JPatternJudge: An Extensible Framework for Design Pattern Specification and Verification

A Thesis

submitted to the designated
by the General Assembly
of the Department of Computer Science and Engineering
Examination Committee

by

Anastasios Tsimakis

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN DATA AND COMPUTER SYSTEMS ENGINEERING

WITH SPECIALIZATION
IN ADVANCED COMPUTER SYSTEMS

University of Ioannina

February 2021

Acknowledgements

I am grateful to all who assisted me in this endeavor, and primarily prof. Apostolos Zarras who supervised this work and guided me until its completion. He was always available for anything, and his advice and suggestions were invaluable throughout the process. I am also thankful to the members of the examining committee, prof. Vassiliades and Papapetrou, for their insight and suggestions. I am also thankful to my family and my good friend Chris, who supported me in every step of the way.

Table of Contents

List of Figures

List of Tables

Abstract

Anastasios Tsimakis, M.Sc. in Data and Computer Systems Engineering, Department of Computer Science and Engineering, School of Engineering, University of Ioannina, Greece, February 2021.
JPatternJudge: An Extensible Framework for Design Pattern Specification and Verification.
Advisor: Apostolos Zarras, Associate Professor.

Design Patterns are a staple of software development in object-oriented programming languages. However, there are multiple reasons for a pattern that has been applied on a piece of code to be erroneous; be it from faulty understanding of its purpose, some mistake in the realization, or even the fact that since systems evolve and change over time, the pattern can be broken inadvertently.

To automatically verify the correctness of a pattern, we must first devise a way of specifying it. This thesis presents JPatternJudge, a framework that defines a specification model for design patterns, represented as a collection of roles and constraints, as well as mechanisms for detecting and verifying instances of design patterns in Java source code. JPatternJudge is implemented as an Eclipse plugin, and includes specifications of 18 different design patterns. We also present a measure of the framework's reusability and extensibility, as well as the results of verification of patterns on real Java projects.

Εκτεταμένη Περίληψη

Αναστάσιος Τσιμάκης, Δ.Μ.Σ. στη Μηχανική Δεδομένων και Υπολογιστικών Συστημάτων, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πολυτεχνική Σχολή, Πανεπιστήμιο Ιωαννίνων, Φεβρουάριος 2021.
JPatternJudge: An Extensible Framework for Design Pattern Specification and Verification.
Επιβλέπων: Απόστολος Ζάρρας, Αναπληρωτής Καθηγητής.

Τα σχεδιαστικά μοτίβα λογισμικού (software design patterns) αποτελούν ένα πολύ βασικό κομμάτι της ανάπτυξης λογισμικού, επιτρέποντας σε προγραμματιστές να υλοποιήσουν πολύπλοκα συστήματα. Κάθε σχεδιαστικό μοτίβο αποτελεί μια γενική λύση σε ένα συχνό πρόβλημα, και αναφέρονται συνήθως σε αντικειμενοστρεφείς γλώσσες προγραμματισμού όπως η Java.

Παρά την εκτεταμένη χρήση τους όμως, πολλές φορές ένα μοτίβο δύναται να είναι λανθασμένο για διάφορους λόγους: ίσως ο προγραμματιστής δεν κατάλαβε πλήρως τον σκοπό του pattern, ή έκανε κάποιο λάθος στην υλοποίησή του. Μπορεί επίσης το σφάλμα να προέκυψε λόγω της αναπόφευκτης εξέλιξης του συστήματος, όπου οι αλλαγές στον κώδικα κατέστρεψαν τη δομή του. Αυτό δεν είναι κάτι που είναι πρακτικά εφικτό να ελεγθεί από ανθρώπους, άρα ένα αυτοματοποιημένο εργαλείο θα ήταν πολύ χρήσιμο.

Για να μπορέσει όμως ένα εργαλείο να επαληθεύσει την ορθή υλοποίηση ενός pattern, θα πρέπει πρώτα να έχει έναν συγκεκριμένο ορισμό για αυτό. Σε αυτή την εργασία, παρουσιάζουμε έναν τρόπο να ορίσουμε τα σχεδιαστικά μοτίβα ως ένα σύνολο από κλάσεις που εξυπηρετούν έναν συγκεκριμένο ρόλο στο μοτίβο, καθώς και ένα σύνολο από δηλωτικούς περιορισμούς (declarational constraints) τους οποίους πρέπει να υπακούν. Οι περιορισμοί αυτοί εκφράζονται σαν σχέσεις μεταξύ κόμβων του αφηρημένου συντακτικού δέντρου (Abstract Syntax Tree, AST) του πηγαίου

κώδικα. Με βάση αυτό το σύστημα, παραθέτουμε επίσης 18 ορισμούς για design pattern.

Τέλος, παρουσιάζουμε το JPatternJudge, ένα framework για το Eclipse IDE, το οποίο χρησιμοποιεί τους ορισμούς αυτούς για αυτόματη επαλήθευση των patterns και εστιάζει ιδιαίτερα στην επεκτασιμότητα. Για αναγώριση των κλάσεων που συμμετέχουν σε ένα pattern και έχουν έναν ρόλο, το JPatternJudge χρησιμοποιεί ένα καινούργιο Annotation το οποίο προσθέσαμε στην Java.

Chapter 1

Introduction

The formal definition of design patterns, as used today, first appears in Christopher Alexander's 1977 book "A Pattern Language: Towns, Buildings, Construction" [1]. Despite originating from architecture and civil engineering, patterns were quickly incorporated in the field of software engineering as well, particularly in the landmark 1995 publication of the so-called "Gang of Four": Design Patterns: Elements of Reusable Object-Oriented Software [2]. In their book, Gamma, Helm, Johnson, and Vlissides introduced 23 software design patterns for object-oriented programming: *general solutions to common problems in software design, presented in a form that can be translated to code from natural language*. Since then, design patterns have been an integral part of software engineering for decades, playing an important role in writing good quality code - in particular, code that is easily extensible and maintainable.

The most common and important design patterns are described in various catalogs, either in print or, more commonly lately, online. These catalogs act as a quick reference guide for developers, giving an overview of how each pattern can be implemented. However, there is a large element of uncertainty as to how exactly each pattern can be defined and implemented; this can lead to various errors, whether due to misunderstanding the pattern's purpose or to an incorrect implementation.

1.1 Motivation

The driving force behind the idea of JPatternJudge is the issue that despite the significant importance of design patterns in software engineering, there are still very common and fundamental errors being committed by developers when the time comes to implement a pattern. For example, a 2020 study [3] illustrated several common mistakes that occur in the implementation of the Command pattern due to inappropriate division of responsibilities among the pattern's constituent classes.

However, we could reduce the number of instances where a pattern is erroneously implemented by using an automated verification mechanism. A tool can analyze a given implementation of a design pattern according to some criteria, and decide if it is correct or not. If there are errors, the tool can describe them to the developers, assisting in better understanding the pattern and guiding them how to fix the code to properly adhere to the pattern structure. In order for such a tool to work though - or in fact for any sort of verification to succeed - we must first define the pattern as strictly as possible. This is in fact not fully possible, since specifying a generic solution is a contradictory notion. The most practical solution to this is to have specifications for the most commonly accepted and used interpretations of each pattern, while using a specification model that has a high degree of extensibility, allowing developers to easily add more variations.

There have been some tools in the past with the same goal of pattern verification, such as HEDGEHOG [4], jStar [5], and others. However, these tools all have several limitations: many of them are outdated and unsupported, with some of them no longer available for use. More importantly, extensibility and user-friendliness is often an issue, since they utilize languages for pattern specification that can be difficult for the average developer to fully utilize - HEDGEHOG, for example, uses a custom made Prolog-derived language called SPINE, and jStar uses the assertion language of separation logic. For that reason, JPatternJudge was conceived with extensibility being a prime characteristic, while still using a simple and intuitive specification language so as to assist the average developer with easily including additional variations of patterns that commonly occur in their own context of work.

Essentially, we are motivated to assist two different sets of people with our framework: First, pattern experts, who wish to quickly and easily create new design pattern specifications. These users want high reusability of concepts to reduce the need for

new additions, while still being easily extensible for the addition of the new pattern specifications. Secondly, software developers, even those without extensive knowledge of design patterns, should be able to use the framework in practice with minimal effort, especially when it comes to specifying instances of design patterns in their code for verification.

In order to have the aspects of reusability, extensibility, and user-friendliness, we must decide on the appropriate way to express the design pattern specifications in such a way that they are both easily understood conceptually, as well as having a formal representation that lends itself to extension. One might think that making pattern specification more easily understood would also make it simpler and thus not supporting as strict a definition as possible, but it is essential if we wish to move the field of pattern specification and verification from theoretical-only tests on small pieces of code to a practical application on real software systems. Previous works have approached the field of pattern specification from many different angles; focusing on the instances where extensibility was a concern, there exist several languages specifically created for design pattern modeling such as DPML [6], BPSL [7], and SPINE. However, none of them satisfied our needs - predominantly because they are not easily understood and utilized, as mentioned earlier. We will explore some of the languages and specification methods in detail in Chapter 2, explaining our decision to use a new form of specification.

1.2 Contribution

This thesis presents JPatternJudge, a framework that allows precise specification of design patterns and allows developers to automatically verify the presence and correctness of these patterns for the Java programming language. A pattern specification is expressed as a set of declarative constraints regarding both static and dynamic aspects of the classes participating in a given pattern, i.e., relations such as the existence of fields of a certain type, invocations of methods, inheritance relations, et cetera.

The core of the proposed approach is an extensible model that specifies highly reusable constraints to facilitate the specification of design patterns and the realization of the respective verification mechanisms. The proposed constraints are defined and combined into pattern specifications using simple, developer-friendly object-oriented

programming constructs of Java, which is the target programming language of our framework. This means that adding constraints and using them for pattern specifications amounts to developing simple classes and combining existing model elements. In addition, we provide specifications for 18 software design patterns, along with comments and notes on certain aspects of the specifications.

A noteworthy addition compared to previous works using declarative constraints is the introduction of negative constraints - so far, such specification methods only took into account the necessity for a given relation between classes to exist. We include certain constraints where a given relation must NOT exist; for example, a certain class does NOT have a field of a given type.

Finally, we showcase the framework implemented as a plugin for the Eclipse integrated development environment, with testing done on two sets of real Java projects to validate the framework's verification capabilities. JPatternJudge successfully verifies design pattern instances and reports on any errors detected, with the instances being specified using a simple custom Java annotation in the source code being examined. We also showcase the high level of reusability of constraints among our pattern specifications, as well as the framework's extensibility in relation to both constraints and pattern specifications as a measure of lines of code.

1.3 Thesis Structure

The thesis consists of 7 chapters as follows:

- Chapter 1, Introduction provides an overview of the contributions of this thesis as well as its structure.
- Chapter 2, Related Work describes some fundamental concepts related to our work, and explores work related to the field of pattern specification and verification.
- Chapter 3, Pattern Specification Model presents the details of our metamodel for design pattern specification
- Chapter 4, Design Pattern Specifications provides a list of specifications for 18 GoF design patterns supported by JPatternJudge.

- Chapter 5, Framework Mechanisms explains JPatternJudge's system architecture, design, and implementation, as well as how it can be used by developers.
- Chapter 6, Validation deals with practical applications of JPatternJudge, showing the results of using the framework on real projects. It also gives a measure of reusability of the constraints defined in the model, as well as extensibility in terms of adding new constraints and design patterns.
- Chapter 7, Future Work and Conclusion gives further ideas for expanding the framework and its utilities and provides a conclusion of the work presented.

Chapter 2

Related Work

In this chapter we will present an overview of some basic concepts necessary for our work. We will also examine works similar to ours in literature.

2.1 Design Patterns

The first mention of design patterns being formally defined in literature appears in a 1977 book by Christopher Alexander, titled "A Pattern Language: Towns, Buildings, Construction" [1]. As per the title, the concept of patterns began in the field of architecture and civil engineering, but can be easily applied to almost any field. In particular:

"Each pattern describes a problem that 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."

This concept was first transferred to the field of software engineering in the 1987 OOPSLA Conference by Kent Beck and Ward Cunningham, in a technical report titled "Using Pattern Languages for Object-Oriented Programs" [8]. In [8], the authors list 5 basic patterns for GUI development in the Smalltalk language, and explain

how creating a pattern language for object-oriented programming can be a great boon to programmers when it comes to facing certain recurring issues in software development. However, the work that was primarily responsible for popularizing software design patterns to the scale they are used today was the 1995 book "Design Patterns: Elements of Reusable Object-Oriented Software" [2] by the so-called Gang of Four: Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. In [2], the writers introduced 23 design patterns, serving as generic solutions for some common problems under specific contexts in software engineering. These patterns in essence show relationships and interactions between classes, without however specifying the actual implementation of each class, providing a layer of abstraction, and were divided in three categories. By the writers' own admission, these categories are somewhat arbitrary, and certain patterns could belong to another category; however, they have remained as such since they serve their purpose as-is. Each pattern was accompanied by examples in C++ and Smalltalk, since the Java language hadn't yet been fully developed at the time.

Creational patterns deal with mechanisms to create new objects in cases where it is important to decide what type of object is being created. Creational patterns include the Abstract Factory, Builder, Factory Method, Prototype, and Singleton patterns.

Structural patterns deal primarily with using object composition to extend the functionality of a class. They include the Adapter, Bridge, Composite, Decorator, Facade, Flyweight, and Proxy patterns.

Finally, Behavioral patterns deal with communication between objects, and how to efficiently do that without overcomplication. They are the most numerous, including the Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, and Visitor patterns.

Besides its definition, the purpose of a software design pattern is to improve two fundamental characteristics of the code it is applied on: extensibility and maintainability, which are very tightly connected with each other. The majority of effort that is spent on industrial software systems goes towards maintenance, rather than the initial development. This means that techniques to reduce defects such as technical debt and code smells are very useful to reduce costs as well. The question that arises is whether software design patterns can indeed help in this aspect.

Empirically, the answer is yes; design patterns are taught worldwide in software engineering courses, and their use is heavily promoted and encouraged. However,

there have also been studies on the matter as well, attempting to measure the impact that design patterns have on the quality of code. A number of such studies were collected and studied as part of a systematic literature review in 2019 [9], with interesting results: Multiple studies ([10],[11],[12]) examined whether a class that participates in a pattern is larger in size or more prone to changes than classes outside of patterns. Size and change proneness in general are agreed to be an obstacle for maintainability; large class size is the archetypal code smell, with all it entails. The results were inconclusive, depending on the pattern examined. Certain pattern classes were more prone to changes compared both to other patterns and non-pattern classes; in other cases, there was no significant difference in change proneness. However, the question here is whether the classes were prone to change primarily because they were part of a pattern, or the opposite: they were part of a pattern because they have an important role in the system and thus are often the subject of changes. This means we cannot say with certainty how design patterns affect code quality in this aspect.

On the other hand, other studies ([13],[14],[15]) used controlled tests to discover that documenting instances of design patterns in code leads to a marked increase of comprehension of the code, and thus obviously easier maintenance and extension. However, this doesn't tell us something about the difference in code quality between using a pattern and not doing so, merely the effects of documenting their presence. Finally, in terms of actual quality metrics, a study [16] showed that design patterns improve metrics such as coupling and cohesion. In general, we can see that the empirical experience holds true: design patterns do indeed have a positive impact on code.

2.2 Design Pattern Specification and Verification

The problem of design pattern verification is the following: given a piece of source code that we suspect or know is the realization of a design pattern, we want to decide whether this pattern was implemented correctly. Obviously, in order to judge the correctness of a given concrete implementation, we must first define the correctness in an abstract level. This is the specification of a design pattern, which is its own problem.

Usually, both in printed catalogs of design patterns such as the one by the GoF as

well as various online sites (e.g. [17],[18]), patterns are described in an abstract way and accompanied by examples. This is usually sufficient for developers to grasp the general concept, purpose and structure of the pattern, but it is obviously not enough when we need verification, and especially in an automated way. In this case, we need a very strict formal specification of patterns.

The problem is that in truth, trying to specify a pattern is a bit of an oxymoron: a pattern implies a solution to a number of similar problems; a generic guide that can be customized to the given context of the problem at hand. A specification, on the other hand, is the exact opposite: an exhaustive list of all possible characteristics, attributes, constraints, et cetera of the concept we wish to explore. As such, we can see that it is impossible to fully match these opposite concepts together. Each design pattern is interpreted in a slightly different way by each programmer, so an exact specification is out of our reach; we cannot account for every single possible interpretation.

Previous works have several approaches to this issue, each with different drawbacks. The main issue in many of them that we wish to avoid is that even though the specification methods are quite strong from a technical standpoint, they are difficult to learn and utilize, meeting that even though theoretically extensibility is achievable, in practice it is quite a difficult task. In addition, for many of them verification of source code was a secondary concern.

Ideally, we want the specification method to be as easy to be expressed in Java as possible, both to allow the developers to understand the patterns themselves, as well as be able to write extensions not only on the theoretical level of the pattern language, but also on the practical level of the automated verification tools. If those tools are developed in Java, they are made much more accessible. Our specification method contains aspects from several of the following languages, but attempts to move more towards OOP concepts. In particular, we make use of declarative constraints expressed as classes of our OO framework, combined to be used in a design pattern specification. We will examine the exact model of specifications in more detail in Chapter 3. In the following subsections, we will go over some of the most influential previous works in the field of design pattern specification and verification.

2.2.1 Balanced Pattern Specification Language

The Balanced Pattern Specification Language (BPSL)[7] can express both structural and behavioral aspects of a design pattern. This is done with two separate mechanisms: using First-Order Logic for the structural aspects, and Temporal Logic of Actions [19] for the behavioral ones. These are both robust mathematical formalisms, that lead to strict, well-defined specifications. However, this also leads to high complexity; an average software developer in an industrial project would find it difficult to understand how exactly a pattern is defined, and even more so to extend the language with new specifications. However, it is important to note that BPSL was designed primarily with the purpose of automatically generating pattern instances in code, rather than verifying the validity of existing ones.

2.2.2 jStar

jStar [5] is a tool aimed at general verification of Java programs, not specifically for design patterns. It uses separation logic, a type of logic that is used to ascertain the correctness of programs and is derived from Hoare logic. It deals primarily with the behaviour of programs, by using assertions that contain a precondition and a postcondition over a single command, in contrast to Hoare logic, whose pre- and postconditions over the entire program state. While separation logic is well suited for verification - it was created for it - it is also a very complex type of logic that is not within the means of the average software developer to understand or properly utilize.

2.2.3 Design Pattern Modeling Language

The Design Pattern Modeling Language (DPML) [6] uses a visual specification diagram to model design patterns. It is essentially an extended form of UML, containing additional notation tailored to design pattern specification. There are specific notations that represent concepts such as interfaces , operations and methods (essentially declarations and definitions of methods respectively, in more OOP terms), as well as more specific binary relations than UML such as “implements”, “declared in”, “creates”, etc. On the surface, DPML seems quite close to our own specification language, albeit in a visual format rather than a textual one - already a disadvantage for tool support. However, DPML seemed to lack more formal definitions for the relations

and constraints of the pattern's participants (classes, interfaces, methods, etc), relying on natural language to convey the necessary information. DPML had two automated tools associated with it, DPTool and its successor MaramaDPTool, which was however discontinued in 2008. Both tools supported the creation of pattern specification diagrams, as well as a mechanism for design pattern instantiation. Finally, they supported the verification of a design pattern UML model - however, not the verification of a pattern in the actual source code, which is the focus of our work.

2.2.4 HEDGEHOG

HEDGEHOG [4] was an influential tool that is the most similar to our own work, focusing primarily on the verification of existing patterns in Java source code - it is, however, no longer available for use. It uses SPINE, a custom declarative language based on Prolog syntax for pattern specification. This language however, in contrast to Prolog, allows the use of evaluable sets and propositions to be used as part of the rules. It was also easier to integrate in a Java tool than Prolog. Regardless, it also has certain weaknesses; the syntax is more complex than Prolog, which is already foreign to someone used to OOP-style concepts. In addition, some of the pattern specifications rely on developers to conform to specific naming conventions, such as method names beginning with "add" or "get". This is not something we can assume will be true, and can lead to inaccuracies.

Chapter 3

Specification Model

In this chapter we will present the details of the conceptual model used to express design pattern specifications. Examples of such specifications can be found in Chapter 5.

3.1 Design Patterns

Each design pattern is applied on a particular set of classes and interfaces, collectively referred to as types in Java. Each type that participates in a pattern has a specific purpose, or role, and multiple types can have the same role. These roles have a unique name in the pattern, signifying approximately what their purpose is in the context of the problem the pattern is supposed to resolve. For example, in the Command pattern, we have an interface serving the Command role, a number of classes serving the Concrete Command role, a class serving as the Receiver, and a class serving the Invoker role.

In Figure 3.1, we can see a UML metamodel of a design pattern specification. It contains a number of types, each with a role, and a number of constraints for each

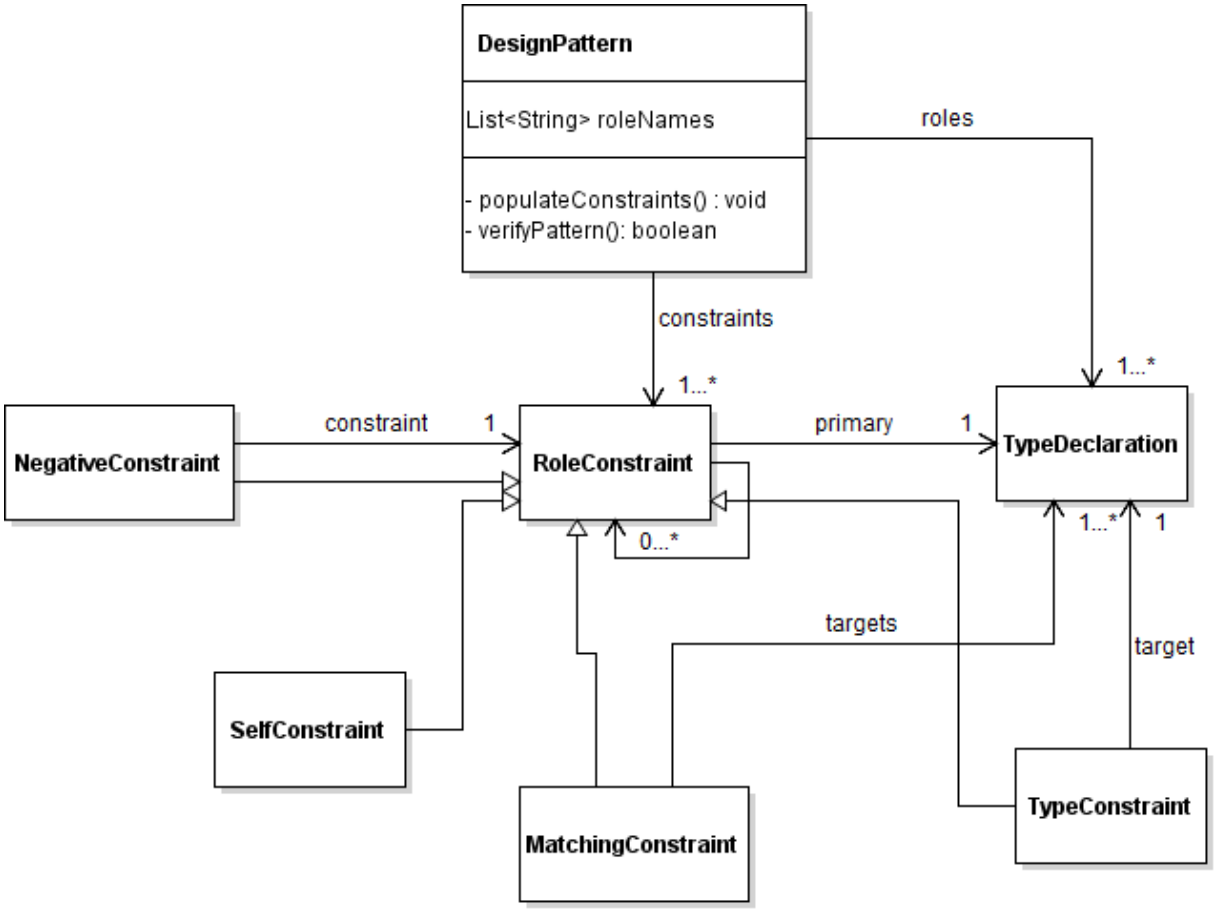

Figure 3.1: UML diagram of pattern specification metamodel

participating type. Each Constraint has the primary participant type, and a number of target types if applicable. In addition, each Constraint may contain additional Constraint elements. Below, we further define the model using OCL [20]:

A DesignPattern has a role collection, which is a set of tuples. Each tuple represents a role, and contains the role's name and the set of types that serve that role in the pattern.

```
context DesignPattern::roles : Set{Tuple{role: String, types: Set{Type}}}
```

The FullRoleConstraints invariant expresses that if a type with a role is a primary participant in a Constraint, then all types with that role must be primary participants in a Constraint of that kind - essentially, as mentioned before, all types with the same role must obey the same Constraints.

```
context DesignPattern inv FullRoleConstraints:
self.roles -> forAll(rt | if
self.constraints -> includes(c | rt.types ->
includes(ct | ct = c.primaryParticipant))
then
rt.types -> forAll(x | self.constraints ->
includes(c2 | c2.primaryParticipant = x and c2.oclIsTypeOf(c)))
```

Additionally, the FullTypeConstraintTargets makes sure that if there is a TypeConstraint with a primary participant of a given role and a target participant with another role, then there is a TypeConstraint of that sort for each type of the primary participants role and each type of the target participant's role - essentially an all-to-all connection between the primary and the target participant roles.

```
context DesignPattern inv FullTypeConstraintTargets
self.roles -> forAll(rt,rt2 | if
self.constraints -> includes(c | c.oclIsKindof (TypeConstraint) and
rt.types -> includes(ct | ct = c.primaryParticipant)) and
rt2.types -> includes(ct2 | ct2 = c.targetParticipant))
then
rt.types->forAll(x | self.constraints ->
includes(c2 | c2.primaryParticipant = x
and c2.oclIsTypeOf(c))) and rt2.types->forAll(x2 | self.constraints ->
includes(c2 | c2.targetParticipant = x2 and c2.oclIsTypeOf(c)))))
```

The RolesExist invariant expresses that for each role in the pattern, there must be

at least one type with that role.

```
context DesignPattern inv RolesExist:
self.roles -> forAll(rt | rt.types -> notEmpty())
```

Finally, the AllConstraintsSatisfied invariant ensures that for a DesignPattern to be verified as correct, all of its Constraint elements are satisfied.

```
context DesignPattern inv AllConstraintsSatisfied:
self.verifyPattern() = true implies
self.constraints -> forAll(c | c.satisfied = true)
```

3.2 Constraints

In order for participating types to properly fulfill their role, they must have certain static and dynamic aspects. We represent those necessary aspects as a set of declarative constraints that the types with a given role must obey, similar to SPINE, the language used by HEDGEHOG. There are, however, several differences. SPINE has a syntax similar to Prolog, allowing the combination of constraints with logical operands (AND/OR). In our case, all constraints must hold true, thus essentially all of them are linked with the AND operand by default. In addition, all types that are part of the design pattern with the same role must all obey the same constraints.

The static or dynamic aspects of a type that a Constraint expresses are specified with respect to specific structures of the Abstract Syntax Tree (AST) of that type. This type is called the *primary participant* of the Constraint. These AST structures essentially express a relation (such as inheritance or dependency) of the primary participant with some other type, called the *target participant* of the Constraint. As mentioned previously, all types with the same role must obey the same Constraint elements. This means that when we mention a primary participant, we usually express it with the name of its role rather than the participant's implementation name, and imply that there are multiple such Constraints; each with a different primary participant implementation for all types of the primary participant's roles. In our model we assume the standard AST model structure of the Java Language Specification [21]. In Figure 3.2, we can see an abstract model of the AST nodes that are most commonly used by our Constraints. For our implementation of JPatternJudge, we used Eclipse's AST library [22].

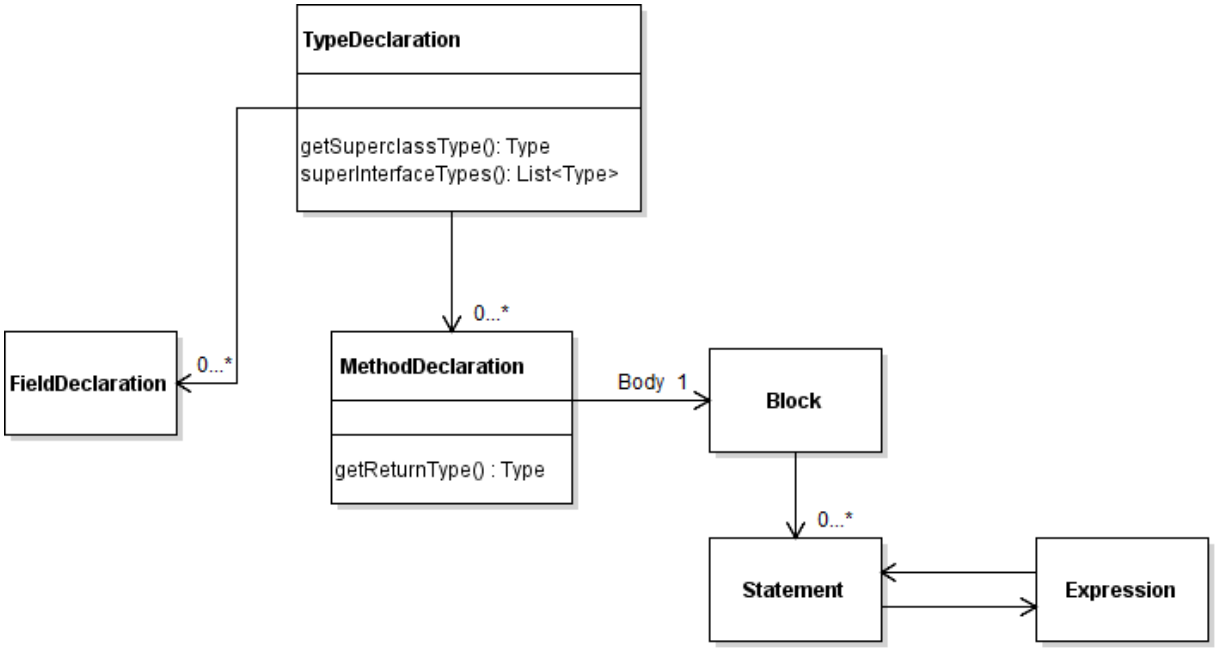

Figure 3.2: UML diagram of part of the AST model.

Constraint has 3 specializations according to the number of target participants: SelfConstraint, TypeConstraint, and MatchingConstraint. In addition, we can characterize a Constraint as being either simple, or composite. A composite Constraint makes use of other Constraint elements, either simple or recursively composite, to express the required AST structure. All of the simple Constraint elements are used very often in pattern specifications, both on their own and as part of composite Constraints, a fact we will see in more detail in Chapter 5. Finally, we can make a distinction between positive and negative Constraints, a concept that previous works do not make use of. A positive Constraint is an assertion that a static or dynamic aspect does indeed exist for a type. A negative Constraint, on the other hand, asserts that a static or dynamic aspect categorically does NOT exist. We will go over each Constraint category in Section 3.3

Composite Constraints have certain limitations to what Constraint elements they can contain, as expressed by the OCL invariants below. The AllConstraintsSatisfied invariant expresses that for the pattern to be verified as correct, all of its Constraint elements must hold true. Finally, the SelfInternalConstraints and TypeInternalConstraints restrict what kinds of Constraint elements can be included in a composite Constraint of the respective sort, in order to maintain the appropriate number of target participants (0 for Self, 1 for Type). MatchingConstraint elements do not need

such an invariant.

```
context SelfConstraint inv SelfInternalConstraints:
self.internalConstraints -> forAll(ic | ic.oclIsTypeOf(SelfConstraints))
context TypeConstraint inv TypeInternalConstraints:
self.internalConstraints -> forAll(ic | ic.oclIsTypeOf(SelfConstraint)
or ic.oclIsTypeOf(TypeConstraint))
```

3.3 Constraint Categories

In the following subsections, we will further explore each category of Constraints, as well as the reusable Constraints used for specifications. Constraints used only in the context of a single pattern will be presented in Chapter 5, in the appropriate pattern specification. Each Constraint will be formalized using the Object Constraint Language (OCL).

3.3.1 SelfConstraint

A SelfConstraint element does not have a target participant. The primary participant type does not have a relation with another type, but rather we examine a particular inherent characteristic. A composite SelfConstraint may obviously only contain other SelfConstraint elements, to maintain the absence of a target participant.

TypeIsInterface: The primary participant is an interface.

```
context TypeIsInterface inv:
self.primaryParticipant.isInterface() = true
```

ClassIsAbstract: The primary participant is a class with the abstract modifier.

```
context ClassIsAbstract inv:
self.primaryParticipant.getModifiers() -> includes(m | m.isAbstract() = true)
```

TypeIsInterfaceOrAbstractClass: The primary participant is either an interface, or a class with the abstract modifier.

```
context TypeIsInterfaceOrAbstractClass inv:
self.constraints() -> includes(c1 | c1.oclIsTypeOf(ClassIsAbstract))
context TypeIsInterfaceOrAbstractClass inv:
self.constraints() -> includes(c2 | c2.oclIsTypeOf(TypeIsInterface))
```

These three are the only SelfConstraint instances utilized for our specifications, but other examples of SelfConstraint are those that would express the existence of specific access modifiers for a class (private, protected, public) or the static modifier.

3.3.2 Type Constraints

A TypeConstraint has a single target participant, meaning that the primary participant has a relation with exactly one other type. This relation can be of any sort as defined by UML, such as inheritance or dependency. A composite type Constraint can contain other TypeConstraint or SelfConstraint elements, but not MatchingConstraints, in order to maintain the single target participant. TypeConstraint are the most numerous and the most widely used in specifications.

ClassExtendsSuperclass: The primary participant is a class that extends the target participant.

```
context ClassExtensSuperclass inv:
self.primaryParticipant.getSuperclass().oclIsTypeOf(targetParticipant)
```

TypeImplementsInterface: The primary participant implements the target participant interface.

```
context TypeImplementsInterface inv:
self.primaryParticipant.getInterfaces() ->
includes(i | i.getType().oclIsTypeOf(targetParticipant))
```

TypeInheritsFrom: The primary participant has an inheritance relation with the target participant, extending it if it is a class and implementing it if it is an interface.

```
context TypeInheritsFrom inv:
self.constraints -> includes(c1 | c1.oclIsTypeOf(ClassExtendsSuperclass))
context TypeInheritsFrom inv:
self.constraints -> includes(c2 | c2.oclIsTypeOf(TypeImplementsInterface))
```

ClassHasFieldOfType: The primary participant has a field whose type is the target participant.

```
context ClassHasFieldOfType inv:
self.primaryParticipant.getFields() -> (f | f.getType().oclIsTypeOf(targetPrticipant))
```

ClassHasFieldCollectionOfType: The primary participant has a field that is a collection of objects whose type is the target participant.

```
context ClassHasFieldOfType inv:
```

```
self.primaryParticipant.getFields() -> (f | f.oclIsTypeOf(Collection(targetParticipant)))
```

ClassHasVariableOfType: The primary participant has a method which contains a variable whose type is the target participant.

```
context ClassHasVariableOfType inv:
self.primaryParticipant.getMethods() -> includes(m | m.getVariables() -> includes(v
| v.getType().oclIsTypeOf(targetParticipant)))
```

ClassHasFieldOrVariableOfType: The primary participant either has a field whose type is the target participant, or has a method which contains a variable whose type is the target participant.

```
context ClassHasFieldOrVariableOfType inv:
self.constraints -> includes(c1 | c1.oclIsTypeOf(ClassHasFieldOfType))
context ClassHasFieldOrVariableOfType inv:
self.constraints -> includes(c1 | c1.oclIsTypeOf(ClassHasVariableOfType))
```

ClassInstantiatesObjectOfType: The primary participant has a method which instantiates an object whose type is the target participnt.

```
context ClassInstantiatesObjectOfType inv:
self.primaryParticipant.getMethods() -> includes(m | m.getExpressions() -> includes(
e | e.oclIsTypeOf(ClassInstanceCreation) and e.getType().oclIsTypeOf(targetParticipant)))
```

ClassHasMethodWithParameter: The primary participant has a method, which has a parameter whose type is the target participant.

```
context ClassHasMethodWithParameter inv:
self.primaryParticipant.getMethods() -> includes(m | m.getParameters() -> includes(p
| p.typeOf().oclIsTypeOf(targetParticipant)))
```

ClassHasMethodWithReturnType: The primary participant has a method whose return type is the target participant.

```
context ClassHasMethodWithReturnType inv:
self.primaryParticipant.getMethods() -> includes(m |
 m.getReturnType().oclIsTypeOf(targetParticipant))
```

ClassHasMethodWithReturnCollectionOfType: The primary participant has a method that returns a collection of objects whose type is the target participant.

```
context ClassHasMethodWithReturnCollectionOfType inv:
self.primaryParticipant.getMethods() -> includes(m |
 m.getReturnType().oclIsTypeOf(Collection(targetParticipant)))
```

ClassInvokesMethodOf: The primary participant has a method which invokes a method that is declared in the target participant.

```
context ClassInvokesMethod inv:
self.primaryParticipant.getMethods() -> includes(m | m.getExpressions() -> includes(e
| e.oclIsTypeOf(MethodInvocation) and self.targetParticipant.getMethods() -> includes(m2
| m2.name = e.methodName)))
```

3.3.3 Matching Constraints

MatchingConstraint instances are always composite, and not cross-pattern reusable. Each MatchingConstraint is used only in a specific design pattern and perhaps its variations. A MatchingConstraint has more than one type as target participants; however, every type must have the same role in the pattern. For TypeConstraint elements, the combination of the FullRoleConstraints and FullTypeConstraintTargets invariants means that if a pattern has a TypeConstraint element with primary participant of role X, and target participant of role Y, it also has such TypeConstraint elements with primary participants every type with role X, and target participants every type with role Y. However, only in certain cases, we don't want the FullTypeConstraintTargets invariant to hold - we have a TypeConstraint with a given target participant, and we know the role of the target participant, but we don't know which of the types serving that role is the correct one. To resolve this, we create a MatchingConstraint: it is composed of a set of TypeConstraint elements, and has as a set of target participants the set of all types that serve a role. The TypeConstraint elements are checked with each individual type as a target participant, and if one of them is satisfied, then the entire MatchingConstraint element is also satisfied.

For example, in the Abstract Factory pattern, we have multiple Concrete Factories and multiple Products, but each Concrete Factory produces only one Product. Thus, each concrete factory has a MatchingConstraint with all products; and for the Constraint to hold true, there must be at least one product for which the internal Constraint elements hold true.

3.3.4 Negative Constraints

As mentioned earlier, a negative Constraint is essentially the assertion that a positive Constraint (any of the Constraints mentioned earlier) does NOT hold true. As

expressed by OCL:

```
context NegativeConstraint:: satisfied : boolean
derive:
not(constraint.satisfied)
```

The reason previous works do not mention negative Constraints might be because they seem quite trivial, and to a certain degree that would be correct. For example, we could add a negative Constraint to every single pattern that all fields of all roles must not be public - something that is taught to programmers from the very beginning of introductions to object-oriented programming as a fundamental necessity of good quality code. That would indeed be a trivial Constraint that doesn't really add something to the specification. However, there are some instances that negative Constraints can be important without being so obvious, especially depending on certain interpretations of patterns, which we will see in Chapter 5. Also, in some cases, it would not be easy to express negative Constraints - for example, in DPML using visual specification diagrams.

Chapter 4

Framework Mechanisms

In this chapter we will discuss the abstract mechanisms of the JPatternJudge framework, as well as certain aspects of our actual implementation. We can divide them in the following parts:

- Constraint Realization: How the framework realizes the constraints defined in the model.
- Design Pattern Realization: How the framework realizes the design patterns defined in the model.
- Design Pattern Instance Detection: Given a piece of source code, how the framework detects which classes participate in a pattern.
- Design Pattern Verification: Given a set of design patterns, how the framework verifies their correctness.

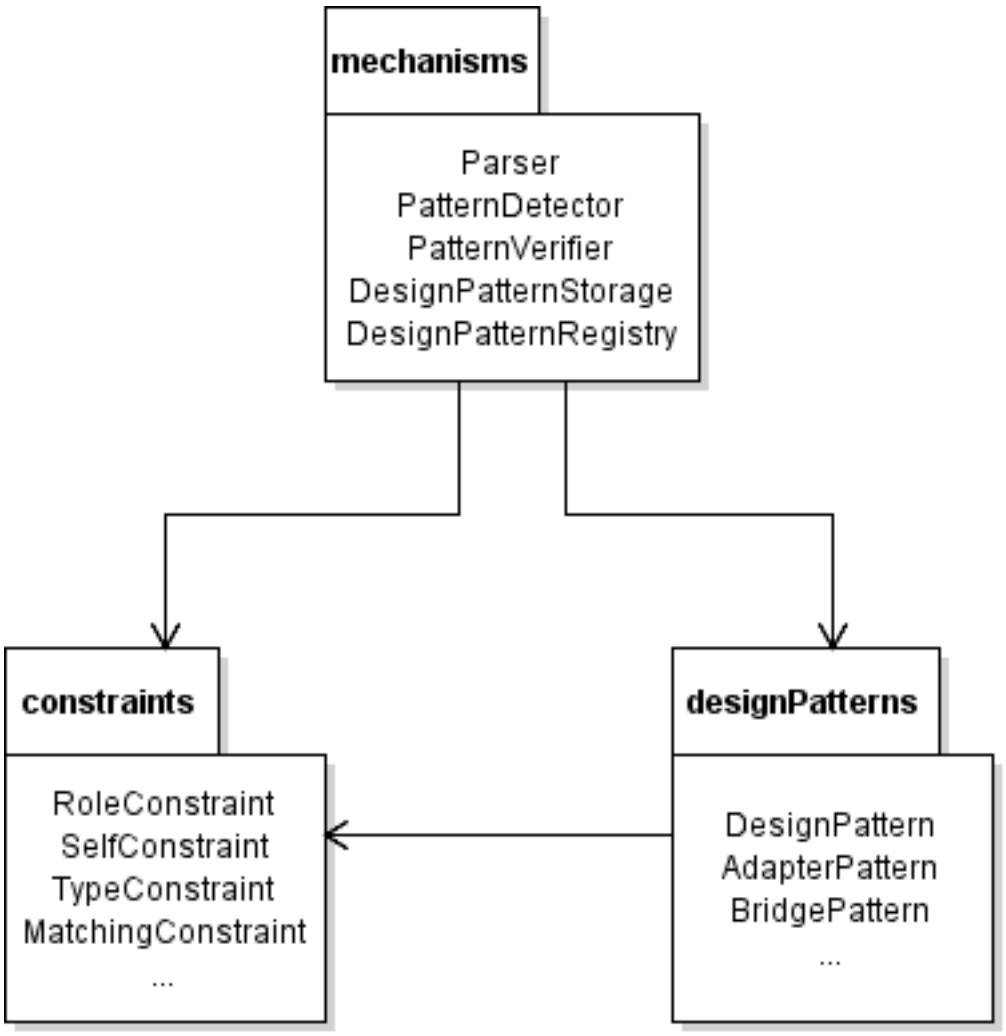

Figure 4.1: Package diagram of JPatternJudge.

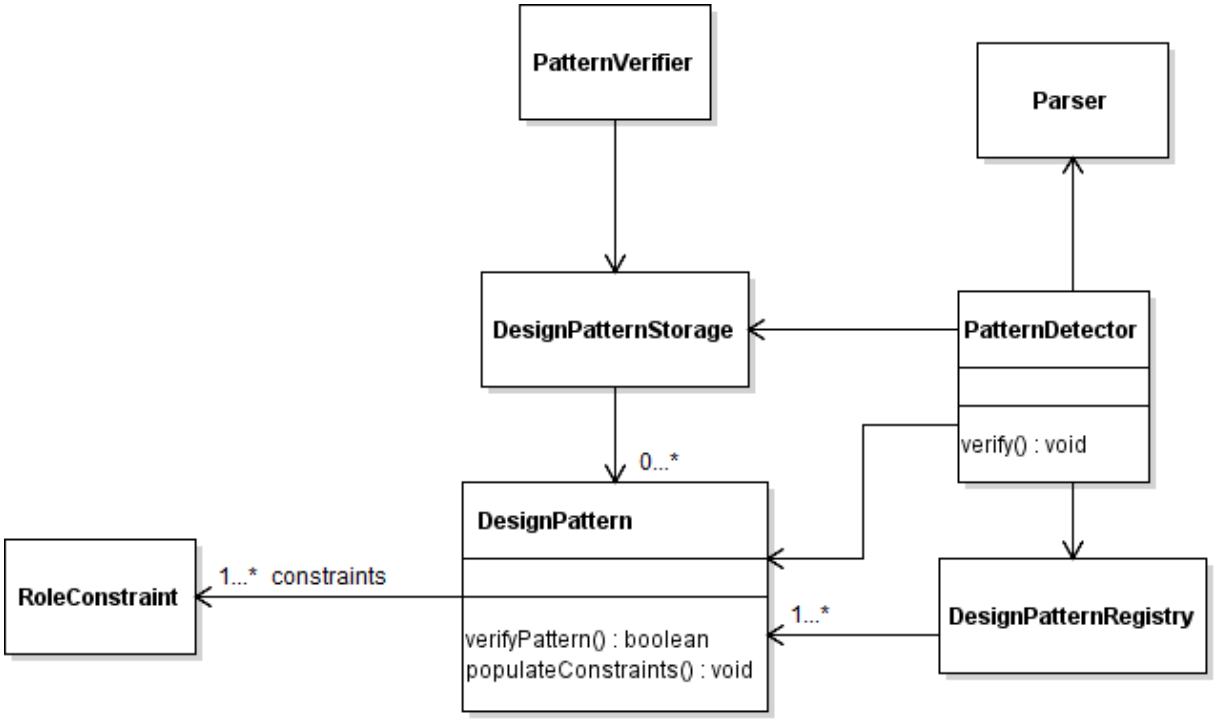

Figure 4.2: UML diagram of JPatternJudge's architecture.

We will discuss each one individually in the following sections. We can see the package diagram of JPatternJudge in 4.1, and a more detailed architecture in 4.2.

The DesignPatternRegistry class contains a collection with a single object for each subclass of DesignPattern - i.e. one of each patterns specified in the framework. It is used as a prototype to check against pattern instances detected that are to be verified. The DesignPatternStorage class on the other hand contains a collection of the actual design pattern instances detected.

4.1 Constraint Realization

The realization of the constraints follows the abstract model presented in 3, and is concentrated in a single package. The class diagram of the package can be seen in 4.3, including the main hierarchy and some cases of the concrete classes.

The RoleConstraint class is an abstract superclass representing the general concept of the constraint as defined in 3. It has a TypeDeclaration AST node as a field, containing the primary participant of any constraint. It also contains abstract methods for error messages, which we will expand upon later. Below that in the hierarchy, we

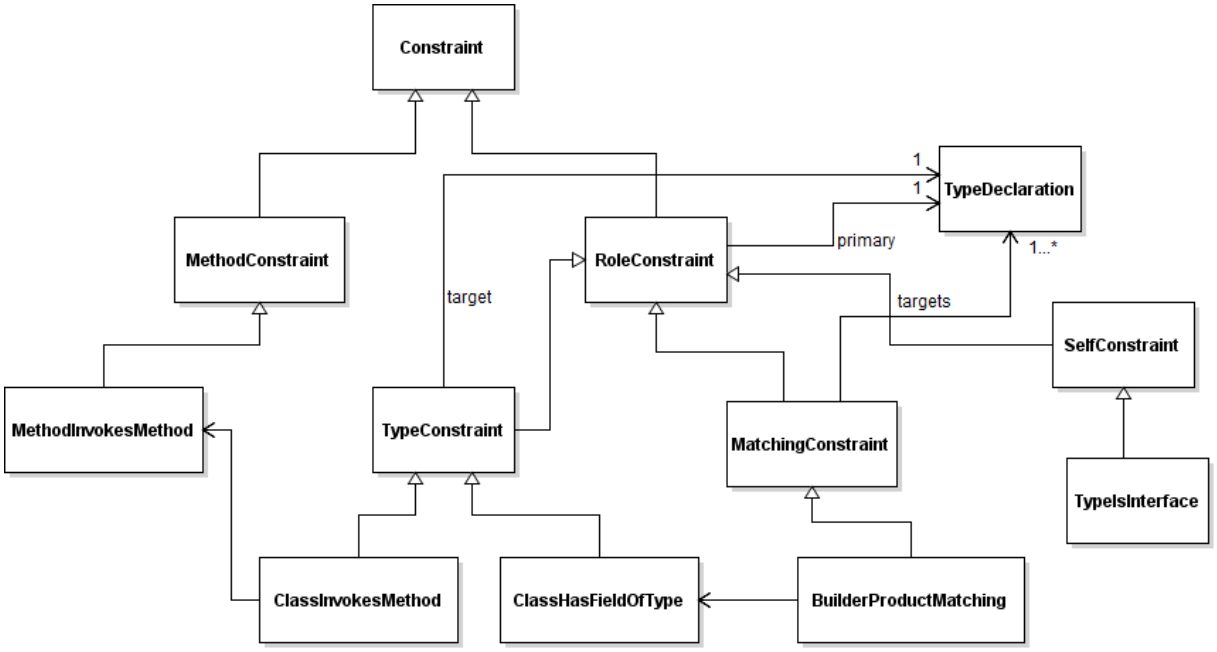

Figure 4.3: UML Class Diagram of the constraint package.

have the three different abstract forms of constraints, self, type, and matching. Each one has its own abstract class, containing the appropriate TypeDeclaration field for the target participants: none for self, one for type, and a collection for matching. In the diagram we can also see a small number of concrete constraints for each kind of RoleConstraint. Each individual concrete constraint is represented in its own class. We can see that the matching constraint BuilderProductMatching is associated with the ClassHasFieldOfType type constraint, indicating that it is a composite constraint as defined in 3. Finally, we have the NegativeConstraint class, which extends the RoleConstraint and also has a field of RoleConstraint, as defined in the model.

At this point we must address the existence of the additional hierarchy and classes present in the diagram, namely the classes Constraint and MethodConstraint. MethodConstraint is an abstract superclass for constraints regarding a single method, rather than a class or interface, and as such is never actually directly used in a design pattern. Rather, method constraints are only auxiliary classes to reduce the size of role constraint classes and offer an element of reusability. For example, the MethodInvokesMethod constraint shown in the diagram checks whether a method invokes another method of a given name, and is used both by the ClassInvokesMethod constraint and the special TemplateMethod constraint, reducing code duplication. The Constraint superclass encompasses both method and role constraints, essentially the

concept of a constraint in general, and has an abstract check() method. This method is responsible for determining whether a constraint is satisfied or not, returning a boolean value of true if the constraint holds, or false if it doesn't.

In order to decide whether a constraint is satified or not, the check() method examines the properties of child AST nodes of the TypeDeclaration nodes of the primary and target participants. For example, in the ClassHasFieldOfType constraint, all the FieldDeclaration nodes of the primary participant are checked; if one of them shares a type with the TypeDeclaration of the target participant, the constraint is satisfied.

As for the error messages, each constraint obviously has its own different message to indicate what went wrong if it is not satisfied. We do however have two different messages for each constraint: one for when it is used on its own, and one when it is used through a NegativeConstraint. The reason they are different is that we can't automatically derive the negative message from the positive without having some awkward phrasing that would impede readability and might cause confusion on what exactly is wrong with the pattern.

Finally, we must make note that some constraints have restrictions on how they were realized in our implementation. One of the main restrictions is that constraints dealing with inheritance (ClassExtendsSuperclass, TypeImplementsInterface) can only work with a single level of inheritance. For example, if we have three classes A, B, and C, A extends B an B extends C. Technically, the constraint ClassExtendsSuperclass(A,C) is satisfied; however, due to the nature of the AST structure, our implementation cannot properly process it since the TypeDeclaration node of A is not directly connected to the TypeDeclaration node of C, only of B. There is a way of bypassing this issue, which is to iteratively search for the TypeDeclaration node of the immediate superclass in the source code and check for the constraint there (e.g. find B, and do ClassExtendsSuperclass(B,C)) but it is not included in the scope of our work. Secondly, another important restriction is the concept of an object collection. While in an abstract specification a collection is well defined, in reality we cannot be sure what is or isn't a collection of objects. In Java, it can be an array, one of the builtin collections provided by the language (e.g. ArrayList, HashSet, et cetera), but it can also be a custom-made class. This is not something we can reliably detect through the AST, so we make the compromise that our constraints detect the following kinds of collections: either it is an array, or a parameterized type with the target participant

as a parameter.

4.2 Design Pattern Realization

Each design pattern is realized in its own class, inheriting from the abstract DesignPattern class. There is no need to make a distinction between creational, structural, and behavioral patterns, or any other categorization; all patterns are treated the same.

First of all, a concrete pattern object has a name and an id. The name is self-explanatory; for example, the Adapter pattern's pattern name is "Adapter". The id field indicates which pattern instance an actual object represents, since a Java project can have multiple distinct instances of the same pattern, and is unique among objects. Secondly, a pattern has a collection - in our implementation an ArrayList - containing the names of all the pattern's roles. Finally, we have the two most important fields: the collection of participating types, and the collection of constraints. The first collection is the "roles" collection defined via OCL in 3, in the Design Patterns section. In the model, it was defined as a set of tuples; each tuple representing a role, and containing the role's name and a set of all types with that role. In our implementation, it is realized by a Java HashMap, with the role names as keys, and sets of TypeDeclarations as values.

The DesignPattern class also has certain methods that are common amongst all patterns. In particular, the checkRoles() method realizes the RolesExist invariant defined in 3: A pattern instance must have at least one type for each of its roles. The verifyPattern() method realizes the AllConstraintsSatisfied invariant, which asserts that for a pattern to be verified as correct, all of its constraints must be satisfied (i.e. their check() method must return true). Finally, populateConstraints() is an abstract method in which the constraints are created for each different pattern.

4.2.1 Specification of Design Patterns in Practice

Essentially, from the perspective of a pattern expert, it is quite easy to create additional pattern specifications in the framework. Due to the high degree of reusability of constraints among patterns, the general extensibility of the model elements, and our utilization of object-oriented concepts, all that is required in the creation of a new class that subclasses DesignPattern.

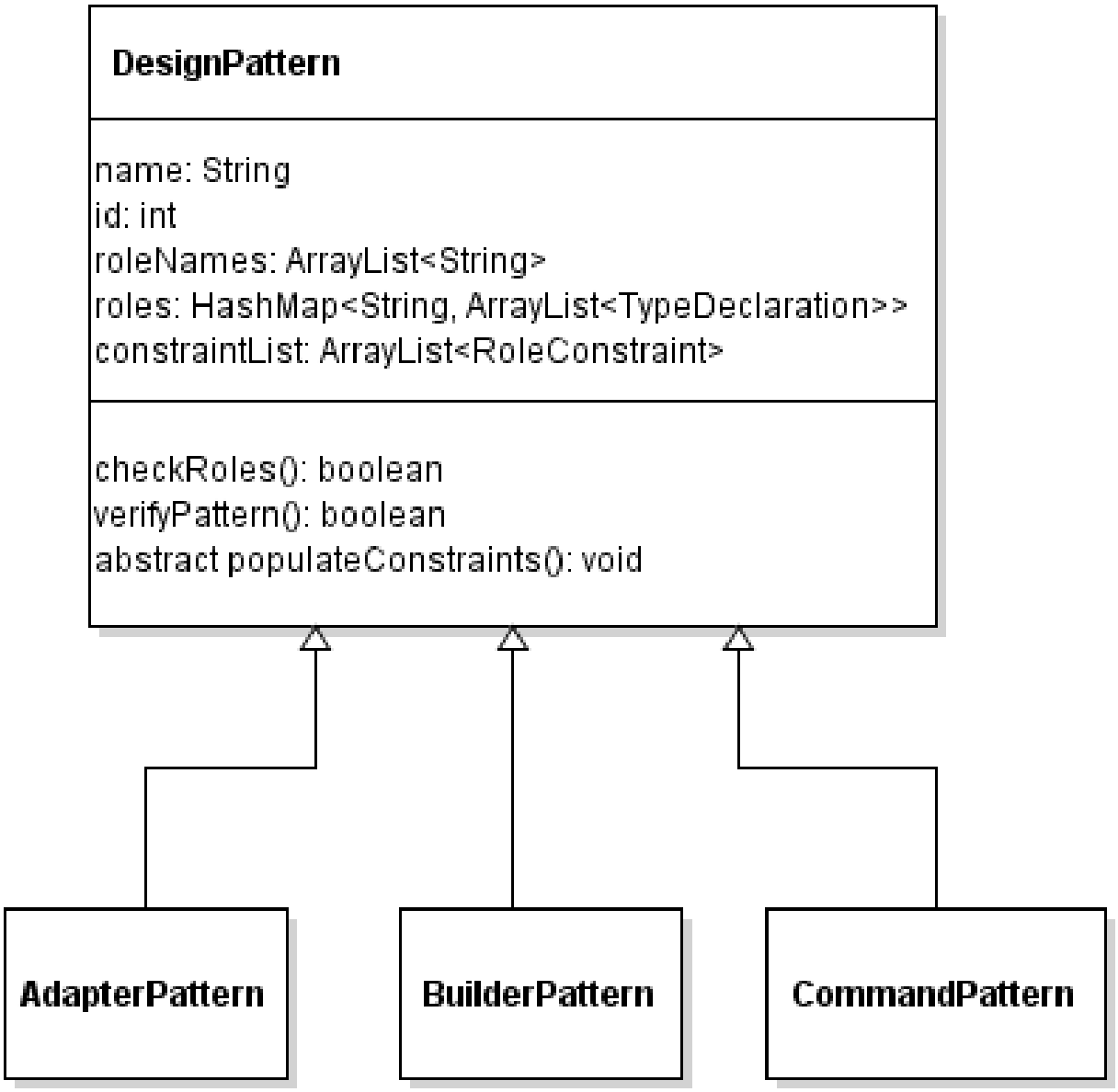

Figure 4.4: UML Class Diagram of the designPattern package.

```
public AdapterPattern()
{
    patternName = "Adapter";

    roleNames.add("Adapter");
    roleNames.add("Adaptee");
    roleNames.add("Adapter Client");
    roleNames.add("Adapter Interface");

}

@Override
public void populateConstraints() {
    TypeDeclaration adapter = roles.get("Adapter").get(0);
    TypeDeclaration adaptee = roles.get("Adaptee").get(0);
    TypeDeclaration adapterClient = roles.get("Adapter Client").get(0);
    TypeDeclaration adapterInterface = roles.get("Adapter Interface").get(0);

    constraintList.add(new TypeIsInterface(adapterInterface));

    constraintList.add(new ClassHasFieldOfType(adapterClient,adapterInterface));
    constraintList.add(new NegativeConstraint(new ClassHasFieldOfType(adapterClient, adaptee)));
    constraintList.add(new NegativeConstraint(new ClassHasVariableOfType(adapterClient, adaptee)));

    constraintList.add(new ClassImplementsInterface(adapter, adapterInterface));
    constraintList.add(new ClassHasFieldOfType(adapter,adaptee));
    constraintList.add(new ClassInvokesMethodOfClass(adapter,adaptee));

}
```

Figure 4.5: AdapterPattern source code

In this class, the names of the roles participating in the pattern are added to the roleNames list, and the appropriate RoleConstraint objects are instantiated in the populateConstraints() method, following the OCL invariants mentioned. In Figure 4.5 we can see an example, showing the implementation of the Adapter pattern. Every other mechanism such as verification of the pattern, checking for the existence of all roles, et cetera, are already present in the abstract DesignPattern class, and do not need to be re-created.

4.3 Detecting Design Pattern Instances

In Chapter 3 we defined the abstract model of design pattern specifications, and in Chapter 5 we also give the exact specifications for each pattern. However, in order to proceed to the verification stage, we must first have a way to extract them from the source code. So, given a large Java project, we must have a way of understanding which classes and interfaces participate in a pattern, and what roles they have.

A vital assumption we make here is that the developers do in fact know these particulars. This is something that other works such as HEDGEHOG also accept,

and is in fact inescapable; assuming otherwise shifts the problem in the domain of automatic pattern detection, which while adjacent to this, is not within the scope of our work.

In jStar, the entire Java project examined is transformed into a Jimple form, and the types that are involved in the verification are included in the specification rules. In HEDGEHOG, only the specific files that contain the source code of the types involved in a pattern are loaded in the tool. We adopt a different approach, by using a custom Java annotation to mark the participating classes directly in the source code. The project is then parsed, and the Abstract Syntax Tree is produced. If the custom annotations are detected in the created AST, JPatternJudge is aware that that particular type is involved in a pattern.

4.3.1 Design Pattern Annotation

An annotation is a form of metadata that can be added to a piece of source code. Java provides certain builtin annotations, such as @Override and @Deprecated, but also gives the ability to create custom annotations, which can have their own fields. Annotations can be applied in various code elements, such as classes, fields, methods, etc; however, our annotation is only applied on classes and interfaces, since only they can have a role and participate in a pattern.

Our custom annotation has two parts, which can be seen in Figures 4.6 and 4.7. The @DesignPattern annotation is the one that is used to annotate the source code, and has three fields. The pattern field is used to specify what kind of pattern the type participates in; e.g. Adapter, Command, etc. The patternID field specifies which instance of that pattern the type belongs to. Finally, the role field specifies what role the type has in the pattern. The @Target meta-annotation specifies that @DesignPattern can only be applied on types, while the @Repeatable meta-annotation specifies that multiple instances of @DesignPattern can be applied on the same type - in case it has a role in multiple different patterns. This also requires the existance of the @Patterns annotation, which essentially acts as a storage for @DesignPatterns annotations applied on the same type.

Figure 4.6: DesignPattern annotation

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Repeatable;
import java.lang.annotation.Target;

@Target(ElementType.TYPE)
@Repeatable(value = Patterns.class)
public @interface DesignPattern {
public String pattern() default "";
public int patternID() default 0;
public String role() default "";
}
```

4.3.2 Specifying Design Pattern Instances in Practice

It is trivially easy for software developers to utilize our custom annotation to specify the existence of a design pattern instance in their code. All that is required is to annotate each class or interface with the @DesignPattern annotation, appropriately filling the fields. For example, to annotate the class CObClass that plays the role of a concrete observer in the first instance of an Observer pattern, the source code would look like this:

```
@DesignPattern(pattern="Observer", patternID=1, role="Concrete Observer")
public class CObClass {......}
```

To annotate the interface FileCommand that plays the role of Command in the homonymous pattern, the annotation would be the following:

```
@DesignPattern(pattern="Command", patternID=1, role="Command")
public interface FileCommand {......}
```

If, in the same project, there was n additional instance of the Command pattern referring to e.g. buttons instead of files, the annotation would be the following:

```
@DesignPattern(pattern="Command", patternID=2, role="Command")
public interface ButtonCommand {......}
```

Something to note here is that these two annotations must be included in the Java source code that is being verified - NOT in the framework itself.

Figure 4.7: Patterns annotation required for @Repeatable meta-annotation

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

@Target(ElementType.TYPE)
public @interface Patterns {
DesignPattern[] value();
}
```

4.3.3 Pattern Detection Mechanism

The detection mechanism relies on the @DesignPattern annotation described above, and is implemented in the PatternDetector class. There are two main steps involved: First, a Parser processes the source code to create the abstract syntax tree for each type in the program. Then, a Pattern Detector examines these ASTs; if the presence of the @DesignPattern annotation is detected, the information in it is processed an the annotated type is added in the appropriate DesignPattern object. We can see a pseudocode algorithm of this process in Figure 4.8:

The DesignPatternRegistry has a collection of DesignPattern objects. In fact, it has a single object for each different pattern specified in the framework. It is used to check whether a pattern name found by the detector is valid. On the other hand, the DesignPatternStorage is a collection of the actual pattern instances detected in the source code.

Specifically, the algorithm has the following steps for each type in the project being examined:

- Parse the type and create its AST. (lines 2-3)
- Examine the AST and extract the annotations applied on the type. (line 4)
- For each annotation applied, examine if it is a @DesignPattern annotation and if the information in it is valid. (lines 5-6)
- To check if the annotation is correct, check if the name of the pattern given exists in the DesignPatternRegistry, and the name of the role exists in the appropriate

Figure 4.8: Design Pattern detection and instantiation algorithm pseudocode

```
1    for (Type type: Project) {
2        ast = Parser.getAST(type);
3        typeDeclaration = ast.getTypeDeclaration();
4        annotations = typeDeclaration.getAnnotations();
5        for (Annotation annotation: annotations){
6            if (annotation.isDesignPattern() and checkInformationValidity(annotation)) {
7                pattern = annotation.getPattern();
8                id = annotation.getID();
9                role = annotation.getRole();
10               if (!DesignPatternStorage.get(pattern).contains(id)){
11                   DesignPatternStorage.addPattern(pattern, id);
12               }
13               DesignPatternStorage.get(pattern).get(id).addRole(role, typeDeclaration);
14           }
15       }
16   }
17   for (DesignPattern p: PatternStorage)
18   {
19       p.populateConstraints();
20   }
```

Pattern. (line 6)

- Extract the (confirmed valid) information from the annotation. (lines 7-9)
- If a pattern with the given ID does not exist in the DesignPatternStorage, create a new instance. (lines 10-12)
- Add the type and its role to the pattern instance. (line 13)

When all the design patterns have been detected and fully populated with their roles and types, we go over them one more time and create the necessary constraints as per their specification, taking care to follow Design Pattern model defined in 3. (lines 17-20)

4.4 Design Pattern Verification

The verification process is extremely simple in our framework. For each pattern instance stored in the DesignPatternStorage collection, the PatternVerifier checks their validity in two steps: first, it checks if the pattern instance has a type for each role, and if so, it checks if all the constraints are satisfied.

```
for (DesignPattern pattern: DesignPatternStorage){
    if(pattern.checkRoles()){
        pattern.verifyPattern();
    }
    print(pattern.getErrorMessage())
}
```

A pattern's error message depends on the type of error. If the pattern is missing role types, as indicated by checkRoles(), then the message outlines which roles are missing. Otherwise, the message is a collation of the error messages of the individual constraints. If all the constraints are satisfied, the message simply indicates that the pattern is verified as correct.

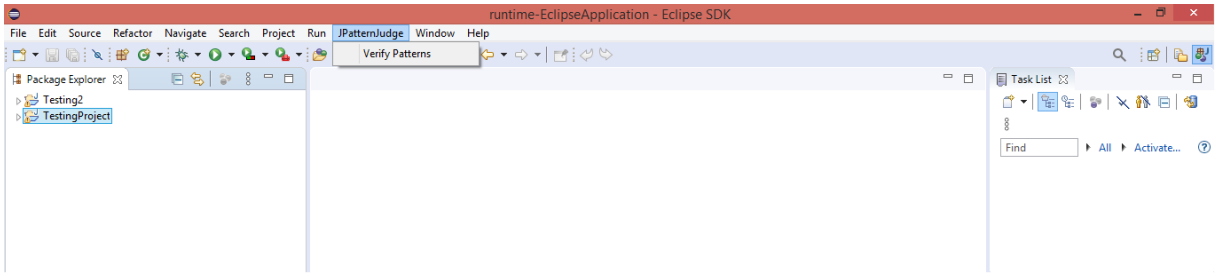

Figure 4.9: The JPatternJudge menu object in Eclipse's menu bar.

4.5 The JPatternJudge Plugin

Our implementation of the framework takes the form of a plugin for the Eclipse Integrated Development Environment. Eclipse is a very popular IDE used by developers in the production of industrial OOP projects, and as such is ideal for integrating our framework directly in the development process.

Using the plugin is very straightforward. First, the user must add the @DesignPattern annotation definitions in the project being examined, and annotate all the classes and interfaces that participate in patterns appropriately. The plugin adds a new menu to Eclipse's menu bar, titled "JPatternJudge". This menu contains a single menu item, which detects and verifies the patterns present and annotated in the selected Eclipse project in the Project Explorer view. The menu can be seen in 4.9.

When JPatternJudge finishes the verification process, it generates a popup window containing a full report. This report includes the error messages for every pattern detected, as they were described in the previous section. This popup window, along with an example report showing 3 kinds of messages (pattern is correct, pattern is missing roles, and pattern has constraints that are not satisfied) is shown in 4.10. The report is also stored in a text file titled "DesignPatternReport.txt", which is saved in the project's workspace folder, for the developer to peruse at their leisure.

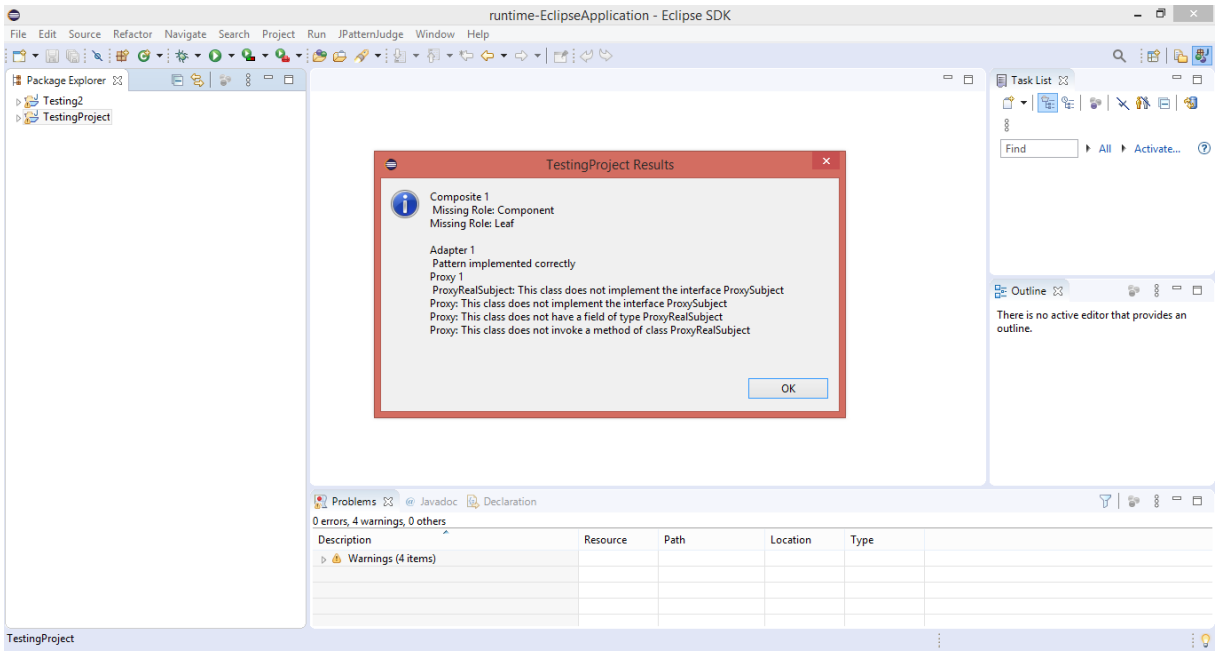

Figure 4.10: Example of a verification report generated by JPatternJudge.

Chapter 5

Design Pattern Specifications

In this chapter we will list the design patterns included in JPatternJudge, as well as their specifications. The specification of each pattern takes form as a list of roles and a list of constraints, obeying the model presented in the previous chapter. We assume the reader has at least a basic familiarity with the GoF patterns, but we do include a short description of each pattern's purpose and what each role represents in the context, as well as a UML diagram of the pattern's structure.

5.1 Notation

In Chapters 3 and 4 we showed that pattern specification writers can develop simple classes for the specifications extending the DesignPattern subclass and overriding the populateConstraints() method, hile still following certain OCL constraints. For brevity, we do not include the entire source code of the class for each pattern, but utilize the following notation when it comes to constraints:

A SelfConstraint element with a primary participant that has a given role is represented as:

SelfConstraint(PrimaryParticipantRole)

Similarly, a TypeConstraint is represented as:

TypeConstraint(PrimaryParticipantRole, TargetParticipantRole)

And a MatchingConstraint with a set of target participants is represented as:

MatchingConstraint(PrimaryParticipantRole, [TargetPrticipantRole])

When a pattern contains a NegativeConstraint element, they are represented as follows, depending on what kind of constraint they contain:

NegativeConstraint(SelfConstraint(PrimaryParticipantRole))

NegativeConstraint(TypeConstraint(PrimaryParticipantRole, TargetParticipantRole))

NegativeConstraint(MatchingConstraint(PrimaryParticipantRole, [TargetParticipantRole]))

5.2 Structural Patterns

5.2.1 Adapter

The adapter pattern is a very common pattern that is used to connect two classes that are normally incompatible, by creating a linking interface. The client is a class that contains some already existing business logic we wish to expand upon, but the adaptee (also known as service, and usually part of third-party or legacy code), which contains useful behavior the client wants, has an incompatible interface. The solution is to create an adapter interface that describes the protocol the client wants to use, and then an Adapter. The adapter implements the adapter interface, and also has a field of adaptee. It realizes the behavior that connects the two incompatible classes, the client and the adaptee. The client can now have a field of adapter, and use it instead of the adaptee directly.

Roles

- Client
- AdapterInterface
- Adapter

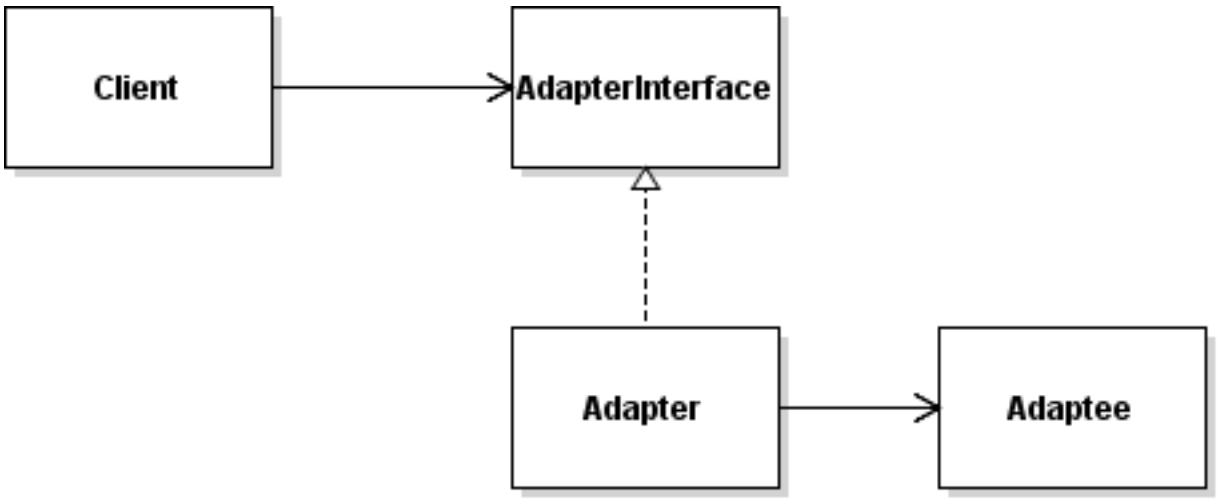

Figure 5.1: UML Diagram of the Adapter pattern structure.

- Adaptee

Constraints

- TypeIsInterface(AdapterInterface)
- TypeImplementsInterface(Adapter, AdapterInterface)
- ClassHasFieldOfType(Adapter, Adaptee)
- ClassInvokesMethod(Adapter, Adaptee)
- ClassHasFieldOfType(AdapterClient, Adapter)
- NegativeConstraint(ClassHasFieldOrVariableOfType(AdapterClient, Adaptee)

5.2.2 Bridge

Bridge is a pattern that is used to split a set of related classes in two different hierarchies, the abstraction and the implementor, allowing each of them to be developed independent of the other. The abstraction provides high-level control, while the concrete implementors take care of the low-level work for each different context they operate in, all through the implementor interface.

Roles

- Abstraction

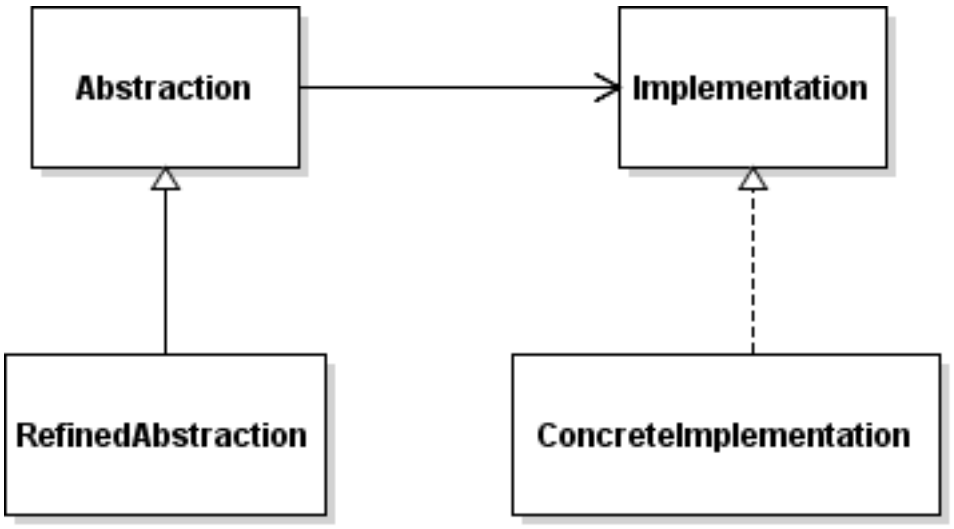

Figure 5.2: UML Diagram of the Bridge pattern structure.

- RefinedAbstraction
- Implementor
- ConcreteImplementor

Constraints

- ClassIsAbstract(Abstraction)
- ClassHasFieldOfType(Abstraction, Implementor)
- ClassExtendsSuperclass(RefinedAbstraction, Abstraction)
- TypeIsInterface(Implementor)
- TypeImplementsInterface(ConcreteImplementor, Implementor)

5.2.3 Composite

The Composite pattern is used to represent sets of objects that can form a tree structure. There are two main variants of Composite, focusing either on type safety or uniformity. Our specification is based on the latter. The component is the abstract tree node; the leaf represents a node that does not contain any components, while the composite represents a tree node that does - whether these are leaves or recursively composites.

Roles

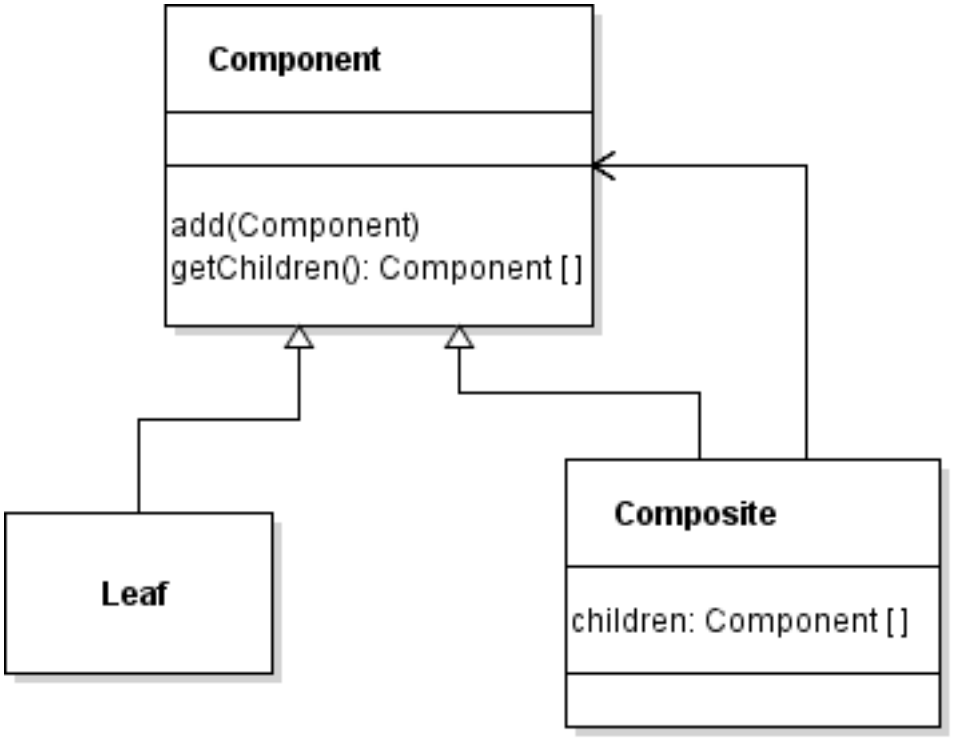

Figure 5.3: UML Diagram of the Composite pattern structure.

- Component
- Composite
- Leaf

Constraints

- ClassIsAbstract(Component)
- ClassHasMethodWithParameter(Component, Component)
- ClassHasMethodWithReturnCollectionOfType(Component, Component)
- ClassExtendsSuperclass(Leaf, Component)
- ClassExtendsSuperclass(Composite, Component)
- ClassHasFieldCollectionOfType(Composite, Component)

5.2.4 Decorator

The Decorator pattern, also known as Wrapper, is used to attach extra behavior to an object. An interface (component) is used for both the wrapper and wrapped objects. The concrete components are the objects being wrapped, containing the basic behavior. The decorator implements the component interface to ensure compatibility, while defining additional behavior. A variant of this pattern has an abstract decorator, and concrete decorators to further extend the wrapped object's functionality by providing alternative behavior.

Roles

- Component
- ConcreteComponent
- Decorator

Constraints

- TypeIsInterfaceOrAbstractClass(Component)
- TypeInheritsFrom(ConcreteComponent, Component)
- TypeInheritsFrom(Decorator, Component)
- ClassHasFieldOfType(Decorator, Component)

5.2.5 Flyweight

Flyweight is used to reduce the memory usage of a program by extracting the common parts of multiple objects and keeping them in a single, shared one. The flyweight class contains the shared state of the objects, which are represented by the context. A flyweight factory manages the production of flyweights when a new one is needed.

Roles

- Flyweight
- Context
- FlyweightFactory

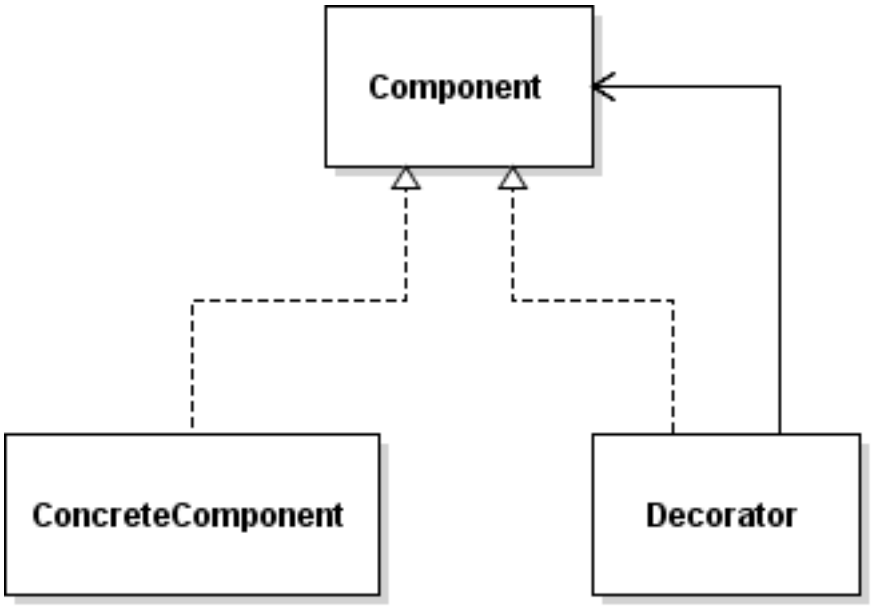

Figure 5.4: UML Diagram of the Decorator pattern structure.

Constraints

- ClassHasFieldOfType(Context, Flyweight)
- ClassInvokesMethod(Flyweight, FlyweightFactory)
- ClassHasFieldCollectionOfType(FlyweightFactory, Flyweight)
- ClassInstantiatesObject(FlyweightFactory, Flyweight)
- ClassHasMethodWithReturnType(FlyweightFactory, Flyweight)

5.2.6 Proxy

Proxy is used to create a substitute of an object, acting as an intermediate to other objects and giving indirect access. To ensure compatibility, a shared interface between the two objects is created. The proxy has a field of the real subject, and invokes its methods, while layering additional functionality.

Roles

- Subject
- Proxy
- RealSubject

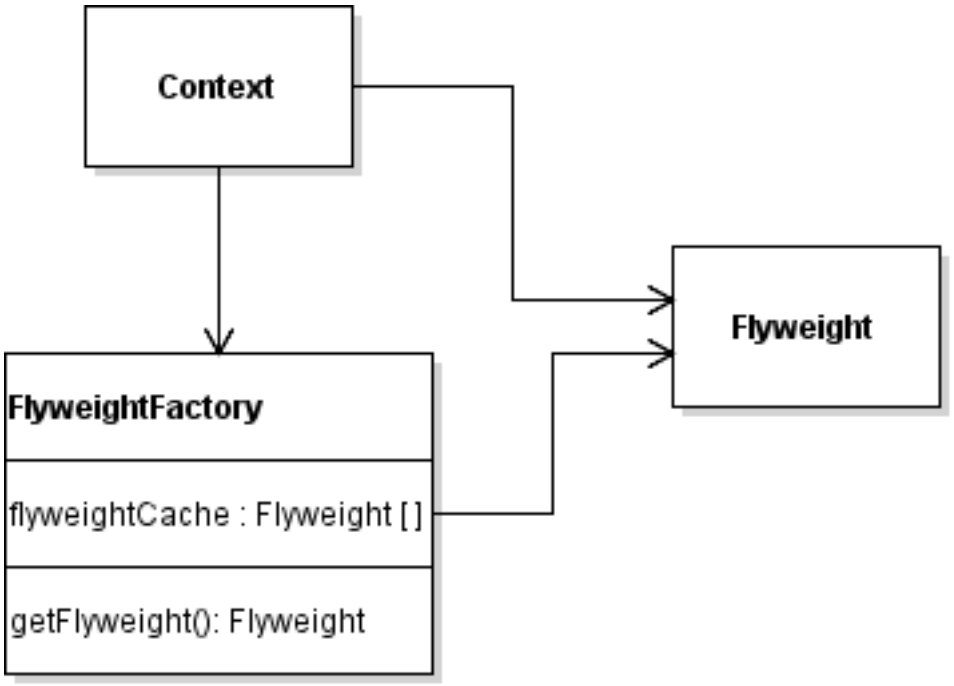

Figure 5.5: UML Diagram of the Flyweight pattern structure.

Constraints

- TypeIsInterface(Subject)
- TypeImplementsInterface(RealSubject, Subject)
- TypeImplementsInterface(Proxy, Subject)
- ClassHasFieldOfType(Proxy, RealSubject)
- ClassInvokesMethod(Proxy, RealSubject)

5.3 Behavioral Patterns

5.3.1 Chain Of Responsibility

Chain of responsibility is a pattern used to create a series of processing objects. Each object is essentially a handler for a request with different behavior. When the request reaches a handler, the handler can either process it if able or pass it along to the next in the chain until it reaches the one that can handle it.

Roles

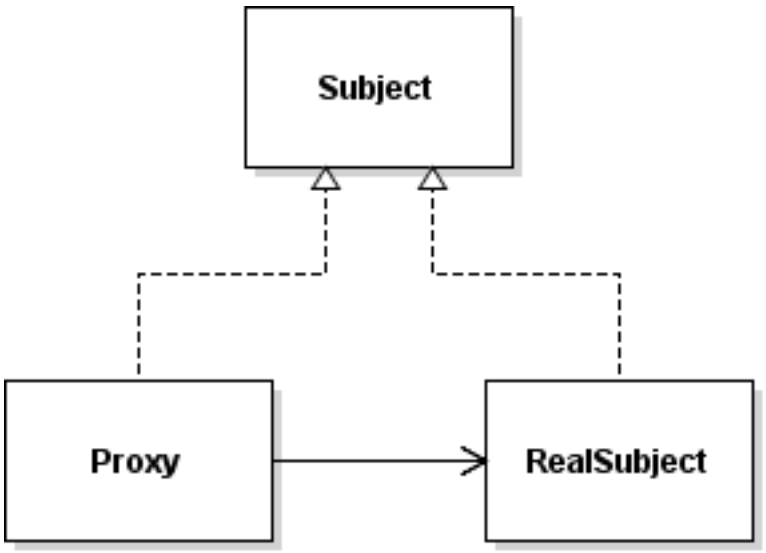

Figure 5.6: UML Diagram of the Proxy pattern structure.

- Handler
- ConcreteHandler

Constraints

- TypeIsInterface(Handler)
- ClassHasMethodWithParameter(Handler, Handler)
- TypeImplementsInterface(ConcreteHandler, Handler)
- ClassHasFieldOfType(ConcreteHandler, Handler)

5.3.2 Command

Command is a design pattern that is used to transform a request into a concrete object,allowing more capabilities when it comes to handling it such as parameterization of methods concerning requests. A concrete command object has a receiver and invokes its method, while an invoker object is the one who executes the commands without knowing a concrete command specifically.

Roles

- Command
- ConcreteCommand

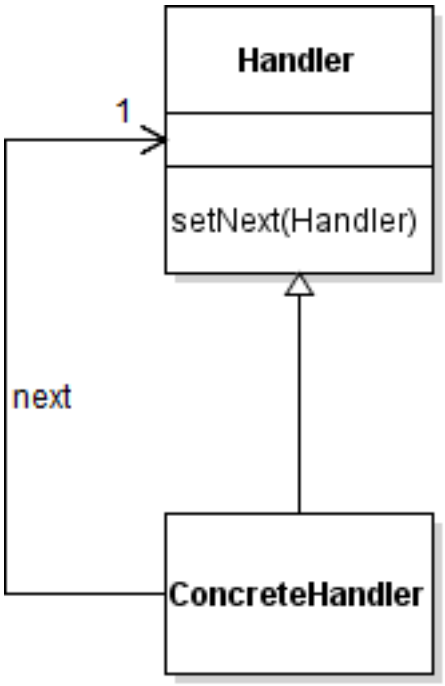

Figure 5.7: UML Diagram of the Chain of Responsibility pattern structure.

- Invoker
- Receiver

Constraints

- TypeIsInterface(Command)
- TypeImplementsInterface(ConcreteCommand, Command)
- ClassHasFieldOfType(ConcreteCommand, Receiver)
- ClassHasFieldOrVariableOfType(Invoker, Command)
- NegativeConstraint(ClassHasFieldOfType(Invoker, ConcreteCommand))

5.3.3 Mediator

Mediator is a design pattern used to reduce coupling between objects by assigning a single mediator object to be responsible for communication between the rest. The communicating objects, called colleagues, do so through the mediator object. We can also have multiple mediators supporting different behavior.

Roles

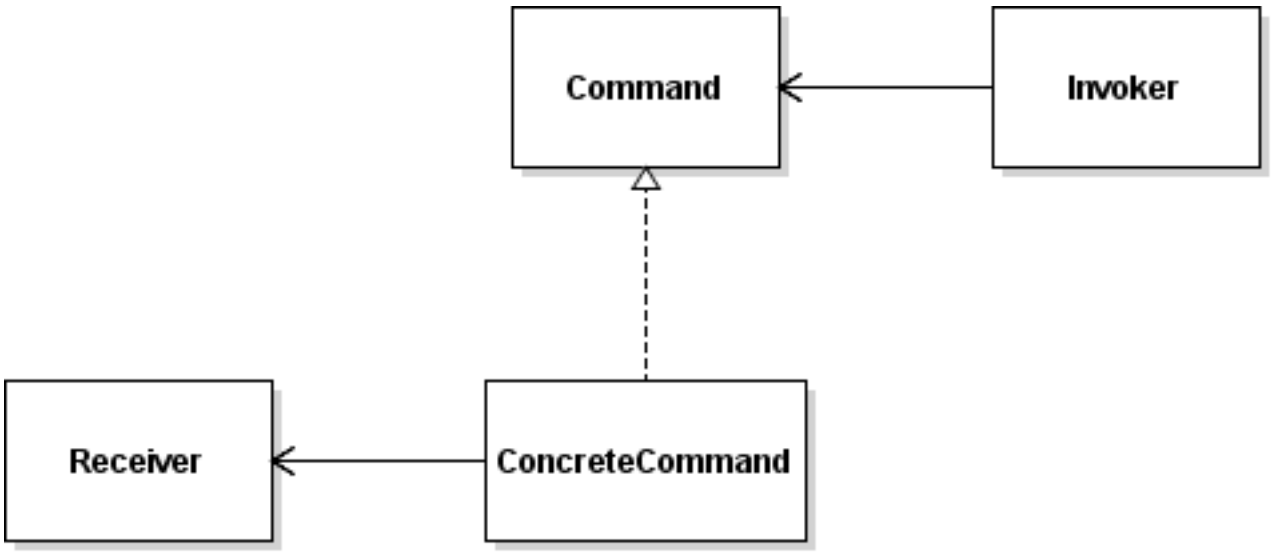

Figure 5.8: UML Diagram of the Command pattern structure.

- Mediator
- ConcreteMediator
- Colleague
- ConcreteColleague

Constraints

- TypeIsInterface(Mediator)
- TypeImplementsInterface(ConcreteMediator, Mediator)
- ClassHasFieldOfType(ConcreteMediator, ConcreteColleague)
- ClassIsAbstract(Colleague)
- ClassHasFieldOfType(Colleague, Mediator)
- ClassExtendsSuperclass(ConcreteColleague, Colleague)

5.3.4 Memento

Memento is a design pattern used to store the state of an object in another object, allowing to save and restore the previous state at any point. It is essentially a data object, but with a specific purpose and behavior. The originator is an object that has

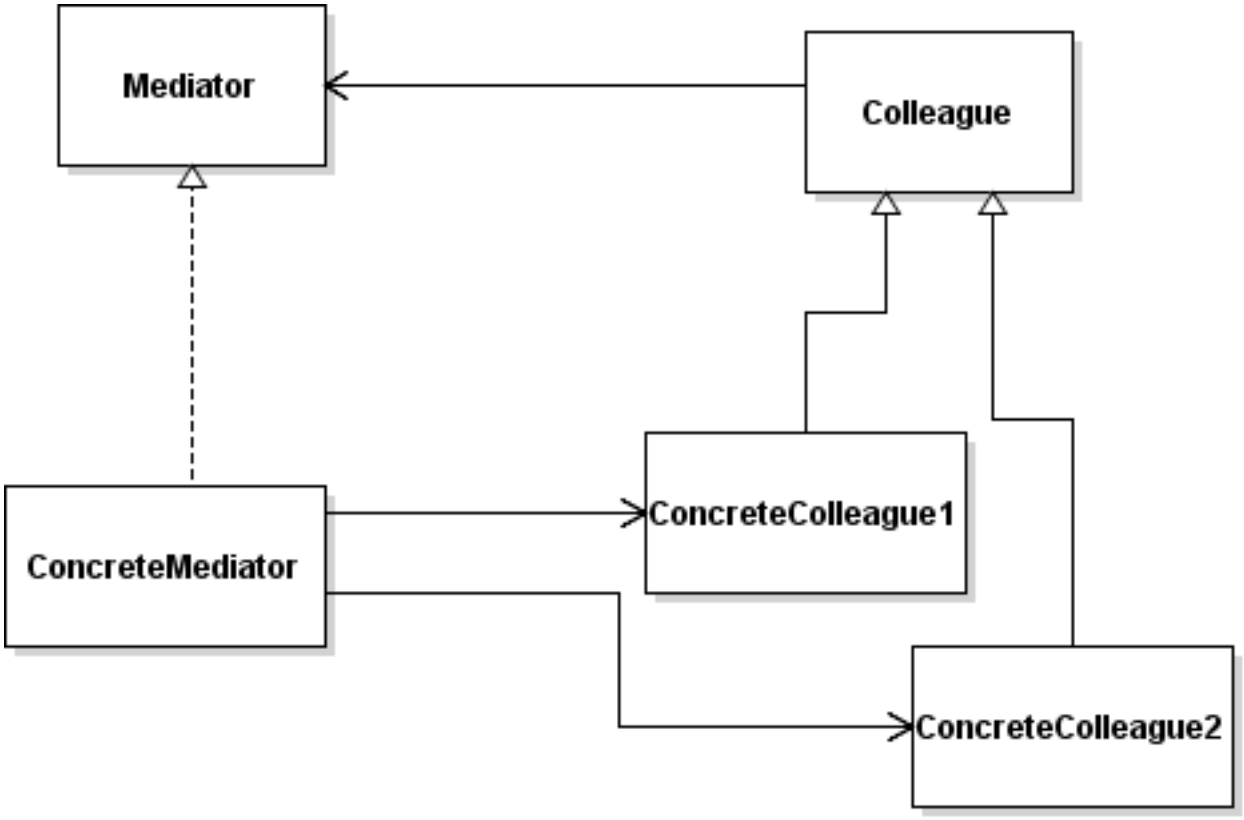

Figure 5.9: UML Diagram of the Mediator pattern structure.

a specific state, however we want to be able to switch between previous states at will; for example, if we want to undo a change or load a previous version. The memento is the object containing that state, created by the originator when it is ”saved”, and the caretaker is an object that manages a list of mementos, loading them on the originator.

Roles

- Originator
- Memento
- Caretaker

Constraints

- ClassHasFieldCollectionOfType(Caretaker, Memento)
- ClassHasFieldOfType(Caretaker, Originator)
- ClassHasMethodWithParameter(Originator, Memento)
- ClassHasMethodWithReturnType(Originator, Memento)

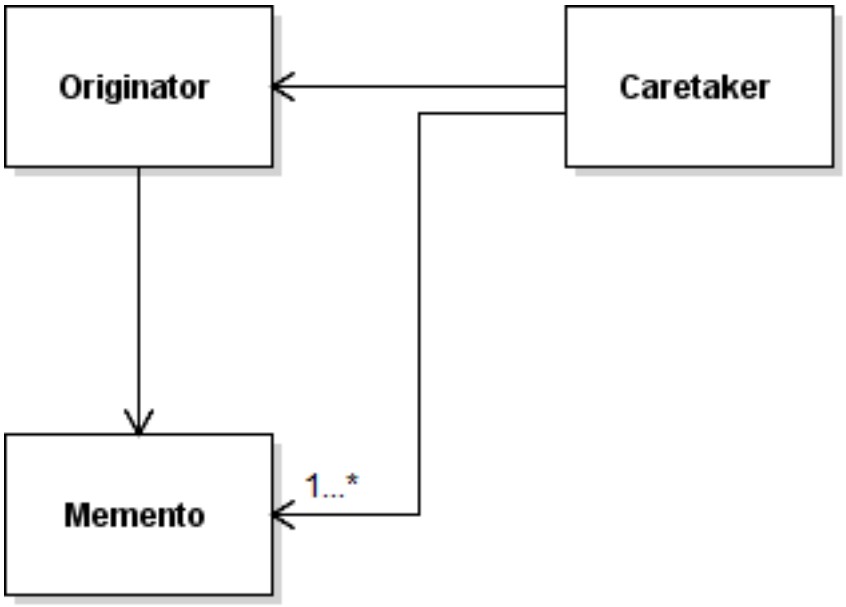

Figure 5.10: UML Diagram of the Memento pattern structure.

- ClassInstantiatesObjectOfType(Originator, Memento)

5.3.5 Observer

Also known as Subscriber or Listener, the Observer pattern is used when a single object want to notify multiple others when a change occurs in it. The object that changes and sends out the notifications is the Subject, while the objects interested in the subject's state changes are concrete observers implementing an observer interface. A variant of observer contains multiple concrete subjects instead of just one, implementing their own interface.

Roles

- Subject
- Observer
- ConcreteObserver

Constraints

- ClassHasFieldCollectionOfType(Subject, Observer)
- ClassHasMethodWithParameter(Subject, Observer)

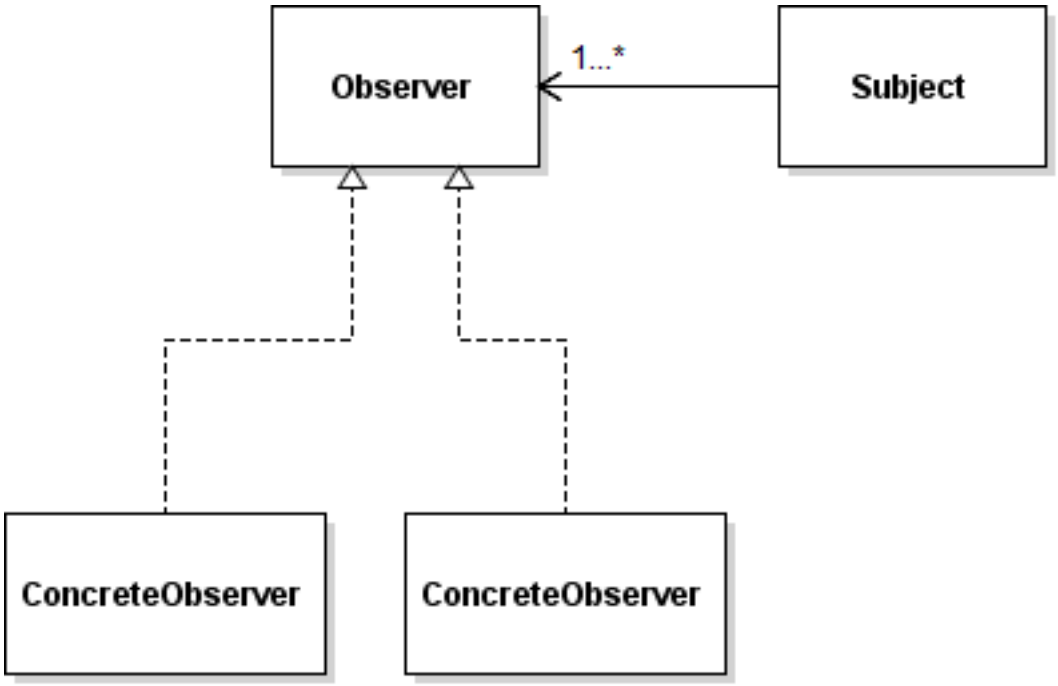

Figure 5.11: UML Diagram of the Observer pattern structure.

- ClassInvokesMethod(Subject, Observer)
- TypeIsInterface(Observer)
- TypeImplementsInterface(ConcreteObserver, Observer)

5.3.6 State

State allows an object to alter it behavior when its internal state is changed, allowing switching from one type of behavior to another at runtime. We have an object, our context, that has a finite number of distinct states, each with different behavior. We represent each of those states in a different concrete state class, allowing the context to switch from one to another in a controlled manner, similar to a finite state machine. This sounds similar to Memento, but there the states all have the same behavior and simply change attributes; in State, each one has completely different behavior.

Roles

- Context
- State
- ConcreteState

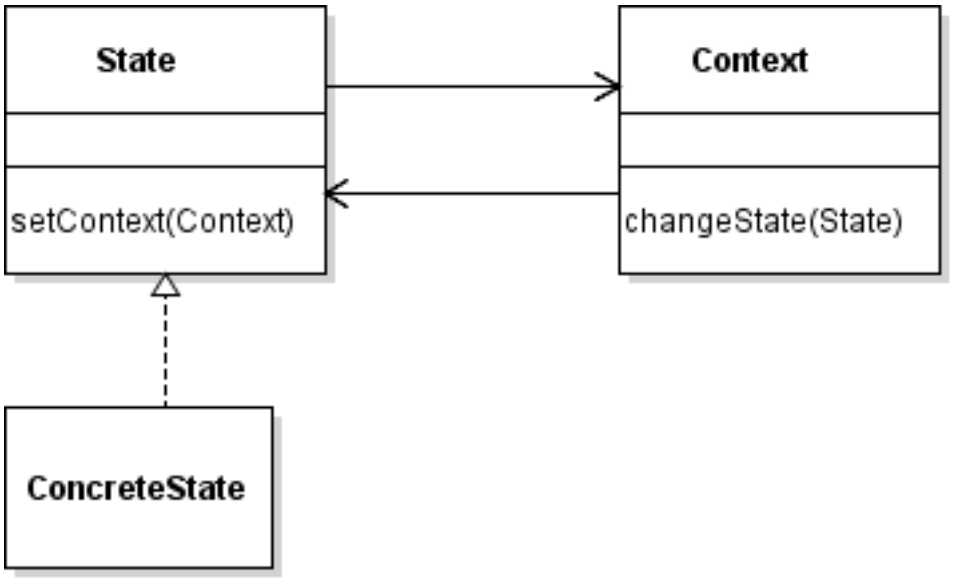

Figure 5.12: UML Diagram of the State pattern structure.

Constraints

- ClassHasFieldOfType(Context, State)
- ClassInvokesMethod(Context, State)
- TypeIsInterface(State)
- ClassHasMethodWithParameter(State, Context)
- TypeImplementsInterface(ConcreteState, State)

5.3.7 Strategy

Strategy is a design pattern that can be used to allow selection of a specific algorithm, out of a family of them, at runtime. A context object has a job to do, which can be done in multiple different ways. Each of those is realized in its own concrete strategy class, and the context uses one of them. This sounds very similar to the State pattern, however a key difference is that in this case the context does not select which strategy it will use; instead, it is provided with one by some other external class.

Roles

- Context
- Strategy

- ConcreteStrategy

Constraints

- ClassHasFieldOrVariableOfType(Context, Strategy)
- ClassInvokesMethod(Context, Strategy)
- TypeIsInterface(Strategy)
- TypeImplementsInterface(ConcreteStrategy, Strategy)

5.3.8 Template Method

Template Method is a design pattern that defines the skeleton of an algorithm in a superclass via abstract methods, allowing the subclasses to implement their own specific steps without changing the algorithm's structure. A template class has an algorithm that has s specific structure, but some of the steps can have variations. Those steps are declared as abstract methods, which are defined in subclasses of the template.

Roles

- Template
- Concrete

Constraints

- ClassIsAbstract(Template)
- ClassExtendsSuperclass(Concrete, Template)
- TemplateMethodConstraint(Template)

The TemplateMethodConstraint is used only in this pattern, and is defined as follows:

TemplateMethodConstraint: The primary participant has at least an abstract method, and a concrete method that invokes the abstract one.

```
context TemplateMethodConstraint inv:
self.primaryParticipant.getMethods() -> includes(m | m.getModifiers() -> excludes(mod
| mod.isAbstract() = true) and m.getExpressions() ->
```

```
includes(e | e.oclIsTypeOf(MethodInvocation) and self.primaryParticipant.getMethods()
-> includes(m2 | m2.getModifiers() -> includes(mod2 | mod2.isAbstract() = true) and
e.methodName = m2.name)))
```

Visitor Visitor is a pattern that is used to separate an algorithm from the objects it operates on, by moving the relevant methods in a single object. The visitor object contains the algorithm itself, and can ”visit” any of the objects being operated on, called elements. Each element also has a method for accepting a visitor, creating a two-way relation between elements and visitors.

Roles

- Visitor
- ConcreteVisitor
- Element
- ConcreteElement

Constraints

- TypeIsInterface(Visitor)
- ClassHasMethodWithParameter(Visitor, ConcreteElement)
- TypeImplementsInterface(ConcreteVisitor, Visitor)
- TypeIsInterface(Element)
- ClassHasMethodWithParameter(Element, Visitor)
- TypeImplementsInterface(ConcreteElement, Element)
- ClassInvokesMethod(ConcreteElement, Visitor)

5.4 Creational Patterns

5.4.1 Abstract Factory

The Abstract Factory pattern is used when we have multiple factories that produce families of objects. We have a number of product interfaces; for each of those, we

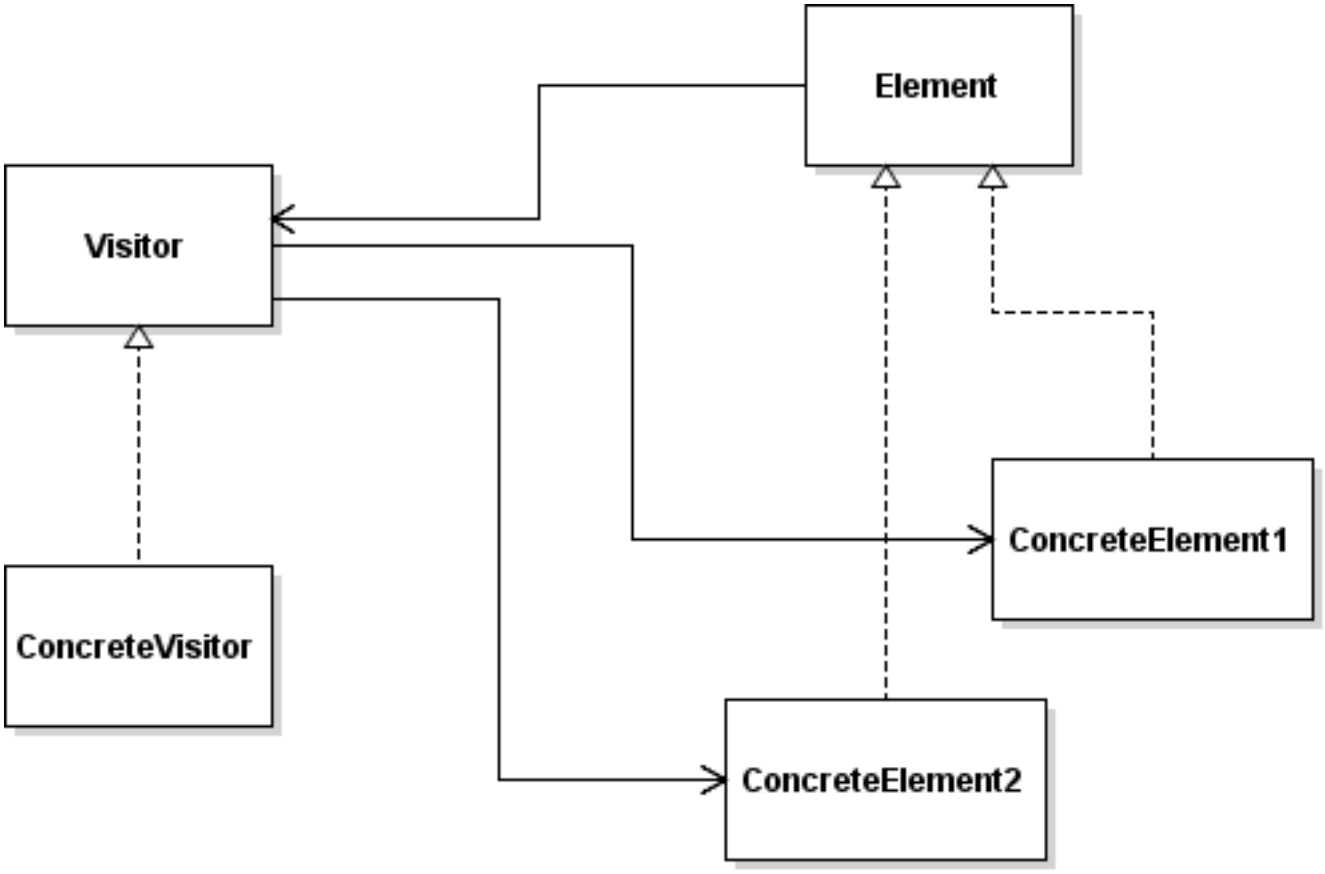

Figure 5.13: UML Diagram of the Visitor pattern structure.

have a number of concrete products. Each concrete factory can produce one concrete product of each product family.

Roles

- AbstractFactory
- ConcreteFactory
- Product
- ConcreteProduct

Constraints

- TypeIsInterface(AbstractFactory)
- ClassHasMethodWithReturnType(AbstractFactory, Product)
- TypeImplementsInterface(ConcreteFactory, AbstractFactory)
- TypeIsInterface(Product)
- ConcreteProductInterfaceMatching(ConcreteProduct, [Product])
- FactoryProductMatching(ConcreteFactory, [ConcreteProduct])

The two matching constraints are defined below:

ConcreteProductInterfaceMatching: Each ConcreteProduct implements one of the Product interfaces, not all of them. In general form, this means that the primary participant implements at least one of the target participant interfaces.

```
context ConcreteProductInterfaceMatching inv:
self.constraints -> forAll(c | c.oclIsTypeOf(TypeImplementsInterface) and
c.primaryParticipant = self.primaryParticipant and
self.targetParticipants -> includes(t | t = c.targetParticipant))
and self.constraints -> includes(c2 | c2.satisfied = true)
```

FactoryProductMatching: Each concrete factory instantiates a concrete product for each different product interface.

```
context FactoryProductMatching inv:
self.constraints -> forAll(c | c.oclIsTypeOf(ClassInstantiatesObjectOfType) and
c.primaryParticipant = self.primaryParticipant and self.targetParticipants -> forAll(t1,t2
| not t1.oclIsKindOf(t2)))
```

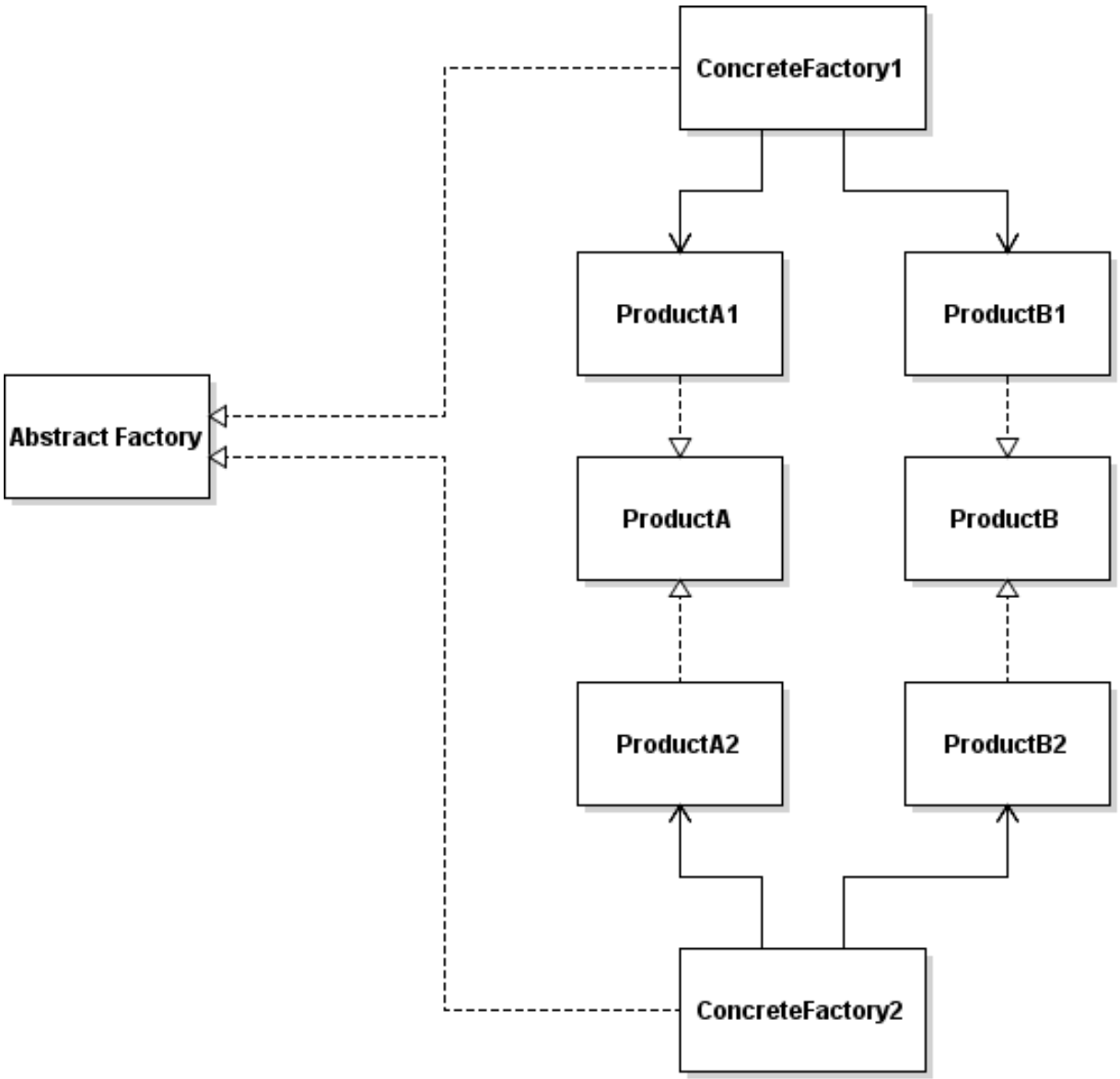

Figure 5.14: UML Diagram of the Abstract Factory pattern structure.

5.4.2 Builder

The Builder pattern is used when we need to create complicated objects with lots of different parts. There are multiple concrete builders, and each one of them uses a different assembly of parts to create a different product object. The building process is managed by a director, who calls on the appropriate builder when necessary to create the desired object.

Roles

- Builder
- ConcreteBuilder
- Product
- Director

Constraints

- TypeIsInterface(Builder)
- TypeImplementsInterface(ConcreteBuilder, Builder)
- BuilderProductMatching(Concretebuilder, [Product])
- ClassHasFieldOfType(Director, Builder)

BuilderProductMatching: This constraint ensures that a given concrete builder is connected to a single product. This means that it has a field of that product type, it instantiates an object of that product type, and it has a method with return type of that product.

```
context BuilderProductMatching inv:
self.constraints -> select(c | c.oclIsTypeOf(ClassHasFieldOfType)) -> forAll(f
| f.primaryParticipant = self.primaryParticipant and self.targetParticipants -> includes(tP
| tp = f.targetParticipant))
context BuilderProductMatching inv:
self.constraints -> select(c | c.oclIsTypeOf(ClassInstantiatesObjectOfType)) ->
forAll(f | f.primaryParticipant = self.primaryParticipant and self.targetParticipants
-> includes(tP | tp = f.targetParticipant))
context BuilderProductMatching inv:
```

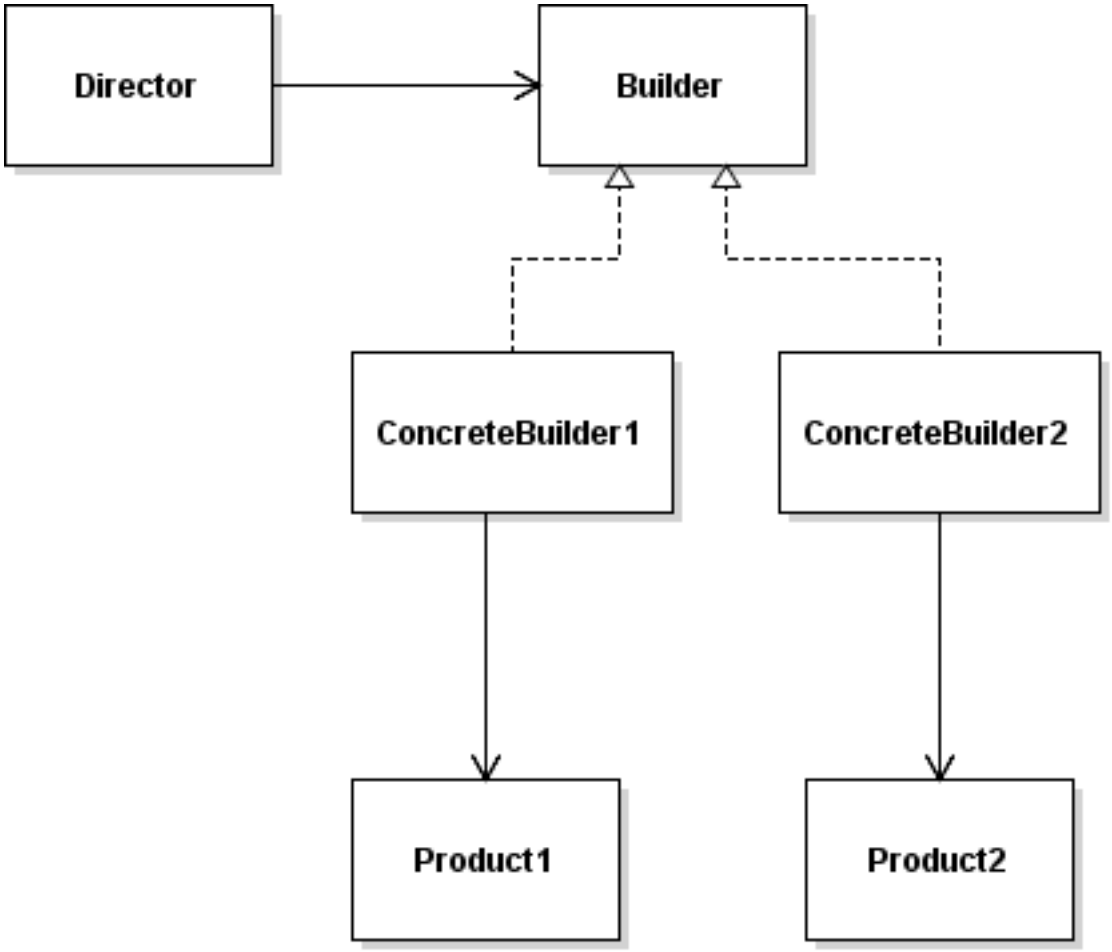

Figure 5.15: UML Diagram of the Builder pattern structure.

```
   self.constraints -> select(c | c.oclIsTypeOf(ClassHasMethodWithReturnType)) ->
forAll(f | f.primaryParticipant = self.primaryParticipant and self.targetParticipants
-> includes(tP | tp = f.targetParticipant))
```

5.4.3 Singleton

The Singleton pattern is used to ensure that a class has only a single instance at any given time, which can be accessed globally. This means that this class has private constructors, and instantiates itself statically.

Roles

- Singleton

Constraints

- SingletonInstantiation(Singleton)

SingletonInstantiation: This constraint ensures 3 things for the singleton: that all of its constructors are private, that it has a static field of itself, and that a static method creates n instance of itself.

```
context SingletonInstantiation inv:
self.primaryParticipant.getMethods() -> select(m | m.isConstructor() = true) ->
forAll(c | c.getModifiers() -> includes(mod | mod.isPrivate() = true))
context SingletonInstantiation inv:
self.primaryParticipant.getFields() -> includes(f | f.getModifiers() -> indludes(mod
| mod.isStatic() = true) and f.getType().oclIsTypeOf(primryParticipant))
context SingletonInstantiation inv:
self.primaryParticipant.getMethods() -> includes(m | m.getModifiers() -> includes(mod
| mod.isStatic() = true) and m.getReturnType().oclIsTypeOf(primaryParticipant))
```

5.5 Unspecified Patterns

Some patterns have not been specified or included in our framework. In particular, there are 3 such patterns: Facade, Iterator, and Prototype.

Facade is a pattern that is so generic, we cannot specify it. It can be defined as a pattern that uses an object as a front-facing interface to obscure more complex behavior behind it - which can be achieved in a myriad of ways. While in some cases of previous patterns with multiple variations we have included at least one interpretation, for Facade we believe there is no particular reason to.

Iterator and Prototype both share the same reason. They are patterns that have become so widespread in their use, the Java Language Specification has included them by default. When we wish for a class to realize the Iterator pattern over a certain type of collection, java provides the java.util.Iterator interface. Similarly, the Prototype pattern can be realized by a class that implements the java.lang.Cloneable interface. This means that there is no need to specify these patterns in terms of declarational constraints, since by simply implementing those interfaces we can be sure the corresponding pattern is indeed correctly applied.

Chapter 6

Validation

In this chapter we will showcase how JPatternJudge can verify patterns in real Java projects. We also present some data concerning the extensibility and reusability of the framework.

6.1 Reusability

In this section we will examine how reusable the realized constraints are. In particular, in Table 6.1 we list the simple constraints used in our pattern specifications, showing how many times each of them is used in a specification in general, and in how many distinct specifications. This includes being used as part of a composite constraint.

As we can see, there are three constraints that are exceedingly common among pattern specifications: TypeIsInterface, TypeImplementsInterface, and ClassHasFieldOfType. In fact, they account for 50 out of the 89 total uses of simple constraints among all pattern specifications (including them being used as part of a composite constraint), or 56% of them. Of course, TypeIsInterface and TypeImplementsInterface are very tightly connected, since if an interface exists in a pattern then necessarily some

Table 6.1: Constraint Reusability Table

Constraint	Total Uses	Pattern Uses
TypeIsInterface	15	13
ClassIsAbstract	4	4
TypeImplementsInterface	17	13
ClassExtendsSuperclass	5	4
ClassHasFieldOfType	18	11
ClassHasFieldCollectionOfType	4	4
ClassHasVariableOfType	2	2
ClassHasMethodWithParameter	7	6
ClassHasMethodWithReturnType	5	5
ClassHasMethodWithReturnCollectionOfType	1	1
ClassInvokesMethod	7	7
ClassInstantiatesObjectOfType	4	4

class will implement it - however, it still goes to show that there is a very high degree of reusability among certain constraints. Perhaps a more indicative example is that 16 out of the 18 pattern specifications (89%) contain at least one inheritance constraint (TypeIsInterface or ClassIsAbstract). In addition, 14 out of 18 specifications (78%) contain a field constraint (ClassHasFieldOfType or ClassHasFieldCollectionOfType). Finally, and perhaps most importantly, there are only 5 instances of constraints that were not specified as part of the abstract model being used in a pattern specification, out of the 87 total constraints among the 18 specifications, or approximately 5,7%. This means that the overwhelming majority of constraints used are already defined in the framework.

6.2 Extensibility

In this section we will analyze JPatternJudge's extensibility in terms of Design Patterns and Constraints. The measure we examine are the lines of code required to add a new design pattern or constraint in the framework, both in terms of total class size and in terms of the populateConstraints() and check() methods for patterns and constraints respectively. These are the methods where the important business logic is located.

In Table 6.2 we can see the number of lines of code for each DesignPattern subclass, as well as how many of them are part of the populateConstraints() method which contains the pattern's constraints.

Table 6.2: LoC measurements in Design Pattern classes

Pattern	**Total LoC**	**populateConstraints() LoC**
Abstract Factory	68	33
Adapter	43	18
Bridge	49	21
Builder	58	25
Chain Of Responsibility	38	13
Command	49	21
Composite	43	15
Decorator	48	21
Flyweight	46	17
Mediator	54	27
Memento	39	13
Observer	42	17
Proxy	40	14
Singleton	31	5
State	42	16
Strategy	40	14
Template Method	39	13
Visitor	48	21
Total	817	324
Average	45	18

As we can see, only a very small amount of code is required to implement pattern specification in JPatternJudge. The caveat is that the constraints required for it are already defined in their own classes, somewhat obscuring the real number of lines. However, the vast majority of constraints used are simple and thus already present in the tool, as was shown in Section 6.1, after the implementation of the constraints defined in Chapter 3, only 5 new constraints were required.

In terms of extensibility of contraints, in Table 6.3 we can see the number of

lines of code for each class, and for the check() method containing the logic that decides if the constraint is satisfied or not for the basic constraints defined in the model (Chapter 3). In Table 6.4, we see the same information for constraints created for a specific pattern (found in Chapter 5), and method constraints (as mentioned in Chapter 4).

Table 6.3: LoC measurements in basic Constraint classes

Basic constraints	**Total LoC**	**check() LoC**
TypeIsInterface	28	3
ClassIsAbstract	31	4
TypeIsInterfaceOrAbstractClass	25	6
TypeImplementsInterface	41	13
ClassExtendsSuperclass	36	11
TypeInheritsFrom	51	25
ClassHasFieldOfType	39	12
ClassHasFieldCollectionOfType	60	28
ClassHasVariableOfType	40	13
ClassHasFieldOrVariableOfType	34	8
ClassHasMethodWithParameter	43	14
ClassHasMethodWithReturnType	42	14
ClassHasMethodWithReturnCollectionOfType	41	14
ClassInvokesMethod	48	18
ClassInstantiatesObject	42	13
Total	601	195
Average	40	13

As we can see, constraint classes are in general quite small, even the ones that express more complicated constraints used in particular patterns. On average, the basic constraint classes contain 40 lines of code, while the pattern-specific constraints contain 58.

A special note should be made for the three extremely large method constraints, MethodHasVariableOfType, MethodInvokesMethod, and MethodInstantiatesObjectOfType. The reason that these classes are so large, is that they need to individually check almost every single different type of Expression and Statement present in a method -

Table 6.4: LoC measurements in pattern-specific and method Constraint classes

Pattern-specific constraints	**Total LoC**	**check() LoC**
BuilderProductMatching	61	27
ConcreteProductInterfaceMatching	41	14
FactoryProductMatching	42	12
TemplateMethodConstraint	82	40
SingletonInstantiation	64	30
Total	290	123
Average	58	25
Method constraints	**Total LoC**	**check() LoC**
MethodHasParameterOfType	30	13
MethodReturnsType	19	4
MethodReturnsCollectionOfType	44	23
MethodHasVariableOfType	141	114
MethodInvokesMethod	471	457
MethodInstantiatesObjectOfType	513	455
Total	1218	1066
Average	244	213

there are 33 different subclasses of Expression, and 22 subclasses of Statement, each with different attributes that need examination. This leads to a total of up to 55 different cases that need to be explicitly examined, and even though the logic for each one is simple, it still leads to this sort of class. However, as we saw in the previous section, the constraints that we have already included in JPatternJudge are sufficient for most specifications, so it is unlikely that such a large class will need to be added.

6.3 Practical Evaluation

In this section we will present the results of using JPatternJudge on a set of Java projects. These were developed by students as part of a university-level Software Engineering course. The use of specific design patterns was explicitly required as part of the assignment, meaning that we both know that patterns do indeed exist in

Table 6.5: Composite pattern constraints

Constraint ID	Constraint
1	ClassIsAbstract(Composite)
2	ClassHasMethodWithParameter(Component, Component)
3	ClassHasMethodWithReturnCollectionOfType(Component, Component)
4	ClassExtendsSuperclass(Leaf, Component)
5	ClassExtendsSuperclass(Composite, Component)
6	ClassHasFieldCollectionOfType(Composite, Component)

the code, and that specific classes have known roles in those patterns. We have two sets of projects from two different years (2018 and 2020).

6.3.1 2018 Project

This project concerned the creation of an application for pattern writing, supporting various templates for patterns as well as generating documents for the patterns in various text formats (simple text, LaTeX).

There are two design patterns utilized in this project that we verified: **Composite** and **Decorator**. Composite is applied on a set of classes that represent an entire pattern language: the leaf is a simple class (PatternPart) that holds some basic data, while the composite (PatternComposite) is subclassed further into a PatternLanguage, a Pattern, and a Decorator class. The Decorator pattern is applied on the PatternComposite class (which has the component role for the decorator pattern); the PatternLanguage and Pattern classes are the concrete components, while the Decorator is obviously the homonymous role and is responsible for adding the commands for a LaTeX document (e.g. declaring sections and subsections).

We examined 15 different projects for that assignment and added the appropriate annotations to each class. We then ran JPatternJudge on each project. The pattern detection and verification proceeded as normal, and the reports were generated. In both Composite and Decorator patterns, there were several common errors that were detected.

In Table 6.5 we can see again the list of constraints in the Composite pattern.

For composite, there were 3 distinct errors that were found:

- CE1: The Component class was concrete. (Constraint 1)

Table 6.6: Decorator pattern constraints

Constraint ID	Constraint
1	TypeIsInterfaceOrAbstractClass(Component)
2	TypeInheritsFrom(ConcreteComponent, Component)
3	TypeInheritsFrom(Decorator, Component)
4	ClassHasFieldOfType(Decorator, Component)

- CE2: The method returning a collection of components was missing. (Constraint 3)
- CE3: No inheritance relations at all, and Composite did not have a field that was a collection of Components. (Constraints 4, 5, 6)

We represent them with the shorthand CE, for **C**omposite **E**rror.

In Table 6.7, we can see the results of the verification: 10 of the 15 projects had a concrete Composite class (CE1), and 4 projects lacked the collection return method (CE2). CE3 was present only in a single instance, but it meant that the pattern structure was completely wrong. Pattern instances that did not contain any errors an were verified as correct are marked with ” - ”.

As for the Decorator pattern, 4 of the projects did not implement it at all, while all except a single one of the others shared the same 2 mistakes. The constraints for the Decorator pattern are shown again in Table 6.6, and the errors are listed below:

- DE1: The Component class was concrete. (Constraint 1)
- DE2: The Decorator class was missing a field of type Component. (Constraint 4)

We represent them with the shorthand DE, for **D**ecorator **E**rror.

Here the issue also lies with the structure of the project itself, rather than only on the developers. The component role of Decorator is the PatternComposite class, which needs to be concrete for the purposes of the Composite pattern in participates in. So there is a conflict in constraints there. However, according to the commonly accepted specification, these implementations are still wrong and were verified as such.

Table 6.7: Verification results for 2018 project

Project	**Composite Errors**	**Decorator errors**
Project 1	CE1	DE1
Project 2	CE1	DE1, DE2
Project 3	CE1, CE2	Missing
Project 4	CE3	Missing
Project 5	CE1	DE1, DE2
Project 6	CE1, CE2	DE1, DE2
Project 7	-	Missing
Project 8	-	DE1, DE2
Project 9	CE1	DE1, DE2
Project 10	CE2	DE1, DE2
Project 11	-	DE1, DE2
Project 12	CE1, CE2	DE1, DE2
Project 13	CE1	Missing
Project 14	CE1	DE1, DE2
Project 15	CE1	DE1, DE2

Table 6.8: Command pattern constraints

Constraint ID	Constraint
1	TypeIsInterface(Command)
2	TypeImplementsInterface(ConcreteCommand, Command)
3	ClassHasFieldOfType(ConcreteCommand, Receiver)
4	ClassHasFieldOrVariableOfType(Invoker, Command)
5	NegativeConstraint(ClassHasFieldOfType(Invoker, ConcreteCommand))

6.3.2 2020 Project

This project concerned the creation of a simple Text-To-Speech application composed of a GUI, a text editor, and the text-to-speech conversion using an external library.

There are three design patterns of interest in this project: **Command**, **Template Method**, and **Strategy**. We examine a total of 10 projects.

We have 3 instances of Command in each project: One for manipulating documents(creating, opening, saving), one for converting entire documents to speech, and one for converting single line to speech. e can see the constraints for the Command pattern in Table 6.8. An issue that appears is that the Command role in all 3 cases is served by the ActionListener interface, provided by java.awt - an external library that we o not have access to. As such, in some cases we cannot annotate it so as to properly detect the pattern instance. This means that JPatternJudge won't proceed with individual constraint verification if it detects a role missing, as outlined in the verification mechanism in Chapter 4. However, in some projects it was implemented by a new interface included in the project. In general, we detected 3 distinct problems with the Command patterns:

- CRM: Command role not detected - belongs to external library.
- CE1: Invoker does not contain a Command field or variable. (Constraint 4)
- CE2: The Concrete Commands do not have a field of Receiver. (Constraint 3)

CE is shorthand for **C**ommand **E**rror, and CRM is for **C**ommand **R**ole **M**issing.

The Strategy and Template Method patterns are combined, and used to implement various different encoding algorithms for the documents: There is a Template Encoding class, and an EncodingStrategy class. One of them is subclassed to the

Table 6.9: Strategy pattern constraints

Constraint ID	Constraint
1	ClassHasFieldOrVariableOfType(Context, Strategy)
2	ClassInvokesMethod(Context, Strategy)
3	TypeIsInterface(Strategy)
4	TypeImplementsInterface(ConcreteStrategy, Strategy)

Table 6.10: Template Method pattern constraints

Constraint ID	Constraint
1	ClassIsAbstract(Template)
2	ClassExtendsSuperclass(Concrete, Template)
3	TemplateMethodConstraint(Template)

other (varying among projects) and then further subclassed by the concrete strategies/template classes (which are one an the same). This 2-layer hierarchy caused the presence of false negatives in every instance, as was described in Chapter 3: The concrete strategies inherit directly from one of the Strategy or Template classes, but not both; and indirect inheritance cannot be detected by our implementation. The constraints for the Template Method and Strategy pattern are given in Tables 6.10 and 6.9 respectively. In general, we found the following errors:

- IH: Indirect Hierarchy. (Template constraint 2 or Strategy constraint 4)
- TE1: Template method was not appropriately defined. (Template constraint 3)
- SE1: Strategy context does not invoke a strategy method. (Strategy constraint 2)

Where TE is shorthand for **T**emplate **E**rror, and SE for **S**trategy **E**rror.

We can see the results of the verifications in Table 6.11. We cannot properly detect any of the Command instances for 4 of the 10 projects, and we cannot detect its first instance for an additional 1. However, in every detectable instance, we can see that there were errors committed that JPatternJudge found during verification. As for Template and Strategy, we do indeed have the false negative of Indirect Hierarchy (IH), but we also detected additional mistakes that were correctly verified.

Table 6.11: Verification results for 2020 project

Project	Command 1	Command 2	Command 3	Template Method	Strategy
Project 1	CRM	CE1, CE2	CE1	-	IH
Project 2	CE1, CE2	CE1, CE2	CE1, CE2	-	IH
Project 3	CRM	CRM	CRM	IH	-
Project 4	CE1, CE2	CE1	CE1, CE2	Missing	SE1
Project 5	CRM	CRM	CRM	-	IH
Project 6	CRM	CRM	CRM	TE1	IH
Project 7	CE1, CE2	CE1, CE2	CE1, CE2	IH, TE1	-
Project 8	CE1	CE1	CE1	TE1	IH
Project 9	CRM	CRM	CRM	TE1	IH
Project 10	CE1	CE1	CE1	-	IH

6.3.3 Conclusions

As we saw form the results of the project verification, JPatternJudge can function in a real environment outside of custom-made projects for testing, with the capacity to correctly verify patterns as well as detect errors.

From the data presented in Tables 6.7 and 6.11, we can extract a measure of correctness over the results of the verifications. We determined the Recall of JPatternJudge to be 1 - i.e., all of the patterns that contained errors were in fact verified as incorrect; whether we examine patterns individually or in total. This was determined through manual examination of the pattern instances. Precision has differences from pattern to pattern, and its lower score is due to the practical limitations of the tool (leading to errors CRM and IH) mentioned earlier. The exact results for Precision can be found in Table 6.12.

In addition, we can measure the amount of effort required for the developers to use JPatternJudge in the following way: The effort required is simply the number of annotations used, each one of which is only one line of code - which in turn is equal to the number of classes participating in design patterns. We can see this metric in Tables 6.13 and 6.14 for each set of projects. The larger effort required in the second set of projects is due to the presence of the Command pattern, and the multiple ConcreteCommand classes associated with it.

We can also see that there is indeed a reason to use JPatternJudge, as mistakes

Table 6.12: Precision scores

Pattern	**Precision Score**
Composite	1
Decorator	1
Command	0.567
Template Method	0.6
Strategy	0.125
Total	0.667

Table 6.13: Developer effort for annotation as LoC - 2018 Projects

Project	**Annotation LoC**
Project 1	6
Project 2	6
Project 3	3
Project 4	3
Project 5	6
Project 6	6
Project 7	3
Project 8	6
Project 9	6
Project 10	6
Project 11	6
Project 12	6
Project 13	6
Project 14	6
Project 15	6
Average	5.4

Table 6.14: Developer effort for annotation as LoC - 2020 Projects

Project	**Annotation LoC**
Project 1	22
Project 2	22
Project 3	27
Project 4	20
Project 5	17
Project 6	21
Project 7	24
Project 8	16
Project 9	12
Project 10	21
Average	20.2

are indeed being committed by programmers when using Design Patterns. From the data presented, we can also see that these mistakes are very commonly shared among programmers; although interpreting this is outside the scope of our work. Of course, as we mentioned these projects were part of a university assignment rather than professional code, which means that a lower level of quality and experience was expected. However, this doesn't mean that professional developers do not make similar mistakes; in addition, patterns can be broken due to the inevitable evolution of a system, meaning that by verifying the patterns after every system update developers can avoid problems occurring down the line unexpectedly. Finally, by using JPatternJudge and annotating the design patterns, developers are also documenting their existence; this can assist later on in maintenance, as was described in [15].

Chapter 7

Future Work and Conclusion

7.1 Future Work

There is a great deal of possible further work that may follow on from JPatternJudge. In this section we will present some possibilities.

7.1.1 Expanding JPatternJudge Functionality

An obvious expansion to JPatternJudge includes the specification of more pattern variants. If done by independent developers, this will also give us a clearer picture of the framework's extensibility, since our experience with JPatternJudge obviously makes it an easier process for us.

Another possibility is to add further functionality to JPatternJudge beyond the pattern specifications themselves. We can define each role of a pattern as having one or multiple classes serving that role, which would allow us to detect and flag instances where a role with a single class is for some reason served by many. However, this would have a slight impact on extensibility without really offering any significant advantage.

Alternatively, we can include supporting multiple variants of a pattern in a single definition, meaning that the class corresponding to the design pattern will include

specifications for all variations. This can make the tool easier on the end user, since there will be no need to be so exact with the class annotation - however, this will introduce more complexity on the back end, further hampering extensibility. For example, if we had two variations of Decorator, normally we would create two separate specifications e.g. ”Decorator 1” and ”Decorator 2” in two different classes. A developer would then have to be careful to select the proper variant in the annotation of his source code, which somewhat defeats the purpose of the tool - if the developer is aware of the differences between pattern variations and can select the correct one for annotation, it’s unlikely they commit an error in the actual implementation. Ideally, they would simply annotate as ”Decorator”, and JPatternJudge would automatically detect which variation is used. This is a very difficult task, since there is the question of how, exactly, we decide which variation is used. One way would be to iterate over all variants, and select the one where all constraints are satisfied; but of course this means we can only decide if the pattern if correctly applied, again essentially losing the main function of JPatternJudge.

7.1.2 Design Pattern Detection

The problem of detecting a design pattern is quite different from the problem of verification. We do not have prior knowledge or suspicion about the presence of a specific pattern - rather, we blindly search the source code for possible structures that correspond to one. Of course, both problems of verification an detection are first reliant of specification of patterns. Using our current method of declarative constraints, pattern detection will not yield good results, even with a large number of pattern variations specified. The general structures used in patterns are very common, leading to a very high number of false positives - meaning that another form of specification must be used. There have been a multitude of works in this field, such as using template matching[23], graph-based similarity scoring[24], and machine learning[25].

Since using JPatternJudge as a basis for full pattern detection is not promising, we can shift to a more approachable problem: partial pattern detection. So far, users of JPatternJudge must annotate every class or interface that participates in a pattern with the correct role. This can become cumbersome in large projects, so we could try and find a way to reduce this workload. Additionally, a developer might be certain about the role of a specific class, but not about the roles of other classes. A possibility

for additional functionality would thus be to ”fill in the blanks”, so to speak, when not every role in a pattern has been annotated. This will require us to examine each pattern to decide in which cases the pattern can be filled in, how many roles are needed for us to be sure of the result, an what is the tolerance (if any) on errors in the application of the pattern.

7.1.3 Automatic Introduction or Repair of Design Patterns

Moving further than detection, we have two more possibilities that might warrant further exploration, although they far exceed the scope of JPatternJudge.

First, automatic suggestion and introduction of design patterns. An almost opposite concept of pattern detection, this is the problem of deciding a piece of code is not part of a design pattern, but could be. This is a very difficult issue, since we cannot know the programmer’s intent based simply on the structure of the code. We could have an empirical list of possible code structures that can be shifted to the form of a design pattern, but this isn’t a general solution.

Alternatively, we can attempt to automatically repair erroneous pattern instances. Meaning that if JPatternJudge detects an error in a specific pattern application, we can not only show what the error was, but also suggest ho it cn be resolved. This isn’t as simple as only altering the classes so that the constraints are obeyed; we must make sure that the function of the code remains the same. In this context, both repairing a pattern and automatically implementing one are related to the concept of Refactoring, since we want to restructure exiting code without altering its external behavior.

7.2 Conclusion

In this thesis, we presented the problem of formulating a specification for a software design pattern, as well as automatically verifying its implementation. Various tools have been developed in the past, but none of them are being supported at this time. In addition, they used complicated pattern languages and were difficult to use. We presented JPatternJudge, an Eclipse-based framework for design pattern specification and verification.

JPatternJudge uses a system of declarative constraints to define patterns. While other works have also used declarative constraints in the past, we introduce a new

concept of negative constraints that have been ignored so far, in order to have a stricter definition of pattern implementations. It is also much easier to extend with new variations of design patterns, with our implementation of the framework requiring on average only 45 lines of code for a new pattern specification, and 40 for a constraint. In addition, the constraints defined are very highly reusable.

Finally, we showed that JPatternJudge can indeed verify design patterns in real Java projects, correctly identifying mistakes according to the specifications.

Bibliography

[1] C. Alexander, *A pattern language: towns, buildings, construction*. Oxford University press, 1977.

[2] E. Gamma, *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[3] A. V. Zarras, "Common mistakes when using the command pattern and how to avoid them," in *Proceedings of the European Conference on Pattern Languages of Programs 2020*, 2020, pp. 1–9.

[4] A. Blewitt, "Hedgehog: automatic verification of design patterns in Java," Ph.D. dissertation, University of Edinburgh. College of Science and Engineering., 2006.

[5] D. Distefano and M. J. Parkinson J, "jStar: Towards practical verification for java," *ACM Sigplan Notices*, vol. 43, no. 10, pp. 213–226, 2008.

[6] D. Mapelsden, J. Hosking, and J. Grundy, "Design pattern modelling and instantiation using DPML," in *ACM International Conference Proceeding Series*, vol. 21, 2002, pp. 3–11.

[7] T. Taibi and D. C. L. Ngo, "Formal specification of design pattern combination using BPSL," *Information and Software Technology*, vol. 45, no. 3, pp. 157–170, 2003.

[8] K. Beck and W. Cunningham, "Using pattern languages for object oriented programs," in *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1987.

[9] F. Wedyan and S. Abufakher, "Impact of design patterns on software quality: a systematic literature review," *IET Software*, vol. 14, no. 1, pp. 1–17, 2019.

[10] J. M. B. D. J. Helen and J. Yang, "OO design patterns, design structure, and program changes: An industrial case study," in *IEEE International Conference on Software Maintenance*, 2001.

[11] D. Posnett, C. Bird, and P. Dévanbu, "An empirical study on the influence of pattern roles on change-proneness," *Empirical Software Engineering*, vol. 16, no. 3, pp. 396–423, 2011.

[12] M. Vokac, "Defect frequency and design patterns: An empirical study of industrial code," *IEEE Transactions on Software Engineering*, vol. 30, no. 12, pp. 904–917, 2004.

[13] L. Prechelt, B. Unger-Lamprecht, M. Philippsen, and W. F. Tichy, "Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance," *IEEE Transactions on Software Engineering*, vol. 28, no. 6, pp. 595–606, 2002.

[14] C. Gravino, M. Risi, G. Scanniello, and G. Tortora, "Does the documentation of design pattern instances impact on source code comprehension? results from two controlled experiments," in *2011 18th Working Conference on Reverse Engineering*. IEEE, 2011, pp. 67–76.

[15] G. Scanniello, C. Gravino, M. Risi, G. Tortora, and G. Dodero, "Documenting design-pattern instances: A family of experiments on source-code comprehensibility," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 3, pp. 1–35, 2015.

[16] A. Ampatzoglou and A. Chatzigeorgiou, "Evaluation of object-oriented design patterns in game development," *Information and Software Technology*, vol. 49, no. 5, pp. 445–454, 2007.

[17] "Refactoring guru design patterns." [Online]. Available: https://refactoring.guru/design-patterns/catalog

[18] "Tutorials point design patterns." [Online]. Available: https://www.tutorialspoint.com/design_pattern/index.htm

[19] L. Lamport, "The temporal logic of actions," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 3, pp. 872–923, 1994.

[20] J. Cabot and M. Gogolla, "Object constraint language (ocl): a definitive guide," in *International school on formal methods for the design of computer, communication and software systems*. Springer, 2012, pp. 58–90.

[21] "Java specification." [Online]. Available: https://docs.oracle.com/javase/specs/

[22] "Eclipse package org.eclipse.jdt.core.dom." [Online]. Available: https://help.eclipse.org/2020-03/ntopic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/package-summary.html

[23] J. Dong, Y. Sun, and Y. Zhao, "Design pattern detection by template matching," in *Proceedings of the 2008 ACM symposium on Applied computing*, 2008, pp. 765–769.

[24] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design pattern detection using similarity scoring," *IEEE transactions on software engineering*, vol. 32, no. 11, pp. 896–909, 2006.

[25] M. Zanoni, F. A. Fontana, and F. Stella, "On applying machine learning techniques for design pattern detection," *Journal of Systems and Software*, vol. 103, pp. 102–117, 2015.