar to EBNF being used to define the syntax of programming languages, a Graphical Extended Backus Normal Form (GEBNF) [11] can be used to define the syntax of a graphical modeling language like UML. This syntax is restricted to graphical modeling and is specific to the domain it is intended for. Following is an abstract syntax of a modeling language defined on GEBNF syntax, defined by the tuple \( R, N, T, S \).

where, 
\( N \) is a finite set of non-terminal symbols, 
\( T \) is a finite set of terminal symbols, 
\( R \in N \) is the root symbol and 
\( S \) is a finite set of syntax rules:

E.g. \( Y ::= f_1 : X_1, f_2 : X_2, \cdots, f_n : X_n \)

where, 
\( Y \in N, f_1, f_2, \cdots, f_n \) are called field names, 
\( X_1, X_2, \cdots, X_n \) are the fields.

As explained in Section 4.4, design patterns are specified with Class Diagrams (CD) in combination with informal explanation. The relationships shown by a CD, can now be expressed using a syntax that is defined with the GEBNF notation. In essence, GEBNF provides a meta-modeling language for defining different components of a CD. Using this language, we can define custom formulae based on first-order predicates to capture the details of the CD. This will be equally readable but much more expressive than a CD.

In Figure 2.2, we had explained our approach of tackling the research problems in Sec-
Figure 5.1 gives an insight into the finer details of our approach. It depicts that, predicates defined on the syntax of GEBNF captures the details of a CD. A CD is a semi-formal representation of a design pattern, as it still needs textual details to describe a design pattern precisely.

5.2.2 Translation of a class diagram

In this section, we explain the GEBNF syntax for describing a CD. A CD is depicted by classes and interfaces, related by associations and generalizations between classes and calls between methods existing in respective classes. The following syntax is adopted:

\[
CD ::= \\
\quad \text{classes} : \text{Class}^+, \\
\quad \text{interfaces} : \text{Interface}^*, \\
\quad \text{assocs} : (\text{Classifier}, \text{Classifier})^*, \\
\quad \text{geners} : (\text{Classifier}, \text{Classifier})^*, \\
\quad \text{calls} : (\text{Method}, \text{Method})^*
\]

A Classifier can be a Class or an Interface shown as:

\[
\text{Classifier} ::= \text{Class} \mid \text{Interface}
\]

A Class is associated with its name, namespace it belongs to, variables defined, methods declared, class access specifier, constructors and finalizers. Any Class can have zero or more Variable, Property, Method or Constructor declarations. A Class can have either zero or one Finalizer declared. Only one AccessSpecifier and Modifier is allowed in each class. Class can be described as:
5.2. Adapting the GEBNF syntax

Class ::= 
  name : String,
  namespace : String,
  methods : Method*,
  constructor : Constructor*,
  finalizer : Finalizer?,
  variables : Variable*,
  properties : Property*,
  accessSpecifier : AccessSpecifier?
  modifier : Modifier?

where, String is a terminal denoting a string of characters.

Interface is similar to the Class, however it does not have variables, constructors and finalizer as defined for a Class. It can be described as:

Interface ::= 
  name : String,
  namespace : String,
  accessSpecifier : AccessSpecifier?
  modifier : Modifier?
  methodsigs : MethodSig*

Methods have a name, input parameters, return types, access specifier, modifier, variables and method calls. Any Method can have zero or more input parameters (inParams) with a particular return type (returnType). There can be zero or more MethodCall within a Method.

Method ::= 
  name : String,
  inParams : Variable*,
  returnType : Class,
  accessSpecifier : AccessSpecifier?
  modifier : Modifier?
  variables : Variable*

returnType represents any data type that can be defined in the programming language. In general, we define it as a Class.

Property defined in the syntax of a Class is similar to a Method, without having input
parameters (\textit{inParams}).

\textit{Constructor} is similar to the syntax of a Method except that Constructor does not have a return type (\textit{returnType}). Moreover, the name of constructor is implicitly understood to be the same as that of the class it belongs to.

A \textit{Finalizer} is similar to the \textit{Constructor} without input parameters (\textit{inParams}) and a \textit{Modifier}.

\textit{Variables} has a name, type and modifier flag.

\begin{verbatim}
Variable ::= 
    name : String, 
    type : Class, 
    modifier : Modifier
\end{verbatim}

\textit{MethodSig} represents the signature of a method and has the same syntax as that of a \textit{Method} except that it does not have a \textit{Variable} and \textit{Methodcalls}.

\begin{verbatim}
MethodSig ::= 
    name : String, 
    inParams : Variable*, 
    returnType : Class, 
    accessSpecifier : AccessSpecifier? 
    modifier : Modifier?
\end{verbatim}

\textit{MethodCall} has a name, input parameters and a class instance that it refers to. The declaration of the method call is found in this Class.

\begin{verbatim}
MethodCall ::= 
    name : String, 
    instance : Class, 
    inParams : Variable*
\end{verbatim}

\textit{Modifier} can be virtual, static, sealed, abstract or override as described:

\begin{verbatim}
Modifier ::= virtual | static | sealed | abstract | override
\end{verbatim}

\textit{AccessSpecifier} can be public, private, protected and internal as described:

\begin{verbatim}
AccessSpecifier ::= public | private | protected | internal
\end{verbatim}

\textbf{Example:}
We can represent any CD using this EBNF syntax. Consider the CD in Figure \ref{fig:example}, this
can be described by the following syntax.
CD has classes: AbstractTemplate, ConcreteClass1 and ConcreteClass2. There is a generalization relationship between ConcreteClass1 and AbstractTemplate, and also between ConcreteClass2 and AbstractTemplate. There is also a method call from Algorithm to Step1 and Step2.

\[ CD ::= \{ \text{AbstractTemplate, ConcreteClass1, ConcreteClass2} \} : \text{classes}, \]
\[ \{(\text{ConcreteClass1, AbstractTemplate})\} : \text{geners}, \]
\[ \{(\text{Algorithm, Step1}), (\text{Algorithm, Step2})\} : \text{calls}. \]

After the basic components of the CD are represented, we get down to the class level. The syntax for the classes AbstractTemplate, ConcreteClass1 and ConcreteClass2 are seen below. Class constitutes name, methods and modifier. An Example for the modifier of the AbstractTemplate class is abstract.

\text{AbstractTemplate} ::= 
\quad \{\text{“AbstractTemplate”}\} : \text{name}, 
\quad \{\text{Algorithm, Step1, Step2}\} : \text{methods}, 
\quad \{\text{abstract}\} : \text{modifier}. 
\quad \{\text{public}\} : \text{accessSpecifiers}. 

\text{ConcreteClass1} ::= 
\quad \{\text{“ConcreteClass1”}\} : \text{name}, 
\quad \{\text{Step1, Step2}\} : \text{methods}. 

\text{ConcreteClass2} ::= 
\quad \{\text{“ConcreteClass2”}\} : \text{name}, 
\quad \{\text{Step1, Step2}\} : \text{methods}. 

We describe the methods seen in each of the class, where Algorithm has a name, two method calls Step1 and Step2 with an access specifier abstract.

\text{Algorithm} ::= 
\quad \{\text{“Algorithm”}\} : \text{name}, 
\quad \{\text{Step1, Step2}\} : \text{methodcalls}, 
\quad \{\text{public}\} : \text{accessSpecifier}. 

\text{Step1} ::= 
\quad \{\text{“Step1”}\} : \text{name}, 
\quad \{\text{abstract}\} : \text{modifier}. 

\text{Step2} ::= 
\quad \{\text{“Step2”}\} : \text{name}, 
\quad \{\text{abstract}\} : \text{modifier}. 

The following syntax describes the method invocations involved in the CD.

\text{Step1} ::= \{\text{“Step1”}\} : \text{name} 
\text{Step2} ::= \{\text{“Step2”}\} : \text{name}
The CD represented by a syntax defined on GEBNF, can be used to express any static information. In the subsequent sections we explain how this syntax can be used to create a formalism based on first-order predicates.

5.2.3 Building predicates

The unique advantage of expressing a CD in a syntax defined on GEBNF is the implicit creation of predicates and formulas. CD is represented with a syntax which explicitly states individual elements forming it. Hence the elements of the syntax can be readily considered as a predicate or a formula. For example, the element methods in GEBNF syntax is a formula returning the list of methods seen in its parent element i.e. Class. Hence for any class c, methods(c) will return a set of method definitions that are part of the class c. This can be represented as:

\[ \text{methods}(c) \equiv \forall m : \text{Method. } m \in c \]

where, methods is the set of all methods in the CD. The formalism implies that methods(c) returns a subset of methods \( m_s \), such that each method \( m \) in \( m_s \) also belongs to the class \( c \).

methods, can have methods with same declaration (like name and variables) in the CD, but it is implicitly assumed that they do not belong to the same class. This is detected by the compiler analyzing the program.

We express a method call existing from method \( m_1 \) to method \( m_2 \) as described below.

\[ \text{calls}(m_1, m_2) \equiv m_1 \rightarrow m_2 \in \text{calls} \]

where calls is the set of all method calls in the CD and \( \rightarrow \) refers to any relationship mapped from \( m_1 \) to \( m_2 \), belonging to the calls subset.

Similar interpretation can be used for gens and assocs which show generalization (inheritance) and association between classes or interfaces. Association is the relationship between different classes or interfaces defined by object creation, namespace inclusion or static calls.

Using a similar approach we can derive predicates from the GEBNF syntax. Consider the predicate isAbstract(x), which is either true or false depending on the modifier of the element \( x \):

\[ \text{isAbstract}(x) \equiv \text{modifier}(x) \in \{\text{abstract}\} \]
The following predicate is adopted from [11], which ensures that all classes in $ys$ can be accessed only through the classes in $xs$.

$$\text{access}(xs, ys) \equiv \forall x \in \text{classes}. \forall y \in ys. x \mapsto y \in \text{deps} \Rightarrow x \in xs \cup ys$$

where, $\text{deps}$ represents a dependency i.e. either an association ($\text{assocs}$) or a generalization ($\text{geners}$).

We also introduce a formula $\text{count}(X)$ to capture cardinality of the set $X$ i.e.

$$\text{count}(X) \equiv |X|$$

where, $|X|$ represents the cardinality of the set $X$.

Consider the following example for describing the number of methods in a class $c$ and the number of input parameters for a method $m$ respectively.

$$\text{count(methods}(c)) \text{ and count(inParams}(m))$$

where, $\text{methods}(c)$ is a set of all methods contained in class $c$. and $\text{inParams}(m)$ is a set of all input parameters declared in method $m$.

For resolving an element of the CD, we use implicit functions to represent its base type. For example, to resolve a method $m$ to the $\text{Class}$ it is declared in, we use $\text{Class}(m)$.

This approach is adapted throughout our work in formalizing all design patterns. The prime advantage with such an approach is the direct correlation with implementation. We can develop programming functions mimicking the predicates or formula derived from the GEBNF syntax. This approach is explained in Section 7.3.

### 5.2.4 Formalization of standard patterns

In this section, we illustrate how we adapt formal specification of design patterns using the predicates derived from GEBNF syntax. The motive of this section is to formally specify design patterns and not to find violations while implementing any, as our research aims at. However, the technique used in formalizing any design guideline remains the same. For building a good understanding, we take the examples of the GoF design patterns listed in Section 4.4.
Builder design pattern

In Section 4.4.1, we have discussed the Builder pattern with suitable examples. The specification of the design patterns is often cumbersome to explain in an efficient and precise manner. We use formulae formed on the basis of first-order predicate logic to describe it.

Guidelines:
The specification of the design patterns is best understood when it is split into a number of guidelines that the pattern must adhere to. Hence a conjunction of all such guidelines will describe the design pattern fully.

1. The Director class must provide a generic method for concrete object creation. This means that Director must have a method accepting IBuilder as its input parameter and should return the concrete Product.

\[
\forall \text{Director} : \text{Class.} \exists \text{IBuilder} : \text{Interface.} \text{assoc}(\text{Director, IBuilder}) \land \exists \text{construct} \in \text{methods(Director).} \exists v \in \text{inParams(construct).type}(v) = \text{IBuilder} \Rightarrow \exists \text{Product} : \text{Class. returnType(construct)} = \text{Product}
\]

2. IBuilder provides an interface using which specific implementations can be generically handled. This means that the IBuilder interface must have at least one method signature returning the concrete Product type to be called via the Director class.

\[
\forall \text{Director} : \text{Class.} \exists \text{IBuilder} : \text{Interface.} \exists \text{construct} \in \text{methods(Director).} \exists m \in \text{methods(IBuilder). calls(Construct, m)} \Rightarrow \exists \text{Product} : \text{Class. returnType(m)} = \text{Product}
\]

3. The client class must never directly create the concrete Product (Monitor in the example), hence it must always be via the Director and the IBuilder implementations. This can be captured by the access that we have defined in Section 5.2.3

\[
\forall \text{Director} : \text{Class.} \exists \text{Product} : \text{Class. access(\{Director\}, \{Product\})}
\]

Adapter design pattern

In Section 4.4.2, we discussed Adapter pattern with suitable examples. We can now formalize this guideline as discussed in the Section 5.2.4
Guidelines:

1. `ITarget` provides an abstract interface for the `Client` to access the `Adapter` class.

\[ \forall ITarget : Interface. \exists Adapter : Class. \text{geners}(Adapter, ITarget) \Rightarrow \exists Client : Class. \text{assocs}(Client, ITarget) \]

2. The `Client` class calls a method in the `Adapter` class \((m2)\), which adapts the `Adaptee` by either inheriting it (class `Adapter`) or instantiating it (object `Adapter`). However, we are not concerned about how `Adapter` accesses the `Adaptee`, hence we only describe the `Adapter` method calls the specific declaration of `Adaptee` method (spReq).

\[ \exists m1 \in \text{methods}(Client). \exists m2 \in \text{methods}(Adapter). \text{calls}(m1, m2) \Rightarrow \exists spReq \in \text{methods}(Adaptee). \text{calls}(m2, spReq) \]

3. The `Client` does not access the `Adaptee` directly, it is always via the `Adapter`. We use the access predicate defined previously to describe this.

\[ \text{access}\{\{Adaptee\}, \{Target, Adapter\}\} \]

Template method design pattern

In Section 4.4.3 with an example, we had discussed how Template Method pattern helps in defining a generic way to access the concrete class functions. We can now formalize this guideline as:

Guideline:
If all abstract classes \((aClass)\) having template methods \((t)\), then that template method should call at least one abstract method \((m)\) declared in the same abstract class.

\[ \forall aClass : Class. \forall t \in \text{methods}(aClass). \text{isAbstract}(aClass) \Rightarrow \exists m \in \text{methods}(aClass). \text{calls}(t, m). \text{isAbstract}(m) \]
In this chapter, we will specify the design pattern guidelines that need to be verified at Philips and adopt the formalization techniques discussed in Chapter 5. These design patterns form the basis of our research and we describe these patterns similar to the standard design patterns in Section 4.4. In the following subsections we will discuss this technique by considering design patterns like the Dispose pattern, the Model View ViewModel pattern and the Dependency property pattern, commonly seen in Philips architecture. The first pattern we will be discussing in Section 6.1 is the Dispose Pattern, which is followed in the architecture to explicitly free the resources and improve performance of the system. In Section 6.2 we will discuss few guidelines of an architectural pattern called MVVM, which is followed to maintain separation of concerns between different functionalities. In Section 6.3 we will discuss the guidelines to be followed to implement a Dependency property.

6.1 Dispose pattern

Implementing proper resource management can be difficult and may distract the developer from the real problems he is working on. To give the developer freedom from managing the resources, the .Net framework provides automatic garbage collection. In object oriented programming every class needs some resources and to use these resources we need to ensure that sufficient memory is allocated. The memory allocated for a resource is represented by the class in which it is used. The resource for which the memory is initialized is set to an initial state and is used by the instances of the class. Once the instance completes its task with the resource, the resource needs to be released. After releasing the resource, the garbage collector will free the memory associated with it. Mistakes like not freeing the memory when no longer required and the attempt to use the memory after it has been freed can occur during the resource management. The
consequence of these two mistakes may lead to resource leaks [21]. In this section we discuss different ways available to handle this issue.

**Garbage Collector:**
The garbage collector is the solution to avoid these mistakes as it will automatically free the memory when no longer required. Any resource that is used in the program needs memory and is allocated from the managed heap. We use the `new` operator to create an instance of a class. If the managed heap has space for the new instance, its constructor is called and the `new` operator allocates the space and returns the pointer to the allocated space. This allocation scheme runs on an assumption that there is an infinite address space and storage [21]. But this assumption is false in real life. The managed heap makes use of the garbage collector to strengthen this assumption. When the heap detects that there is no space to allocate memory for a new object, it uses the garbage collector to free the objects that are no longer required. To improve the performance, the managed heap is divided into parts and each part is called a generation [22]. During the garbage collection, the garbage collector frees the memory in parts ensuring the fraction of the heap to be freed each time. Most of the objects that we create in our program can rely on garbage collector for memory management. If the object encapsulates unmanaged resources then it is necessary to release it explicitly [23]. The unmanaged resources in our current codebase may be the DirectX or OpenGL artifacts.

**Finalizer:**
The garbage collector knows when to reclaim the memory owned by an object but does not have specific information about how to clean up the resources. During the garbage collection an optional member that a garbage collector calls for this task is the finalizer. The finalizer has the following syntax structure:

```
~ClassName ()
{
    // . . .
}
```

Listing 6.1: Finalizer declaration in C#

If an object needs finalizer to be called during garbage collection then it should explicitly be implemented in the class. The garbage collector maintains a finalization queue [21] to keep track of the objects having Finalize methods. The problem with garbage collection is that it knows the instance whose memory has to be freed but is unaware of the state of the resource in that instance. To properly free the memory of an instance and the resources in that instance, the developer has to explicitly write the code for resource clean-up. The developer can place the code for resource cleanup in the finalizer.

**Pattern name:**
Dispose Pattern.

Problem:
Whenever an object having a finalizer is created, the garbage collector makes an entry in the finalization queue that points to the object. Therefore the garbage collector maintains the information of all the objects that need to be finalized in the finalization queue. If the garbage collector finds a dead object that needs to be finalized, it does not reclaim the space for that object at that time. Instead of reclaiming the space, the garbage collector will remove the entry for the object from the finalization queue and add an entry to the list of objects that are marked as ready for finalization. As the garbage collector has not reclaimed the memory, the pointers within the object are valid till finalization is complete. After garbage collection the finalization thread will go through these objects that need finalizer to be invoked. Once these objects complete the invocation of the finalizer they become ready for garbage collection and the memory associated with these objects is later reclaimed. So an object that needs to invoke finalizer lives longer than objects that do not need finalization. Due to the internal pointers of these longer lived objects, the objects referred from these pointers also live long. The finalization thread has more work if an object needs finalizer, this could also lead the thread to spend more time when there is complex code in the finalizer. So the complex code in the finalizer will also pile up other objects from being freed as there is a single finalization thread for this job [22].

Solution:
It is not a good idea to depend on garbage collector to free resources using finalizer, especially for some applications which runs continuously without restart. The disadvantages of finalization can be avoided by implementing the dispose pattern. The dispose pattern is an alternative way to free the resources an object owns rather than using finalization. The Dispose method is an explicit way to free resources and the finalization is an implicit way. Providing an explicit way to free resources can release the resource before garbage collector attempts to free the object.
The guidelines in Section [6.1.1] are followed as a solution to the problem.

6.1.1 Dispose pattern guidelines

In this section, we discuss the guidelines that provides a standard design solution for the problems discussed in Section [6.1]. These guidelines ensure that the Dispose pattern is implemented as designed. The detection of any design pattern involves static analysis of the source code. The formalism defined to capture the design patterns precisely from source code will involve low-level descriptions. In order to abstract low-level details of the formalism, we use the following seven predicates:

1. isDisposeVoid(m):
This predicate is used to check if the method \( m \) is a \textit{Dispose} method, without any input arguments. See Appendix \ref{appendix:disposable} for details.

2. \textbf{isDisposeBool}(\( m \)):
   This predicate is used to check if the method \( m \) is a \textit{Dispose} method, with only one input argument of type \textit{bool}. See Appendix \ref{appendix:dispose-bool} for details.

3. \textbf{hasDisposeVoid}(\( c \)):
   This predicate is used to validate if there exists a \textit{Dispose} method without any input arguments in class \( c \). See Appendix \ref{appendix:dispose-void} for details.

4. \textbf{hasDisposeBool}(\( c \)):
   This predicate is used to validate if there exists a \textit{Dispose} method with only one input argument of type \textit{bool} in class \( c \). See Appendix \ref{appendix:dispose-bool} for details.

5. \textbf{isDispose}(\( m \)):
   This predicate is used to validate if the method \( m \) is a \textit{Dispose} method. So it can either be a \textit{Dispose(\textit{void})} or a \textit{Dispose(\textit{bool})} method. See Appendix \ref{appendix:dispose} for details.

6. \textbf{hasDispose}(\( c \)):
   This predicate is used to validate if there exists a \textit{Dispose} method in class \( c \). The method \( m \), can either be a \textit{Dispose(\textit{void})} or a \textit{Dispose(\textit{bool})} method. See Appendix \ref{appendix:dispose} for details.

7. \textbf{hasIDisposable}(\( ci \)):
   This predicate is used to validate if classifier \( ci \) or any of its parent classifiers implement the \textit{IDisposable} interface. A classifier \( ci \) can be either a class or an interface. See Appendix \ref{appendix:dispose} for details.

Formal expressions to define the above seven predicates are explained in Appendix \ref{appendix:dispose}.

Following are the guidelines that are formulated using the above predicates as the solution to the problems described in Section 6.1.

1. Implement \textit{IDisposable} if there is a \textit{Dispose} method.
   \textbf{Problem:} How to abstract the implementation of the \textit{Dispose} method of a class to an interface?
   How to allow all Disposable objects to be treated in the same way for disposal?
   \textbf{Solution:} Microsoft provides an \textit{IDisposable} interface to implement any classes having resources to be disposed. If there is a \textit{Dispose} method in a class then there should be an \textit{IDisposable} interface implemented in it or any of the parent classes or interfaces.
   \textbf{Example:}
   - Listing \ref{listing:violation} shows a violation of the guideline due to the unavailability of
Listing 6.2: Missing IDisposable interface

- Listing 6.3 shows a class implementing an interface IExample, which implements the IDisposable interface. This adheres to the design guidelines.
Listing 6.3: Correct implementation of IDisposable interface

Figure 6.1: Class diagram for implementing IDisposable

Figure 6.1 shows the class diagram showing IDisposable interface is implemented in the class having the Dispose method. The child class, Child and the parent class, Parent have IDisposable interface as one of their ancestors.

**Formalization:**
The predicate developed for identifying the IDisposable interface and the Dispose method in a class can be used here to make the description clear. This formalization verifies for all classes if the class has a Dispose method then it should have an IDisposable interface implemented.

\[ \forall c : \text{Class. } hasDispose(c) \Rightarrow hasIDisposable(c) \]

**Consequence:** All the objects deriving from the IDisposable interface can be treated in the same way for Disposal.

2. Define Dispose bool.

**Problem:** How to free resources explicitly and share the code required for disposing?

**Solution 1:** The Dispose(bool) method should be implemented if a class has a
finalizer. If the class is a sealed class and it does not have a finalizer but has few resources to be released, then Dispose(bool) method is not necessary. It can implement the Dispose method to release the resources. A class that cannot be extended from any other class is termed as a sealed class. In C# a keyword sealed is used to define a class as a sealed class.

**Example:**

- Listing 6.4 has no violation as the class is sealed. Since there is no finalizer implemented in this class, we do not need to implement a Dispose(bool) method. But in case of an unsealed class this is a Violation because it should have a Dispose bool method defined.

```csharp
// missing Dispose(bool) but not a violation.
sealed class SealedClass : IDisposable
{
    public void Dispose()
    {
        ...
    }
}
```

Listing 6.4: Sealed class having only Dispose(void) method

![Figure 6.2: Sealed class has a finalizer and a Dispose(bool)](image)

Figure 6.2 depicts a sealed class having a Dispose(bool) method and a finalizer. The Dispose(bool) method is required only because of the presence of the finalizer.

- Listing 6.5 has a sealed class named `SealedClass`, which has a finalizer but is missing a Dispose(bool) method. The guideline is violated in this case. Even if the class was not sealed there should be a Dispose bool method with the finalizer. So this is a violation in case of an unsealed class also.

```csharp
// missing Dispose(bool) hence a violation as there is a finalizer.
sealed class SealedClass : IDisposable
{
    public void Dispose()
}
```

Listing 6.5: Sealed class having only Dispose(void) method
6.1. Dispose pattern

Listing 6.5: Sealed class with a finalizer but without Dispose(bool)

```csharp
sealed class SealedClass
{
    public SealedClass()
    {
    }
}
```

Figure 6.3: Sealed class does not have a Dispose(bool) method

In Figure 6.3, `Class1` is a sealed class and does not have a `Dispose(bool)`. Since `Class1` does not have a finalizer, the guideline is not violated.

**Formalization:**
For all classes, if there is a finalizer in the class then there should be a `Dispose(bool)` method definition in the class.

\[ \forall c : \text{Class}. \ isFinalized(c) \Rightarrow \exists m : \text{Method}. \ isDisposeBool(m) \]

3. Implement `IDisposable`.

**Problem:** How to free resources explicitly?

**Solution 2:** Any class that defines a finalizer should implement the `IDisposable` interface.

**Example:**
- Listing 6.6 shows a violation as the `Dispose(bool)` method is not implemented even in the presence of finalizer.

```csharp
public class Class1
{
    ...
    public Class1()
    {
    }
}
```
Listing 6.6: Dispose method missing in the presence of a finalizer

```
{ ...
}
// IDisposable should be implemented if there is a
// finalizer defined.

~Class1()
{
    Dispose(false)
    ...
}
```

- Listing 6.7 has no violation due to implementation of the Dispose method with the finalizer.

```
public class Class1 : IDisposable
{
    public Class1()
    {
        ...
    }
    protected virtual void Dispose(bool disposing)
    {
        ...
    }
    public void Dispose()
    {
        ...
    }
    // This will run only if Dispose is not called
    ~Class1()
    {
        ...
    }
}
```

Listing 6.7: Both Dispose method and finalizer present

Figure 6.4 shows a class diagram with a finalizer in Class1, implementing the IDisposable interface.

**Formalization:**
If there is any class c that has a finalizer, then it should also have IDisposable
6.1. Dispose pattern

∀c: Class. isFinalized(c) ⇒ hasIDisposable(c)

Consequence: By implementing IDisposable and Dispose method with the finalizer, we can explicitly free the resources and reuse the code used to dispose. Using finalizer will make release of resources unpredictable, as it may be called any time after the resources are available for releasing. But calling Dispose method would free the resources sooner than the finalizer and is deterministic, hence it is a good design guideline.

We need not implement this guideline as the guideline 1 and 2 will imply this guideline as shown below.

∀c: Class. isFinalized(c) ⇒ ∃m: Method. isDisposeBool(m, c)
∀c: Class. ∃m ∈ methods(c). isDispose(m) ⇒ hasIDisposable(c)
by transitivity we have
∀c: Class. isFinalized(c) ⇒ hasIDisposable(c)

4. Define override Dispose bool method.

Problem: How to encapsulate the logic required to free the resources?
Solution: For an unsealed class that implements the IDisposable interface, there should be a Dispose(bool) method that is either virtual or overridden from the parent class. All the logic required to free the resources should be implemented in this method.

Example:

- In Listing 6.8, ClassOne violates the guideline as the class is unsealed and there is no definition of the Dispose(bool) method.
public ClassOne()
{
    this.reader = new TextReader();
}

// There must be a virtual or an overridden Dispose(bool) method.
public void Dispose()
{
    reader.Dispose();
    GC.SuppressFinalize(this);
}

Listing 6.8: Unsealed having an IDisposable interface and missing Dispose(bool)

- In Listing 6.9, ClassOne does not violate the guideline as there is a virtual Dispose(bool) method.

```csharp
public class ClassOne : IDisposable
{
    private TextReader reader;
    private bool disposed = false;
    public ClassOne()
    {
        this.reader = new TextReader();
    }
    // This is a virtual method. Hence guideline satisfied.
    protected virtual void Dispose(bool disposing)
    {
        ...
        reader.Dispose();
        ...
    }
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
}
```

Listing 6.9: Declaration of virtual Dispose(bool)

Figure 6.5 shows the Parent class implementing the IDisposable interface and has a virtual Dispose(bool) method, which is overridden in the Child class satisfying the guideline.
6.1. Dispose pattern

Figure 6.5: The Child class overrides the Dispose(bool) method from the Parent Class

**Formalization:**
For all unsealed classes implementing IDisposable, the class should have a Dispose bool method and it should be virtual or overridden.

\[ \forall c : \text{Class. } \neg \text{isSealed}(c) \land \text{hasIDisposable}(c) \Rightarrow \exists m : \text{Method. } \text{isDisposeBool}(m) \land (\text{isVirtual}(m) \lor \text{isOverride}(m)) \]

**Consequence:** By having a Dispose method in a class we can free resources explicitly and need not wait for the garbage collector. By providing a virtual method, the child class can also implement the Dispose method and free the resources.

5. Call Dispose(false) from the finalizer.

**Problem:** How to reuse the logic used to free the resources?

**Solution:** If the class implements an IDisposable interface and defines a finalizer then the finalizer should call the Dispose(false). The argument false in the method call Dispose(false) indicates that the method call is from the garbage collector.

**Example:**
• Listing 6.10 has a class `Class1` which satisfies the guideline by calling the method `Dispose(bool)` with `false` as an argument.

```csharp
public class Class1 : IDisposable
{
    public Class1()
    {
        ...
    }
    protected virtual void Dispose(bool disposing)
    {
        ...
    }
    public void Dispose()
    {
        Dispose(true);
        // stop garbage collector from calling finalizer
        // if Dispose is already called.
        GC.SuppressFinalize(this);
    }
    ~Class1()
    {
        // This call should be made indicating the call is from
        // the garbage collector.
        Dispose(false);
    }
}
```

Listing 6.10: Finalizer calls Dispose(bool) with false as an argument

Figure 6.6: The class `Class1` has a finalizer and a Dispose(bool)

Figure 6.6 shows the class `Class1` which has a finalizer and the `Dispose(bool)` method. In this case, the finalizer of `Class1` should call the `Dispose(bool)` method with `false` as an argument. The sequence diagram in Figure 6.7 shows the method call scenario, where the garbage collector calls the finalizer and the finalizer calls the `Dispose(bool)` method.

Formalization:
For every class $c$ implementing `IDisposable` and has a finalizer, must call `Dispose(false)` in its finalizer.

$$\forall c : \text{Class}. \ (\text{isFinalized}(c) \land \text{hasIDisposable}(c) \Rightarrow (\exists mc : \text{Methodcall}. mc \in \text{Finalizer}(c) \land \text{name}(mc) = "Dispose" \land \text{inParam}(mc) = \text{false}))$$

**Consequence:** By calling the Dispose method from the finalizer, we can reuse the logic required to free the resources between the explicit cleanup and the implicit cleanup from the garbage collector.


**Problem:** How to avoid the finalizer if we have freed the resources explicitly?

**Solution:** If a class has a `Dispose` method and a finalizer, then the `Dispose` method should call `GC.SuppressFinalize()`. Calling `GC.SuppressFinalize()` would ensure that the garbage collector will not call the finalizer, if the developer has called the `Dispose` method explicitly. This saves the time required for garbage collection.

**Example:**
Listing [6.10] has a `Dispose(void)` method in `Class1` which calls `GC.SuppressFinalize()` to stop garbage collector from calling finalizer.

If a class has a `Dispose(bool)` method and a finalizer, then the Dispose method should call the `GC.SuppressFinalize` as shown in Figure [6.8]
For every finalized class $c$ having a Dispose method, there should be a method call $\text{SuppressFinalize}$, in the $\text{Dispose}$ method.

$$\forall c : \text{Class}. \forall m_1 \in \text{methods}(c). \, (\text{isDispose}(m_1) \land \text{isFinalized}(c) \Rightarrow \exists m_2 : \text{methods}. \, \text{calls}(m_1, m_2), \, \text{name}(m_2) = \text{"SuppressFinalize"})$$

**Consequence:** Finalizer will be called only if the Dispose method is not called, this helps in faster garbage collection.

7. Call base class Dispose method.

**Problem:** If the Dispose(bool) method in a class is overridden, then an explicit call should be made to the Dispose(bool) method in the parent class to free its resources.

**Example:**

- In Listing 6.11 an overridden Dispose(bool) method is calling its base class Dispose(bool) method satisfying the guideline.

```csharp
protected override void Dispose(bool disposing)
{
    // check if Dispose has already been called.
    if (!disposed)
    {
        if (disposing)
        {
            // Dispose Managed Resource
        }
        // Dispose Unmanaged Resources
    }
    disposed = true; // indicates Dispose is called for this instance.
```
6.1. Dispose pattern

Figure 6.9: The overridden Dispose bool method should call the base Dispose bool method.

```
base.Dispose(disposing)
}
```

Listing 6.11: A overridden Dispose(bool) method is calling its parent Dispose(bool).

Figure 6.9 shows the overridden Dispose bool method in the Child class and Figure 6.10 shows the method call from the overridden Dispose bool.

**Formalization:**
First we define another predicate for describing a method call \( mc \) whose declaration exists in another class \( c \).

\[
\text{hasMethodDecl}(mc, c) \equiv \text{instance}(mc) = c
\]

This predicate can be read as *method call mc has declaration in class c*. Similarly a base class method call can be formalized as:

\[
\text{hasBaseMethodDecl}(mc, c) \equiv \\
\forall c_1 : \text{Class. } \forall mc : \text{Methodcall. } \text{hasMethodDecl}(mc, c_1) \Rightarrow \text{geners}(c_1, c)
\]

We now define the following formalization for the guideline: \( \forall c_1 : \text{Class. } \forall m \in \text{methods}(c_1). \text{isDisposeBool}(m) \land \text{isOverride}(m) \Rightarrow \exists mc \in \text{methodcalls}(m). \text{name}(mc) = "\text{Dispose}" \land \text{hasBaseMethodDefn}(mc, c_1) \)

**Consequence:** Base class resources will be disposed effectively while disposing
52 Chapter 6. Philips specific design patterns

Figure 6.10: Shows two sequence diagrams, where the Child class Dispose bool method of the parent class.

derived class resources.

8. Dispose disposable types

**Problem:** How to dispose the disposable types owned by a class?

**Solution:** All member fields of the Disposable type owned by a class must be explicitly disposed. This can be done by calling the corresponding Dispose method for that field in the Dispose bool method of the class. The Dispose bool method can call other methods which in turn disposes the owned disposable fields.

**Example:**

- In Listing 6.12, Class2 has two disposable fields (obj1, obj2) and a non-Disposable field (nonDisposable1). In this case, obj1 is explicitly disposed by calling obj1.Dispose() in Dispose(bool) method of Class2. Since Dispose(bool) in Class2 calls Dispose method of Class1 for the instance obj1, the guideline is satisfied in this case. Also Dispose bool in Class2 calls Cleanup2 which in turn calls obj2.Dispose(). This ensures correct implementation of the guideline.

```csharp
// NonDisposable class does not have Dispose pattern implemented.
public class NonDisposable
{
    ...
}
```
public class Class1 : IDisposable
{
    // Member fields. (having resources that need to be Disposed).
    ...
    protected virtual void Dispose(bool disposing)
    {
        ...
    }
    public void Dispose()
    {
        ...
    }
}

public class Class2 : IDisposable
{
    // Dispose method call is not needed for nonDisposable1.
    private NonDisposable nonDisposable1;

    // Dispose method call is needed for obj1
    private Class1 obj1;
    private Class1 obj2;
    private bool disposed = false;

    public void Clean2()
    {
        ...
        obj2.Dispose();
        ...
    }
    public void Dispose()
    {
        ...
    }
    protected virtual void Dispose(bool disposing)
    {
        if (!disposed)
        {
            if (disposing)
            {
                // Dispose method is not called for nonDisposable1 but obj1 has one
                // Dispose() call as Class1 implements Dispose pattern.
                if (obj1 != null)
                {
                    obj1.Dispose();
                    // Clean2 calls Dispose method for obj2.
                    Clean2();
                }
            }
            disposed = true;
        }
    }
Listing 6.12: Instances of Class1 in Class2 are explicitly disposed in Class2 by calling Dispose method of Class1.

Figure 6.11: Class2 owns an instance of Class1 and needs to dispose this instance.

Figure 6.11 shows the association between the class Class2 and the class Class1. Since Class1 implements dispose pattern, Class2 should call the Dispose method of Class1 for the instance of Class2 used in Class1. The call from Class2 to Dispose method of Class1 is shown in Figure 6.12.

Formalization:

If there is any disposable objects created in a class \( c \), it must be explicitly disposed in the Dispose(Bool) method. We initially define a recursive formula \( \text{CallsDispose}(m) \) to specify if a method \( m \) eventually calls a Dispose method.

\[
\text{CallsDispose}(m_1, m_2) \equiv \exists (\text{calls}(m_1, m_2) \land \text{isDisposeVoid}(m_2)) \\
\lor \exists m_3 : \text{Method}. \text{CallsDispose}(m_2, m_3)
\]

The formalism below states that if there is any Disposable variable \( v \) in any Class \( c \) then the DisposeBool method of \( c \) should call the Dispose method for that variable \( v \).

\[\forall c : \text{Class}. \forall v \in \text{variables}(c). (\text{hasIDisposable}(\text{Class}(v)) \Rightarrow (\exists m_1 \in \text{methods}(c). \text{isDisposeBool}(m_1) \land \exists m_2 : \text{Method}. \text{CallsDispose}(m_1, m_2) \land \text{Class}(m_2) = \text{type}(v)))\]

Consequence: By calling the Dispose method of the field in the Dispose bool method, the code can be reused for explicit cleanup and cleanup through the finalizer. This guarantees the disposal of the Disposable fields in the class.
6.2 MVVM Pattern

Background Details:
The iXR viewing architecture developed at Philips is built using Windows Presentation Foundation (WPF). WPF is used to build user interfaces in an application. In most of the recent applications WPF has replaced Windows Form. The key features of WPF that makes it different from Windows Form are XAML, command binding and dependency property. XAML (Extensible Application Markup Language)\[24\] is a markup language supported by WPF to build user interface. Data binding binds a control in user interface with a property in the code. Command binding allows reuse of logic by binding the user actions in the user interface to the commands in the code. More details about WPF and its features is provided in Appendix A.1. These key features of WPF allows a clear separation between the user interface and the code-behind. In this section we discuss

Figure 6.12: Class2 will call the Dispose method of Class1 to dispose the instance of Class1 in class2.

A simple example is presented in Appendix A.4 which follows all the guidelines of the Dispose pattern. To ensure correct implementation of the Dispose pattern, all the guidelines listed in Section 6.1.1 should be followed.
about an architectural design pattern emerged from the key features of WPF.

Problem:

- How to separate the user interface functionality from the business logic?
- How to allow independent modifications to each part of the application?
- How to provide independence to the user interface designer from the code and the application developer from the user interface related issues?
- How to share the business logic among multiple user interfaces?
- How can we implement the application so that independent components can be tested easily?
- How to have a View layer without code-behind or very less code behind?
- How to reuse the logic among different View?
- How to utilize the key features of WPF to solve above problems?

Name: Model View ViewModel (MVVM).

Solution: MVVM \cite{25} is an architectural design pattern that is supported by WPF applications. MVVM has three layers, The Model, View and the ViewModel as shown in Figure 6.13.

Model Layer The Model Layer is the core layer and contains the business logic of an application. The Model layer is independent of the other two layers and it has no knowledge of the other layers. So the changes in the other layers are not affected to the Model layer.

View Layer In MVVM, the View layer has clear separation between presentation and logic. This separation is possible due to the key features provided by XAML. In WPF applications most of the code in View layer is in XAML, this allows the designers to create a user interface with very less or no code behind. Complete code of the user interface will be in declarative XAML tags. This feature of less code-behind allows the designer to improve the user interface without affecting the code in other layers. In MVVM, the designers have the freedom to improve the user interface because the View layer does not have any presentation logic or code behind. The View layer knows the ViewModel layer and is bound to few properties and commands in the ViewModel layer. The data binding and command binding provided by WPF allows the View to be the observer of
the ViewModel. Here the View is the observer of ViewModel but ViewModel is unaware of the View. This feature makes MVVM different from other design patterns discussed in Appendix A.1.5

**ViewModel Layer**
The ViewModel layer lies in between the Model layer and the View layer and it manages the state and the presentation logic of the application.

The main task of ViewModel layer is to adapt Model elements so that they can be used in the View layer. The View layer is the observer of the ViewModel layer and it binds to the ViewModel layer properties. The ViewModel layer has the responsibility to expose the Model layer data to the View. For safety of the Model layer data, the ViewModel should not expose the Model layer elements directly but provide its own properties that expose the Model. By exposing its own properties for the View, the ViewModel will keep the separation between View and the Model layer. To notify the change in property values in ViewModel layer to the data bound elements of View layer, the ViewModel layer implements `INotifyPropertyChanged` interface. This interface exposes a `PropertyChanged` event, which notifies the change in the property values in the ViewModel layer to the View layer elements. MVVM provides a command binding between the View and the ViewModel. The command binding is used whenever a command needs to be executed in response to the user action. The commands for user action are placed in the ViewModel layer. The advantage of having commands in View layer is the possibility to test the commands separately in the absence of View and also the commands can be reused among multiple Views. The command binding and the provision for creating
custom commands in WPF has replaced code required to execute few actions in the View layer by commands in ViewModel layer. If the application needs only the inbuilt commands then the command binding to inbuilt commands can be done in View layer and ViewModel layer need not have any commands. The inbuilt command binding can be done in the XAML code and no logic is needed in the code-behind. If the application needs some custom commands then they should be implemented in ViewModel layer. The commands can be implemented in the ViewModel layer by encapsulating the commands with the ICommand interface or by using the RoutedCommand or the RoutedUICommand class. These are the options provided by WPF for creating our own commands. The alternative way to create custom commands in ViewModel layer is by using the RelayCommand. If the commands are created using ICommand interface in ViewModel then they should be exposed as public properties to make commands available for binding.

Consequence: In MVVM pattern, the Model layer has no references to other layers and is independent by itself. So the business logic is separated from the presentation logic. The ViewModel layer is also independent from the View layer and the modifications in the View will not be affected to any other layers. This enables the reuse of the Model layer and the ViewModel layer. These two layers can be tested independently and are easy to maintain. Most of these advantages of MVVM pattern are through effective utilization of WPF features. So the drawback of this pattern is, it requires a complex framework that supports binding and a mark up language like XAML which eliminates the need for code-behind.

The IXR architecture at Philips Healthcare is based on the MVVM pattern. Few guidelines that are followed in the architecture are listed in Section 6.2.1.

6.2.1 MVVM design guidelines

1. The ViewModel Layer should access the Model layer through Model layer interfaces.

   Problem: How to provide multiple implementations of the Model?
   How to hide the implementation details of the Model from the above layers?

   Solution: The Model layer classes should be programmed to the interface and the ViewModel layer should access the Model layer through the interfaces as shown in Figure 6.14. By accessing the Model layer through the interfaces, the implementation details of the Model layer will be hidden from the ViewModel layer. The Model layer interfaces forms the abstraction of the Model layer reducing the complexity of the architecture and hiding the implementation details.
6.2. MVVM Pattern

Figure 6.14: The ViewModel layer should access the Model layer through the Model layer interfaces.

- In Listing 6.13 the Model layer class `ModelClass` is directly used in the ViewModel layer violating the guideline.

```java
// ViewModel Layer
public class Sample1ViewModel : ViewModel
{
    // ModelClass is a Model Layer class, therefore this is a violation.
    private ModelClass domainModel;
    ...
    public SampleViewModel(ModelClass domainModel)
    {
        SampleModel = domainModel;
    }

    public DoCalculations()
    {
        // use domainModel for some calculations.
    }
    ...
}

// Model Layer
public class ModelClass
{
    // ...
```
Listing 6.13: The ViewModel layer directly accesses the Model layer class.

- In Listing 6.14, the Model layer element is accessed through the interface `IModelClass` satisfying the guideline.

```java
// ViewModel Layer

public class Sample1ViewModel : ViewModel
{
    // IModelClass is a Model Layer interface.
    private IModelClass domainModel;
    ...
    public SampleViewModel(IModelClass domainModel)
    {
        SampleModel = domainModel;
    }

    public DoCalculations()
    {
        // use domainModel for some calculations.
    }
    ...
}

// Model Layer

interface IModelClass
{
}

class ModelClass: IModelClass
{
    // ...
}

class DomainDataModel: IModelClass
{
    // ...
}
```

Listing 6.14: Correct implementation of the guideline.

**Formalization:**
This guideline states that none of the ViewModel layer class must access any Model layer class. This can be formally specified as follows:
∀c₁, c₂ : Class. isViewModelLayer(c₁) ∧ isModelLayer(c₂) ⇒ ¬deps(c₁, c₂)

However a Model layer Class can be accessed by a ViewModel class via an interface belonging to the Model layer. This can be specified as:

∀c₁ : Classifier. isViewModelLayer(c₁) ∧ (∃c₂ : Classifier. isModelLayer(c₂) ∧ deps(c₁, c₂)) ⇒ isInterface(c₂)

Consequence: By abstracting the Model through its interfaces, we can have multiple implementations of the Model and hide the implementation details of the Model.
For example, if the Model accesses a file and a database and it is abstracted through the interface then the ViewModel will use the interface to access the database or the file. The ViewModel layer elements will not know whether the file is accessed or the database. Thus hides the implementation details of the Model.

2. Independent Model layer.

Problem: How to implement algorithms in Model layer independent of the user interface issues and provide the developers freedom from user interface issues?

Solution: The Model layer elements should not have any references to the ViewModel layer or the View layer as shown in Figure 6.15. Since the business logic of an application is implemented in the Model layer, the Model layer should be independent of other layers. So that the changes in above layers will not affect the Model layer.

- In Listing 6.15, the Model layer class DomainModel has reference to the ViewModel layer class SampleViewModel and the View layer class WPFControlClass. So the guideline is violated in this example.
Formalization:
This guideline states that none of the Model layer classifiers must have a dependency with any of the View or ViewModel layer classifiers. This can be formally specified as follows:

\[
\forall c_1 : \text{Classifier.isModelLayer}(c_1) \land ((\exists c_2, c_3 : \text{Classifier.isViewLayer}(c_2) \land \text{isViewModelLayer}(c_3)) \Rightarrow \neg (\text{deps}(c_1, c_2) \lor \text{deps}(c_1, c_3)))
\]

Consequence: The developers need not worry about the user interface issues and the business logic is independent of the affects of other layers.

3. Separation between the ViewModel and the View layer.
   Problem: How to test the presentation logic independent of the View?
   How to reuse the ViewModel among multiple Views?
   How to restrict the affects of changes in the View to the other layers?

Solution: There should be separation between elements of the View layer and the ViewModel layer. In the MVVM pattern, the ViewModel layer can interacts with the Model layer and it should not have any knowledge of the View layer. The View layer elements will have references to the ViewModel layer, but the ViewModel layer should not have any references to the View layer as shown in Figure 6.16. The use of the command binding and the data binding between the View layer and the ViewModel layer allows the ViewModel layer to be free from the View layer references.

- In Listing 6.16, the ViewModel layer class SampleViewModel has a reference
6.2. MVVM Pattern

Figure 6.16: The ViewModel layer should not have any reference the View layer.

to a View layer class \textit{WPFControlClass} thus violating the guideline.

\begin{verbatim}
public class SampleViewModel : ViewModel
{
    // This is not a violation as SampleViewModel class is belongs to
    // the ViewModel layer.
    private SampleViewModel object1;
    ...

    // Violation of the guideline as there is a reference to a
    // WPFControlClass class which belongs to the View Layer in the
    // IXR architecture.
    private WPFControlClass object2;
    ...
}
\end{verbatim}

Listing 6.16: The ViewModel layer class has a reference to the View layer class.

\textbf{Formalization:}
This guideline states that none of the ViewModel layer classifier must have a
dependency with the View layer classifier. This can be formally specified as follows:

\[ \forall c_1 : \text{Class.} \; (\text{isViewModelLayer}(c_1) \land \exists c_2 : \text{Class.} \; \text{isViewLayer}(c_2) \Rightarrow \neg \text{deps}(c_1, c_2)) \]

\textbf{Consequence:} Separating the ViewModel layer from the View layer will allow
multiple Views to reuse the ViewModel layer elements. The ViewModel can be
easily tested without a user interface. Also the affects in the user interface will not
cause any affects to ViewModel layer elements.

4. No references from the View to the Model layer.
\textbf{Problem:} How to keep changes in the View independent of the Model layer?

\textbf{Solution:} The View layer should not have any direct references to the Model as
Figure 6.17: The View layer should not have direct reference to the Model layer.

shown in Figure 6.17. Any changes in the View should be propagated through the ViewModel.

- The View layer class in Listing 6.17 has a direct reference to the Model layer class by setting the View data context to a Model layer instance, thus violating the guideline.

```csharp
// VIEW LAYER
//
public partial class ViewClass
{
    public ViewClass()
    {
        InitializeComponent();
        // The use of DomainModel class is a violation here as it belongs
        // to the Model Layer.
        DataContext = new DomainModel();
    }
    ...
}
```

Listing 6.17: The View layer class has a direct reference to the Model layer class.

- In Listing 6.18, the View sets its data context to a ViewModel class following the guideline.

```csharp
// VIEW LAYER
//
public partial class ViewClass1
{
```
6.2. MVVM Pattern

Listing 6.18: The guideline is not violated in this example.

Formalization:
This guideline states that none of the View layer classifiers must have a direct dependency with a Model layer classifier. Such a dependency should be via a ViewModel layer classifier. This can be formally specified as follows:

∀c1 : Classifier. (isViewLayer(c1) ∧ (∃c3 : Classifier. isModelLayer(c3) ∧ deps(c1, c3))) ⇒ (∃c2 : Classifier. isViewModelLayer(c2) ∧ deps(c1, c2) ∧ deps(c2, c3))

Consequence: Defining the ViewModel to be between the Model layer and the View layer, will restrict the affect of changes in View to the Model.

5. RaisePropertyChangedEvent

Problem: How to reduce the side effects of refactoring of property names in View-Model layer?

Solution: Whenever the properties of an object changes, PropertyChanged event is fired. The PropertyChange event handler takes the property name as an argument. The property name is a string and can be changed during refactoring. If the changes in the property name are not reflected in the PropertyChanged arguments, the notification of the property change will not be handled. To overcome the problems caused in the PropertyChanged event handler during refactoring, the RaisePropertyChangedChangeEvent is used.

- Listing 6.19 uses the PropertyChanged event handler with the String Name as an argument. This may cause problems during refactoring if the developer forgets to update the string value.

```csharp
public class SimpleViewModel : INotifyPropertyChanged
{
    private string _name;
    ...
    public string Name
```
Listing 6.19: OnPropertyChanged event handler is used with a String value.

- RaisePropertyChangedEvent is used in Listing 6.20 thus satisfying the guideline.

```csharp
public abstract class ViewModel : IViewModel, INotifyPropertyChanged
{
    ... public void RaisePropertyChangedEvent(MethodBase sourceProperty)
    {
        if (PropertyChanged != null)
        {
            string propertyName = sourceProperty.Name.Substring(4);
            // Event Handler
            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
        }
    }
    ...
    // = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
    public class SimpleViewModel : ViewModel
    {
        private string _name;
        ...
        public string Name
        {
            get { return _name; } set
            {
                _name = value;
                // RaisePropertyChangedEvent is used instead of PropertyChanged.
                RaisePropertyChangedEvent(MethodBase.GetCurrentMethod());
            }
        }
    }
```
Listing 6.20: RaisePropertyChangedEvent is used in the place of OnPropertyChanged.

**Formalization:**
This guideline states that there should not be any property declared with a method call to PropertyChanged event handler. This can be formally specified as follows:

\[ \forall p : \text{Property. } \exists mc \in \text{methodcalls}(p). (\text{name}(mc) = "PropertyChanged" \Rightarrow \neg \text{namespace}(\text{instance}(mc)) = "System.Windows") \]

**Consequence:** The changes in the property name is now managed by the RaisePropertyChanged event as it does not take the hard-coded string as an argument.

6. Use Commands in the ViewModel Layer.

**Problem:** How to reuse the logic related to the user interface actions among multiple Views?

**Solution:** Event handling of the user interface elements should not be implemented in the View layer. By implementing the events related to user interactions in the View layer we cannot reuse the logic. So the code related to the user interface event handling should be replaced by the command binding.

**Example:**

- In Listing 6.21 a button event is implemented in the View layer. This is a violation of the guideline and the logic within the event should be moved to a command in the ViewModel layer.

```csharp
//button click event
private void Button_Click(object sender, System.Windows.RoutedEventArgs e)
{
    //logic
}
```

Listing 6.21: Button click event is implemented in the View layer

- In Listing 6.22 the button click event is replaced by the AddCommand command
in the ViewModel layer and command binding in the View layer.

```csharp
// VIEW LAYER
// ViewClass.xaml
// shows binding of a command.
<Button Content="Add" Command="{Binding AddCommand}"/>

// VIEWMODEL LAYER
//
public class MouseHandlingViewModel : ViewModel
{
    ...
    public ICommand AddCommand
    {
        get { ... }
        set { ... }
    }
}
```

Listing 6.22: Button click event replaced by command binding.

**Formalization:**
We initially define a predicate to identify if a method has an argument for handling Windows events.

\[
\text{isWindowsEvent}(m) \equiv \exists v \in \text{inParams}(m).\text{namespace(Class}(v)) = \"\text{System.EventArgs}\"
\]

All methods having arguments for handling Windows events, must contain all variables and method calls belonging to the View layer only.

\[
\forall m : \text{Method}. \ (\text{isWindowsEvent}(m) \Rightarrow \forall v \in \text{variables}(m), \forall mc \in \text{methodcalls}(m). \ \text{isViewLayer}(\text{type}(v)) \land \text{isViewLayer}(\text{instance}(mc))\)
\]

Here \(\text{type}(v)\) returns the class type of variable \(v\) and \(\text{instance}(mc)\) returns the class instance serving the method call \(mc\).

**Consequence:** Multiple Views can reuse the commands in the ViewModel layer and reduce the amount of code in the View Layer. The commands can be tested independent of the View and reused. So it is easy to maintain and test the commands.

For example, Microsoft Word reuses the commands and provides multiple views to run the command. If we consider changing the font color as user action then
we can do it by a right click or by selecting the icon in menu bar. Both the user actions share the same command and eliminates the redundant code.

6.3 Dependency Property Pattern

In this section we list the guidelines for to implementing Dependency property. Most of the user interface element properties have default initial value, which will not be modified throughout the implementation. Even though they have default values, each object needs separate memory for each common .Net property. WPF provides the concept of dependency properties, which is the extended version of the common .Net properties. Dependency property should be added to the class deriving from the DependencyObject class. The value of dependency property is resolved using dynamic value resolution [24]. The default value is selected only when there is no value set to the dependency property. The value of the dependency property depends on multiple sources and the value is selected from a source depending on the level of precedence. The source may be a binding expression, a local value, a trigger, a style or a default value, listed according their precedence. Description of these sources is given in Section [A.1.2] The dependency property notifies the data binding service of WPF when the value is changed [24]. Thus the changed value can be used in data binding.

**Name:** Dependency Property Pattern

**Problem:** How can this change of behavior among properties be communicated to the developers?

How to preserve the separation of concerns between layers from the dependency property?

**Solution:** The guidelines listed in Section [6.3.1] are followed to solve these problems.

6.3.1 Guidelines

The guidelines to be followed while implementing dependency property are:

1. Name of the dependency property should be suffixed with Property. This is a convention followed to differentiate between Dependency property and CLR property.

   **Example:** In Listing [6.23] the dependency property name is suffixed with Property following the guideline.

   ```csharp
   public static readonly DependencyProperty Sample1Property ;
   
   Listing 6.23: Naming convention for dependency property.
   ```

   **Formalization:**

For abstracting the details of a Dependency property, we define the following pred-
icate:

\[ \text{isDependencyProperty}(p) \equiv \text{name}(\text{type}(p)) = "\text{System.Windows.DependencyProperty}" \]

The following formalism describes that all DependencyProperty instances must have the suffix Property. The regular expression (*) indicates zero or more occurrences of characters.

\[ \forall p : \text{Property}, \text{isDependencyProperty}(p) \Rightarrow \text{name}(p) = "*Property" \]

2. Dependency property should be implemented in a class that inherits DependencyObject class. This guideline is followed for correct implementation of the pattern, as this allows wrapping of dependency property and supports WPF data binding.

**Example:** In Listing 6.24, SomeClass inherits the Window class, whose ancestor is DependencyObject class. Thus satisfies the guideline.

```csharp
//Window class inherits DependencyObject class
public partial class SomeClass : Window
{
    //Sample1Property is a dependency property
    public static readonly DependencyProperty Sample1Property;
    ...
}
```

Listing 6.24: Window class inherits DependencyObject.

**Formalization:**

\[ \forall c : \text{Class}, (\forall p \in \text{properties}(c)) \Rightarrow \exists \text{DepObject} : \text{Class}. \text{deps}(c, \text{DepObject}) \land \text{namespace}(\text{DepObject}) = "\text{System.Windows}" \]

3. Dependency property should be used only in the View layer because the Dependency properties are used with the WPF elements.

**Example:** Listing 6.25 shows the dependency property defined in the View layer, this is correct implementation of the guideline.

```csharp
//VIEW LAYER
public partial class SomeClass : DependencyObject
{
    //Sample1Property is a dependency property
    public static readonly DependencyProperty Sample1Property;
    ...
}
```

Listing 6.25: Wrapping dependency property without violation.
Formalization:
\[ \forall c : \text{Class.}\ (\forall p \in \text{properties}(c). \text{isDependencyProperty}(p)) \Rightarrow \text{isViewLayer}(c) \]

Consequence: By following the dependency property guidelines we will ensure that the developer using the property can easily differentiate between the common .Net properties and the dependency property. The developer should be aware of the behavior of the property he is using. This helps in eliminating the confusion and reduces possible errors due to confusion. Also the pattern allows wrapping the dependency property and maintaining separation of concerns.
Chapter 7

Tool development

7.1 Background

Formally specifying all the design rules is one of the important milestones in this research, but it is also equally important to identify a mechanism to automatically verify if such design guidelines hold in the software. For automatically validating design rules we have come up with a tool which can help the developers at Philips Healthcare with the following activities:

- Validate the design rules that are defined by the architects.
- Share the design rules across teams for consistency.
- Build custom rules based on the formalization techniques discussed in Chapter 5.
- Save these custom rules for future use.

It can be noted that all the design patterns talk about how different elements in the software can interact with each other. In order to implement a tool to validate guidelines defined on such elements, we have to parse the entire code structure. Instead of implementing a custom parser, we look into commercial tools which can be extended to meet our requirements. The following are few of the requirements that must be met in the tool:

- It should parse the C# code.
- It is ideal if it can be integrated into the Visual Studio editor.
- It should offer an open API structure allowing with extend the existing built-in
functions.

- It should perform an on-the-fly static analysis of the design patterns.
- It must ease the development and the testing aspects of the implementation.
- The tool should be available to all the developers in the project without any licensing issues.
- The tool should be stable and the future versions should not affect our implementation.
- The tool should work well with other plugins used at Philips.

Considering the steep expectations of the tool, it is ideal to reuse or extend any of the available commercial tools. There are a number of commercial tools that can perform either static or dynamic code analysis and offer a variety of operations. The following subsection gives an introduction into these two categories of code analysis.

### 7.1.1 Static Code Analysis:

Static code analysis refers to the technique of deriving facts without actually executing the program built from that software. In simple terms it is an automatic replacement for program comprehension or code review. It can either be done by accessing the plain source-code of the software or by accessing the assembly file of the built code. One can gauge the structure of the program source code, for example namespace, class hierarchy details, type declarations etc. Static code analysis is seen with many commercial tools, where Microsoft .Net Reflector [27] is commonly used for class browsing and disassembling purposes. Reverse engineering tools also employ static code analysis technique to extract a model from a source code.

In an attempt to reduce errors in the program, static analysis can help detect code, likely to contain syntax errors or bad formatting. The main advantage of this technique is that rule violations can be detected very early in the coding phase hence resulting in considerable saving in time and money. Generally correction of an error at the testing stage is ten times more expensive than its correction at the construction (coding) stage [28].

There are many commercial tools that can be used to realize this project goal. Following are a list of some popular tool-sets:

- Resharper [29]
Chapter 7. Tool development

- NDepend [30]
- FXCop [31]
- Code Analyzer [32]
- CodeRush [33]

Each tool has a distinct advantage as compared to the others. We show this comparison in the Section 7.1.3.

7.1.2 Dynamic Code Analysis:

Dynamic code analysis is often complicated as compared to static analysis, as this is performed by executing the programs. The tests are carried out at the program runtime, so the results depend on the range of test inputs supplied. Often it is very difficult to judge if the set of test inputs are sufficient to declare the program to be error-free. Code coverage is one of the most commonly known metric that uses this technique. Code coverage is useful is testing the different behaviors of the system.

As compared to static analysis, this technique is able to detect dependencies that are not possible in static analysis like dynamic dependencies using reflection, dependency injection, polymorphism etc. Details that are temporal or generated during runtime can only be tested using dynamic analysis.

There are many commercial tools available; following are few of the popular tool-sets: Avalanche [34] and BoundsChecker [35].

7.1.3 Survey of Commercial Tools:

The design patterns discussed in the Chapter can be validated by static code analysis, however none of the commercial static analyzers directly support building such custom rules. The tool should support defining custom design rules for validation through the open API framework. Open API allows extensions to the existing framework by using the libraries in the same way the actual tool was implemented. In the subsequent subsection we provide a survey of tools which can be considered to perform design pattern validation.

NDepend:

Pros: NDepend supports large scale code metrics management by visualizing dependencies using directed graphs and dependency matrix. Like Resharper, NDepend also supports validation of architectural and design rules coupled with a full integration with
Visual Studio. Custom rules are written using a language called Code Query Language (CQL), the structure of which is similar to traditional Structure Query Language (SQL).

**Code Query Language (CQL):** CQL is the proprietary language used by NDepend for querying the code, modeled as a database. Hence it supports a SELECT TOP FROM WHERE ORDER BY pattern as in SQL [30]. We have tried to implement our guidelines using NDepend and the guidelines can be written as in the examples shown in Listing 7.1 to 7.3.

- Listing 7.1 returns list of classes which implement an IDisposable interface.

  ```cql
  SELECT TYPES WHERE IsClass AND Implements "System.IDisposable"
  ```

  Listing 7.1: CQL query to obtain a list of classes that implements an IDisposable interface

- Listing 7.2 shows a warning if the name of a namespace contains a string View and is using a code element named ModelLayer.

  ```cql
  WARN IF Count > 0 IN SELECT NAMESPACES WHERE IsDirectlyUsing "ModelLayer" AND NameLike "View"
  ```

  Listing 7.2: CQL query to show display a warning when a dependency is found.

- Show a warning if a method named MyMethod has more than one parameter.

  ```cql
  WARN IF Count > 0 IN SELECT METHODS WHERE ( NbParameters < 2 AND NameIs "OurMethod" )
  ```

  Listing 7.3: CQL query that restricts the method to have maximum of one parameter.

**Cons:**

- NDepend is a commercial tool and the license is not available for free. So the tool needs many licenses to share among all the developers in the project.

- NDepend does not support multi-level query that is commonly seen with SQL. Consider the following design rule: Show a warning if a class uses View layer namespaces and implements the IDisposable interface. This would need a multi-level query that gives an option to access (SELECT) both NAMESPACES and TYPES implementing IDisposable.
FXCop:

Pros: FXCop is a Visual Studio extension from Microsoft for reporting possible design, performance, security improvements by analyzing the code assemblies. It is now part of the Microsoft Windows SDK for Windows 7 and .NET Framework 4. It supports development of class libraries, hence custom design rules can be defined. FXCop can be interfaced with other applications as it can be run from the command line in addition to its console options.

Cons: FXCop caters the requirements set in this project, but the analysis of code assemblies has an inherent issue. To validate a design rule, the assembly file of the project must be input to FXCop. The warnings relevant to any design rule violation is then generated, leading to an offline verification. Hence considerable effort is needed to check the warnings and correct as considered to an on-the-fly checking.

TICS:

Pros: TICS is another static code analyzer for generating coding standard violations. It is very similar to FXCop in terms of majority of the features. It can also be initiated from command line or from the IDE. Like other tools TICS can also combine validation results in a uniform way. Like other tools, TICS can be used to categorize violations based on severity levels.

Cons: TICS is very similar to FXCop, hence has a similar flaw. The design violations are detected by running the tool. The warnings are displayed in a separate window for correction. Hence there is no on-the-fly validation of design rules.

Resharper

Resharper is commercial static analyzer developed by JetBrains [13]. It performs a static code analysis with an on-the-fly error detection and correction. The on-the-fly checking helps the developer with instant suggestions and code fixes. This saves considerable time as the design flaws are detected without the code being compiled. It also provides additional features for error correction, code completion, navigation, syntax highlighting, formatting and optimization. Using the Open API framework enables us to extend Resharper to validate custom design rules defined in a particular project. Our attempt is to implement the design rules formalized in Chapter 6 to develop a plug-in which would ensure the rules are adhered in all the product implementations. Rule violations can be displayed as errors, warnings, suggestions or hints and also an option to re-factor the code also exists.

Pros:
1. Resharper supports on-the-fly verification of the source code.

2. Provides a plugin development framework.

**Cons:**

1. The plugin development framework is complex.

2. Difficult to test the plugins as the only option is to check the location of error on the Visual studio editor.

3. Offline verification is not possible with Resharper, i.e. it cannot be used outside Visual Studio IDE. Only way to extend the tool is by developing plugins, which makes testing very difficult.

4. With every release of Resharper, there are few modifications to the open APIs [36]. So the plugins developed with Resharper is difficult to maintain.

**CodeRush**

CodeRush is a commercial tool provided by Devexpress [37] for maintaining the quality of source code. CodeRush supports static code analysis and provides features to build plugins to Visual Studio for on-the-fly verification of the source code. CodeRush provides most of the features provided by Resharper with few advantages. The advantages in CodeRush is mainly because of its architecture. CodeRush provides a DXCore library which can be used to generate an object model of a solution and also separates the plugin framework from DXCore. This helps us to develop the logic required to analyze the code and verify the logic using the DXCore object model. Once the logic is developed and tested we can develop the plugins for on-the-fly notifications of the violations.

**Pros:**

1. Provides a plugin development framework for Visual Studio.

2. Supports on the fly verification of the code.

3. Supports plugin development for refactoring the source code.

4. The separation between the CodeRush plugin framework and the DXCore object model allows us to keep the logic of the plugins separate from the plugins. This helps in simpler testing and easier to maintain the code. Even the logic can be reused among multiple plugins.

5. CodeRush Xpress is available for free. Which has most of the features required in our requirements.
Cons:

1. It has a good architecture to be used independent of Visual Studio, but the tool does not come with this feature.

2. The tool gives the details of the Abstract Syntax Tree but there are no methods with which we can directly access the relationship between different classes and interfaces outside the plugin framework.

Looking upon the features provided by these tools we selected to work with NDepend as it provides a good visualization of the statistics. But we found that complex relationships between classes could not be represented by the CQL queries. As complex relationship between classes was essential we could not implement the design pattern guidelines with NDepend.

Then we selected Resharper as it was available at Philips and the developers were using the features of this tool. During the development process we found the disadvantages listed in section 7.1.3 to be more strong to move to another tool. Later we tried to overcome few problems of Resharper by using the DXCore object model and the Resharper plugin framework. We were able to parse and verify the current document in the Visual Studio editor but were unable to get the structure of the whole project. This made us to move the development completely to CodeRush. With CodeRush we have built our current tool which overcomes both the disadvantages of CodeRush listed in Section 7.1.3. Thus CodeRush was helpful in satisfying most of our requirements and provided the features to extend the tool easily.

7.2 Architecture of the Tool

The architecture of the tool is shown in Figure 7.1. The architecture shows our tool in the dashed box and the CodeRush framework used by our tool. The CodeRush framework contains the DXCore libraries and the Coderush plugin framework. The DXCore libraries can be used to parse the entire solution and provide the details of the parsed solution. The components of our tool shown in the dashed box in Figure 7.1 will use the DXCore services to generate the object model. Once the DXCore object model is generated, the tool iterates over the code elements in the model to verify the guidelines, for this task the Solution Explorer component is implemented in our tool. The problem with the DXCore object model is to extract the details from it. As DXCore was designed to work as a plugin, most of the functionalities work only when it is used as a plugin. To use DXCore independent of the plugin framework, a component called Fact Extractor is implemented in our tool. The Fact Extractor extracts the details from the object model generated for the input solution. The separation between the DXCore and the CodeRush plugin framework allows the tool to access the DXCore libraries independent of the plugin framework. The plugin framework provides the APIs required to develop
the plugins to the Visual Studio. As the plugin works with the editor, it will have the access to the current file the developer is working with. This helps in analyzing the source code in the active file in the editor and to provide on-the-fly verification of the source code.

Figure 7.1: Architecture of the tool (within dashed box) with the arrows showing the dependencies between different components.

The tool shown in the dashed box in the Figure shows the dependencies between four components namely the Fact Extractor, the Rule Builder, the Solution Explorer for report generation and the plugins to Visual Studio. The Fact Extractor is in the lowest level and provides basic methods required to access the structure of classes and methods in the current project. The Rule Builder uses the methods in the Fact Extractor to verify the guidelines and the methods in Rule Builder are used in the plugins and the Solution Explorer.

The advantages of this separation between these four components are.

1. The rules are reusable among different components like the Solution Explorer and various plugins.

2. Easy to formulate the guidelines by reusing the basic methods in the Fact Extrac-
3. Easy to test the rules and to maintain the code.

4. Easy to write a plugin and to understand the code.

5. The code for a plugin is separated from the logic this makes the tool to be extended easily and increases the ease of testing.

The functionalities of each of the components are discussed in the Section from 7.2.1 to 7.2.4.

7.2.1 Fact Extractor

The Fact Extractor contains the basic methods required to formulate the guidelines and it gets the language elements from the Rule Builder. The language elements may be the class, the interface, the parsed file or any instance of the code element depending on the guideline. The task of the Fact Extractor is to extract the details from these elements. The details that can be obtained from the Fact Extractor are the implementation equivalent of the basic predicates in the formalization. The class diagram for Fact Extractor is shown in the Figure 7.2.

For each element that needs to be analyzed while building the rule, we have a class in the Fact Extractor. The class in the Fact Extractor provides the Rule Builder with the methods required to get the details of the source code for the formalization. The Fact Extractor also gives the details of the structure of the source code. The details of the structure is the information about the relationship between classes and interfaces and with other language elements. The Fact Extractor provides additional functionalities to access the DXCore object Model to get the structural details of the source code outside the plugin framework. Given a class or any element in the solution, the entire structure and the details can be extracted in the Fact Extractor. The Fact Extractor can take any element like an expression, interface, method or a method call and return the required details.

Few features provided by Fact Extractor are listed below.

- Given a file, the Fact Extractor can be used to get all the classes and the list of namespaces used in that file and also details of the file like the complete file path can also be obtained. Also the information about the layer to which the file belongs. All these information can be obtained through FileInfo class.

- Given a class, the Fact Extractor can be used to get details like class name, number of methods, each method in the class, modifiers of the class, access specifier of the class, associations with other class, ancestors of the class and all the implemented
7.2. Architecture of the Tool

Figure 7.2: Class diagram of the Fact Extractor

interfaces and also the descendants of the given class. All this information is available from ClassInfo class in the Fact Extractor.

- We can also get the information related to a method from the MethodInfo class of the Fact Extractor. The method information may be its name, access specifier, modifier, return type, parameters of the method and the methodcalls from the method and also the variables and type reference expression used. Type reference expressions give the references to other classes.

- InterfaceInfo in the Fact Extractor inherits few properties and methods from ClassInfo to give the details of an interface with rest of the classes and interfaces. So the relationships that can be obtained from InterfaceInfo are the descendants of an interface, ancestor interfaces, methods in an interface and access specifier of the interface.

- VariableInfo in the Fact Extractor gives the details of the type of the variable, variable name and information of the type. The type can be resolved to its class and its details can be obtained.
• PropertyInfo in the Fact Extractor gives the details of a property, the name of the property, the variable for which the property is defined, the setter and the getter of the property and the method calls in the property.

• MethodCallInfo in the Fact Extractor gives the details of the method call, the name of the method call, the location of the method call (inside the property or inside a method), list of arguments, the method definition of the called method.

• The ExpressionResolver class is used by the Fact Extractor to resolve the references between different language elements in the code. For example the association between the class can be obtained through resolving the type reference expression in a class.

7.2.2 Rule Builder

The guidelines that are formulated in Chapter 5 are implemented in this layer of the tool. Each pattern that is to be verified will have a class in this layer. The pattern to be checked is implemented by a class that must be derived from the DesignRules class (this is an abstract class) as shown in Figure 7.3. The DesignRules class will initialize few of the Fact Extractor fields and its own fields. These fields are used during pattern verification. Every guideline of the pattern will be implemented as a method in the corresponding class. These methods are called from the plugin or the Solution Explorer. To implement the formulated guidelines, we have used the basic methods provided in the Fact Extractor as described in Section 7.2.1. So the methods in the Fact Extractor can be tested separately and reused for implementing the guidelines. The implementation details of a sample guideline is given in Section 7.4.

7.2.3 CodeRush Plugins for Visual Studio

Plugins contains the code required to work with Visual Studio. The main task is on-the-fly design pattern verification. It notifies the developer about the possible violations of design pattern guidelines. The formulated guidelines in Rule Builder are called in the plugin to check whether they are applicable in the current file and to verify its correctness of the implementation. If there are any violations then the error pops up with the information of the violated guideline. The plugin will have code that calls the methods required to verify guidelines and also the location where the error should pop-up in case of violation. To verify the guidelines, the plugin calls the methods in the Rule Builder.

We are using two types of CodeRush plugins to verify the code in the Visual Studio editor. One is the codeprovider and the other is the action hint. The codeprovider is
used to show the violation message on the Visual Studio editor. It verifies the source code on-the-fly and helps the developer to correct the violation of the guideline. The action hint plugin is also used to verify the source code in the Visual Studio editor. The errors will be pointed out by the arrows with the name of the design pattern. The errors will be pointed out when the developer presses a key combination. The key combination can be assigned by the developer. The main idea behind action hint is to point out the errors after the developer has completed the coding in a file. The output of the plugin in the Visual Studio editor is shown in Figure 7.5. The implementation details of a sample plugin is given in Section 7.4.
7.2.4 Solution Explorer

Even though we have on-the-fly verification of the source code, we may need to know the results of the entire solution where the violations might still exist. It is necessary to have the list of violations at a single location. To provide this feature in our tool, we have listed the violations in a report, which is easy to evaluate. The report contains the details of the violations with the name of the guidelines that are violated and the location in the code where the guidelines are violated. To generate the report we have developed a Solution Explorer. The Solution Explorer is mainly needed to evaluate the entire solution with many projects. The Solution Explorer is developed using few services of the DXCore. The main task of Solution Explorer is to read solution from the disk and parse it using the DXCore parser. The Solution Explorer also uses the DXCore libraries to generate the object model for the input solution. From the object model obtained for the solution we iterate through each project in the solution. From each project we iterate through each class in the project. For each class we use the Rule Builder to verify the correctness of the implementation of the guidelines. If any violations are returned from the Rule Builder we include it in the report and generate a solution wide report with list of violation in the solution.

The Solution Explorer is also useful in verifying the implemented tool. As the verification of the tool is difficult through plugins, we can use Solution Explorer to verify the tool. The Solution Explorer can be used in verifying the correctness of the implemented rules. The correctness of the implemented rules is verified by providing the test cases as the input to the Solution Explorer. By using some test cases we can check for false positives and false negatives that result from the implemented guidelines in the rule builder. With these results we can improve the implementation of the methods used to verify the guidelines ensuring correctness of the implementation.
7.3 Formalization - Implementation Co-Relation

The relationship between the formalization and the implementation is given in this section. For every detail that is used in the formalization there is an equivalent method or a property in the implementation. Table 7.1 shows the relationship between the formalization and the implementation of the class. Formalization details of a class can be obtained from ClassInfo in the implementation.

<table>
<thead>
<tr>
<th>Formalization</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>ClassInfo</td>
</tr>
<tr>
<td>class: Class</td>
<td>ClassInfo class;</td>
</tr>
<tr>
<td>name(class)</td>
<td>class.Name</td>
</tr>
<tr>
<td>methods(class)</td>
<td>class.AllMethods</td>
</tr>
<tr>
<td>constructor(class)</td>
<td>class.GetConstructor</td>
</tr>
<tr>
<td>finalizer(class)</td>
<td>class.GetFinalizer</td>
</tr>
<tr>
<td>variables(class)</td>
<td>class.AllVariables</td>
</tr>
<tr>
<td>modifier(class)</td>
<td>Modifiers are implemented as properties and can be obtained as in Table 7.2</td>
</tr>
<tr>
<td>accessSpecifier(class)</td>
<td>Access Specifiers are implemented as properties and can be obtained as in Table 7.3</td>
</tr>
</tbody>
</table>

The modifier used in a class are listed in Table 7.2 which has the details of implementation corresponding to formalization. Given any formalization detail there is a property or a method that gives the equivalent implementation detail. The co-relation between the implementation and the formalization is discussed in Section 7.4.

<table>
<thead>
<tr>
<th>Formalization</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>isAbstract(class)</td>
<td>class.IsAbstract</td>
</tr>
<tr>
<td>isVirtual(class)</td>
<td>class.IsVirtual</td>
</tr>
<tr>
<td>isStatic(class)</td>
<td>class.IsStatic</td>
</tr>
<tr>
<td>isSealed(class)</td>
<td>class.IsSealed</td>
</tr>
<tr>
<td>isOverriden(class)</td>
<td>class.IsOverride</td>
</tr>
</tbody>
</table>
Table 7.3: Co-relation between the formalization and implementation for Access Specifier.

<table>
<thead>
<tr>
<th>Formalization</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>isPublic(class)</td>
<td>class.IsPublic</td>
</tr>
<tr>
<td>isPrivate(class)</td>
<td>class.IsPrivate</td>
</tr>
<tr>
<td>isProtected(class)</td>
<td>class.IsProtected</td>
</tr>
<tr>
<td>isInternal(class)</td>
<td>class.IsInternal</td>
</tr>
</tbody>
</table>

7.4 Implementation Details

In this section we have included the implementation details of a guideline. We have also shown the formalization and the equivalent implementation of the guideline in the Rule Builder. We have a short proof to emphasize that the implementation co-relates to the formalization.

1. **Guideline**
   If there is a Dispose method in a class then the class should implement IDisposable interface in it or in any of its parent classes. This guideline is discussed in Section 6.1.

2. **Formalization**
   The formalization of the guideline is given in Equation (7.1). In the formalization we have used predicates like isDispose, hasIDisposable and the implementation of these predicates are shown in Listing 7.5.

   \[
   \text{hasDispose}(c) \Rightarrow \text{hasIDisposable}(c) \quad (7.1)
   \]

3. **Implementation**
   The formalization of the guideline is implemented in the Rule Builder. The implementation details of the guideline is given in Listing 7.5. This implementation requires few basic methods and properties from the Fact Extractor.

   - **Fact Extractor**
     The basic methods and properties used from the Fact Extractor are given in Listing 7.4. The first method in Listing 7.4 is the GetAllParentInterfaces(), which returns all the interfaces implemented by the class directly and indirectly through the parent classes and interfaces. 
     AllMethods is an enumerator which is used to get all the methods in a class and the property Name is used to get the name of the class or a method, depending on the instance accessing the property.
### Rule Builder

The predicate hasDispose(c) in the formalization is co-related to the hasDispose(c) method in the implementation. The hasDispose(c) predicate is used to check if there is a Dispose method in the class c. According to the implementation hasDispose(c) returns true if there is Dispose method else it returns false.

```csharp
public bool IDisposableIfDispose()
{ // has dispose method but does not implement IDisposable Interface.

    if (hasDispose(_classInfo) &&
        !hasIDisposable(_classInfo))
    {
        return true; // violation
    }
    return false; // no violation
}

// returns true if the class implements an IDisposable interface
private bool hasIDisposable(ClassInfo myclasinfo)
{
    if (myclasinfo != null)
    {
        if (myclasinfo.GetAllParentInterfaces().Count > 0)
        {
            foreach (InterfaceInfo int1 in myclasinfo.GetAllParentInterfaces())
            {
                if (int1.Name.Equals("IDisposable"))
                {
                    return true;
                    // classinfo implements IDisposable interface
                }
            }
        }
    }
}
```
The predicate hasIDisposable in the formalization checks recursively if the class c implements a IDisposable interface. In the implementation, the method GetAllParentInterfaces gets all the implemented interfaces of the class C and the hasIDisposable method checks if there is an interface named IDisposable in the list returned by the GetAllParentInterfaces method and returns true if there is one else returns false.

Using these two methods, the method IDisposableIfDispose() checks if there is a violation of the guideline. In the formalization of the guideline, we check for every class if there is a Dispose method then there has to be an IDisposable interface implemented. In the implementation of the guideline we return true if there is a Dispose method in a class and the IDisposable interface is not implemented. So we return true in the implementation if the formalized guideline is not satisfied. The co-relation between the formalization and the implementation is given in Equation 7.1 to 7.3.

To satisfy the guideline, the formalized rule in Equation 7.1 should hold.

\[ \text{hasDispose}(c) \Rightarrow \text{hasIDisposable}(c) \]

This can also be written as

\[ \neg \text{hasDispose}(c) \lor \text{hasIDisposable}(c) \] \hspace{1cm} (7.2)

negating Equation 7.2 we have

\[ \neg(\neg \text{hasDispose}(c) \lor \text{hasIDisposable}(c)) \]

Applying De-Morgans law, we get Equation 7.3

\[ \text{hasDispose}(c) \land \neg \text{hasIDisposable}(c) \] \hspace{1cm} (7.3)
We have implemented Equation 7.3 in Listing 7.5 for the formalization of the guideline in Equation 7.1. So the implementation returns true if there is a violation.

- **Solution Explorer and the plugin.**
  The guidelines implemented in the Rule Builder can now be used to create a plugin or to verify the solution through the Solution Explorer.

  - **Solution Explorer**
    In the Solution Explorer we parse the entire solution and pass each class and interface to the Rule Builder to verify the guideline. The call to verify the guideline is given in Listing 7.6.

    ```csharp
    if (disposeRule.IDisposableIfDispose())
    {
        // Print Violation
    }
    
    Listing 7.6: Implementation in Solution Explorer
    
    - **Plugin**
      We use have implemented two types of plugins to Visual Studio. The details of the plugins are discussed in Section 7.2.3. To display the errors on the editor, we need to handle the events of the codeprovider plugin. For this we have to write an event handler for the CheckAvailability event. We have to specify the context and the location on the editor where the message has to pop-up.

      The code for the event handler of the guideline is given in Listing 7.7.

      ```csharp
      private void IDisposableIfDispose_CheckAvailability(object sender,
                                                          CheckContentAvailabilityEventArgs ea)
      {
        if (CodeRush.Source.ActiveClass != null)
        {
          DesignRules dRules = new DisposePatternRules
          (CodeRush.Source.ActiveClass);
          if (dRules.IDisposableIfDispose())
          {
            ea.Available=true;
          }
        }
      }
      
      Listing 7.7: CheckAvailability event for CodeProvider
For the action hint plugin we need to handle the CheckAvailability event and the Execute event. The CheckAvailability event handler should have the context of the source code and the Execute event handler should have the location on the editor to point the action hint arrow. We have provided the context and location of the code in Execute event handler and the CheckAvailability event is always made available. The code snippet is shown in Listing 7.8.

```csharp
private void IDisposableIfDispose_CheckAvailability(CheckActionAvailabilityEventArgs ea)
{
    ea.Available=true;
}
```

Listing 7.8: CheckAvailability event for the action hint plugin.

The code snippet for the Execute event handler is given in Listing 7.9.

```csharp
private void IDisposableActionExecute(ExecuteEventArgs ea)
{
    if (CodeRush.Source.ActiveSourceFile != null)
    {
        foreach (Class myClass in CodeRush.Source.ActiveSourceFile.AllTypes)
        {
            DesignRules dRules = new DisposePatternRules(myClass);
            if (dRules.IDisposableIfDispose())
            {
                CodeRush.ActionHint.PointTo(m.NameRange.Bottom, "Dispose Pattern", Color.Red);
            }
        }
    }
}
```

Listing 7.9: CheckAvailability event for the action hint plugin.
Chapter 8

Verification of the tool

The formal specification of the design patterns must be verified to make sure they precisely cover all the requirements described by the pattern. We have adopted a reverse engineering approach to do this. Initially a translation from the formal specification of the design patterns to the implementation is done. This implementation is verified extensively in three phases: Unit testing, Batch Testing and Brute force approach. These techniques are explained in the subsequent sections.

Following these verification procedures will let us derive the following advantages:

- Issues caught during verification have resulted in implementing stronger constraints for detection of design flaws.

- By implementing stronger constraints, we have adopted a reverse engineering approach to strengthen the formal specification of the design patterns. This will precisely specify each of the design guideline by depth.

- Large number of false positives and false negatives are eliminated by these testing schemes.

- Building a tool for generating a solution wide violation report, gives an overall idea of a project. This is useful for creating and managing development artifacts related to design adherence.

Each of these advantages will be revisited explicitly in the following verification methodology.
8.1 Unit Testing

Unit testing is a conventional method of testing the individual components of the software separately. White box unit testing is initially performed to debug and solve logical issues in the code. This is followed up with black box unit testing to test the software unit for a varied input combination. The implementation of tool as explained in Section 7.2 is implemented by building a simple framework for validating generic design rule specification. Unit Testing of the Fact Extractor and the Rule Builder are both important to eliminate false positives and false negatives. The plug-in version of the tool is a class library hence debugging it like a console application is not possible. It is also the case that few of the DXCore libraries do not work in a console application. Unit testing can be performed in the following ways:

1. Develop a new unit test within Visual Studio and assert that the output of any function is as expected.
2. Use a specialized unit-testing framework like NUnit.
3. Implement a test console application that imports the Fact Extractor (*.dll file) and verifies output for each function in it.

We adopt the third approach as it gives us the flexibility of creating a generic console application which can be extended to check the Fact Extractor as well as the Rule Builder. This test application is also extended to form the basis of Solution Explorer, as it is a console application with function calls to validate all the design patterns implemented in Rule Builder.

Fact Extractor Unit Testing:
Section 7.2 described the Fact Extractor as a simple framework consisting of functions for gathering details about elements such as classes, interfaces and their relationship between each other. It creates a structure that closely relates to the Class Diagram details described in GEBNF notion in Section 5.2.2 for example, there exists functions in the Fact Extractor to capture class properties like name, namespace, methods etc. Agile development of these functions enables the tool to grow steadily from a simple functionality to a complex one. As a result, each of the function in the Fact Extractor is sound for the following reasons:

1. Each function in the Fact Extractor is manually tested by passing a C# file. This is repeated with different samples of the C# file.
2. Manual tests are performed for checking the correctness of the structural details of a C# file. For example, details like different methods declared inside a class can be verified manually.
3. The Class Diagram is generated for a test C# file, and the structure is compared with the details fetched from the Fact Extractor. This clearly validates the functions in the Fact Extractor for details on each element of the Class Diagram.

**Rule Builder Unit Testing:**
Testing the Rule Builder is the most important part of this verification process. As explained in Section 7.2.2, the Rule Builder houses the actual rules for validating the design patterns discussed in Chapter 6. In Section 7.3, we described about how formal specifications are translated to implementation. Unit testing step is done to verify that this translation captures all the requirements of the design pattern correctly. This is tested similar to the Fact Extractor, by using a console test application which imports the Rule Builder and validates all custom design pattern (in Section 6) implementation against sample C# files created as test cases to cover all the variations in the individual patterns. These test cases cover positive as well as negative cases created by injecting violations of design patterns.

**Pros and Cons:**
Unit Testing of FactExtractor and RuleBuilder has the following merits:

- This technique is useful in debugging and building an initial prototype of the tool.
- Detection and elimination of false positives is straight forward by strengthening design guidelines.
- By preparing defect injected test cases, false negatives can be eliminated.
- This prototype framework can be easily extended to implement stronger design constraints.

However, it poses the following concerns:

- Forming an exhaustive list of test cases is a challenge.
- Complex test cases might need continuous tweaking as the tool evolves.
- Lot of effort should be spent in making a good set of test cases.

**8.2 Build Testing**

At Philips Healthcare, design statistics like the number of violations seen and severity of the violation are tracked for every new build of the software in the build sever. Solution Explorer provides these statistics directly.
Although Unit testing is an irreplaceable approach for verifying the tool, preparing an exhaustive list of test cases is not practical. A suitable alternative is to test the tool against a large code repository. The Solution Explorer performs a solution-wide validation of design patterns. By solution-wide, we mean that all files under the same C# solution file (*.sln file in C#) are verified for design violations. The Solution Explorer being a console application allows easy debugging and analysis. Build testing is done in three stages:

1. Perform Unit Testing on SolutionExplorer by creating sample test cases.
2. Compare the results from the previous iterations with the new results.
3. Perform acceptance Test of the tool by testing the tool with the Philips codebase and interaction with designers and developers in the team.

Initial results have shown a lot of false positives being reported. This is discussed below:

**Pros:**

1. Validation of the entire solution aids in detection and elimination of false positives.
2. Ability to parse the entire solution helps in implementing rules based on relationship existing between different namespaces or projects. This is not possible in Unit Testing, where only a single C# file is parsed.
3. By comparing the results from the previous iterations with the new results we can verify if the true violations of previous iteration still exists in the new results. This will help in finding few possible false negatives that can occur in the verification process.

**Cons:**

1. However it might result in large scale fine tuning of the implementation.
2. Detecting all false negatives is unlikely in a large codebase, unless specific defect injected test cases are used.

The prime advantage of Build testing is the detection of false positive violations in the entire codebase. E. W. Dijkstra pointed out, *testing can only capture the presence of bugs and not its absence* [38], hence detection of false negatives must be carefully considered. We approach this problem by a Brute force technique.
8.3 Brute force technique

Both Unit Testing and Build Testing are very efficient in detecting implementation flaws in the tool. Once these flaws are corrected, the tool is sound but not complete. Completeness can be claimed only when the false negatives are eliminated. The Dispose pattern and Dependency pattern was discussed in Section 6.1 and Section 6.3 respectively. Both these patterns have specific keywords like “Dispose(false)” and “DependencyProperty”, which make it unique as compared to others. We can perform a Brute-Force search for this keyword and manually validate if the design pattern holds true. In this technique, we directly use the Visual Studio IDE for searching the keywords in any file/solution. This method might seem laborious but the design patterns dealt in this research are not found as commonly as any standard design pattern, so its merits clearly overweights the drawbacks. The report generated by Solution Explorer is marked against each of the keywords searched using Visual Studio, while manual verification is done on all unmarked entries.

Once the verification tests are covered and implementation issues are fixed by strengthening the rules, we reverse engineer to strengthen the formal specification initially constructed. This step is repeated until issues are resolved to give a clear formal specification and implementation.
Chapter 9

Results

In this chapter, we will discuss about the results seen by using our tool when it is tested against sample test cases and Philips codebase. The results recorded are after performing multiple iterations and refinements based on the testing methodologies discussed in Chapter 8.

As we were building the tool, we have tested its functionalities on few simple test cases to ensure the correctness of the tool. While implementing the methods in the Fact Extractor we have tested each method to avoid errors in the tool. Once the Fact Extractor was ready with little functionality, we implemented few rules in the Rule Builder and verified their correctness against our test cases. As we were using DXCore outside the plugin framework, we found few exceptions when we used the tool with multiple projects. We then reworked on the Fact Extractor to support verification of the guidelines over multiple projects. We tried many different ways to implement the basic methods required in the Fact Extractor to support multiple projects and updated the Fact Extractor. After updating the Fact Extractor, we used the test cases to verify the rules in the Rule Builder. The results of the verification is discussed in this chapter.

Table 9.1: Details of the Philips codebase.

<table>
<thead>
<tr>
<th>Number of Projects</th>
<th>74</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of files</td>
<td>1911</td>
</tr>
<tr>
<td>Number of classes</td>
<td>1359</td>
</tr>
</tbody>
</table>
9.1 Dispose pattern results

We have tested Dispose pattern guidelines with two test cases and the Philips codebase. The results of both the cases are discussed in this section.

Table 9.2: Details of the sample test case for Dispose pattern

<table>
<thead>
<tr>
<th></th>
<th>Testcase 1</th>
<th>Testcase 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of projects</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Number of files</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Number of classes</td>
<td>9</td>
<td>12</td>
</tr>
</tbody>
</table>

First, we verify the guidelines in our test cases (details of the test cases is given in Table 9.2). In each test case, we had injected few violations of the guidelines. Based on the test results, we improved the implementation to satisfy the guidelines. After updating the implementation of the rules our tool worked well with the test cases providing the results in the Figure 9.1.

As we tried our tool on the Philips iXR codebase (Table 9.1) we obtained the results in the Figure 9.2.

The guideline 8 had 817 violations and we found few reasons behind these violations. So we made the following improvements:

1. Initially, the disposable fields were restricted to be disposed in the Dispose bool
method. Later the guideline was slightly modified to allow Dispose bool method to call any other method which calls the Dispose method for the corresponding field. By this change the number of violations observed had reduced from 817 to 568.

2. In the second improvement we modified the guideline from *All disposable objects in a class should be disposed* to *Only those disposable objects owned by the class should be disposed*. By improving the guideline we have got the number of violations from 568 to 84.

3. From the results of the previous iterations, we found few variations in the implementation of the guideline in the codebase. These variations were not violations of the guideline, hence these were caught as false positives by the tool. After considering these variations in the Rule Builder the number of violations reduced from 84 to 38.

The results of the tool in Figure 9.2 have shown the violations for dispose pattern guidelines to be true positives. If there are any future guidelines that will be followed in the team then the Rule Builder should be updated to allow verification with respect to the new guidelines. So the tool in its current implementation can be used to verify the current guidelines followed at Philips and supports improvements for the future guidelines.

Table 9.3: Time taken to complete the verification on the Philips codebase.

<table>
<thead>
<tr>
<th>Pattern Name</th>
<th>Time Taken (in minutes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dispose Pattern</td>
<td>3:48</td>
</tr>
<tr>
<td>MVVM Pattern</td>
<td>5:39</td>
</tr>
<tr>
<td>Dependency Pattern</td>
<td>1:41</td>
</tr>
</tbody>
</table>

9.2 MVVM pattern results

In this section, we will present our tool results for an MVVM pattern. We have prepared one test case for testing and the Philips codebase. The results of both the cases are discussed in this section.

Table 9.4: Details of the sample test case for MVVM design pattern

<table>
<thead>
<tr>
<th>Testcase 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Projects</td>
</tr>
<tr>
<td>Number of files</td>
</tr>
<tr>
<td>Number of classes</td>
</tr>
</tbody>
</table>
The design guidelines are initially verified against the test cases seen in Table 9.4. Similar to the approach followed for Dispose pattern, we have injected few violations to cover all the guidelines. The results for this approach is seen in Figure 9.3.

![Figure 9.3: MVVM design pattern results](image)

The Philips codebase in Table 9.1 offers more challenge due to the variations on how MVVM design pattern is implemented. The result is shown in Figure 9.3.

![Figure 9.4: Results from the Philips codebase for MVVM design pattern](image)

The guideline 3 in Figure 9.3 is relevant to the View Model violations as seen in Section 6.2.1. After testing the the MVVM design pattern implementation against the test cases, the guidelines are checked against the Philips codebase. Initially a lot of false positives were generated due to strict design guidelines. We employ the following improvements to curd false positives in the MVVM pattern.

- **MVVM** is a 3-layered architecture as depicted in Figure 3.2. Our initial approach considered *Visuals* and *Interactors* to be a View layer component. After a feedback based on our first results, *Visuals* is considered as a ViewModel component. This decreased the violations to 816.

- In the second iteration, we found out that there were many classes that did not belong to any layer. They were categorized into an auxiliary layer and no design pattern verification was performed on this set. This significantly reduces more false positives by returning 341 violations.

- Furthermore, we made many other small refinements, leading to 94 violations being caught. We assume that further refinement of design guidelines, will minimize the
false positives positively.

9.2.1 Use Commands in the ViewModel Layer.

To verify the implementation of the guideline described in Section 6.2.1, we implement few test cases. To have all possible violations of this type, we strengthened the guideline by disallowing all Windows event handlers used in the codebase. This resulted in 998 violations. Later we implemented the actual guideline, which disallows only those Windows event handlers that are in View layer and has the ViewModel layer code in it. (In such cases commands should be used instead of events.) By implementing this we obtained one violation which was true positive.

9.3 Dependency Property pattern results

The guidelines for Dependency Property pattern were verified with the test cases we have implemented and the iXR codebase. The test results are as shown in the Figure 9.5. The test cases developed focuses on the first two guidelines and the third guideline is Philips specific, so the test case could only provide results based on the assumptions made on layering of the application. When we run the test on the Philips codebase, the results for first two guidelines showed up true violations but the third guideline resulted in four false positives. By analyzing the results we observed that few improvements were required while separating the layers. The results of the tests on the codebase are provided in the Figure 9.6.

![Figure 9.5: Results obtained from test cases for dependency property pattern.](image)

![Figure 9.6: Results obtained from Philips codebase for dependency property pattern.](image)
The details of the system used to run these test cases is given in Table 9.5 and the time taken to verify these patterns on Philips codebase is given in Table 9.3.

Table 9.5: System details.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Intel Pentium dual-core 3.20GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System</td>
<td>Windows 7 (64Bit)</td>
</tr>
<tr>
<td>Memory (RAM)</td>
<td>4GB DDR3</td>
</tr>
<tr>
<td>Hard Drive</td>
<td>320GB</td>
</tr>
</tbody>
</table>
Chapter 10

Conclusion and Future work

10.1 Conclusions

Large software systems built on specific design pattern guidelines result in a consistent and flexible architecture. Performing static analysis of source code to detect design violations is one of the most novel ideas to curb degradation of software. Verification of design patterns by static analysis is a reverse engineering methodology, often difficult considering the complexities involved. It is ideal if an early feedback is given on incorrect implementations of design patterns. This will ensure that any slack in the coding standards can be corrected and integrity of the codebase can still be maintained.

The main objective of our thesis in developing a simple yet robust framework for verifying the implemented design patterns, is achieved. First, we have formalized the design patterns guidelines in first-order predicate logic using a GEBNF syntax. This syntax can be used to define a domain specific modeling language for specifying any design pattern precisely. This syntax can also be extended to adapt to any new object oriented language paradigms. Due to its close association with the implementation, even a novice developer can directly infer the design guidelines and define new ones when needed. Second, the use of an existing commercial tool, DXCore for parsing the source code will enable future support with new technologies brought in with C# language. The implementation of our tool on top of DXCore is a smooth translation from the formal specifications used. This makes sure that our approach can be used to specify new design patterns also. The resultant tool is robust enough to handle new design pattern verification needs. Third, we have built the tool by separating the concerns on each component. This allows a reusable and flexible architectural scheme in defining any design guideline. The verification of future design patterns can be implemented in the RuleBuilder component using the functions available in the FactExtractor component. Finally, we incorporate an offline as well as an on-the-fly verification scheme in detecting
design pattern violations. They ensure that the design patterns are adhered in legacy codebase as well new implementations. We have also employed effective testing schemes to minimize false-positive and false negative violations being reported. We also draw the results of our tool against sample test cases and Philips codebase. We conclude that repeated refinements done to the design guidelines will minimize the false-positive results.

10.2 Future work

While this thesis has proposed a simple and effective solution for performing static analysis to verify design pattern violations, there are many challenges. In this section, we recommend few areas that can be researched upon.

Although our tool is based on formal specification of design patterns, there are several areas of improvement:

- As an improvement to this work, the formalization of the guidelines can be automatically mapped to the implementation. This is possible because every predicate in formalization has an equivalent method or property in implementation. This has two significant advantages: first, that it automatically relates formalization and implementation and second, it would allow the formal language defined to be commonly used by designers to communicate and specify design patterns. A mock user-interface can be seen in Figure 10.1.

![Figure 10.1: Mockup for an automatic formal verification scheme](image)

This tool should allow design patterns to be stated formally in a text box. As an example to illustrate, we have used one of the Dispose pattern guidelines seen in Section 6.1.

- Another improvement to this tool would be to perform dynamic analysis of the code along with static analysis to reduce the false negatives and false positives. As static analysis is faster and dynamic analysis provides more insights about the
programming paradigm, the combination would achieve good results in verifying
the design patterns.

- The plug-ins provided with the tool can be enhanced by providing refactoring func-
tionalities in addition to the violation notifications. This would help the developer
in taking the necessary actions to overcome from the violations and have a good
implementation.

- The tool can also be used to get the statistics of the code and analyze the existing
structure. We can also visualize the relationship of a language element like class
with other language elements and its role in the application.
References


Appendix A

Appendix

A.1 Windows Presentation Foundation (WPF)

In this section we discuss about the Windows Presentation Foundation and its key features. In Appendix A.1.5 we discuss how these features lead to an architectural design pattern.

A.1.1 Introduction

The user interface is the main part in the system that exposes the underlying system to the user and also guides the user through existing functionalities. Having this in mind Microsoft released .NET applications with Windows Forms having simple APIs for creating interactive Windows applications. The programming model used in Windows Form is an event driven programming model, where the user interactions are handled through events. Every user interface designed using Windows Form will have the controls and each control has its own handle. While creating a Windows Form application, every control drawn on the screen is defined in C# code [24]. So this would be difficult to integrate the UI design with the code. The developers have to rework on the UI design provided by the UI designers in Windows Form. But in the case of WPF applications, the UI designed is translated into XAML tags. So the main advantage of designing UI with XAML is separation of graphical content from the code-behind. Designing custom look and feel for the UI controls is easy in WPF. Few key features of WPF are discussed in this chapter. In Appendix A.1.2, we have included a short introduction about XAML followed by data binding in Appendix A.1.3 and use of commands in Appendix A.1.4.
A.1.2 XAML

WPF provides similar concepts as Windows Form for designing the user interface, by allowing the designer to drag and drop the controls on the window. It also provides declarative programming style for coding related to the user interface with XAML (Extensible Application Markup Language)[24]. XAML is a markup language which defines the arrangement of controls that make up the user interface and is used to create user interfaces. WPF provides an optional Windows Form style of event driven programming, but most of the tasks of user interfaces can be achieved through its declarative style. Microsoft has provided the UI designer in Visual Studio and Expression Blend that generates XAML tags automatically allowing drag and drop of controls. A basic UI can be created using Visual Studio and then the advanced graphic can be added by the graphics designer. This allows developers and designers to integrate their work easily through XAML.

Few features WPF available in XAML are:

1. Data binding: WPF provides declarative data binding, allowing us to describe the relation between the data and the user interface elements in XAML tags. This simple way of binding the user interface elements reduces the code behind the user interface. This feature is enabled by the introduction of dependency property in WPF.

2. Style: Styles are used to improve the look and feel of a user interface by reusing the code. The common property values of user interface elements are defined and reused. The reuse of code would reduce the lines of code and is easy to maintain.

3. Resource: Reusable pieces of an application can be placed in a resource dictionary, WPF uses ResourceDictionary type to do this[39]. Resources can be defined at different levels; if a resource is defined for a specific user control then it is visible only for that user control. Resources can be combined with the previously defined resources and can be referenced by using StaticResource extension. The referenced resource is searched from local scope to the global scope.

4. Control Template: Control template is used to change the structure of a control without modifying its behaviors[39].

5. Data Template: describes how the data needs to be visualized to display on the user interface. We need to explicitly specify how the data should appear in the controls like List Box, Combo Box or list view otherwise WPF will call ToString() on the data to be displayed.

6. Triggers: This feature is used to perform an action when a condition is satisfied.
By using triggers we can respond to mouse events and changes in data model. Triggers can be used declaratively and is mostly used with templates.

Three key features of XAML are **expressivity, comprehensiveness, and extensibility**. Expressivity is mainly through the features it provides. XAML allows the designers to include various controls, layout features and text features in their user interface. XAML also provides complete syntax for 2D vector graphics and can represent few 3D scenes. Comprehensiveness of XAML is through the collaboration that it allows between the developers and the designers. Various important features of XAML that makes it different compared to other markup languages are data binding, command binding, styles triggers and templates. XAML is easily extensible, as the controls provided by WPF are not fixed but can be extended and reused using templates and resources.

### A.1.3 Data Binding

Another major advantage of WPF is its data binding. Data binding is used to synchronize the values between two elements by binding them together. Usually data binding is required to bind some control in a user interface with a property of an object in the business logic. So that when there is a change in the value, the change is propagated automatically. The data binding model described by Microsoft is shown in Figure A.1.

![Data binding model](image)

Figure A.1: Data binding model

Figure A.1 shows the four components involved in data binding namely the binding target dependency object, the target dependency property, a binding source and a path for data flow. The data binding model also shows the direction of the data flow possible between the source and the target. The direction can be one way, where the target object is automatically updated due to the changes in the source and the source does not get affected by the changes at the target. Two way data binding affects both the source and the target property if any one of them is updated. OneWayToSource updates the source object when the target object is updated. The source is updated in case of OneWay-ToSource and TwoWay data binding depending on the value of UpdateSourceTrigger.
Any value towards the source is propagated depending on the value of the UpdateSourceTrigger. One of the ways to communicate the change of property value between the source and target is through INotifyPropertyChanged interface. If there is a change in the value of a property in a class that implements INotifyPropertyChanged then the class will raise an event to communicate the change. This way of communication can be used when there is a change in the property concerning a single instance of an object. If we have a collection of objects then WPF provides an ObservableCollection<T> class to communicate the change. ObservableCollection<T> is a generic class which implements INotifyPropertyChanged interface.

The majority of WPF elements that are used to build user interfaces derive from System.Windows.FrameworkElement base class and this class has a DataContext property. This property allows the developer to select the framework elements data context for data binding. By using the data context, every element within the control will use the same DataContext for data binding by default. XAML allows the designer to specify what triggers an update in the data binding. This can be done by providing UpdateSourceTrigger.

Listing A.1: Syntax Of data-binding

<
object property="{Binding declaration}"/>

In A.1 the object is the user-interface object and may be a text-box for example. In Binding declaration we specify the path, which is a property name to which the view object binds to.

A.1.4 Commands

By using the commands we can separate the logic of the command from its invoker. This separation between the command logic and the invoker of the command will help in reuse of the command logic between multiple invokers. This separation is useful while designing and implementing the user interface. Where multiple actions in the user interface can reuse the single command in the code behind. WPF supports the command model for executing the actions in the user interface. WPF has few built-in commands and also allows the developer to create his own custom command. This section gives a short description of the built-in commands and the custom commands in WPF.

1. Built-in commands

WPF provides a developer with few built-in commands present in one of the command libraries mentioned in this section. There are five main command libraries namely ApplicationCommands, ComponentCommands, EditingCommands, MediaCommands and NavigationCommands.
2. **Custom Commands** The built-in commands provided by WPF are not sufficient for all applications. Developers are provided with the independence to create their own commands and bind them to the objects they need to. One of the ways to create our own custom commands is to use ICommand interface.

- **ICommand interface**
  
The ICommand interface provides two methods and an event to create our own custom commands \[11\]. Two methods provided by the interface are the CanExecute method and the Execute method. The CanExecute method determines whether the command is available to be executed in the current state and it returns true if the command is enabled else false if the command is disabled. Execute method is called when the command is invoked and contains the logic of the command. Execute method can only be invoked if CanExecute method returns true. The event provided by the ICommand interface is CanExecuteChanged event. This event will be raised when there is a change in the state and it notifies the invoker about the state of the command. The method signature of the methods in ICommand is shown in Listing A.2.

```csharp
bool CanExecute(Object parameter)
void Execute(Object parameter)
```

Listing A.2: Signature of methods in ICommand interface.

The RoutedCommand class and the RoutedUICommand class implements ICommand interface. So if we use these classes to create our own commands then we will implement ICommand interface indirectly.

In this short introduction to WPF we discussed few features of WPF like XAML, data binding and the commands. In Appendix A.1.5 we will discuss different architectural design patterns and how they utilize the functionalities of WPF. Also an architectural design pattern called MVVM which is specifically used in WPF applications to take the advantage of the WPF architecture.

### A.1.5 Patterns for User Interface related Applications

In this section we discuss about the architectural design pattern similar to MVVM discussed in Section 6.2. The main goal of the architectural patterns discussed in this section is to separate the presentation logic and the user interface from the business logic. The separation between the presentation logic and the business logic is important as frequent changes in the user interface design should not affect the business logic. The other reason is the independence between the developers and the designers. A designer
can create the user interface based on the requirements of an application and is free to make any changes in the user interface. He can later integrate his work with the developers' work. Even the developer can work on the business logic without worrying about the user interface-related issues. To allow this separation, few patterns like Model View Controller (MVC), Model View Presenter (MVP) and Model View ViewModel (MVVM) were developed.

Key features of these patterns are

1. Dividing the application into layers and allowing the developers to work on different layers independently and reduce dependencies.

2. Separation of the user interface from the business logic.

3. Model Layer: Every pattern separates the model layer from the rest of the application. Model layer will be independent and will not refer to any other layers in an application.

Model View Controller

MVC pattern has three layers namely the Model, the View and the Controller. The Model layer contains the business logic of the application and is unaware of the View and the Controller. The View is the user interface of an application which is aware of both the Controller and the Model. The Controller is the third layer which receives the user input from the View and makes corresponding changes in the Model. The user interaction in the View is communicated to the Model through the Controller by making some changes in the Model. The changes in the Model will be noticed by the View directly as the View knows the Model. The relationship between the View and the Model is that the View acts as the observer of the model and reacts to changes in the Model. Since the View reacts to changes in the Model, the Controller will communicate the changes only to the Model and not back to the View. In the MVC pattern the View and the Controller forms the Presentation and provides separation from the Model. Providing the separation between the presentation and the Model will allow multiple presentations for a Model. Between the View and the Controller, the Controller reacts to the user interactions in the View and the View reacts to the changes in the Model through the Controller. So the communication between the View and the Controller is through the Model. Figure A.2 shows the MVC pattern having three layers and their interactions.

In the passive View approach of MVC, the View contains no logic and is completely handled by the Controller. By having a passive View, we can separate the Model and the View completely and have complete interaction managed by the Controller. The passive View approach is shown in Figure A.3 where the View and the Model are not aware of each other. The advantage of this approach is ease of testing. As View contains no logic, it is easy to test the controller in the absence of the View. It is also possible
to test the View by a sample test model in controller. So the view and the controller can be tested easily as the View is unaware of the Model and is completely managed by the Controller. The disadvantage of this approach is the Controller having complete responsibilities to maintain the interaction between the view and the Model. Any change in the View or the model should be completely managed by the controller and is time consuming. Controller has to make changes in the model layer for a user action in view and update the state of view.

MVC allows multiple Views to share the Model layer data. Since MVC is a event driven approach, the logic behind the user actions cannot be reused. Since there is no direct dependency between the View and the Model, changes in View will not directly affect the Model. In the supervising controller, the Model notifies the View directly about the changes in the model. This can be a disadvantage as the View may take some time
to render and few notifications may be lost. In passive View, the controller has more tasks to perform. It gets the notifications from both the View and the Model. In complex applications the controller may be overloaded with notifications.

Model View Presenter (MVP)

The components of the MVP pattern are the Model, the View and the Presenter. The Model is the domain model of the application containing the business logic; The View contains the UI logic and is responsible for handling the user interactions. Presenter is between the View and the Model and has the presentation logic and interacts between the view and the model. The Presenter is aware of both the view and the model but the view and the model are not aware of each other. Any action in the view is delegated to the presenter and it updates the model. Similarly if there is a change in the model then the state of the view is updated by the presenter. MVP pattern has two type of implementations one is the passive view implementation and the other is the supervising presenter. In the passive implementation, the view does not have any logic and is completely dependent on the presenter when there is a user interaction. View is completely unaware of the model. The passive implementation of MVP pattern is shown in the Figure A.4.

Supervising presenter is the second approach used for implementing MVP. In supervising presenter approach, the View has information about the Model and needs data binding engine to bind to the domain model data. Since there is a direct data binding between the Model and the View, the Model should implement INotifyPropertyChanged interface to allow the data binding engine to know about the changes. Here the presenter will maintain the presentation logic and the state of the View. Presenter will be used
by the view only when there are complex user interactions. The complex interactions may be to run a command in response to the user interaction. So the commands will be implemented in the presenter and the model will be updated when a command is executed. The supervising presenter approach is shown in the Figure A.5 where presenter is aware of both view and the model, which will be used during complex interactions.

In the Passive View implementation. The Presenter is notified by both the view and the model about their changes and it updates the changes accordingly. This could be a disadvantage as the presenter has to update both the components for every change in the view. Since the view has some code about the presenter and the data binding of WPF will not be utilized effectively, MVP is not a suitable pattern to be used in WPF applications.

In the Supervising presenter approach, the View is aware of the model due to the data binding, but the model does not know about the view or the presenter. Due to presence of commands in the presenter, this pattern looks suitable to be used with WPF. But the disadvantage of this approach is the direct interaction between the View and the Model. Due to these interactions, it is difficult to test different components independently and also difficult to maintain the application.

A.2 Basic predicates for formalization

We can formalize the design patterns by directly using the GEBNF notation discussed in Chapter 5. However, this would create a low-level (source code level) detailed description, resulting in a complicated specification. We intend to abstract the code level details by
form high-level specifications so that they can be easily interpreted. This helps both the designer as well as the developer to define predicates based on his needs.

For a Dispose pattern, there are essentially three important aspects to help the designer abstract the design guidelines in a simple and abstract manner. It is detailed in the following points:

- **Dispose method:**
  Listing [A.3](#) shows a fragment of the entire dispose pattern. It is a public method with no parameters and is implemented from the `IDisposable` interface.

```csharp
sealed class Class : IDisposable
{
    public void Dispose()
    {
        ...
    }
}

Listing A.3: Dispose method implementation
```

This is the most essential part in a Dispose pattern; hence it will be encountered in most of the Dispose pattern guidelines. Due to this reason it becomes logical if we abstract it into a predicate. The following predicate validates if a method \( m \) is a `Dispose` method without any input arguments.

\[
\text{isDisposeVoid}(m) \equiv 
\text{name}(m) = \text{"Dispose"} \Rightarrow \text{count}(\text{inParams}(m)) == 0
\]  

(A.1)

This predicate can be read as is \( m \) a Dispose void method.

The following predicate validates the presence of a method \( m \) without any input arguments, named `Dispose` in a class \( c \):

\[
\text{hasDisposeVoid}(c) \equiv \exists m \in \text{methods}(c).\text{isDisposeVoid}(m)
\]  

(A.2)

This predicate can be read as does class \( c \) have a Dispose void method.

- **Dispose Bool:**
  Dispose(bool) is another method defined in a class with a modifier as virtual or overridden. However this method takes an input parameter of `boolean` type. Listing [A.4](#) shows this method:

```csharp
public class ClassOne : IDisposable
```

(A.4)
With a similar technique of developing predicates, we can specify this fragment using the following predicate. This predicate indicates the presence of a method \( m \), named \( \text{Dispose} \) with an input parameter of type Boolean in a class \( c \).

\[
is\text{DisposeBool}(m) \equiv \\
\quad \text{name}(m) = "\text{Dispose}" \Rightarrow \\
\quad \exists p \in \text{inParams}(m).\text{type}(p) = \text{bool} \land \text{count}(\text{inParams}(m)) == 1 \tag{A.3}
\]

This predicate can be read as \( m \) is a \( \text{Dispose Bool} \) method.

The following predicate validates the presence of a \( \text{Dispose} \) method with only one input argument of type \( \text{bool} \) in a class \( c \):

\[
\text{hasDisposeBool}(c) \equiv \exists m \in \text{methods}(c).\text{isDisposeBool}(m) \tag{A.4}
\]

This predicate can be read as \( c \) is a \( \text{Dispose Bool method} \) in class \( c \).

The following two predicates abstract the definitions seen in Equation A.1 to A.4:

\[
is\text{Dispose}(m) \equiv is\text{DisposeVoid}(m) \lor is\text{DisposeBool}(m) \tag{A.5}
\]

Equation A.5 shows a generic representation of a \( \text{Dispose} \) method, as it can be a \( \text{Dispose(void)} \) or \( \text{Dispose(bool)} \) method.

\[
\text{hasDispose}(c) \equiv \text{hasDisposeVoid}(c) \lor \text{hasDisposeBool}(c) \tag{A.6}
\]

Equation A.6 shows a generic representation of the class having a \( \text{Dispose} \) method, as the class can have a \( \text{Dispose(void)} \) or \( \text{Dispose(bool)} \) method.

• \( \text{IDisposable} \):

\( \text{IDisposable} \) is an interface defined in the namespace “\text{System}” of the .NET framework by Microsoft. It gives a template for implementing the \text{Dispose} methods.
The existence of this interface would have to be checked in most of the Dispose pattern guidelines; hence a standard predicate would keep the description simple and abstract.

The following predicate indicates the presence of an interface $i$ named `IDisposable` being implemented by any class $c$. The class $c$ has an association relationship with the interface $i$.

$$\text{hasIDisposableInterface}(c) \equiv \exists i : \text{Interface.geners}(c, i) \land \text{name}(i) = "IDisposable"$$

This predicate can be read as “classifier $c$ has IDisposable implementation”. In Section 5.2.2 we defined Classifier $c$ to be either a Class or an Interface.

There is an implicit issue with the above predicate; it checks only the immediate parent relationship. There can be a situation where the `IDisposable` interface is implemented by one of the ancestors in the hierarchy. Listing A.5 shows an example of such a situation:

```java
public class Parent : IDisposable
{
    public Parent1() { }
}

public class Child : Parent
{
    public void Dispose()
    {
        ...
    }
}
```

Listing A.5: IDisposable implementation

In Listing A.5, the Child class implicitly implements `IDisposable` via its inheritance of Parent class. The above formulation of `hasIDisposable` is refined to describe the lookup for `IDisposable` in the class hierarchy in a recursive way, with the stopping condition: `name(i) = "IDisposable"`. In the formalism below, if $c1$ does not implement `IDisposable` interface, $c1$’s parent $c2$ can implement `IDisposable`.

$$\text{hasIDisposable}(c1) \equiv \forall c2 : \text{Classifier.geners}(c1, c2) \Rightarrow (\text{isInterface}(c2) \land \text{name}(c2) = "IDisposable") \lor \exists c3 : \text{Classifier.geners}(c2, c3).\text{hasIDisposable}(c3)$$  (A.7)
This predicate can be read as “does classifier c1 or any of its parent implement the IDisposable interface ”.

A.3 Additional Guidelines

In this section we have included the additional guidelines that we have formalized and implemented.

1. This guideline belongs to the Dispose pattern guidelines.

**Problem:** The only virtual method call contained in the finalizer of a class should be the Dispose bool method. Because the most derived implementation of the method will be run and it may not chain to its base class when it should in some cases /cite.

**Consequence:** Implementation of this guideline has resulted in many false positives, as it is not possible to decide when the overridden method should chain to the base class. The results of this guideline are difficult to predict. **Example:**

- In Listing[A.6] the finalizer makes a call to a virtual method Method1 which is not allowed according to this guideline.

```csharp
public class Class1 : IDisposable
{
    public Class1()
    {
    }
    protected virtual void Dispose(bool disposing)
    {
        ...
    }
    public void Dispose()
    {
        ...
    }
    public virtual void Method1()
    {
        ...
    }
    // Method1 is a virtual method, hence this is a violation.
    ~Class1()
    {
    }
```
Listing A.6: Finalizer of the class Class1 calls the virtual method Method1.

Figure A.6: Only virtual method, the finalizer can call is the Dispose bool.

Figure A.6 shows the class Class1 having the finalizer and the note indicating that the virtual method that can be called in the finalizer is the Dispose bool method.

Formalization:
We now define a new predicate for identifying if modifier of a method call mc is virtual.

\[ isVirtualMCall(mc) \equiv \forall mc: Methodcall. \exists m: Method. assocs(mc, m) \land isVirtual(m) \]

Similar predicates can be developed to all other method modifier types as well. Any finalized class implementing IDisposable, must not have any method calls other than Dispose(bool).

\[ \forall c: classes.hasIDisposable(c) \land isFinalized(c) \Rightarrow \exists mc: methodcalls.mc \in finalizer(c).isVirtualMCall(mc) \land name(mc) \neq "Dispose" \]

2. This guideline is similar to the above Dispose guideline and shares the same problem. Dispose bool method should not call any other virtual methods other than the base class Dispose bool method. As the most derived implementation may not chain to base class as required. Which is difficult to predict and analyze using static code analysis.

Example:

- In Listing A.7, the guideline is violated as Dispose method calls Method1 which is a virtual method.
public override void Method1()
{
    ...
}

protected virtual void Dispose(bool disposing)
{
    ...

    // Call to Method1() is not allowed as it is a virtual method.
    Method1();
    base.Dispose(disposing);
}

Listing A.7: Dispose method is calling a virtual method other than Dispose(bool).

**Formalization:** The only virtual method call allowed in any Dispose method should be Dispose(bool).

∀c : Class. ∀m : Method. isDispose(m, c) ⇒
¬∃mc : Methodcall. mc ∈ m.isVirtualMCall(mc) ∧ name(mc) ≠ ”Dispose”

### A.4 Dispose Pattern Example

In this section we have included a simple example in Listing A.8 satisfying dispose pattern guidelines. We have considered StreamWriter objects to be disposed in this case. A text file testing.txt is used as a resource which need to be disposed by the DisposableBaseClass and all the class that use the instance of this class.

```
// Implement IDisposable for Disposal
public class DisposableBaseClass : IDisposable
{
    private const string STR_Dtestingtxt = "d:\\testing.txt";
    // _report is a disposable field
    private StreamWriter _report;
    protected bool disposed = false;

    public DisposableBaseClass()
    {
        _report = new StreamWriter(STR_Dtestingtxt);
    }

    // Dispose() call GC.SuppressFinalize
    public void Dispose()
    {
        Dispose(true);
    }
}
```

123
GC.SuppressFinalize(this);

// Dispose bool method should be virtual as this not a sealed class
protected virtual void Dispose(bool disposing)
{
  if (!disposed)
  {
    if (disposing)
    {
      // call Dispose() method for every Disposable fields
      // in the class
      _report.Dispose();
    }

    disposed = true;
  }
}

// Finalizer should call Dispose(false)
~DisposableBaseClass()
{
  Dispose(false);
}

public class DisposableChildClass : DisposableBaseClass
{
  const string STR_DtestResultstxt = "d:\\testResults.txt";
  // _testResults is a disposable field
  private StreamWriter _testResults;

  public DisposableChildClass()
   : base()
  {
    _testResults = new StreamWriter(STR_DtestResultstxt);
  }

  // override Dispose(bool) as there is a base class
  // and this class is not a sealed class
  protected override void Dispose(bool disposing)
  {
    if (!disposed)
    {
      if (disposing)
      {
        // Dispose the disposable fields
        CleanUpResults();
      }
    }
  }
}
Listing A.8: Simple example satisfying dispose pattern guidelines

A.5 Work Division

The work distribution between the authors of this thesis, is shown in Table A.1.

Table A.1: Details of the sample test case for MVVM design pattern

<table>
<thead>
<tr>
<th>SL No.</th>
<th>Task</th>
<th>Assigned to</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction to standard patterns</td>
<td>Shravan</td>
</tr>
<tr>
<td>2</td>
<td>Introduction to WPF and MVVM patterns</td>
<td>Vinod</td>
</tr>
<tr>
<td>3</td>
<td>Design Pattern Catalogue (Requirement Analysis)</td>
<td>Vinod</td>
</tr>
<tr>
<td>4</td>
<td>Formalization of standard and custom design rules (Requirement refinement)</td>
<td>Shravan</td>
</tr>
<tr>
<td>5</td>
<td>Implementation of generic framework</td>
<td>Shravan and Vinod</td>
</tr>
<tr>
<td>6</td>
<td>Implementation of Philips specific design patterns and plug-ins</td>
<td>Shravan and Vinod</td>
</tr>
<tr>
<td>7</td>
<td>Validation of the implementation.</td>
<td>Shravan and Vinod</td>
</tr>
</tbody>
</table>