The Difficult Art of Combining Design Patterns with Modern Programming Languages

New Design Pattern Insights via a Classification of Design Patterns Relative to Languages

Eirik Emil Neesgaard Årseth

Thesis submitted for the degree of
Master in Informatics: Programming and System Architecture
60 credits

Institute of informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2020

The Difficult Art of Combining Design Patterns with Modern Programming Languages

New Design Pattern Insights via a Classification of Design Patterns Relative to Languages

Eirik Emil Neesgaard Årseth

The Di□cult Art of Combining Design Patterns with Modern Programming Languages

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

Software design patterns are high-level design solutions to common occurring problems within software development. The topic of this study is the relationship between design patterns and modern programming languages, and whether or not design patterns are replaceable by language features.

The main contribution of this thesis is a classification scheme: Previous discussion of language support for design patterns, lack nuance. Our research show that the different classifications of support, that a language can exhibit for a pattern, exists on a spectrum. We propose a model that establish four different classifications of support, as opposed to a binary model, that only consists of the two classifications, supported and unsupported.

This classification scheme is used for analysing concrete pattern-language relationships. The patterns Singleton, Strategy, Command, Decorator, Prototype, Iterator and Proxy are implemented in multiple languages, including Emerald, Java, JavaScript, Python and Scala. By applying our proposed classification scheme, the level of support the languages show for the different patterns is determined. Singleton is the only pattern shown to be completely redundant, i.e., fully supported, in some languages. The six other patterns are all shown to have partial support in one or more languages. Object model, lambda functions, prototypical inheritance, library features and dynamic typing are the language features that the pattern support is attributed to, in the examined patterns.

Results confirm that design patterns and programming languages are two closely related concepts. The study indicates that some patterns are replacements for missing language features, whereas others are independent of language.

Acknowledgements

Foremost, I would like to thank my thesis advisor, Eric Bartley Jul. I am greatly appreciative of his helpful guidance, humorous digressions and our many conversations.

I must also acknowledge my parents. Their support and encouragement throughout my studies, has been unwavering.

Contents

List of Figures

List of Tables

Listings

Part I

Introduction

Chapter 1

Introduction

The topic of this master thesis is the relationship between design patterns and programming languages. In particular, the study is interested in whether or not design patterns are replaceable by programming language features. This chapter introduces the study at a glance.

1.1 Motivation

25 years have passed since the publication of *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma et. al [10]. Despite its age, the book still serves as the main authority on the subject of object oriented design patterns. However, a lot has happened in the world of programming languages since then. New languages have emerged, and the languages still in use have evolved. In particular, fewer languages seem to be purely object oriented. Instead, industry-leaders such as Java, Python, JavaScript and C++ are multi-paradigm languages that exhibit elements of both object oriented and functional programming. In addition, dynamic and weak typing schemes seems to become more prevalent. This shift calls the need for certain design patterns into question. Is it possible that some design patterns are made redundant by modern language features? Or perhaps that that modern languages facilitate improved or simplified implementations of the traditional design patterns. These questions are the motivation of this thesis.

1.2 Problem Statement

Software design patterns are high-level descriptions of solutions to common occurring problems within software development. Since the advent of de-

sign patterns, the world of programming languages has changed. Languages have evolved and new languages have appeared. What can be learned about the relationship between design patterns and programming languages from this shift in programming languages? Are design patterns and language features, interchangeable concepts? More specifically, are any design patterns made redundant or replaced by language features? And do any patterns see significantly improved implementations, in light of modern programming languages? This is our main problem statement.

Further we wonder what levels of support a language can have for a specific pattern. What does it mean that a programming language support a certain design pattern? Does full support mean that the pattern is superfluous? And what grades of distinction exists between full and no support? Can we create a classification scheme that accurately describes this spectrum of pattern-language relationships?

Finally, if our conjecture that modern programming languages support some design pattern holds true, which language features are responsible for such support?

1.3 Goal

This thesis aims to gain new insights regarding the relationship between design patterns and programming languages. The thesis should contribute to the body of research on both topics, by formulating a theory on how they interrelate.

On a more technical level, we wish to create a classification scheme for describing the relationship between patterns and languages, in order to analyse concrete pattern-language relationships. By applying that classification scheme, we wish to determine whether or not a selection of design patterns are made redundant, or see implementation improvements in light of modern programming languages.

1.4 Approach

The approach of this thesis is experimental: We implement several programs in various programming languages. The programs should mimic use cases where selected design patterns can be applied. In addition to implementing the design patterns, alternative implementations utilizing language features are also explored. These alternative implementations will be evaluated and

compared to the original design pattern, in order to classify the pattern-language relationships through a classification scheme.

To carry out the work proposed above, we will firstly create a scheme for classifying the relationships between languages and patterns. Then we will look into specific patterns and languages to determine which relationships that seem interesting to investigate. The implementations and classification described above will then be carried out. Finally we will summarize and attempt to answer the questions put forth in section 1.2 in light of our findings.

1.5 Work Done

This thesis proposes a classification scheme for the relationship between design patterns and programming languages. The scheme is applied to a selection of implementations, to analyse the pattern-language relationships. A more detailed outline of the work done can be divided into four sections:

1. The existing body of research on the relationship between design patterns and programming languages is studied.
2. A scheme for classifying the relationship between programming languages and design patterns is created. The scheme introduced four classifications of support that a programming language could exhibit for a design pattern. This allows us to be more formal and precise in our evaluation of pattern-language relationships.
3. We present seven di□erent use cases that should mimic systems where design patterns can be applied. Each of the seven programs are designed to showcase a single design pattern. The designated design patterns are Singleton, Strategy, Command, Decorator, Prototype, Iterator and Proxy. For each of the proposed use cases, several implementations are created in various programming languages. The languages utilized in this study are EMERALD, JAVA, JAVASCRIPT, PYTHON and SCALA. For each of the design patterns we classify their relationship to the programming languages utilized in the use case implementations.
4. Lastly we evaluate the results of the implementations, and formulate a theory on design patterns in light of the current world of programming languages.

1.6 Evaluation Criteria And Results

The evaluation criteria and results of this thesis can be divided between two levels: There are evaluation criteria and results concerning the project as a whole, and there are technical criteria and results regarding specific design patterns and their implementations.

1.6.1 Project Evaluation Evaluation Criteria

The criteria for the evaluation of this project is whether or not new insights have been made, regarding design patterns and how they relate to programming languages. Is it meaningful to study how languages and patterns a□ect one another, or are they completely independent concepts? Evaluations of these questions are subjective.

1.6.2 Design Patterns Evaluation Criteria

Our classification scheme defines the criteria used in the evaluation of design patterns and programming languages. Each pattern-language pair are evaluated in accordance to this scheme. The classification within this scheme is largely subjective and empirical. When deciding whether or not an alternative implementation is an improvement over the original design pattern, we mainly consider complexity.

By complexity we simply mean how complicated a program is to implement or understand. We define readability and conciseness as components of complexity. Lines of code can be a useful metric for complexity, but we will mainly make subjective evaluations.

1.6.3 Project Results

This study presents new insights as to how design patterns relate to programming languages. The study confirms that some design pattern are replaceable by language features.

The main finding is that the relationships between pattern and language exists on a spectrum. That is, there exist classifications of their relationship, other than supported and unsupported. This spectrum of classifications of support resulted in a classification scheme, in which we established four di□erent classifications of support, that a language can show for a pattern. More classifications likely exists, but we find four enough to create a practical

model. This is in contrast to a traditional binary model, that only has the classifications supported and unsupported.

Another result, though not as novel, is that we have determined the relationship between design pattern and programming language to be a meaningful area of study: They are not unrelated concepts, as they do indeed a□ect one another. The results suggest that more research can be done on the topic.

1.6.4 Design Pattern Results

The results of our study show that certain design patterns are made completely redundant in some programming languages. That is to say, the programming language supports the design pattern fully, and provides LANGUAGE LEVEL SUPPORT, as per our classification scheme. Some design patterns are partly supported or have improved implementations, whereas some patterns appears to be unchanged. We also found that some design pattern are explicitly supported for standard library objects, e.g., JAVA's built in ArrayList supports the Iterator pattern, whereas arbitrary JAVA objects have no such support.

Most of the supported patterns can be attributed to a selection of language features, comprising of:

- Functions as objects
- Alternative object models
- Prototypical inheritance
- Dynamic typing
- Library provided features

Our classification scheme is also a technical result of this thesis. Through its creation we have identified 4 di□erent classifications of support, that a programming language can exhibit for a specific design pattern.

1.7 Conclusion

Our results lead us to the conclusion that design patterns and programming languages greatly a□ect one another. Some design patterns are made redundant by modern programming language features. Other patterns appear to be partially supported by several languages.

The main contribution of this study is a scheme for classifying the the level of support, that a language exhibits for a pattern. We find previous discussion

of design pattern support to be lacking in nuance. Our studies show that the pattern-language relationships exists on a spectrum. We have established a model, proposing four different classifications.

By applying the classification scheme to multiple design pattern implementations, it is shown that many design patterns are partially supported in various programming languages. Support for design patterns can be attributed to numerous factors. Intentional pattern implementation, library support and coincidal support from new language features. In particular it seems as though the shift in modern programming, towards multi-paradigm languages plays a part.

Thus we conclude that the need for some design patterns will vary, whereas some patterns probably are language-independent.

1.8 Contributions

The main contribution of this thesis is the proposal of a scheme for classifying the relationships between design patterns and programming languages. The classification scheme is used for evaluating to what degree a pattern is supported by a language.

Applying this classification scheme to a selection of design pattern implementations, provides another contribution: The tables in chapter 9 outlines the specific patterns that are supported by a selection of programming languages.

1.9 Limitations

This study have several limitations.

- Subjective evaluation of code
- Limited number of design patterns and programming languages
- Lack of depth: We are only working with simple programs that are trivial to implement. Implementation in large enterprise systems might not yield the same same results.
- Developer experience and expertise: All programs are implemented by a single student. A more skilled developer might have ideas for better implementations.

- Limited number of implementations: The fact that a design pattern can be replaced for a single use case, does not guarantee that the same holds true for every instance.

1.10 Outline

This thesis consists of four parts: "Introduction", "Background", "Implementation and Classification" and "Evaluation and Conclusion"

Part I: Introduction

Chapter 1 - Introduction: The first section gives a short overview of the thesis. Sections such as motivation, work done and results briefly summarizes the entire study.

Part II: Background

Chapter 2 - Design Patterns: Gives an introduction to design patterns, including GoF's classification scheme and a selection of design patterns.

Chapter 3 - Programming Languages: Covers central topics in the world of programming languages. Concepts such as language paradigms and typing schemes are presented.

Chapter 4 - Language Presentation: The programming languages used in this study are briefly introduced, along with explanations of language features relevant to this study.

Chapter 5 - Previous Works: This chapter outlines the research previously done on the relationship between design patterns and programming languages.

Part III: Implementation and Classification

Chapter 6 - Classification Scheme: A scheme for classifying the relationship between programming languages and design patterns is proposed.

Chapter 7 - Proposed Relationships: This chapter introduces pairs of design patterns and programming languages that are analysed.

Chapter 8 - Implementations: Seven design pattern are presented followed by presentations of use cases where the patterns can be applied. These use cases are implemented with and without the use of the design pattern in various programming languages. Each pattern-language relationship is then classified.

Part IV: Results and Conclusion

Chapter 9 - Design Pattern Results: The results of the previous chapter are evaluated, and we make note of language features influencing the results.

Chapter 10 - Project Results: Results concerning the project as whole are presented. That is, what have been learned about the relationship between programming languages and design patterns.

Chapter 11 - Conclusion: We summarize our findings and make conclusions in light of them.

Chapter 12 - Future Work: A Brief proposal of potential future work on the relationship between patterns and languages.

Part II

Background

Chapter 2

Design Patterns

In this chapter a general introduction to design patterns is given. Firstly, a definition of design patterns is presented, in addition to an account of design patterns origins. Three design pattern examples are covered, including a simple implementation of each pattern.

2.1 What are Design Patterns?

Within software engineering, design patterns are general solutions to recurring problems that can be addressed by design decisions when creating an application [10]. Over time, the observant system designer might recognize that the same designs can be applied to a multitude of seemingly unrelated applications. The reappearing design can then be identified as a design pattern. Design patterns are not implementations or algorithms that trivially can be introduced into a system or application, but rather an approach to how design issues can be solved.

The concept of design patterns was first introduced in the world of architecture by Christopher Alexander [1]. In the world of software engineering, design patterns were popularized by the group of authors known as the Gang of Four, consisting of Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Henceforth the Gang of Four will be abbreviated as GoF. In their book *Design Patterns: Elements of Reusable Object-Oriented Software*, that we shall refer to as GofBook, GoF presents their definition of design patterns, introduced a classification scheme for design patterns and catalogued 23 patterns, each identified more than once in di□erent systems [10]. This thesis is strongly based on GoF's interpretation and classification of design patterns.

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

Table 2.1: Classification scheme of design patterns

2.2 Classification of Design Patterns

The classification scheme introduced by GoF, organize patterns by two criteria: purpose and scope [10, p. 9]. The purpose of a pattern can be creational, structural or behavioral. Creational patterns are concerned with creating objects. Structural patterns attempt to impose a certain arrangement of objects and classes. Behavioral patterns deal with the interaction between classes and objects and the delegation of their responsibilities.

Another way of categorizing patterns is scope. By the definitions of GoF, the scope of a pattern can either be classes or objects. Class patterns mainly organize designs through class-hierarchies whereas object patterns are concerned with objects that can be altered at runtime.

Table 2.1 lists the patterns catalogued by GoF and, and their arrangement within the classification scheme. The patterns presented in this chapter are highlighted in bold font.

2.3 Design Pattern Examples

In this section, three design patterns are presented. The Singleton, Composite and Observer patterns are explained and illustrated through simple JAVA implementations and UML 2.5 [19].

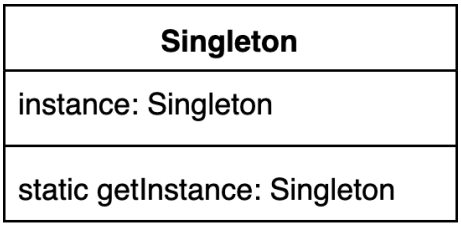

Figure 2.1: Singletom UML specification

2.3.1 Singleton

Singleton is a creational design pattern [10, p. 127]. Its purpose is to ensure that there only is a single object instance of some class or type. In addition, the pattern provides a global access point to the singleton object.

In many cases, it is important that some class only has a single instance. For example a file system: Although real-life implementations might differ, one can imagine that an operating system only has a single file system. It would then be important that all attempts to interact with the file system should be directed to the single object representing it. To guarantee that the single object is used, one needs to ensure that no accidental object instantiations occur, by making multiple instantiations impossible. In class-oriented languages, such as Java and C++, this is achieved by making the class itself responsible for its own initialization. As showcased by the UML in figure 2.1, the Singleton design pattern is very simple. Listing 2.1 provides a generic implementation of the pattern, and listing 2.2 gives an example implementation. The term *generic implementation* is used to describe general implementations that presents the structure of the pattern, without a specific usage.

Nevertheless, Singleton can be a highly useful pattern that ensures that classes or objects only are instantiated when, and in the manner they are intended.

Listing 2.1: Java Generic Singleton Example

```
1 class Singleton {
2   Singleton instance;
3   private Singleton(){}
4
5   public getInstance(){
6     if (instance == null){
7       instance = new Singleton();
8     }
9     return instance;
```

```
10    }
11    // Application specific code...
12  }
```

Listing 2.2: Java file-system Singleton example

```
1  class FileSystem {
2    FileSystem instance;
3    ArrayList<Folder> folders;
4    ArrayList<File> files;
5
6    private FileSystem(){}
7
8    public getInstance(){
9      if (instance == null){
10       instance = new FileSystem();
11     }
12     return instance;
13   }
14
15   File getFile(String fileName)(){...}
16   Folder getFolder(String folderName){...}
17 }
```

2.3.2 Composite

Composite is a structural design pattern pattern [10, p. 163]. It enables the programmer to access single objects, and compositions of the same objects uniformly. It does so by composing objects into tree structures, where each subtree can be accessed as an independent tree. Each node in the tree can either be a leaf, representing some final object, or it can be a composition of multiple objects, in the form of a subtree with one or more child-nodes. In order to access the leafs and compositions of objects uniformly, that is treating the objects as they were of the same type, the two types of nodes must both implement a common interface. The UML of the pattern can be found in figure 2.2, and a generic Java implementation in listing 2.3.

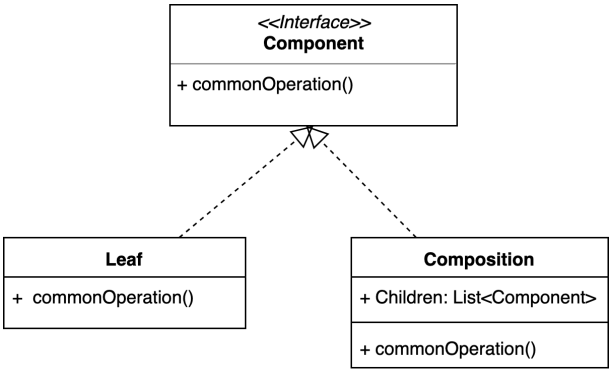

Figure 2.2: Composite UML specification

Listing 2.3: Java generic implementation of Composite pattern

```
interface Component {
 void commonOperation();
}

class Leaf implements Component {
 void commonOperation(){
  // Common operation for self
 }
}

class Composistion implements Component {
 List<Component> children;

 void commonOperation(){
  // Initial setup
  for(Component child: children){
   child.commonOperation()
  }
  // Final configuration
 }
}
```

The pattern is useful when modelling systems that have a tree-like structure, or at least can be modelled by a tree-structure. An example of such use is a graphics application where you can draw graphical elements such as circles, triangles and boxes. Each of these elements are represented by their own classes, that provides the method draw. These will be our leaf-node classes. In addition one can create more complex graphical elements by combining the already mentioned elements. We call this new graphical element a figure.

In order to treat all the elements a figure is made up of as a single graphical element, e.g., when moving or deleting the figure, we must have a class that represent the composition of the elements in the figure. We call this class FIGURE, which has the responsibility of providing the same interface as the leaf-node classes, but implementing it in a way that makes sense for the whole of the figure. A Java implementation of this simple application is provided in listing 2.4

Listing 2.4: Simple Java implementation of Composite pattern

```
1  interface GraphicalElement{
2   void draw(int xCordinate, int yCordinate);
3  }
4
5  class Triangle implements GraphicalElement {
6   void draw(int xCordinate, int yCordinate){
7      // Draw Triangle in right place
8   }
9  }
10
11 class Box implements GraphicalElement {
12  void draw(int xCordinate, int yCordinate){
13     // Draw Box in right place
14  }
15 }
16
17 class Figure implements GraphicalElement {
18  List<GraphicalElement> children;
19
20  void draw(int xCordinate, int yCordinate){
21   int offset = 0;
22   for(GraphicalElement child: children){
23    child.draw(xCordinate + offset, yCordinate)
24    offset += 10;
25   }
26  }
27 }
```

2.3.3 Observer

Observer is a behavioral design pattern which establishes a one-to-many relationship, from the master subject, to a collection of observers [10, p. 293]. The intention is that when the state of the subject changes, all observers

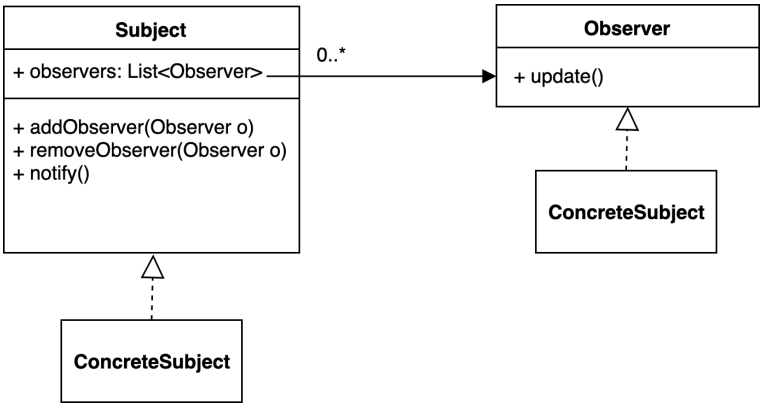

Figure 2.3: Observer UML specification

should be notified of this. This allows the observers to respond to the change of state immediately, without constantly having to check or monitor the state of the subject.

The UML in figure 2.3 presents how this functionality is achieved. The SUBJECT class is responsible for maintaining a collection of all of its observers. The observers can be of any custom type, but must extend the OBSERVER class, which requires an UPDATE method. When the subject's state changes, the NOTIFY method in SUBJECT must be called. The subject will iterate over all of its observers, and invoke their UPDATE methods. A generic Java implementation is given in listing 2.5.

Listing 2.5: Java generic implementation of Observer pattern

```
1  abstract class Subject{
2   List<Observer> observers;
3
4   void notify(){
5    for(Observer o: observers){
6     o.update;
7    }
8   }
9  }
10
11 abstract class Observer{
12  void update();
13 }
```

A common use for the Observer pattern is when one wish to have multiple views, representing the same model or data. For example, one can have a class named EXPENSES where a single instance represents a person's expenses

through the last 30 days. Such a class can be thought of as model, and will serve as the concrete subject in the observer pattern. In addition we have two classes named BarChart and PieChart which respectively renders a bar chart and a pie chart of the running expenses.. These are views of the model, and will be the concrete observers of the observer pattern. By using the observer pattern, one can make sure that the charts always are up to date, because any change made to the subject Expenses, will force the charts to re-render. A simple Java implementation is given in listing 2.6

Listing 2.6: Java example implementation of Observer pattern

```
class Expenses extends Subject {
 Map<String, Integer> expenseNameToCost; // State of
    ↪ subject

 Map<String, Integer> getState(){
  return expenseNameToCost;
 }
}

class PieChart extends Observer {
 Expenses exp;
 void update(){
  render();
 }
 void render(){
  Map<String, Integer> data = exp.getState()
  // Do something to render data into a pie chart
 }
}

class BarChart extends Observer {
 Expenses exp;
 void update(){
  render();
 }
 void render(){
  Map<String, Integer> data = exp.getState()
  // Do something to render data into a bar chart
 }
}
```

2.4 Summary

This chapter presents design patterns as general design solutions to reoccurring problems in software development. The GoF are introduced, along with their book GofBook which was responsible for popularizing design patterns. In GofBook a classification scheme, that classifies the purpose of a pattern to be either creational, structural or behavioral, is presented. This classification of patterns is used throughout this study. Lastly, the three design patterns Singleton, Composite and Observer are covered, along with simple Java implementations and proposed use cases in which the patterns can be applied.

Chapter 3

Programming Languages

In this part a very brief introduction to programming languages is given. In particular, concepts relevant in the investigation of the relationship between design pattern and programming languages are covered.

3.1 About Programming Languages

Programming languages are tools that allows the programmer to express meaning, behavior or structure in a form that is comprehensible to a computer. There exist many di□erent programming languages, of many di□erent types, intended to accomplish many di□erent things. Because programming languages di□er greatly, the design decisions will also do so, when using different languages. In the same manner that building a tree house with either rope, or hammer and nails would yield di□erent constructions, the use of di□erent programming languages will result in di□erent designs. Despite the fact that all programming languages are di□erent, there are certain language concepts and features shared between languages. These are similarities such as type systems, programming paradigm, method of compilation or execution, control flow mechanisms, modularity, level of abstraction, etc. The concepts we find most relevant in the context of design patterns, are programming paradigms and type systems.

3.2 Programming Paradigms

Programming paradigm is an abstract concept used to classify programming languages. It is an approach for describing how programming languages can express meaning. The most common programming paradigms include

object-oriented, functional and imperative. They are also the paradigms most relevant to this thesis.

Most popular programming languages are multi-paradigm languages, that mean that they are combinations of more than one programming paradigm. A language might be tightly associated with a single paradigm, but very rarely can it be said that a language is purely functional or object oriented. For example, JAVA, which mainly is known as an object oriented language, can also be considered an imperative language. And newer versions of Java even provide features associated with functional programming [27].

3.2.1 Object Oriented Programming

As implied by its name, the object oriented programming paradigms is focused mainly on the concept of objects [13, ch. 10.1]. These objects are used both to hold state, and to provide functions. That is, an object can store data, including other objects, and at the same time provide access to functions associated with that single object. The functions of an objects are allowed to mutate the state of its object.

Most popular object oriented-languages, e.g., JAVA are class based [32]. This means that objects are instantiated as an instance of a class, where the class defines the blueprint for all instances of said class. An alternative approach found in JAVASCRIPT is prototype based object-oriented. Prototype based languages have no notion of a class. Instead the objects themselves serves as blueprints for creating objects with similar properties [6]. Objects created this way have a pointer to the Prototype from which they are created. The Emerald programming language exemplifies another way of creating objects [24]. Instead of needing a class or a prototype as a blueprint for new objects, plain objects without direct association to any given type can be created.

3.2.2 Functional Programming

Functional programming is a programming paradigm that solves problems by linking one or more pure mathematical functions and avoiding state, mutability and side-effects. In object oriented programming, one often invokes functions in order to cause some change in the state of an object. In pure functional programming functions are invoked only to get some return value. Avoiding state and side-effects guarantees idempotency, i.e., that two invocations of the same function, will return the same value, given that the input parameters are identical. The same can not be said for object oriented programming, where two seemingly identical function calls can results in two different values. This is one of the desired traits of functional programming,

because it limits complexity and increases the ease of debugging. Another benefit of idempotency is that is allows for concurrent computing. Because there are no side-effects, the order of execution will not affect the results of any function call. Therefore the function calls can be executed concurrently.

While avoiding mutability and side-effect has many benefits, it also causes some limitations. Often mutability, or global state is necessary in real-life applications. Imagine a highly complex object or structure, which needs to be updated frequently. In object oriented programming, one could simply update part of the state that needs updating. Whereas in pure functional programming the entire structure, or chain of functions must me evaluated for each update. Therefore it is often infeasible to implement applications with pure functional programming. Instead multi-paradigm languages are used, and state and mutability is used warily where necessary.

3.2.3 Programming Paradigms and Design Patterns

The design patterns presented by GoF are mainly applicable to object-oriented programming. There do however exist many other design patterns applicable to other paradigms, e.g., functional programming [3]. Different programming paradigms have different limitations, and therefore result in different design patterns. However the intention is to mainly focus on patterns found in object oriented programming, but investigate them under multiple programming paradigms.

3.3 Type Systems

A type system is a mechanism for grouping elements that share some common properties, by associating each element with a grouping which is called a type [13, ch. 6]. A type can be appointed to a multitude of different programming constructs, i.e., variables, functions and expressions. Static typing and dynamic typing are two interrelated concepts in type systems, that are very relevant in the context of design patterns.

3.3.1 Static Typing

Static typing means that the type of all expressions, be it a variable, a function call or a arithmetic expression must be type checked during compilation [13, ch. 6.2]. A type error discovered at compile-time will usually make the compiler reject the program. A consequence of static type checking is that a type must be determined for all expressions during compilation. Static

typing also implies that the type of expressions, remains static, that is it can not change at run-time. While static typing limit flexibility to some extent, it allows type errors to be detected early. Therefore many parts of type checking can be omitted at run-time, which results in more e□cient executable code.

3.3.2 Dynamic Typing

The counterpart to static typing is dynamic typing. Dynamic typed languages will handle the type checking during run-time [13, ch. 6.2]. The type of some expression will be deducted to the type associated with its value, and not what it statically is declared to be. As a result, there is generally no need for the programmer to specify types. Another consequence is that types can change during execution. Therefore dynamic typing o□ers flexibility, but often at the cost of e□cient execution.

Often programming languages will incorporate a combination of both static and dynamic type checking, as they both have their benefits. For example, JAVA is mainly a static typed language; all variables and methods are declared to be of a certain type, and poorly typed programs will typically not compile. JAVA do however utilize some elements of dynamic typing to support features such as dynamic dispatch and down-casting [7].

3.3.3 Strong and Weak Type Systems

Programming languages can be considered strongly typed or weakly typed. Strong or weak typing denotes how strictly type security is enforced. A weakly typed language allows the developer to write code that strictly speaking should raise a type error. For example casting a variable to an incompatible type. Such operations are not possible in strongly typed languages.

3.4 Language Features

This section briefly defines a selection of language features relevant to this thesis.

3.4.1 Lambda Functions

Lambda functions are functions that can be encapsulated into objects and treated as data [31]. A lambda function, or a lambda expression can be treated as an entity that can be passed around to other functions or objects.

3.4.2 Higher Order Functions

Higher Order Functions denote functions that does at least one of the following:

1. Returns a function
2. Takes one or more functions as arguments

A typical use is taking several functions as arguments and returning a new function that is the composition of the argument functions. Lambda functions is a prerequisite for higher order functions.

3.4.3 Interface

Interfaces, in object oriented programming, is a language construct similar to classes and types. It is a specification of which functions an object must support. Classes can be declared to implement an interface, and must then support all the functions specified by the interface.

3.4.4 Abstract Class

An abstract class is a class that is not directly instantiatable. They are used as superclasses, that must be subclassed with concrete implementations.

3.4.5 Duck Typing

Duck typing is an approach to typing where type conformity is decided based on the presence of the correct functions and fields, instead of some their specified types [33]. That is, two objects are type compatible if they have the same functions and properties. This is in contrast to other type systems where to objects are type compatible if they are specified to be of the same type. A benefit of Duck typing is that types need not be specified explicitly.

3.5 Summary

In this chapter programming languages are presented as a tool that allows programmers to express meaning, behavior or structure in a form that is comprehensible to a computer. Followingly the two programming language concepts programming paradigm and type system are described.

Programming paradigm is an abstract concept in programming languages that describes how programming languages convey meaning. The object oriented programming paradigm, is centered around the concept of objects, that have state and side effects. The design patterns presented by the GoF are mainly intended to be used in object oriented languages. Functional programming is presented as another notable programming paradigm, where computation is achieved by linking one ore more function calls, and avoiding shared state, side effects and mutability.

Type systems are also introduced as a programming language concept, that associates elements with a property named type. Dynamic and static typing are presented as two typing schemes. In static typing type checking is completed during compilation, whereas dynamic typed languages are type checked at runtime.

Chapter 4

Language Presentation

This chapter presents the programming languages used in this project. For each of the chosen languages a brief overview is given. In addition to the brief overview, design features and language aspects particularly relevant to design patterns are covered. The process resulting in this selection of programming languages is addressed in chapter 7.

4.1 Java

JAVA is a multipurpose programming language. It was first released in 1996 and has since then been an industry leader in the world of programming languages. As of 2019 JAVA is the third most used programming language on the GitHub platform [28]. JAVA is traditionally considered an object oriented language, but draws inspiration from several paradigms, and thus should be known as a multi-paradigm language. It has a static and strong type system, uses a class based object model and is a compiled language.

4.1.1 Functional programming in Java

JAVA was originally an object oriented language, but has since the launch of Java 8 gained support for language features from functional programming [11, §15.27]. These feature include:

- Lambda functions
- Higher order functions
- Function composition: Combining several functions to create new ones.

The lambda expressions evaluates to objects [11, §15.27.4.]. Such an object is an instance of an anonymous class that implements the functional interface the lambda function conforms to, if one such exists. A functional interface in JAVA must contain exactly one abstract method. [11, §9.8].

4.2 Scala

SCALA is truly a multi-paradigm language. Its origin makes it particularly interesting for this study: The developers behind SCALA wished to unify and generalize object oriented and functional programming concepts in one language [18]. Thus, it provides both an unified object model, and higher order functions. By unified object model, it is meant that all objects inherits from a single base type, which in the case of SCALA is named *Any*. This is opposed to JAVA's object model which has di□erent base types for primitive types such as INT and referenced types that inherits from OBJECT. SCALA has a static and strong type system that also support type inference and is a compiled language.

4.3 JavaScript

JAVASCRIPT is known as the programming language of the web, and was in 2019 the most used programming language on GitHub [28]. In its earlier years, it was mainly used as a client-side language, thus rarely resulting in large and complex systems where design patterns are needed.[1] In later years however, JAVASCRIPT has many uses, including building large enterprise systems where design pattern are relevant. JAVASCRIPT is a multi-paradigm language combining functional, imperative and object oriented programming. Given its functional nature, JAVASCRIPT support higher order functions. It is weakly typed and uses dynamic type checking. Node.js version 10 is used to execute the JAVASCRIPT programs [15].

JAVASCRIPT is an implementation of ECMASript, which is a standardized language specification [34].

4.3.1 Prototype-based Object Orientation in JavaScript

As noted, JAVASCRIPT is an object oriented programming language. Its object orientation is based on prototypal inheritance, as opposed the the more

[1] Client-side languages can be defined as languages that are executed on the client side in a client-server relationship. The term is most commonly used to refer to languages being executed in a web browser.

typical class-based object orientation [8, p. 6]. With prototype-based object orientation, behavior inheritance is delegated to other objects, namely the prototypes. All non-literal JavaScript objects holds an implicit reference to the object that instantiated it, known as the prototype. When some property of an object is attempted accessed, the object will be examined for said property. If it is not found, the request is forwarded to the objects prototype. This process is repeated and the request will traverse the chain of prototypes, until the property is obtained or determined to be undefined in the whole of the chain.

4.3.2 JavaScript Closures

JavaScript creates its functions through closures. Closures is a model of how functions are defined and executed in relation to its enclosing environment. A closure can be described as a pair consisting of a function definition, and a pointer to the environment in which the function was declared. During execution of functions implemented as closures, variables from the environment pointer are semantically speaking copied into the function body, so they can be referenced directly from within the function. They will naturally be overshadowed by any variables by the same name inside the function body. In an execution model without closures, the only variables available inside a function are the ones defined inside the function body, in addition to the argument list.

4.4 Python

Python is a popular multi-paradigm programming language. It has a rich standard library and is used across a multitude of domains. As of 2019 it was the second most used language on GitHub [28]. Python supports object oriented, imperative and functional programming. It is a high-level interpreted language. Python has a dynamic type system which makes it particular interesting in light of Norvigs's postulation: Dynamic languages reduce the need for 16 of GoF's design patterns [16]. The findings of Norvig is further covered in section 5.1.

4.4.1 Duck Typing, interfaces and design patterns

Python uses Duck typing, as defined in chapter 3.4.5. Therefore, interfaces and abstract classes are somewhat redundant concepts in Python. Most of the design patterns in GofBook relies on either interfaces or abstract classes. It is therefore inevitable that those design pattern are changed in

Table 4.1: Taxonomy of Programming Languages

	Emerald	Java	JavaScript	Python	Scala
Object Oriented	Yes	Yes	Yes	Yes	Yes
Functional	No	Yes	Yes	Yes	Yes
Type Checking	Static	Static	Dynamic	Dynamic	Static
Type Safety	Strong	Strong	Weak	Weak	Strong
Inheritance Model	Class	Class	Prototype	Class	Class

PYTHON. Because Duck typing is used, the interfaces and abstract classes often can be completely disregarded. Instead the developer must ensure that the objects implement the appropriate functions.

4.5 Emerald

EMERALD is a distributed and object oriented programming language developed at the University of Washington in the 80s. It is strongly and statically typed. EMERALD is included in this project because of its interesting relationship to the Singleton design pattern: EMERALD uses object constructors as a replacement for classes, thus any object inherently is a singleton [5].

4.6 Summary

Table 4.1 summarises the content presented in this chapter. The chapter has presented JAVA, SCALA, JAVASCRIPT, PYTHON and EMERALD, that are the five programming languages used in this study. In addition to basic facts of the languages, languages features particular interesting in relation to design patterns are covered. All of the five languages support object oriented programming to some extent. JAVA, SCALA, JAVASCRIPT and PYTHON are considered multi-paradigm languages; as they also support functional programming to varying degrees. JAVA, SCALA and EMERALD are statically typed, whereas JAVASCRIPT and PYTHON are dynamically typed.

Chapter 5

Previous works

The purpose of this thesis is to investigate the relationship between design patterns and programming languages. This is not an entirely unexplored area of research. In this chapter relevant literature that exists on the subject is presented.

5.1 Design Patterns and Programming Languages

Design patterns is not an altogether uncontroversial topic. It has been demonstrated by several, that many of the design pattern presented by GoF, become obsolete or easier to implement in dynamic programming languages [12, 16]. Therefore it has been proposed that design patterns only exist as a replacement for missing language features [4].

In 1996, Petter Norvig gave a presentation on the topic of design patterns in dynamic programming languages [16]. He put forth a categorization scheme for specifying the level of support for a design pattern in a given programming language, and used this when analyzing the 23 patterns presented by GoF, in dynamic languages.

His conclusion was that 16 of the 23 become invisible or qualitatively easier to implement in the dynamic languages Lisp and Dylan, compared to C++, for some use of each pattern. He attributed this to six language features or details. Table 5.1 presents how many design patterns each of these language features simplified implementation of.

Norvig also made another observation: Many features of today's programming languages were once considered design patterns. For example, there is no formalized concept of subroutine or function call in assembly programming. Instead patterns existed to mimic the same behavior. In this case it

Language feature	Simplified patterns
First class types	6
First class functions	4
Macros	2
Method combination	2
Multimethods	1
Modules	1

Table 5.1: Language feature resulting in simplified implementation of pattern

is clear that certain patterns just are missing language features. A counter-argument would be that this might be true for some patterns, but that some patterns are impossible to implement as language features.

In 2002 Hannemann and Kiczales presented a study similar to that of Norvig [12]. They implemented the GoF patterns in JAVA and in the aspect-oriented programming language AspectJ, to compare implementations. 17 of the 23 GoF pattern implementations in AspectJ showed improvement, compared to the JAVA implementations. This strengthens the assumption that design patterns are a□ected by programming languages.

Recently three masters theses has also been done on the subject [2, 26, 30].

Alfredov conducted the first of the three studies, when he explored how programming languages a□ect design patterns [2]. His conclusions is that not only are design pattern implementations a□ected by programming languages, programming languages are also a□ected by design pattern, e.g., when a language adopts a pattern as a language feature. Thus he conclude that there exists a bi-directional relationship between programming languages and design patterns.

Sjølyst researched the impact of programming languages on design patterns[26]. He concludes that the main factor a□ecting a patterns applicability in a given language, is whether the language is dynamic or statically typed.

Webjørnsen observed the interaction between design patterns and programming languages, and concludes that there are a variety of elements that a□ect the implementation of a pattern in a given language [30]. Of these elements dynamic typing seemed to a□ect the implementation the most. He also noted that there seem to be a trend of patterns whose purpose is to mimic functional programming, for use in object oriented programming.

5.2 Summary

This chapter presents relevant previous works, on the subject of design patterns and programming languages. Two studies suggested that most of the GoF patterns become obsolete or easier to implement in dynamic programming languages [12, 16]. A masters thesis conclude that there is a bi-directional relationship between programming languages and design patterns, and two other masters theses suggest that dynamic typing is the language feature that seem to a□ect design patterns the most [2][26, 30].

Part III

Implementation and Classification

Chapter 6

Classification Scheme

It is not trivial to determine whether or not a programming language implements a given design pattern. In order to analyze the relationship between programming languages and design patterns in a meaningful way, it is helpful to have some scheme for classifying their relationship. This chapter introduces such a scheme.

6.1 Classification Scheme Overview

The purpose of this study is to determine which patterns are supported or implemented in various programming languages. Assessing whether or not a language supports a given pattern is not straight forward. It is often claimed that "Programming language X supports design pattern Y" [14, 25]. We find such claims to be lacking in nuance. Our research suggest that classifications of pattern support, exist on a spectrum. That is, there are other levels than supported and unsupported, which are the only levels proposed by a traditional binary model. Labeling the pattern-language relationships with the right level of support is difficult. In this chapter we put forth the different classifications of support and criteria for the different levels.

Figure 6.1 describes our classification scheme. We make distinctions between levels of support and classifications of support, so that the levels can be sorted into an hierarchy. Each level of support contains on or more classifications of support. Classifications of support have no hierarchical ordering amongst each other

We have identified 4 different classifications of pattern-language support, associated with 3 levels of support. More classifications and levels do in all likelihood exist, but we find that these present a practical model.

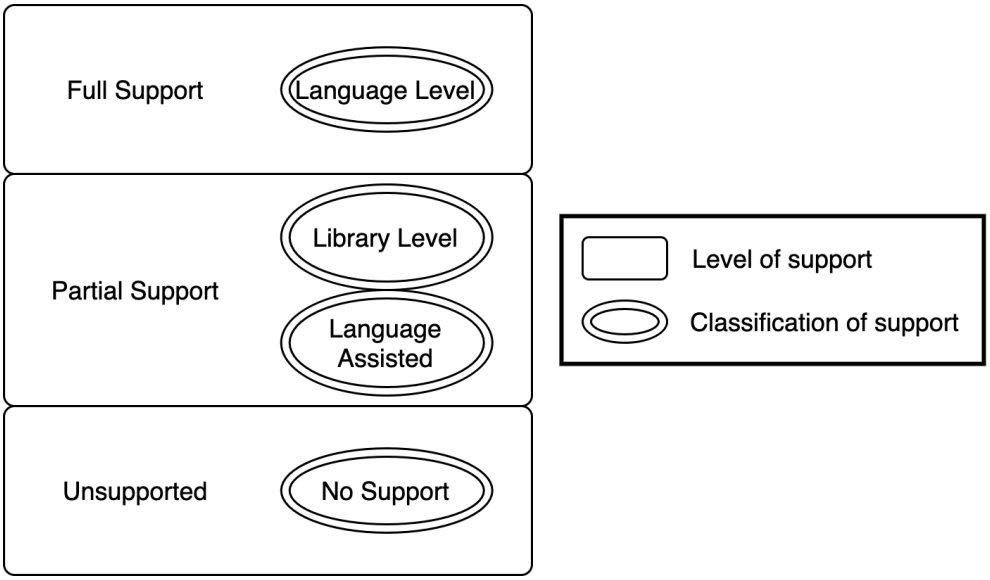

Figure 6.1: Classification Scheme for the level of support a programming language can show for a design pattern.

The three levels of support can be sorted into a hierarchy, ranging from FULL SUPPORT to UNSUPPORTED and PARTIAL SUPPORT in between them.

The levels FULL SUPPORT and UNSUPPORTED only consist of a single classification each. PARTIAL SUPPORT consist of two classifications.

A brief summary of each of the classifications of support follows.

LANGUAGE LEVEL SUPPORT denotes instances where the design pattern is fully supported by the language in question. This means that the pattern is made redundant.

LIBRARY LEVEL SUPPORT means that the design pattern is supported by some library implementation. It is not a language feature, but somebody has provided an implementation of the pattern.

LANGUAGE ASSISTED SUPPORT labels the relationships where some language features assists in achieving the same goal as the design pattern. A language feature might provide an improved implementation, or an alternative to the design pattern, but has not made the pattern completely redundant.

NO SUPPORT means that the pattern is not inherently supported in any way by the language.

6.2 Language Level Support

Language Level Support will designate the highest level of support a language can have for a design pattern. It implies that the design pattern in question is fully supported. That is, it is built into the language. Language Level Support is a catch-all classification for instances where a design pattern is made redundant. Language Level Support can be a consequence of a conscious design decision of implementing the pattern. A conscious design decision means that the language designers intentionally have attempted to support the design pattern, be it through some language mechanism, feature or the overall design of the language.

Language Level Support can also occur coincidentally, as a consequence of the nature of the programming language, e.g., typing scheme or object model. That is to say the motivation for a design pattern is non-existent in the given language. Thus making any implementation of the design pattern redundant.

An example of Language Level Support as a consequence of the nature of the programming language is the Singleton pattern in The Emerald Programming Language. As described in chapter 8.1, Emerald's object model renders the Singleton pattern redundant. It is not necessarily accurate to say that Emerald implements the Singleton pattern, but the need for Singleton implementations in Emerald is non-existent. Thus we say that Emerald provides Language Level Support for the Singleton pattern.

6.3 Library Level Support

Library Level Support is different from Language Level Support. It means that as a part of a language's standard library, or through some external package or framework, there exist some implementation or support for the design pattern.

The important distinction between Language Level Support and Library Level Support is that Library Level Support means concrete implementations made through code in the programming language in question. In a sense we can say that the design pattern is bootstrapped in the programming language. An example of Library Level Support for a design pattern is Java's iterator class for Collections:

```
1 ArrayList<String> list = new ArrayList<String>();
2 Iterator iterator = list.iterator();
```

This is merely an implementation of the Iterator pattern, that comes as part

of the JAVA Standard Library. If the library-provided implementation of the design pattern is general and extensive enough, one could argue that such instances exhibit FULL SUPPORT and LANGUAGE LEVEL SUPPORT for the design pattern. We have however chosen to classify such relationships as LIBRARY LEVEL SUPPORT, and associate the classification with PARTIAL SUPPORT.

6.4 Language Assisted Support

LANGUAGE ASSISTED SUPPORT designates the group of relationships where the language does not provide LANGUAGE LEVEL SUPPORT, but some language mechanism greatly reduces the complexity of implementing the design pattern, or partially alleviates the need for the pattern.

LANGUAGE ASSISTED SUPPORT differs from LANGUAGE LEVEL SUPPORT in that the motivation for the design pattern is not completely non-existent. It differs from LIBRARY LEVEL SUPPORT by the fact the the partial support can be attributed to the nature of the programming language, instead of some library feature.

An example of such a relationship is languages with lambda functions and the Strategy pattern, e.g., JAVASCRIPT in the follwoing example:

```
1 ...
2 const someStrategy = function(){
3     return "bar"
4 }
5 client.setStrategy(someStrategy)
```

In the traditional Strategy pattern, each strategy must be encapsulated by a class [10, p. 315]. In JAVASCRIPT functions are natively defined as objects, and can thus be passed to other objects without an encapsulating object. This solution is not powerful enough to cover all uses of the Strategy pattern, however the language feature utilized would simplify the full implementation significantly. A more detailed account of the Strategy pattern in context of lambda functions is given in chapter 8.2.

It seems natural that LANGUAGE ASSISTED SUPPORT is associated with PARTIAL SUPPORT. It is clear that languages attributed with this classification does not support the pattern in question fully. Likewise it is clear that the pattern is supported to a higher degree than NO SUPPORT. Thus PARTIAL SUPPORT seem reasonable.

6.5 No Support

No Support designates the relationships where the programming language provide no inherent support for the design pattern in question. There is no out-of-the box functionality attempting to implement or alleviate the need for the design pattern. No Support does not mean that the design pattern can not be implemented. It simply means that the programmer must implement the pattern themselves.

6.6 Summary

In this chapter a scheme for classifying the relationships between programming languages and design pattern is presented. We have identified four different classifications of support:

1. Language Level Support
2. Library Level Support
3. Language Assisted Support
4. No Support

These classifications are distributed amongst the three levels of support:

1. Full Support
2. Partial Support
3. Unsupported

Chapter 7

Language and Pattern Selection Process

The world of programming languages is large and ever evolving. Furthermore, new design patterns emerge frequently. Consequently, the process of deciding on which pattern-language relationships to analyse is challenging. This chapter describes that process and the selection criteria for the design patterns and programming languages used in this study.

7.1 Design Pattern Selection Criteria

Although design patterns have been prevalent in the world of software engineering since 1994, they have undergone no formal standardization. Thus the term *Design Pattern* is used in varying contexts.

For this study however, we are particular interested in design patterns applicable to object oriented languages. GoFBook is still an acknowledged authority on object oriented design patterns, and perhaps the closest thing we have to standardized design patterns. Therefore, the only criterion for inclusion of a pattern in this study, is that it is one of the 23 patterns covered in GoFBook. There exists plenty other object oriented patterns that could be interesting to this study. But as our scope is limited, such explorations are omitted. Additionally having an agreed upon interpretation of a pattern ensures that all included patterns have a proposed implementation for comparison. GoF had their own criteria for selecting patterns, and our patterns have therefore already gone through a round of selection.

During the research for this thesis all 23 patterns from GoFBook were studied. Some appeared to simply be workarounds for missing language

features, e.g., the Singleton. Whereas others seemed to be more language-agnostic, with no apparent improvement or replacement in light of modern programming languages. Given the capacity of this study, only a subset of the 23 patterns are included. Those are selected based on interesting findings in their relations to modern languages. For example, patterns that appear to be replaced or improved by language features are more likely to be included. Section 7.3 outlines the background for the inclusion of each specific pattern.

7.2 Programming Language Selection Criteria

The only fixed criterion for the programming languages included in this study is that object orientation is supported. The languages need not be purely object oriented, but must support it to some degree. In addition, our chosen languages should be a representative selection of modern object oriented programming.[1] For example, both dynamic and static type languages are included, as well as class-based and prototype-based languages. Further, it is preferred that overlap between similar languages is limited. Analysing a pattern in both JAVA and C#, which are similar in both typing scheme, supported paradigms and object model, might not yield particular interesting findings. Languages that seem particular relevant in relation to a design pattern, may be included despite not being considered modern; They can strengthen the argument that some design pattern indeed can be replaced by language features. Section 7.3 outlines why specific languages are chosen.

7.3 Proposed Relationships

This section briefly describes why each pattern in this study is included, and why specific languages are chosen for analysing said pattern. Each subsection can be viewed as a hypothesis on the relationship between pattern and language. The proposed classifications of support are based on the classification scheme from chapter 6.

7.3.1 Singleton

At a glance, the Singleton pattern seems to be redundant in EMERALD and SCALA: Their approach to object instantiation means that every object inherently is a singleton object. A JAVA implementation will also be provided

[1] By *modern* it is meant that the use of the language is common in today's world of software development.

for comparison. It is expected that Emerald and Scala shows LANGUAGE LEVEL SUPPORT, whereas Java should show NO SUPPORT.

7.3.2 Strategy

Norvig has proposed that the Strategy pattern can be improved or replaced by first-class functions [16]. By treating functions as data, they can be passed around freely, in similar fashion as strategy-objects in the Strategy pattern. JAVASCRIPT and PYTHON both support first class functions, and JAVA supports it to some extent. Thus, the Strategy pattern will be analysed in relation to these three languages. From Norvig it is also clear that the original pattern might be preferable in some instances, therefore we expect all three language to show LANGUAGE ASSISTED SUPPORT for the Strategy pattern.

7.3.3 Command

Upon investigation, the Command pattern seem to be similar to the Strategy pattern: They are both attempts at encapsulating executable code in referenceable objects. Consequently we expect first-class functions to also aid implementation of the Command pattern. JAVA and JAVASCRIPT are used in these experiments. We expect both of them to show LANGUAGE ASSISTED SUPPORT support for the pattern.

7.3.4 Decorator

At first sight, the Decorator pattern appears to be a way of dynamically attaching new properties to objects, in statically typed languages. It would therefore be interesting to analyse the pattern in a dynamic language, to see if this assumption holds true. JAVASCRIPT is dynamic, and hence a suitable candidate. Again, JAVA will be used for comparison. JavaScript is expected to show LANGUAGE LEVEL SUPPORT or LANGUAGE ASSISTED SUPPORT support, whereas Java should show NO SUPPORT.

7.3.5 Prototype

The Prototype pattern enables object creation based on prototypes. Some languages, e,g. JAVASCRIPT, have prototype based inheritance built-in, as described in chapter 4.3.1. Intuitively one would think prototypical inheritance eliminates the need for a Prototype design pattern. JAVA has a di□erent approach at implementing the pattern, through its CLONEABLE interface. As

we examine these techniques, we expect JavaScript to show Language Level Support, and Java to show Library Level Support for the Prototype pattern.

7.3.6 Iterator

It is often claimed that languages like Python and JavaScript have built in support for the Iterator pattern [14, 25]. While they clearly provide some useful mechanisms for iterating over standard library collections, it seems like only half of the work is done: Developers are still required to implement parts of the pattern. Java provides the Iterable interface, that requires developers to be explicit in their use of the pattern [22]. The Iterator pattern will be analysed in relation to these languages. We expect Python and JavaScript to show Language Assisted Support, and Java to show Library Level Support or No Support.

7.3.7 Proxy

The Proxy design pattern provides a placeholder object, to control access to a target object. JavaScript provides a built in object named Proxy that appears to have the same intent [8, p. 688]. It would therefore be interesting to explore the Proxy pattern, to see if such a built in implementation is possible. A Java implementation will be used for comparison, in addition to the implementation in JavaScript. Java is expected to show No Support for the pattern, whereas JavaScript should exhibit Library Level Support or Language Assisted Support.

7.4 Selection Process Summary

This chapter establishes the criteria used in the selection of design patterns and programming languages for this study. In sections 7.1 and 7.2 it is decided that the patterns should be selected from GofBook, and that the included languages all should support object orientation. Section 7.3 provide background for the inclusion of each specific pattern, and our assumptions for their relationships with selected languages.

Chapter 8

Implementations

In this chapter selected design patterns are implemented in various programming languages. Each section has the following structure:

1. A general introduction to selected design pattern
2. A generic implementation is proposed
3. An example use case is presented
4. Example use cased implemented in various languages
5. Our proposed classification scheme is applied to the implementations, in order to establish the languages support for the patterns

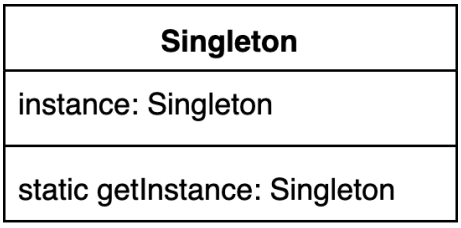

Figure 8.1: Singletom UML specification

8.1 Singleton

This section investigates the Singleton design pattern in relation to three di□erent programming languages: JAVA, EMERALD and SCALA. Firstly the section presents the pattern and the issue it attempts to resolve, followed by a generic implementation. Then a use case for the pattern is introduced, followed by three implementations of that use case. It will be shown that Singleton is a simple pattern, and that some languages provide LANGUAGE LEVEL SUPPORT for it, that is to say there is no need to implement it at all.

8.1.1 About the Singleton Pattern

Singleton is a creational design pattern [10, p. 127]. Its purpose is to ensure that there only is a single object instance of some class or type. In addition the pattern provides a global access point to the singleton object. In this section *Singleton* refers to the Singleton design pattern, whereas the uncapitalised *singleton* denotes objects or classes that are implemented in the Singleton design pattern.

In many cases, it is important that some class only has a single instance. For example in classes representing state or an abstract factory. To ensure that the single instance is used, the design patter makes multiple instantiations impossible. This is achieved by making the class responsible for its won instantiation.

The structure of the Singleton pattern is shown in in figure 8.1.

8.1.2 GoF Singleton Implementation

Listing 8.1: Generic Singleton Java implementation

```
1  class Singleton {
2    Singleton instance;
3    private Singleton(){}
4
5    public getInstance(){
6      if (instance == null){
7        instance = new Singleton();
8      }
9      return instance;
10   }
11   // Application specific code...
12 }
```

The implementation given in 8.1 is inspired by the standard C++ implementation in GoFBook. It limits the instantiation of new objects my making the constructor private, and storing the initialised object as a static variable. Therefore all access to the object from outside the singleton class happens through the getInstance-method.

8.1.3 Singleton Example Use Case

There are many use cases for the Singleton pattern. GoF uses Singletons for factory objects [10].[1] Another common use case for the pattern is to implement some logger as a Singleton, and this is the use case we will use in our investigation of the pattern [17].

A logger is an object that might be used for error handling, debugging or for inspecting the execution of some code.

A very basic logger should allow the user to pass strings, and the string should followingly be outputted to some appropriate destination, e.g., an output stream or a file.

However we do not want the logger object itself to handle the outputting of its entries. Instead the logger should provide access to its entries by providing an iterator.

A final requirement for the logger is that a single logger object should be readily available from any scope within the namespace of the project. By

[1] Factory objects are objects implementing the factory design patterns. See [10]

having this requirements, we only have to pass the logger object to our desired output once (e.g., in our main-method). We can think of our class hierarchy as a tree, with the class containing the main-method as the root-class. Then by the final requirement we can utilize the logger in any class, including the leafs in our tree, without passing the logger through numerous constructors. For example, we might unexpectedly discover that we need some logging in one of our leaf-classes, and instead of modifying the constructor of every class along the path from our root-class, we wish to simply get hold of the logger object used across the project.

Thus we have arrived at the following highly simplified requirements for a logger:

A logger should:

- receive strings and insert them into its internal data structure
- provide an iterator that allows an outside party to traverse its entries
- be readily available from any scope

In Java we might use the logger in the follwing manner:

Listing 8.2: Java snippet demonstrating how we wish to get hold of and use the logger object

```
1 // Get the logger object from any scope
2 Logger logger = Logger.getInstance();
3
4 // Pass a string to the logegr object
5 logger.log("This is a bug!");
```

Listing 8.3: Java snippet demonstrating how we wish to output the log entries

```
1 /* Get an iterator that traverses
2 the log entries and output to stdout */
3 Iterator iter = Logger.getInstance().getIterator();
4 while(iter.hasNext()){
5   System.out.println(iter.next());
6 }
```

8.1.4 Java Singleton Implementation

Listing 8.4 gives an implementation of our Singleton use case. The design pattern is manually implemented in accordance to GoFBook and ensures that there only can exist a single logger object, and that the sinlgeton object

is readily available from any code in the LoggingExample package. The implementation relies on static varriables and methods. Although the design pattern is easy to implement, there are no language aspects of Java that implicitly supports the Singleton pattern.

Listing 8.4: Java implementation of logger use case

```
1  package LoggingExample
2
3  import java.util.*;
4
5  class Logger {
6   static Logger instance;
7
8   List<String> entries;
9
10  private Logger(){
11   entries = new LinkedList<String>();
12  }
13
14  static Logger getInstance(){
15   if(instance == null){
16    instance = new Logger();
17   }
18   return instance;
19  }
20
21  void log(String entry){
22   entries.add(entry);
23  }
24
25  Iterator getIterator(){
26   return entries.iterator();
27  }
28
29 }
```

8.1.5 Emerald Singleton Implementation

An implementation of the logger use case in the Emerald Programming language is given in listing 8.5. By Emerald's Object Model, any object is inherently a singleton [24, p. 1]. There is no need to implement the Singleton pattern. The implementation already possess the same features as the

JAVA implementation in listing 8.4, and thus the Singleton pattern becomes superfluous in EMERALD. There is no need for the getInstance-method, as the logger object per default is a singleton.

Listing 8.5: Emerald implementation of the logger use case

```
export Logger

const Logger <- object Logger
 const entries <- Array.of[String].empty

 export operation log[entry: String]
  entries.addUpper[entry]
 end log

 export function getIterator[] -> [iterator: Iterator
    ↪ ]
  iterator <- Iterator.generateIterator[entries]
 end getIterator
end Logger
```

8.1.6 Scala Singleton Implementation

A third implementation of the logger example is given in listing 8.6. This SCALA implementation is similar to the EMERALD implementation, in that is does not explicitly implement the Singleton pattern. Because of SCALA's object model , every object created through the *object* keyword, is a singleton [29]. Thus it appears that the Singleton design pattern is superfluous in SCALA as well.

Listing 8.6: Scala implementation of logger use case

```
package LoggingExample

import scala.collection.mutable.ListBuffer

object Logger {
 var entries: ListBuffer[String] = ListBuffer()

 def log(entry: String): ListBuffer = entries +=
    ↪ entry
 def getIterator(): Iterator = entries.iterator()
}
```

8.1.7 Singleton Comparison and Classification

The three implementations provided in subsections 8.1.4, 8.1.5 and 8.1.6 will largely yield the same behavior. They will be used in the same way, and they all fulfill the requirements put forth in subsection 8.1.3. Because the EMERALD and SCALA implementations does not need to implement the Singleton Design pattern, they are more brief and concise. Additionally they might be easier to understand to the uninitiated. The JAVA implementation is more verbose and perhaps somewhat esoteric: The business logic of logging is intertwined with the design pattern implementation, which is sub optimal. As outlined in table 8.1, we classify JAVA's support for the Singleton pattern as NO SUPPORT, in accordance to the classification scheme presented in chapter 6. Although JAVA shows no inherent support for the Singleton design pattern, the pattern is still easy to implement and utilize in JAVA. Followingly, we determine that EMERALD and SCALA exhibits LANGUAGE LEVEL SUPPORT for the Singleton design pattern. Therefore it is clear the Singleton pattern can be integrated into an object oriented programming language. We attribute EMERALD and SCALA's support to their object models.

Table 8.1: Classification of support for the Singleton pattern

Programming Language	**Level of language support**
Java	No support
Emerald	Language Level
Scala	Language Level

8.1.8 Singleton Conclusion

In this section, the Singleton design pattern is presented. A generic implementation in JAVA is given and logging is presented as a common use case for the Singleton pattern. Then an implementation of the logger use case is given in the three programming languages JAVA, EMERALD and SCALA. These implementations allowed us to classify the relationship between the three languages and the Singleton pattern. JAVA is established to have NO SUPPORT for singletons whereas EMERALD and SCALA show LANGUAGE LEVEL SUPPORT for the pattern. This classification leads us to the conclusion that the Singleton pattern is integrateable with some object oriented programming languages.

8.2 Strategy

This section examines the Strategy design pattern and its relationship with the programming languages JAVA, PYTHON and JAVASCRIPT. In particular we are interested in whether functional language features, such as lambda functions or method references are suitable replacements for the Strategy pattern. A general presentation of the design pattern and its motivation is given, followed by a generic implementation. Then a use case for the Strategy pattern is introduced, along with implementation of that use case in each programming languages. Based on these implementations we will categorize the relationship between the pattern and said languages, and it is shown that several languages exhibit LANGUAGE ASSISTED SUPPORT for the Strategy design pattern.

8.2.1 About the Strategy Pattern

Strategy is a behavioral design pattern. Its intent it to define a family of related algorithms, behaviors or strategies, and make them interchangeable with one another [10, p. 315].

Often a single task can be performed in many di□erent ways, and which method we determine to be best suited to solve the task may vary. For example, we know of many di□erent algorithms for sorting numbers. The choice of algorithm will depend on what we optimize for. It may be speed, space or some other factor. Therefore we wish to be able to change which algorithm to use freely. This can be achieved by decoupling the algorithm and client, and encapsulating each algorithm into objects. Clients denote the objects that use the algorithms.

By making the algorithm loosely coupled, i.e., partially independent, to its client, we can vary which algorithm to use and decide on the appropriate one at runtime. Varying the algorithm could also be achieved by hard-wiring all algorithms into the client-class, and choosing the appropriate one through if-statements, but this would not allow for the same code-reuse. In stead, the Strategy pattern allows for sharing algorithms horizontally in a class hierarchy. Additionally we wish to simply be able to execute the algorithm associated with the client, without taking into account exactly which algorithm that is. In short we want to separate the concern of choosing algorithm and executing it.

In this section we use the terms *strategy*, *behavior* and *algorithm* interchangeably.

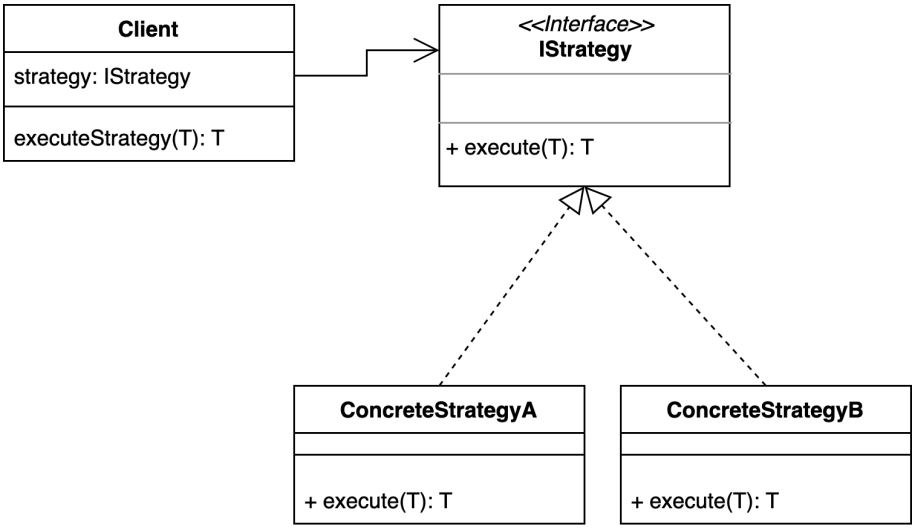

Figure 8.2: Strategy UML specification

8.2.2 GoF Strategy implementation

Figure 8.2 outlines the structure of GoF's proposed Strategy implementation. Instead of implementing the strategies, the client holds a reference to a strategy implementation. When the client wish to execute its strategy, it simply invokes the EXECUTE-method of its STRATEGY-instance.

Before we provide an implementation of the Strategy design pattern, a program modelling the same system without using the patter, is provided in listing 8.7. We have a class named CLIENT that wish to execute some algorithm. There exists two di□erent algorithms that may be appropriate, namely COMMONOPERATIONA and COMMONOPERATIONB. Which algorithm to use depends on some internal state of the Client-instance.

By applying the Strategy design pattern, we get the implementation provided in listing 8.8. It achieves the decoupling by creating an interface which each concrete algorithm must implement. The client has a reference to an object of the STRATEGY-type, and will invoke the reference's COMMONOPERATION-method to execute the algorithm. The strategy-variable must be assigned either in the constructor of the client, or with a designated method SETSTRATEGY as showed in our implementation. The variable can followingly be re-assigned to any object that implements the STARTEGY-interface at runtime.

In this simplistic example, the Strategy pattern might seem redundant, but if we were to add another 5 algorithms to the CLIENT-class it would become clear that the Strategy pattern serves a purpose: The if-statements for deter-

mining the right algorithm could seem obscure, and all the different methods would clutter up the CLIENT-class. By encapsulating each algorithm into classes and keeping a reference to the correct one, we have separated the act of choosing the algorithm and executing it.

Listing 8.7: Client implementation without the Strategy design pattern

```
class Client {

 String operation(){
  if (/* some condition related to Client-object's
     ↪ state  */){
   return commonOperationA();
  } else {
   return commonOperationB();
  }
 }

 String commonOperationA(){
  return "foo";
 }

 String commonOperationB(){
  return "bar";
 }

}
```

Listing 8.8: Generic Strategy Java implementation

```
class Client {

 Strategy strategy;

 String operation(){
  return strategy.commonOperation();
 }

 void setStrategy(Strategy strategy){
  this.strategy = strategy;
 }

}

interface Strategy {
```

```
16   String commonOperation();
17  }
18
19  class ConcreteStrategyA implements Strategy {
20   public String commonOperation(){
21    return "foo";
22   }
23  }
24
25  class ConcreteStrategyB implements Strategy {
26   public String commonOperation(){
27    return "bar";
28   }
29  }
```

8.2.3 Strategy Example Use case

The Strategy design pattern can be applied to many use cases. For our exploration of the pattern, we will create a mock ticket-system for the subway. This ticket system should have a method named CALCULATEPRICE, which will calculate the price for the customer using our system. For this system, there exist two pricing models: One for rush hour tickets, and one for normal tickets. In the rush hour pricing model, all tickets cost NOK 50. In the non-rush hour model, senior tickets cost NOK 20 and normal tickets cost NOK 30. The price models is illustrated in table 8.2 The ticket system will be the Client in our implementation of the Strategy pattern, and the pricing models will be the strategies.

Table 8.2: Price models for subway ticket system

	Normal Customer	**Senior Customer**
Normal Hours	NOK 30	NOK 20
Rush Hour	NOK 50	NOK 50

8.2.4 Java Strategy implementations

Listing 8.9 provides a JAVA implementation of the Ticket System in accordance to the structure proposed by GOFBOOK. Here we chose to set the correct strategy in the constructor of the client. It could also be set through a custom method. Which of the two we prefer is domain dependent.

Listing 8.9: Ticket System Java implementation

```
class TicketClient {

 Customer customer;
 Time time;
 PriceStrategy priceModel;

 TicketClient(Time time, Customer customer){

  this.time = time;
  this.customer = customer;

  if(time.isRushHour){
   priceModel = new RushHourStrategy();
  } else {
   priceModel = new NormalStrategy();
  }
 }

 int calculatePrice(){
  return priceModel.calculate(this);
 }
}

interface PriceStrategy {
 int calculate(TicketClient client);
}

class RushHourStrategy implements PriceStrategy {
 public int calculate(TicketClient client){
  return 50;
 }
}

class NormalStrategy implements PriceStrategy {
 public int calculate(TicketClient client){
  if(client.customer.isSenior){
   return 20;
  }
  return 30;
 }
}
```

As noted in the language chapter 4.1.1, JAVA 8 supports lambda functions. When a lambda expression is evaluated it produces a reference to an object. That object is an instance of a class that implements the functional interface the lambda function conforms to, if one such exists. A functional interface in JAVA is an interface that has a single abstract method. That is, given an interface with exactly one abstract method, e.g., the PRICESTRATEGY-interface from listing 8.9, a lambda function that has the same return- and parameter-types as the interface's single abstract method, will evaluate to an object of a class that implements that interface.

Listing 8.10 demonstrates how this type of object creation can be used: We assign the variable NORMALSTRATEGY to a lambda function. Because the lambda function types conforms to the abstract CALCULATE-method of the PRICESTRATEGY-interface, the lambda expression is an object that implements the PRICESTRATEGY-interface.

```
1 PriceStrategy normalStrategy = (Client client) -> {
2     if(client.customer.isSenior){
3         return 20;
4     }
5     return 30;
6 };
```

Listing 8.10: Java snippet demonstrating lambda expression of type PriceStrategy

This allows us to create encapsulated and interchangeable strategies, without cluttering our class-hierarchy. We can place these PRICESTRATEGY-variables inside the interface itself, so that we still achieves the desired decoupling of client and strategy implementation.

Using this new approach we get the complete implementation provided in listing 8.11. It provides the exact same functionality as listing 8.9, but in a more concise and understandable manner. The class definitions of the concrete strategies disappear, and we are left with only the function-bodies.

Listing 8.11: Improved Ticket System Java implementation using lambda functions

```
1 class Client {
2
3  Customer customer;
4  Time time;
5  PriceStrategy priceModel;
6
7  Client(Time time, Customer customer){
8
```

```
9     this.time = time;
10    this.customer = customer;
11
12    if(time.isRushHour){
13     priceModel = PriceStrategy.rushHourStrategy;
14    }
15     else {
16     priceModel = PriceStrategy.normalStrategy;
17    }
18   }
19
20   int calculatePrice(){
21    return priceModel.calculate(this);
22   }
23  }
24
25  interface PriceStrategy {
26   int calculate(Client client);
27
28   PriceStrategy normalStrategy = (Client client) -> {
29      if (client.customer.isSenior) return 20;
30      return 30;
31    };
32
33   PriceStrategy rushHourStrategy = (Client client) ->
       ↪ 50;
34
35  }
```

There are two main downsides to this improved approach: Although lambda functions theoretically can mimic persistent state, it is not as straight forward as in a complete class definition. This issue can however often be avoided by keeping all state in the client-object. The second downside is the instances where we wish the strategy-interface to have multiple related abstract methods.

8.2.5 JavaScript Strategy Implementation

JavaScript supports lambda functions and assigning functions to variables. This can be utilized to implement our improved Strategy pattern. Listing 8.12 provides this new implementation. JavaScript is weakly- and dynamic typed, and uses duck-typing [34]. Therefore there is no notion of interface. Instead the developer is responsible for implementing the correct

functions and defining the necessary fields. This results in a less verbose implementation. However, the JavaScript implementation is less type-safe, and type-errors will result in a runtime-error.

Listing 8.12: JavaScript implementation of TicketSystem

```
1
2  const normalStrategy = client => {
3   if (client.customer.isSenior) {
4    return 20
5   }
6   return 30
7  }
8
9  const rushourStrategy = client => 50
10
11 class TicketClient {
12  constructor(time, customer){
13   this.time = time
14   this.customer = customer
15   if(time.isrushHour){
16    this.priceStrategy = rushourStrategy
17   } else{
18    this.priceStrategy = normalStrategy
19   }
20
21   this.calculatePrice = () => this.priceStrategy(this
       ↪ )
22
23  }
24 }
```

8.2.6 Python Strategy implementation

Python also supports assignment of functions to variables, thus the modified Strategy pattern showcased in sections 8.2.4 and 8.2.5 can also be implemented in Python and is presented in listing 8.13. Python is dynamic and strongly typed, and also utilizes duck typing [35]. Therefore there is no need for a PriceStrategy-interface.

Listing 8.13: Python implementation of Ticket System

```
1
2  def normalStrategy(self, client):
3   if client.customer.isSenior:
4    return 20
5   return 30
6
7  def rushhourStrategy(self, client):
8   return 50
9
10 class TicketClient:
11  def __init__(self, time, customer):
12   self.time = time
13   self.customer = customer
14
15   if time.isRushhour:
16    self.priceStrategy = rushhourStrategy
17   else:
18    self.priceStrategy = normalStrategy
19
20  def calculatePrice(self):
21   return self.priceStrategy(self, self)
```

8.2.7 Strategy Comparison and Classification

The modified JAVA implementation from listing 8.11 will mostly result in the same functionality as the JAVASCRIPT and PYTHON implementation from listings 8.12 and 8.13. They can be used in the same way, and all implement the use case presented in section 8.2.3 in a satisfyingly manner. These implementations are more concise and are less cluttered than the original Strategy pattern from listing 8.9. However, they have not made the complete Strategy pattern superfluous. They do not achieve the same level of encapsulation, thus making state-management less eloquent. For example, for complex strategies consisting of many lines of code, it would be useful to encapsulate that strategy into a class-object defined in its own file. For simpler strategies, such as our use case, our modified implementations is appropriate. Therefore we classify the JAVA, JAVASCRIPT and PYTHON's support for the Strategy design pattern as LANGUAGE ASSISTED SUPPORT in accordance to the classification scheme put forth in chapter 6. This classification is outlined in table 8.3.

Table 8.3: Classification of support for the Strategy pattern

Programming Language	Level of language support
Java	Language Assisted
JavaScript	Language Assisted
Python	Language Assisted

8.2.8 Strategy Conclusion

In this section, the Strategy design pattern is presented. Strategy's purpose is to define related algorithms and let the client vary the algorithm freely. This has traditionally been achieved by encapsulating each strategy into class-objects that implement a common interface. A generic implementation of the pattern is provided and we present a use case that the pattern can be applied to. Implementations of the use case are provided in JAVA, JAVASCRIPT and PYTHON. Language features such as lambda functions and assignment of functions to variables allows for modified implementations that are more concise. And for many use cases, we believe that the techniques used in the modified implementations is a convenient alternative to the Strategy design pattern. For use cases with complex strategies or algorithms, the Strategy pattern as presented by GoF is often more suitable. Thus we have established that JAVA, JAVASCRIPT and PYTHON all exhibits LANGUAGE ASSISTED SUPPORT for the Strategy pattern.

8.3 Command

The Command design pattern is a behavioral pattern whose purpose is to encapsulate requests or *commands* [10, p. 233]. This section gives a brief overview of the pattern, presents an example use case in the form of a Graphical User Interface (GUI) framework, and demonstrates how the pattern can be improved in light of JAVA's lambda functions, and JAVASCRIPT closures.

8.3.1 About the Command Pattern

The Command pattern let programmers encapsulate requests and commands. Encapsulating commands in objects allow for several benefits:

- Commands are loosely coupled from their invokers
- Command-objects can be passed around and assigned to the appropriate executors
- Command-objects can be enqueued in a suitable data structure that can be programmatically manipulated
- Multiple command-objects can be composed to a single command
- Support for non-implemented operations

During development of some system it is often unbeknownst to the programmers what action some component should trigger. GOFBOOK uses a GUI framework as an example: In the development of a GUI framework, the developer might create a generic button component. It can however not be determined what specific action the button should trigger. The programmers must defer this decision to the users of the framework. So instead of implementing some concrete action to be performed, only the format of the action is agreed upon, namely a command-object.

The Command pattern is somewhat similar to the Strategy Pattern presented in section 8.2. Both of them encapsulate code to be executed into objects. They di□er however in their intent: Whereas the Strategy pattern concern *how* something should be done, the Command pattern let programmers specify *what* should be done.

8.3.2 GoF Command Pattern Implementation

Figure 8.3 describes the structure of GoF's proposed implementation of the Command pattern[10]. In the figure, INVOKER designates the objects that

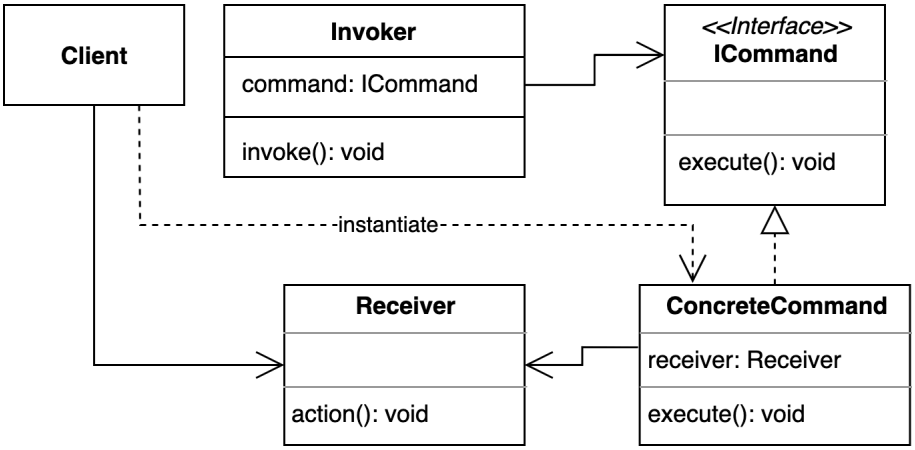

Figure 8.3: Command UML specification

might trigger a command. For example, in the instance of the GUI framework desribed in section 8.3.1, a button would be an INVOKER. Invokers hold a reference to an implementation of the ICOMMAND interface. When the invoke method of the INVOKER is called, it will call the execute method of its command object. The command implementation will typically hold a reference to a RECEIVER. A RECEIVER is the object the command is targeted at. The client is the object that stitches all of this figure together. It will implement or instantiate a command with the appropriate receiver.

The crux of the pattern is that the INVOKER is loosely coupled with its concrete command. Because of this design, the INVOKER class and ICOMMAND interface can, and often are, developed separately from the rest of the system. In the case of the GUI framework, only INVOKER and ICOMMAND would be implemented by the framework developers. The CLIENT, RECEIVER and CONCRETECOMMAND would instead be implemented by the users of the framework.

Listing 8.14 gives a generic implementation of the pattern in accordance to figure 8.3.

Listing 8.14: Generic GoF Java Command implementation

```
1 class Client {
2  public static void main(String[] args) {
3   Receiver receiver = new Receiver();
4   ICommand command = new ConcreteCommand(receiver);
5   Invoker invoker = new Invoker(command);
6  }
7 }
8
```

```
class Invoker {
 ICommand command;

 Invoker(ICommand command){
  this.command = command;
 }

 void invoke(){
  command.execute();
 }
}

class Receiver {
 void action(){
  // Do some action
 }
}

interface ICommand {
 void execute();
}

class ConcreteCommand implements ICommand {
 Receiver receiver;

 ConcreteCommand(Receiver receiver){
  this.receiver = receiver;
 }

 public void execute(){
  receiver.action();
 }
}
```

8.3.3 Command Pattern Example Use Case

The creation of GUI frameworks was mentioned as an applicable use case for the Command pattern. We will carry along with this example, and imagine that we have a very simple GUI framework for creating desktop applications.

The framework consists of two types out-of-the-box components: A blank canvas and buttons. In addition, the framework users can create their own visual components through the framework API. The following snippet demon-

strates how we wish to use the framework: It creates a canvas, a button, and places the button on the canvas, before it draws the canvas:

Listing 8.15: Usage of GUI framework

```
import UIFramework;
...
 Canvas screen = new Canvas(...);
 Button button = new Button(...);
 screen.placeComponent(button, ...);
 screen.draw();
```

8.3.4 Command Pattern Java Implementations

When applying the Command pattern to the application, we must identify the INVOKER, RECEIVER and CLIENT. The BUTTON class is the INVOKER. When the button is pushed, it should execute some command. For our example use of the framework, we want the canvas to close when the button is pushed. Thus the CANVAS class is the RECEIVER, as it is the target of the command. We also create a class named FRAMEWORKCLIENT which is responsible for orchestrating the application, and simulates the Button being pressed. An interface CLICKCOMMAND and an implentation of that interface CLOSECANVASCOMMAND is also needed. The CLOSECANVASCOMMAND will be instantiated with the CANVAS as its RECEIVER, and the BUTTON instance will have a CLOSECANVASCOMMAND as its CONCRETECOMMAND. A simple implementation of the system is given in listing 8.16.

Listing 8.16: GUI Framework Java Implementation

```
class FrameworkClient {
 public static void main(String[] args) {

  Canvas screen = new Canvas();
  CloseCanvasCommand command = new CloseCanvasCommand
     ↪ (screen);
  Button button = new Button(command);
  screen.placeComponent(button);

  // Simulate button being pushed
  button.click();
 }
}

class Canvas {
 void closeCanvas(){
```

```
16    System.out.println("Canvas was closed.");
17   }
18
19   void placeComponent(GraphicalComponent component){
       ↪ /*...*/}
20
21  }
22
23  abstract class GraphicalComponent { /*...*/ }
24
25  class Button extends GraphicalComponent {
26   ClickCommand command;
27
28   Button(ClickCommand command){
29    this.command = command;
30   }
31
32   void click(){
33    command.execute();
34   }
35
36  }
37
38  interface ClickCommand {
39   void execute();
40  }
41
42  class CloseCanvasCommand implements ClickCommand {
43   Canvas canvas;
44
45   CloseCanvasCommand(Canvas canvas){
46    this.canvas = canvas;
47   }
48
49   public void execute(){
50    canvas.closeCanvas();
51   }
52  }
```

In their description of the Command pattern, GoF mentions that the pattern is an object oriented replacement for callbacks [10, p. 235].[2] Since then,

[2] Callbacks can be defined as code passed as an argument to a function. The argument is executed upon completion of the function.

several object oriented languages have gained support for callbacks or similar behavior. As described in section 4.1.1, JAVA 8 supports lambda functions that can be passed as arguments to methods. This results in the simplified CLIENT implementation found in listing 8.17. The BUTTON class, CANVAS class and the CLICKCOMMAND interface are unchanged. The double-colon syntax at line 4 is method-referencing. It is semantically the same as a lambda function calling the CLOSECANVAS method, as showed on line 7. The lambda function conforms to the functional interface CLICKCOMMAND. Therefore the CLOSECANVASCOMMAND class can be omitted, and we have simplified our class hierarchy.

Listing 8.17: Simplified GUI Framework Java Implementation

```
1  class SimplifiedClient {
2   public static void main(String[] args) {
3
4    Canvas screen = new Canvas();
5    Button button = new Button(screen::closeCanvas);
6    // Alternatively:
7    // Button button = new Button(() -> screen.
        ↪ closeCanvas());
8    screen.placeComponent(button);
9
10   button.click();
11  }
12 }
```

This technique for simplifying our class hierarchy, and overall application complexity, is largely the same as the improved Strategy JAVA implementation in section 8.2.4. It also has the same caveats: The command can only contain a single method. The Command pattern is often used to support two related request: An action, and an undo of the same action. This is trivial to implement if our command is a class defined object. We would simply have the two methods *execute* and *undo*. Because JAVA's functional interfaces only can contain a single abstract method, implementing the command object through a single lambda is impossible.

8.3.5 Command Pattern JavaScript Alternative

JAVASCRIPT has always supported callbacks. Additionally it is dynamically typed. Therefore our improved Command pattern is trivial to implement in JAVASCRIPT. Listing 8.18 provides an implementation similar to our JAVA implementations. The FRAMEWORKCLIENT class is strictly speaking not

necessary, but it encapsulates the framework-instantiation in a useful manner. Because JavaScript uses duck typing, interfaces like the ICommand in listing 8.16 are superfluous.

Listing 8.18: GUI Framework JavaScript Implementation

```
class FrameworkClient {
 constructor(){
  const canvas = new Canvas()
  const closeCanvasCommand = () => canvas.closeCanvas
     ↪ ()
  const button = new Button(closeCanvasCommand)
  canvas.placeComponent(button)
  button.click()
 }
}

class Canvas {
 placeComponent(graphicalComponent){ /*...*/}

 closeCanvas(){
  console.log("Canvas was closed.")
 }

}

class Button {
 constructor(clickCommand){
  this.clickCommand = clickCommand
 }

 click(){
  this.clickCommand()
 }
}

const client = new FrameworkClient()
```

Stateful command objects using JavaScript closures

Another caveat of our simplified command implementation is the command objects lack of state persistence. The Command pattern as defined by GoF allows for free distribution of the command objects. That is, the command objects can be passed around to several clients. One might imagine a system

where the command objects are passed through and processed by a complete pipeline of clients. In such an instance, the command object will often need to hold state, e.g., in the form of a status code that provide other clients necessary information on how to process the command. The command objects in listings 8.17 and 8.18 will not persist state between each invocation. One could reference a stateful object in the command function-bodies, and use said object for state management, but that would not guarantee exclusive access through the command object, nor would it be particularly eloquent.

Closures, as desbribed in chapter 4.3.2, provide a better alternative: Closures encapsulates functions, along with their surrounding environment. This is exploited in the generic Command pattern implementation in listing 8.19. By nesting a function, inside another, we can have a stateful encapsulated command object. By invoking the COMMANDCREATOR function defined at line 27, a new function is returned. This new function can be passed around for execution, and will persist state, e.g., the COUNTER and STATE variables at line 28 and 29, between executions. In fact, the returned function will be the only point of access to said variables. Thus we have arrived at an alternative to GoF's Command pattern, that does support state persistence.

Listing 8.19: Generic JavaScript Command Implementation using Closures

```
1  class Invoker {
2   constructor(command){
3    this.command = command
4   }
5
6   invoke(){
7    this.command()
8   }
9  }
10
11 class Receiver {
12  constructor(initialState){
13   this.state = initialState
14  }
15
16  setState(newState){
17   this.state = newState
18  }
19
20  printState(){
21   console.log(this.state)
22  }
23 }
```

```
24
25 const receiver = new Receiver({})
26
27 const commandCreator = function(){
28  let counter = 0
29  const state = { counter: counter }
30
31  return function(){
32   receiver.setState(state)
33   counter++
34   state.counter = counter
35  }
36 }
37
38 const invoker = new Invoker(commandCreator())
39
40 receiver.printState() // prints "{}"
41 invoker.invoke()
42 receiver.printState() // prints "{ counter: 1 }"
43 invoker.invoke()
44 receiver.printState() // prints "{ counter: 2 }"
```

8.3.6 Command Comparison and Classification

In this section, five di□erent implementations of the Command design pattern has been presented: The two generic implementations are the GoF inspired JAVA implementation in listing 8.14 and our alternative JAVASCRIPT implementation using closures in listing 8.19. They are both appropriate for various use cases. The traditional GoF implementation allows for multiple related methods, and easy state management. The JAVASCRIPT alternative is more concise, and doesn't complicate the class hierarchy.

Three implementations of the GUI Framework example were also presented in listings 8.16, 8.17 and 8.18. Because they all achieve the desired functionality, it is clear that our simplified implementation using lambda functions instead of class-based objects, is a suitable replacement for the GoF Command pattern in some instances.

Support for multiple related methods, in the form of *execute* and *undo*, is explicitly mentioned as a feature in GoF's description of the Command pattern. As this feature is not supported in our alternative Command implementations, the traditional Command pattern as presented by GoF is clearly not made superfluous. There are however many use cases where

Table 8.4: Classification of support for the Command pattern

Programming Language	Level of Language Support
Java	Language Assisted
JavaScript	Language Assisted

our simplified implementations are preferable. Therefore we determine both JAVA and JAVASCRIPT to exhibit LANGUAGE ASSISTED SUPPORT support for the Command pattern. This classification is outlined in table 8.4. We establish JAVA's LANGUAGE ASSISTED SUPPORT to be caused by its support for Lambda functions and functional interfaces, that allows us to pass functions as data. JAVASCRIPT's LANGUAGE ASSISTED SUPPORT support can be attributed to its dynamic typing scheme, which nullifies the need for a Command interface, and it's support for closures which facilitates function encapsulation.

8.3.7 Command Conclusion

This section presents the Command design pattern. The Command pattern allows for encapsulation of request or commands, so they can be delegated to the appropriate clients, or composed into new commands. Two generic implementations of the pattern are provided. Additionally development of a Graphical User Interface Framework is presented as an example use case for the pattern, and three implementations of that example is given. A simplification in JAVA using lambda functions instead of class-based object is proposed. Another alternative in JAVASCRIPT allowed us to create stateful command objects, through closures. Both of these alternatives only partly replace the traditional Command pattern. For commands or request that need multiple methods, or complex state management, the pattern as proposed by GoF is still needed. For simpler use cases, our alternative implementations are more concise and reasonable. Therefore we have arrived at the conclusion that JAVA and JAVASCRIPT provide LANGUAGE ASSISTED SUPPORT for the Command pattern.

8.4 Decorator

Decorator is a structural design pattern. It allows for altering of behavior and properties of object instances at runtime. The following section gives a presentation of the pattern, introduces an example use case in the form of text editor creation, and provides implementations of the use case in JAVA and JAVASCRIPT. It is shown that dynamic typing somewhat replaces the pattern in JAVASCRIPT.

8.4.1 About the Decorator Pattern

The typical approach for adding or altering features to existent classes or objects is inheritance. There are however instances were such an approach would be infeasible or sub optimal: Creating subclasses for every combination of features will often yield an enormous and impractical class hierarchy. And there are situations were we wish to attach features dynamically at runtime, without altering all objects of that class. The Decorator design pattern addresses this by using object composition instead of inheritance. That is, instead of creating a subclass for all possible cases, the client holds references to new objects, responsible for some special behavior.

Imagine we were to build a text editor: Then we might model each word as an object. The WORD class would be a subclass of a GRAPHICALELEMENT class. We later decide that it should be possible to make words appear in italic text. The inheritance based approach would be to create a new class named ITALICWORD which is a subclass of WORD. But if the editor also were to support bold, underlined and colorized text, the issue becomes apparent: ITALICBOLDUNDERLINEDYELLOWWORD would be one of many new strange classes. The Decorator pattern's approach is to attach, or *decorate* the original WORD object with new objects with the desired properties. A WORD object would be wrapped by an ITALICDECORATOR object, which again could be wrapped by an BOLDDECORATOR object.

8.4.2 GoF Decorator Implementation

The structure of the Decorator pattern as proposed by GoF is shown in figure 8.4. The objects we wish to attach new properties to, must be a subclass of some abstract class COMPONENT. This is needed so that components and decorators can share a common interface. Another abstract class DECORATOR, which is the superclass of decorators, extends the COMPONENT abstract class. In addition to inheriting from the COMPONENT class, DECORATOR also holds a reference to an inner COMPONENT object, which is the

object being decorated. Because DECORATOR is both a COMPONENT object, and holds a reference to a COMPONENT object, the decorator can be transparent to its clients: The DECORATOR objects will take care of the added properties, and then defer all other requests to its inner component. The CONCRETEDECORATOR class implements the concrete decoration behavior.

In the text editor example presented in section 8.4.1, the COMPONENT class would correspond to the GRAPHICALELEMENT class. The WORD class would correspond to the CONCRETECOMPONENT class. ITALICDECORATOR and BOLDDECORATOR would correspond to CONCRETEDECORATOR and would extend a DECORATOR class.

A generic JAVA implementation of the pattern in accordance to figure 8.4 is provided in listing 8.20. In this implementation the DECORATOR and CONCRETEDECORATOR class introduce no new methods, but only override the existing COMMONOPERATION method. It would also be possible to add new methods in the decorators, but these would only be visible to clients aware of the decorators.

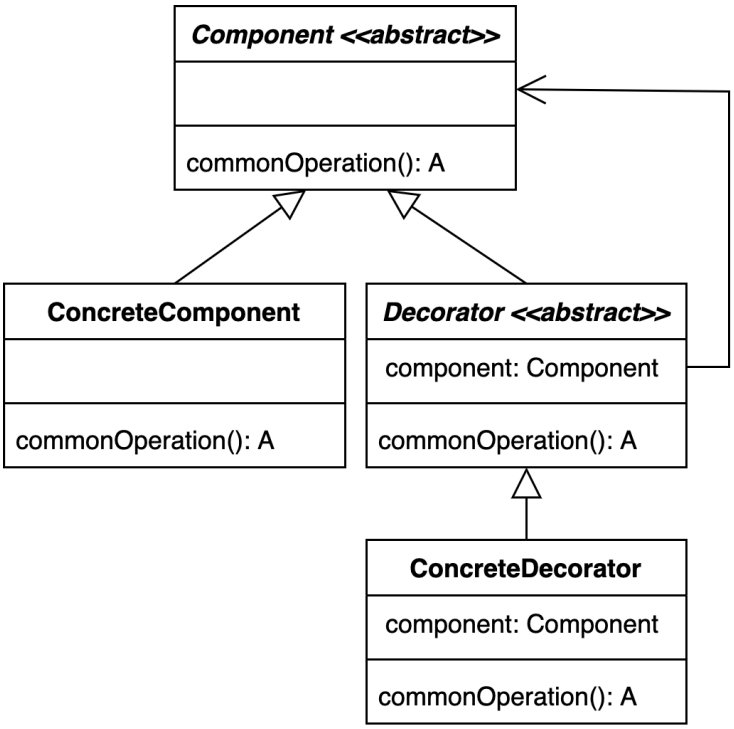

Figure 8.4: Decorator UML specification

Listing 8.20: Java generic Decorator pattern implementation

```
1  abstract class Component {
2   void commonOperation(){ /* ... */}
3  }
4
5  class ConcreteComponent extends Component { /* ... */
       ↪ }
6
7  abstract class Decorator extends Component {
8   Component component;
9  }
10
11 class ConcreteDecorator extends Decorator {
12  @Override
13  void commonOperation(){
14   // Carry out additional responsibility
15   // ...
16   this.component.commonOperation();
17  }
18 }
```

8.4.3 Decorator Example Use Case

There are many possible uses cases for the Decorator pattern. Section 8.4.1 suggested the creation of a text editor as an example. This section elaborates on that example, so that in can be implemented and analyzed:

The requirements for our simple system are the following:

- Words are stored as objects
- WORD objects are a subclass of GRAPHICALELEMENT
- Words can be made italic, bold or colorized
- These properties must be attached to the word objects dynamically, i.e., at runtime.

8.4.4 Decorator Java Implementation

Listing 8.21 gives a possible JAVA implementation of the text editor proposed in section 8.4.3. The design is in accordance to the generic UML from section 8.4.1. The class CLIENT demonstrates how objects can be instantiated and used. The system and pattern is clearly trivial to implement in JAVA.

Listing 8.21: Text Editor Java implementation

```
abstract class GraphicalElement {
 abstract void drawSelf();
}

class Word extends GraphicalElement {
 String word;

 Word(String word){
  this.word = word;
 }

 void drawSelf(){
  System.out.println(this.word);
 }
}

abstract class GraphicDecorator extends
    ↪ GraphicalElement {
 GraphicalElement element;

 GraphicDecorator(GraphicalElement element){
  this.element = element;
 }
}

class ItalicDecorator extends GraphicDecorator {
 ItalicDecorator(GraphicalElement element){
  super(element);
 }

 void drawSelf(){
  System.out.println("Make italic");
  this.element.drawSelf();
  System.out.println("Make italic");
 }
}

class Client {
 public static void main(String[] args) {

  GraphicalElement foo = new Word("foo");
  foo.drawSelf();
```

```
42
43    foo = new ItalicDecorator(foo);
44    foo.drawSelf();
45
46   }
47  }
```

8.4.5 Decorator JavaScript implementation

JavaScript simplifies the implementation greatly: Because it is dynamically typed, object decoration is trivial to implement, and does not require the supporting structure needed in the Java implementation. A possible JavaScript implementation is provided in listing 8.22. Instead of having the decorated object inherit from some class, it inherits directly from the object being decorated. At line 12 this inheritance is established, as it instantiates an object decorated which will have wordObject as its prototype object.

Listing 8.22: Text Editor JavaScript implementation

```
1  class Word {
2   constructor(word){
3    this.word = word
4   }
5
6   drawSelf(){
7    console.log(this.word)
8   }
9  }
10
11 function italicDecorator(wordObject){
12  const decorated = Object.create(wordObject)
13
14  decorated.drawSelf = function(){
15   console.log("make italic")
16   wordObject.drawSelf()
17   console.log("make italic")
18  }
19  return decorated
20 }
21
22 const foo = new Word("foo")
23 foo.drawSelf()
```

```
24 const decoratedFoo = italicDecorator(foo)
25 decoratedFoo.drawSelf()
```

8.4.6 Decorator Comparison and Classification

In the previous section we have implemented the text editor example in JAVA and JAVASCRIPT. The JAVA implementation largely conforms to the traditional implementation of the Decorator pattern, whereas the JAVASCRIPT implementation is more experimental. Both of the implementations provide the same functionality. There are discovered no language or library feature in JAVA that implements or assists the Decorator pattern. Therefor we establish JAVA to exhibit NO SUPPORT for the Decorator pattern, as per our classification scheme. The experimental JAVASCRIPT implementation is clearly more concise, and easier to implement compared to the JAVA implementation. In this instance, conciseness comes at the cost of type safety, as any implementation errors would result in a runtime error. The dynamic nature of JAVASCRIPT clearly a□ects the implementation and need for the Decorator pattern. There is still situations where the traditional Decorator pattern is preferred, but often this alternative implementation will su□ce. Therefore we establish JAVASCRIPT to exhibit LANGUAGE ASSISTED SUPPORT for the Decorator pattern. Similar results are expected in other dynamically typed languages. Our classification of the languages in relation to the Decorator pattern is outlined in table 8.5.

Table 8.5: Classification of support for the Decorator pattern

Programming Language	Level of Language Support
Java	No Support
JavaScript	Language Assisted

8.4.7 Decorator Summary

This section presents the Decorator pattern. It is established that the patterns motivation is to attach new properties of responsibilities to objects at runtime, without relying on inheritance. Creation of text editors is suggested as an example use case for the Decorator pattern, and implementations of the example are provided in JAVA and JAVASCRIPT. JAVA show no inherent support for the pattern, whereas JAVASCRIPT's dynamic type system alleviate the need for much of the patterns structure. We therefore determine JAVASCRIPT to show LANGUAGE ASSISTED SUPPORT for the Decorator pattern.

8.5 Prototype

The following section explores the Prototype design pattern. Firstly, a general introduction to the pattern is provided. A generic JAVA implementation is given, and the creation of a network request generator is proposed as an use case example. Alternative implementations using JAVA's CLONEABLE interface and JAVASCRIPT's prototypical inheritance are discussed, and demonstrates that the Prototype pattern can be built into languages to some degree.

8.5.1 About the Prototype Pattern

Prototype is creational design pattern, where objects are created by copying or inheriting from a prototype object. The Prototype pattern deals with how to instantiate new objects, where the classes to instantiate are determined at runtime [10, p. 117]. Other patterns, like Abstract Factory, also support object creation and configuration at runtime. However, the Prototype pattern di□ers in that it relies on prototypical inheritance, instead of subclassing. This mainly has two benefits: For some uses, subclassing cause a cluttered and somewhat contrived class hierarchy. The other advantage is performance: Copying objects, which the Prototype pattern does, is often more e□cient than creating new ones from scratch.

8.5.2 GoF Prototype Implementation

Figure 8.5 describes GoF's proposed implementation of the Prototype pattern. It uses object composition, instead of subclassing. That is, the client object holds a reference to a prototype object, instead of extending some class. This design decouples the client from its object instantiations: The client object need no knowledge of which classes to instantiate. The prototype objects themselves are responsible for that.

Listing 8.23 provides a generic JAVA implementation of GoF's design. The client holds one or many prototype objects. By prototype object, we mean objects that extend the abstract Prototype class. When asked to create a new object, the client simply asks its prototype object to make a copy of itself. Because the prototype objects are the ones who implement the copying functionality, the client class does not need to know which specific object to create. The abstract Prototype class may implement the copying functionality, if a general implementation is possible.

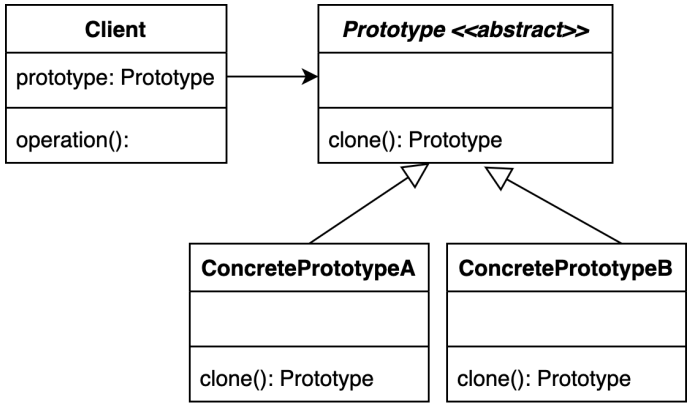

Figure 8.5: Prototype UML specification

Listing 8.23: Generic Prototype Java implementation

```
class Client {
 Prototype prototype;

 Prototype createNewObject(){
  return prototype.clone();
 }

}

abstract class Prototype {
 public abstract Prototype clone();
}

class ConcretePrototypeA extends Prototype {
 public ConcretePrototypeA clone(){
  ConcretePrototypeA copy = new ConcretePrototypeA();
  // Configure copy to be equal to this
  return copy;
 }
}
```

8.5.3 Prototype Example Use Case

For our analysis of the pattern, the creation of a network request generator will be our use case example. There are many situations where the Prototype pattern may be applicable. A common use case is when multiple objects, e.g., network request objects, should have a similar configured state.

Our network request generator should o□er functionality for generating network request objects. Network request objects are objects representing a network request, e.g., a Hypertext Transfer Protocol (HTTP) request. Such objects holds state, or rather a set of fields. The values of these fields varies depending on the type of request. Listing 8.24 demonstrates how we wish to use the network request generator: We import the library, and from that we can generate di□erent network request objects.

For the network request objects to be valid, they must be configured correctly. By storing pre-configured requests, i.e., prototypes in the network request generator, correct and e□cient creation of new network request objects are ensured.

Listing 8.24: network request generator example usage

```
1 import NetworkGenerator;
2
3 NetworkRequest httpRequest = Generators.httpGenerator
    ↪ .newRequest(/*custom data*/);
4 NetworkRequest ftpRequest = Generators.ftpGenerator.
    ↪ newRequest(/*custom data*/);
```

8.5.4 Prototype Java Implementations

Implementing the proposed system with the Prototype pattern is trivial in JAVA. Listing 8.25 follows the design proposed by GoF. The class NETWORKREQUESTGENERATOR corresponds to the Client class from figure 8.5, the abstract class NETWORKREQUEST is the Prototype class and HTTPREQUEST is the concrete prototype implementation. The class GENERATORS is mainly an wrapper object, that stores available request generators.

Listing 8.25: network request generator example JAVA implementation

```
1 class Generators {
2  static NetworkRequestGenerator httpGenerator = new
     ↪ NetworkRequestGenerator(new HttpRequest());
3 }
4
5 class NetworkRequestGenerator {
6  NetworkRequest prototype;
7
8  NetworkRequestGenerator(NetworkRequest prototype){
9   this.prototype = prototype;
10  }
11
```

```
12   NetworkRequest newRequest(){
13    return prototype.copy();
14   }
15  }
16
17  abstract class NetworkRequest {
18   abstract NetworkRequest copy();
19   abstract void sendRequest();
20  }
21
22  class HttpRequest extends NetworkRequest {
23   String state;
24
25   HttpRequest(){
26    this.state = "foo";
27   }
28   HttpRequest(HttpRequest copy){
29    this.state = copy.state;
30   }
31
32   public HttpRequest copy(){
33    return new HttpRequest(this);
34   }
35
36   public void sendRequest(){
37    //...
38    System.out.println("HTTP-request was sent.");
39   }
40  }
```

The JAVA standard library contains an interface named CLONEABLE [21]. It is used to indicate that CLONE is a legal operation for classes that implement the interface. CLONE is equivalent to our use of the word Copy: The operation returns a new object that is copy of the object the operation is invoked on [20]. At a glance, this appears to be a an out-of-the-box generic implementation of the Prototype pattern, and that Java inherently has LIBRARY LEVEL SUPPORT for the pattern.

However, there exists no viable general implementation of the CLONE method. All classes must either implement their own CLONE method, or inherit it from some appropriate superclass. Therefore only half of the work is done by the library implementation. In our contrived and simplified example, implementing CLONE or Copy is straigtforward. But that is rarely the case: GoF states that implementing the CLONE operation correctly, is the most di□cult of the

pattern [10, p. 121]. Therefore the CLONEABLE interface provide no clear benefits in relation to the Prototype pattern in JAVA.

8.5.5 Prototype JavaScript Implementation

Interesting to this study, GoF explicitly claims that the Prototype pattern is built into prototype-based languages, because all object creation inherently happens by cloning a prototype [10, p. 121]. JAVASCRIPT is a prototype based language, as described in chapter 4.3.1. Therefore we expect the Prototype pattern to be redundant, or at least simplified when implemented in JAVASCRIPT.

Listing 8.26 shows that it indeed is simplified. The variable HTTPREQUEST loosely corresponds to a concrete prototype implementation. It is defined as a single object, instead of a class. The variable GENERATOR is comparable to the Client and wrapper object from the JAVA implementation, merged into one. This JAVASCRIPT implementation provides the same functionality as the one in JAVA, in a much more concise manner. Instead of implementing the Clone functionality, JAVASCRIPT's Object.create is used: It creates a new object, and uses the provided object as the prototype for the new one.

When using design patterns, the developer must implement both the domain specific code, and the structure supporting the design pattern. In our JAVASCRIPT implementation, no structure concerning the pattern is needed, and the developer can focus solely on the domain specific code. This come at a cost: By using JAVASCRIPT's native prototypical inheritance, the developer retain no control over how the objects are copied or cloned. Therefore the full Prototype pattern may still be needed for some cases.

Listing 8.26: network request generator example JAVASCRIPT implementation

```
1
2  // Library
3  const HttpRequest = {
4   state: "foo",
5   sendRequest(){
6    console.log("HTTP-request was sent.")
7   }
8  }
9
10 const Generator = {
11  newRequest(){
12   return Object.create(HttpRequest)
13  }
```

```
14 }
15
16 // Usage
17 const req = Generator.newRequest()
18 req.sendRequest()
```

8.5.6 Prototype Comparison and Classification

The previous sections have implemented the network request generator example in JAVA and JAVASCRIPT. The JAVA implementation is inspired by GoF's Prototype implementation, whereas the implementation in JAVASCRIPT relies on the language built-in prototypical inheritance. The implementations provide the same functionality. No built-in support was discoverd in JAVA, and thus we have established that JAVA exhibit NO SUPPORT for the Prototype pattern. The implementation in JAVASCRIPT nearly eliminates the need for the pattern, as the Prototype pattern largely is an interpretation of prototypical inheritance. The are however uses where the full Prototype pattern might be required in JAVASCRIPT. Therefore we establish JAVASCRIPT to exhibit LANGUAGE ASSISTED SUPPORT for the Prototype pattern. These classifications are presented in table 8.6.

Table 8.6: Classification of support for the Decorator pattern

Programming Language	Classification of Language Support
Java	No Support
JavaScript	Language Assisted

8.5.7 Prototype Summary

This section presents the Prototype design pattern. It is a creational design pattern that allows for dynamic object instantiation, i.e., determining which objects to instantiate at runtime. This is achieved by letting the objects implement some cloning functionality, so that new objects can be created from a prototype object. The creation of a network request generator is used as an example use case for the pattern. This use case is implemented in JAVA and JAVASCRIPT. Given these implementation, it is established that JAVA show NO SUPPORT and that JAVASCRIPT show LANGUAGE ASSISTED SUPPORT for the Prototype design pattern.

8.6 Iterator

This section investigates the Iterator design pattern. A general presentation of the pattern is provided, along with GoF's proposed implementation. Then a music library is suggested as a use case example for the pattern. Implementations for this use case in JAVA, PYTHON and JAVASCRIPT are proposed. Given these implementations, we will assess whether or not library and language features can replace the traditional Iterator pattern.

8.6.1 About the Iterator Pattern

Iterator is a behavioral design pattern that provides a way to access elements in aggregate objects, without exposing the objects structure [10, p. 257]. In this context, aggregate objects mean a single object encapsulating a collection or some other grouping. The pattern provides an approach for sequentially querying an aggregate object for its next element, regardless of the objects internal data representation.

There are several motivating aspects for the pattern. Firstly we wish to keep the aggregate objects interfaces as simple as possible. Cluttering such objects interfaces with multiple methods for accessing its elements is sub-optimal. By introducing the Iterator pattern, only a single method, NEWITERATOR, is needed to access to an objects elements. Secondly the Iterator pattern lets clients treat aggregate objects with di□erent internal structures, uniformly. That is, the client need not consider the internal structure of the object it is accessing. Such concerns are handled by the objects iterator-object, not the client. Therefore the client can access di□erent data structures, e.g., a tree and a list, by the same interface.

8.6.2 GoF Iterator Implementation

Figure 8.6 presents GoF's proposed structure for the Iterator pattern. Without the Iterator pattern, the aggregate classes would be responsible for providing external access to their own elements. Instead the pattern introduces iterator objects. Iterator objects becomes the point of access for aggregates. They have a basic interface supporting operations for sequentially accessing elements of some aggregate object. In order to access these elements, all concrete iterator objects must hold a reference to the aggregate object they are accessing. The aggregate objects are responsible for instantiating their own concrete iterators.

Listing 8.27 gives a generic implementation of the pattern in accordance with the UML from figure 8.6. As proposed, the CONCRETEAGGREGATE

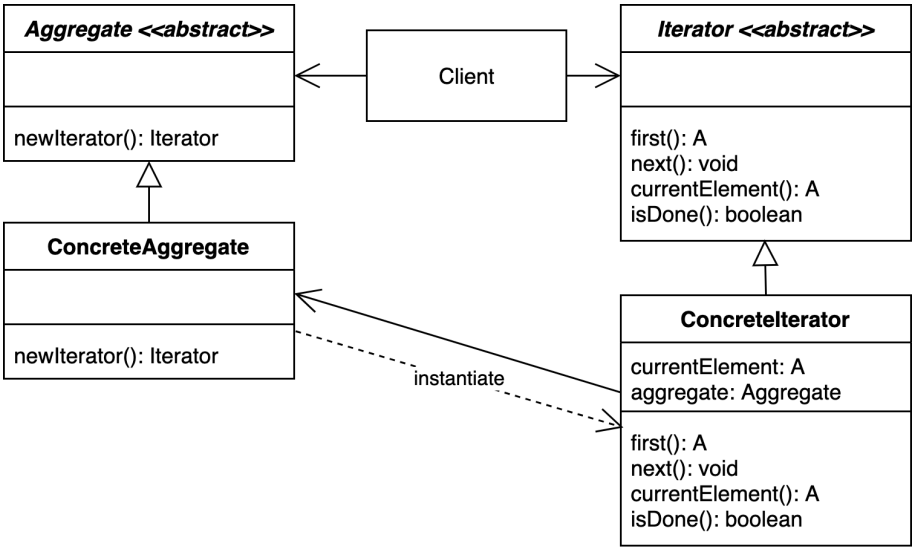

Figure 8.6: Iterator UML specification

class instantiates its own iterator object. By passing itself as an argument to the ConcreteIterator constructor, the iterator object gains access to the internal structure of the aggregate object. The specific implementations of the methods in ConcreteIterator cannot be generalized, as they will depend on the internal structure of the aggregate object.

Listing 8.27: Generic Iterator Java implementation

```
1  abstract class Aggregate {
2   abstract Iterator newIterator();
3  }
4
5  class ConcreteAggregate extends Aggregate {
6   Iterator newIterator(){
7    return new ConcreteIterator(this);
8   }
9  }
10
11 abstract class Iterator {
12
13  abstract Object first();
14  abstract void next(); // Advanses the iteration one
       ↪ step
15  abstract Object currentElement();
16  abstract boolean isDone();
17 }
```

```
18
19 class ConcreteIterator extends Iterator {
20  Aggregate aggregate;
21  Object currentElement;
22
23  ConcreteIterator(Aggregate aggregate){
24   this.aggregate = aggregate;
25  }
26
27  Object first(){/* ... */}
28
29  void next(){/* ... */}
30
31  Object currentElement(){
32   return currentElement;
33  }
34
35 boolean isDone(){/* ... */}
36
37 }
```

8.6.3 Iterator Example Use Case

The creation of a music library will be the example use case for this exploration of the Iterator pattern. This simple music library should simply store songs, and allow for adding new songs to the library. For accessing songs in the library, an iterator object will be used. The iterator should provide the same methods as the generic Iterator implementation in listing 8.27. Listing 8.28 demonstrates roughly how the music library should be used.

Listing 8.28: Proposed music library usage

```
1 ...
2 MusicLibrary library = new MusicLibrary();
3 library.addSong(imagine);
4 MusicIterator iterator = library.newIterator();
5 ...
```

8.6.4 Iterator Java Implementations

Implementing the library using the Iterator pattern is trivial in Java. Listing 8.29 is such an implementation. It uses the abstract classes defined from the

generic implementation in listing 8.27.

Listing 8.29: Java music library implementation

```
class MusicLibrary extends Aggregate {
 ArrayList<Song> songs;

 MusicLibrary(){
  this.songs = new ArrayList<Song>();
 }

 void addSong(Song newSong){
  this.songs.add(newSong);
 }

 MusicLibraryIterator newIterator(){
  return new MusicLibraryIterator(this);
 }
}

class MusicLibraryIterator extends Iterator {
 MusicLibrary library;
 Song currentElement;
 int index;

 MusicLibraryIterator(MusicLibrary library){
  this.library = library;
  this.index = 0;
  this.currentElement = library.songs.get(0);
 }

 Song first(){
  return library.songs.get(0);
 }

 void next(){
  if(this.isDone()) return;
  this.currentElement = library.songs.get(++index);
 }

 Song currentElement(){
  return currentElement;
 }

 boolean isDone(){
```

```
42     return this.index+1 >= library.songs.size();
43   }
44 }
```

It should be noted that Java's standard library collections, e.g., LIST and MAP all implement the Iterator pattern [22]. That could have been used in our JAVA implementation of the music library, as the different songs are stored in an ARRAYLIST: Instead of implementing our own iterator, line 13 could simply be replaced with *return this.songs.iterator()*, and the MUSICLIBRARYITERATOR class could be discarded. We have chosen not to do so, so that we can be general about JAVA's support for the Iterator pattern. That is, we want to analyse the Iterator pattern in relation to all usages in JAVA, not just the ones relying on standard library collections.

JAVA does provide some general support for the Iterator pattern. As part of its standard library, it provides interfaces named ITERABLE and ITERATOR [22][23]. They could be used instead of the abstract classes in our implementation. ITERABLE would then replace the class AGGREGATE, and Java's built in ITERATOR interface would be used instead of our custom ITERATOR abstract class. One would still have to implement the the interfaces, which is the bulk of the work. Therefore it cannot be claimed that Java have full native support for the pattern.

8.6.5 Iterator Python Implementation

As python is dynamically typed, and uses duck typing as discussed in chapter 4.4.1, the use of abstract classes or interfaces can be omitted when implementing the Iterator pattern.

In addition, our music library is built using PYTHON's standard list implementation which inherently supports the iterator pattern, through the built in function ITER [9]. Utilizing this support, we arrive at the implementation presented in listing 8.30. As discussed in regards to the JAVA implementation in section 8.6.4 we want to be general in our analysis of the pattern-language relationship. The ITER function will not be natively supported by custom aggregate objects, as such it is evident that the Iterator pattern is not fully supported in PYTHON.

Listing 8.30: Python music library implementation

```
class MusicLibrary:
 def __init__(self):
  self.songs = []

 def addSong(self, newSong):
  self.songs.append(newSong)

 def newIterator(self):
  return iter(self.songs)
```

8.6.6 Iterator JavaScript Implementation

JavaScript largely yields the same results as our Python implementation in section 8.6.5. The implementation is presented in listing 8.31. It relies on JavaScript's built in array, and the built in function iterator [8, p. 575]. Yet again we have arrived at an implementation not applicable to custom aggregate objects, which indicates that the Iterator pattern is not fully supported in JavaScript.

Listing 8.31: JavaScript music library implementation

```
class MusicLibrary {
 constructor(){
  this.songs = []
 }

 addSong(newSong){
  this.songs.push(newSong)
 }

 newIterator(){
  return this.songs[Symbol.iterator]()
  }
 }
```

8.6.7 Iterator Comparison and Classification

In this section, three di□erent implementation of the music library use case example has been presented. Only the Java implementation explicitly implements the Iterator pattern. The Python and JavaScript implementations

uses the languages built in support for iterators for standard library collections. The same could be achieved in JAVA. None of the languages seems to have built in support for custom aggregate objects. The motivation for the Iterator pattern is therefore unchanged in all three languages, as the pattern has not been made redundant. It is however clear that the languages support for built in iterators for standard library collections, e.g., lists, arrays and maps, often alleviate the need for the Iterator pattern. We therefore establish JAVA, PYTHON and JAVASCRIPT to all exhibit LIBRARY LEVEL SUPPORT for the Iterator design pattern. These results are outlined in table 8.7.

Table 8.7: Classification of support for the Iterator pattern

Programming Language	**Level of language support**
Java	Library Level
Python	Library Level
JavaScript	Library Level

8.6.8 Iterator Summary

This section explores the Iterator design pattern. The Iterator pattern provides functionality for accessing elements sequentially in an aggregate object, without exposing the underlying data structure of that object. The creation of a music library is used as an example use case. Implementations in JAVA, PYTHON and JAVASCRIPT of that use case are presented. It is shown that all languages have built in support for iterators for standard library aggregate objects, such as lists and maps. No language appear to have built in support for iterators for custom aggregate objects. We therefore conclude that all three languages exhibit LIBRARY LEVEL SUPPORT for the Iterator pattern.

8.7 Proxy

This section looks into the Proxy design pattern. A general presentation of the pattern is given, including a generic JAVA implementation. Using the pattern for user authentication is proposed as an example use case, and is implemented using JAVA and JAVASCRIPT.

8.7.1 About the Proxy Pattern

Proxy is a structural design pattern, that provides a placeholder object for another target object, in order to control or limit access to the target [10, p. 207]. In some situations, accessing a target object directly may be impractical, or costly with regards to performance. Instead, a proxy object can be created. The proxy object ensures controlled access to the target. It acts as a transparent interface in front of the target. That is, the proxy supports the same interface as the target, so that the targets clients can access the proxy, believing it is the target.

8.7.2 GoF Proxy Implementation

Figure 8.7 presents the structure of GoF's proposed implementation of the pattern. By having the REALTARGET class and PROXY class inherit from a shared abstract class, clients of the target can treat them uniformly.

Listing 8.32 provides a generic implementation of the pattern. In accordance to the UML from figure 8.7, the abstract class TARGET is extended by the REALTARGET class and Proxy class. Upon invocation of the OPERATION method in a Proxy instance, the proxy will control the access to the target object, and perhaps forward the method invocation to that object. The details of the access control will vary depending on the use case.

Listing 8.32: Generic Proxy Java implementation

```
1  abstract class Target {
2   abstract void operation();
3  }
4
5  class Proxy extends Target {
6   Target target;
7
8   Proxy(Target target){
9    this.target = target;
10  }
11
```

```
12   public void operation(){
13    // Some action to control access to target
14    // ...
15    target.operation(); // Not required
16   }
17  }
18
19  class RealTarget extends Target {
20   public void operation(){ /* ... */}
21  }
```

8.7.3 Proxy Example Use Case

There are many uses for the Proxy pattern. For our exploration of the topic, a very simple banking application is created. The system consists of users and accounts that the users can withdraw money from. The Proxy pattern will be used for authenticating the users, i.e., verifying that they are allowed to withdraw money from the account.

8.7.4 Proxy Java Implementations

Listing 8.33 gives a JAVA implementation of the proposed use case. Implementing the pattern in JAVA is trivial, yet quite verbose. The proxy object represented by the PROXYACCOUNT class holds a reference to the actual account. When the WITHDRAW method in the proxy object is invocated, it will check whether or not the user is authenticated, before forwarding the request to the actual account.

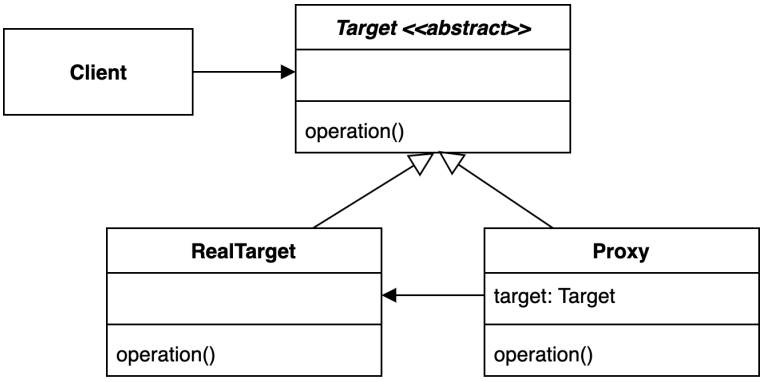

Figure 8.7: Proxy UML specification

Listing 8.33: Java Proxy use case implementation

```
abstract class Account {
 abstract boolean withdraw(int ammount);
}

class RealAccount extends Account {
 int balance;

 public boolean withdraw(int amount){
  if(amount <= balance){
   balance = balance - amount;
   return true;
  }
  return false;
 }
}

class ProxyAccount extends Account {
 Account account;
 User user;

 ProxyAccount(Account account, User user){
  this.account = account;
  this.user = user;
 }

 public boolean withdraw(int amount){
  if(user.isAuthenticated){
   return account.withdraw(amount);
  }
  return false;
 }
}

class User {
 boolean isAuthenticated;
 User(boolean isAuthenticated){
  this.isAuthenticated = isAuthenticated;
 }
}
```

8.7.5 Proxy JavaScript Implementation

Whereas JAVA appear to o□er no built in support for the pattern, the JAVASCRIPT standard library consists of a constructor named PROXY [8, p. 688]. That constructor allows for creation of new objects proxying a target. The proxy object can then intercept and redefine behavior of the target. Listing 8.34 gives an implementation utilizing the PROXY constructor. Instead of the entire account object, only the WITHDRAW function is proxied on line 13. Line 14 redefines the new functions behavior: Firstly the user authentication is verified, and then on line 16, the function call is forwarded to the target.

This alternative design provides the same functionality as the implementation in listing 8.33. However, it is not completely true to the original pattern as proposed by GoF, as the interface of the proxy is slightly di□erent than the targets. Furthermore, this alternative implementation is rather esoteric: Without a deep understanding of JAVASCRIPT and its standard library, the proposed alternative is di□cult to understand, even for such a simple use case. Supporting multiple di□erent functions within the same proxy would be even more cumbersome. Therefore it seems as even though the structure supporting the design pattern can be omitted, the original proxy pattern often is preferable.

Listing 8.34: JavaScript Proxy use case implementation

```
1  const account = {
2   balance: 1000,
3   withdraw: function(amount) {
4    if(amount <= this.balance){
5     this.balance = this.balance-amount
6     console.log("Withdrawal successful")
7    } else {
8     console.log("Insufficient funds")
9    }
10  }
11 }
12
13 const withdrawProxy = new Proxy(account.withdraw, {
14  apply: function(target, thisArg, argumentsList){
15   if(argumentsList[0].isAuthenticated){
16    return Reflect.apply(target,
17                account,
18                argumentsList.slice(1));
19   }
20   console.log("User not authenticated")
```

```
21   }
22 })
23
24 const authUser = { isAuthenticated: true }
25 const NonAuthUser = { isAuthenticated: false }
26
27 withdrawProxy(authUser, 10); // Withdrawal successful
28 withdrawProxy(NonAuthUser, 10); // User not
     ↪ authenticated
```

8.7.6 Proxy Comparison and Classification

The previous sections provides two implementations of the Proxy pattern. The JAVA implementation conforms to GoF's proposed design, whereas the JAVASCRIPT implementation uses a standard library feature. No built in support was discovered in JAVA, and we therefore established that JAVA exhibits NO SUPPORT for the pattern. The JAVASCRIPT alternative has several downsides: It is not applicable to all uses, and is somewhat di□cult to understand. It can practical be useful for some uses. We therefore establish JAVASCRIPT to exhibit LIBRARY LEVEL SUPPORT for the Proxy pattern. These classifications are presented in table 8.8.

Table 8.8: Classification of support for the Proxy pattern

Programming Language	Level of language support
Java	No Support
JavaScript	Library Level

8.7.7 Proxy Summary

This section analyses the Proxy design pattern. The pattern provides a placeholder object, i.e., a proxy, for a target object. The creation of a simple banking system is used as an example use case for the pattern. The use case is implemented in JAVA, that shows NO SUPPORT and JAVASCRIPT that shows LIBRARY LEVEL SUPPORT for the Proxy pattern.

8.8 Implementations Summary

This chapter provides implementations of 7 different design patterns. Each pattern is briefly presented, a long with the proposal of an example use case, that is implemented in various programming languages. Our proposed classification scheme is applied to the implementations, in order to classify the languages support for the different patterns.

Part IV

Evaluation and Conclusion

Chapter 9

Design Pattern Results

This chapter summarizes the results of chapter 8. Creational, structural and behavioral patterns will be evaluated separately, in an attempt to associate language features with specific classifications of patterns. The results show that programming languages can support design patterns, but the extent of the support varies.

9.1 Creational Design Patterns Results

Table 9.1: Creational Design Patterns Results

	Emerald	**Java**	**JavaScript**	**Scala**
Prototype	-	No Support	Language Assisted	-
Singleton	Language Level	No Support	-	Language Level

Table 9.1 shows the results from our analysis of creational design patterns. The Prototype pattern is explored using JAVA and JAVASCRIPT, which are established to have NO SUPPORT and LANGUAGE ASSISTED SUPPORT for the pattern, respectively. JAVASCRIPT's partial support for the pattern is attributed to its object model, and approach to inheritance. The Singleton design pattern was analysed in relation to EMERALD, JAVA and SCALA. Our implementations demonstrated that both EMERALD and SCALA exhibit LANGUAGE LEVEL SUPPORT for the pattern, whereas JAVA showed NO SUPPORT. We attribute EMERALD's and SCALA's support to their object models.

The results show that some creational design patterns indeed can be added to programming languages. The languages object models are the central language feature to support these creational patterns. Whether or not object

model is a pivotal language feature for other creational design patterns is unknown.

9.2 Structural Design Patterns Results

Table 9.2: Structural Design Patterns Results

	Java	**JavaScript**
Decorator	No Support	Language Assisted
Proxy	No Support	Library Level

Two structural design patterns are analysed in this study. The results from this analysis are presented in table 9.2. The Decorator pattern is established to have LANGUAGE ASSISTED SUPPORT in JAVASCRIPT, due to its dynamic typing scheme. JAVA showed NO SUPPORT for the pattern. We expect other dynamically typed languages to also have LANGUAGE ASSISTED SUPPORT for the Decorator pattern. The Proxy pattern was also explored, and it was shown that a JAVASCRIPT standard library object provides LIBRARY LEVEL SUPPORT for the pattern. No improved implementation of the pattern was found using JAVA, which therefore was classified to exhibit NO SUPPORT.

The results indicate that structural design patterns may be aided by language or library features. No single language feature could be established to be particular influential in relation to structural patterns. This shortage of findings may be attributed to the limited scope of this project, as only two out of eight structural patterns are analysed.

9.3 Behavioral Design Patterns Results

Table 9.3: Behavioral Design Patterns Results

	Java	**JavaScript**	**Python**
Command	Language Assisted	Language Assisted	-
Iterator	Library Level	Library Level	Library Level
Strategy	Language Assisted	Language Assisted	Language Assisted

The results from our experiments with behavioral design patterns are presented in table 9.3. The three behavioral design patterns Command, Iterator and Strategy were studied. Command and Strategy are very similar patterns, and yielded equal results: JAVA and JAVASCRIPT both provided LANGUAGE ASSISTED SUPPORT for the patterns. PYTHON was omitted when studying the Command pattern, but also exhibited exhibited LANGUAGE ASSISTED

SUPPORT for the Strategy pattern. The support for Command and Strategy is attributed to the languages support for lambda functions. The Iterator pattern was analysed in relation to JAVA, JAVASCRIPT and PYTHON which all exhibited LIBRARY LEVEL SUPPORT, i.e., the pattern was supported for some standard library objects.

These results show that none of the investigated behavioral design patterns are made completely superfluous in any of the programming languages utilized. However, lambda functions appears to nearly eliminate the need for several of the patterns. From this, it seems likely that lambda functions also can be used in place of other behavioral patterns, though such a claim need further research.

9.4 Evaluations of Language Features

Previous studies have shown that certain language features or elements are of particular importance in regarding programming languages relationship to design patterns [16]. Similar results are found in our experiments. This section discusses a selection of language features and their e□ect of design patterns.

9.4.1 Dynamic Typing and its Effect on Design Patterns

Dynamic typing do a□ect how one implement design patterns. The need for interfaces or abstract classes, which is heavily used in GOFBOOK, disappears. Thus implementation is simplified to some degree. We did however find that design patterns are still highly relevant in dynamically typed programming languages, as few patterns are rendered completely redundant. Decorator was the only pattern explored that greatly benefited from dynamic typing. We expect dynamic typing to reduce the need for other patterns, not included in this study, as well.

9.4.2 Lambda Functions Effect on Design Patterns

Lambda functions and higher order functions are two closely related language features. As they have become more prevalent in modern programming languages, the gap between object oriented and functional languages have shrunk. In our analysis of behavioral design patterns we lambda functions to be a highly useful feature. For some uses, it is a viable alternatives to the original design patterns. Thus the need for object oriented design patterns, mimicking functional behavior, has lessened. We expect lambda functions to also be of benefit to other design patterns.

9.4.3 Object and Inheritance Models Effect on Design Patterns

Programmings languages object models and approach to inheritance directly a□ect the need for some design patterns. It is therefore clear that some design patterns are workarounds for language limitations. The Singleton pattern is a useful example: The lack of granularity in some languages object models prompted the need for the pattern.

9.5 Design Pattern Results Summary

This chapter presents the results from implementing a selection of design patterns in chapter 8. The results show that some creational design patterns can be replaced by language features. Object and inheritance models appears to be of particular importance for such patterns. The need for some structural design patterns are also influenced, and partly decreased by language features. The three examined behavioral design pattern are all partly supported by various programming languages. In particular, lambda functions appear to be limit the need for certain patterns.

Chapter 10

Project Results

The following chapter outlines the overall results of this thesis. Firstly the classification scheme is discussed, as the main new insight and contribution from this project. Following that a theory on the relationship between design patterns and programming languages is presented.

10.1 New Insights through Classification Scheme

The main novel insight gained from this thesis is the di□erent classifications of support a programming language can exhibit for a design pattern. The phrase "Programming language X inherently supports design pattern Y" is not uncommon in discussions of design patterns. Such statements are lacking in nuance: Whether or not a language supports a certain design pattern is not a binary question. One must be more precise in the definition of design pattern support. Our studies show that the relationships between languages and patterns exists on a spectrum. This spectrum can be modelled using a classification scheme.

Figure 10.1 compares the traditional binary classification scheme to our new proposed scheme. Four di□erent classifications of support, that a language can exhibit for a pattern, have been established. They are Language Level Support, Language Assisted Support, Library Level Support and No Support. Language Level Support denote what many call full support. Language Assisted Support describes instances where language features partly replaces design patterns. Library Level Support designate relationships where support exists for standard library objects. No Support intuitively mean that the pattern has no support in the specific programming language. More classifications may exist, but we find this model to be suitable.

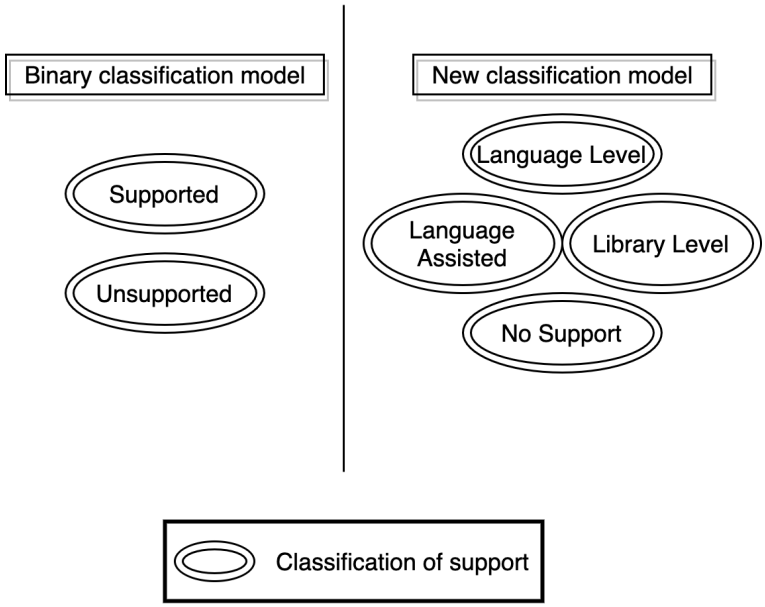

Figure 10.1: Comparison of classification schemes.

By applying this classification scheme to a selection of design pattern implementations, we arrive at two important conclusions:

- Few patterns have LANGUAGE LEVEL SUPPORT in any languages. That is, few patterns are completely superfluous in any programming languages.
- Many patterns appear to have some support in modern programming languages. Both LANGUAGE ASSISTED SUPPORT and LIBRARY LEVEL SUPPORT seem to be prevalent in multiple languages. Modern language features, e.g., lambda functions, appear to aid implementation or provide viable alternatives for many design patterns.

10.2 A Theory on the Relationship between Design Patterns and Programming Languages

The results of our studies show that design pattern and programming languages are two closely related concepts. They are clearly not independent of one another, and their relationship is therefore a relevant topic to study.

Design patterns and programming languages a□ect one another. From the previous section we know that some patterns, e.g., the Singleton pattern, are workarounds for language limitations. As time passes, it is then natural that

languages will adopt certain design patterns. It is therefore expected that design patterns from GofBook see alternative implementations in light of modern languages.

Our research also indicate that some design pattern are completely independent of programming language. That is, no language feature appear to assist or improve their implementation. These patterns may be impossible to incorporate into programming languages in an appropriate manner.

Lastly it should be noted that even though a design pattern is supported by a programming language, the pattern may still be a useful construct. It allows for programmers to communicate about code more clearly. Design patterns gives standardized names to abstract concepts. Discussing abstract code is difficult, so by having a common understanding of patterns, one can be more precise and eloquent in such discussions.

10.3 Project Results Summary

This chapter summarizes the insights gained through this master thesis: The main on being that there is a spectrum of classification that describe the relationship between design patterns and programming languages. That is, other classification than Supported and Unsupported exists. Further it is determined few patterns are fully supported by any language, but many patterns are partially supported by certain languages. Finally the section formulates a theory on the relationship between design patterns and programming languages. It states that pattern and languages are two closely related concepts, that both affect one another. Some pattern are workarounds for missing language features. Other patterns are completely independent of language. Regardless of a pattern is supported by languages or not, design patterns is a useful construct that allows for more precise and concise discussion about code.

Chapter 11

Conclusion

This study determines the relationship between design patterns and programming languages to be a relevant area of study. The two separate topics are closely related, and how they interrelate is therefore not trivial nor insignificant. Both topics must be analysed in light of each other.

The relationship between design patterns and programming languages is complex. Early in the process, it became apparent that a tool for analysing such relations is needed. Determining whether or not a programming language supports a design pattern, is matter of nuance. Our research show that the traditional binary model, consisting of the classifications supported and unsupported, is inaccurate. There are important distinctions between supported and unsupported. Thus a classification scheme for modeling pattern-language relationships is proposed. Four different classifications of support is established, as opposed to the two classifications of the binary model. The scheme puts forth various criteria for classifying the relationship between a specific pattern and language. We determine this classification scheme to be the most important contribution and insight from this master thesis.

By applying our classification scheme to a selection of design pattern implementations, new insights pertaining to specific patterns have been gained: Few design patterns are rendered completely redundant by any languages. Only a single design pattern was determined to be superfluous in light of certain programming languages. However, all seven analysed patterns see partial support in various languages. That is, language or library features gave rise to improved or alternative implementations. It must be noted that all seven design pattern were chosen in this study, because it appeared as though they have some support in selected languages. We do not expect all 23 of GoF's design patterns to have partial support. The established language support is attributed to various language features and aspects: Lambda functions, higher order functions, typing scheme, object model and

approach to inheritance all affected one or more design patterns to varying degrees. Creational design patterns seem to be particular influenced by object model and approach to inheritance. Whereas behavioral patterns were assisted by the presence of lambda functions, higher order functions and dynamic typing. No decisive language feature were found for structural patterns.

From these results we conclude that some design patterns are indeed workarounds for language limitations, and will therefore become redundant as new languages are introduced, whereas other patterns are highly language independent and will likely never have full support in any language.

Chapter 12

Future Work

Our studies suggest that the relationship between design patterns and programming languages is an interesting are of study, that can researched more. Due to the scope of this study, only 7 of the 23 GoF design patterns are examined. More research can therefore be done on the remaining patterns. Further, all programs presented in this study are relative simple and short. Future work may preferably examine larger and more complex pattern implementations.

Although the classification scheme is the main contribution from this study, it may still be in need of some refinement. Further development of the scheme may therefore be explored by future work. More classifications likely exists, and the current classifications would benefit from more explicit criteria. Additionally, the di□erentiation between level of support and classification of support, is somewhat contrived, and can possibly be eliminated.

References

[1] Christopher Alexander. *A pattern language : towns, buildings, construction*. eng. Vol. 2. Center for Environmental Structure series. New York: Oxford University Press, 1977. ISBN: 0195019199.

[2] Vladislav Georgiev Alfredov. "How Programming Languages A□ect Design Patterns." eng. MA thesis. 2016.

[3] S. Antoy and M. Hanus. "New functional logic design patterns." In: vol. 6816. Springer Verlag, 2011, pp. 19–34. ISBN: 9783642225307.

[4] *Are Design Patterns Missing Language Features*. 2013. URL: http://wiki.c2.com/?AreDesignPatternsMissingLanguageFeatures (visited on 05/22/2019).

[5] Andrew P. Black et al. "The Development of the Emerald Programming Language." In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. San Diego, California: Association for Computing Machinery, 2007, pp. 11–12. ISBN: 9781595937667. DOI: 10.1145/1238844.1238855. URL: https://doi.org/10.1145/1238844.1238855.

[6] Christophe Dony, Jacques Malenfant, and Pierre Cointe. "Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation." In: Jan. 1992, pp. 201–217. DOI: 10.1145/141936.141954.

[7] Christopher Dutchyn et al. "Multi-dispatch in the Java Virtual Machine (Poster Session): Design and Implementation." In: *Addendum to the 2000 Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications (Addendum)*. OOPSLA '00. Minneapolis, Minnesota, USA: ACM, 2000, pp. 115–116. ISBN: 1-58113-307-3. DOI: 10.1145/367845.368011. URL: http://doi.acm.org.ezproxy.uio.no/10.1145/367845.368011.

[8] European Computer Manufacturers Association (ECMA). *ECMAScript® 2019 Language Specification*. 2019. URL: https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf (visited on 05/16/2020).

[9] Python Software Foundation. *Python 3.8.3 documentation: Built-in Functions - iter*. URL: https://docs.python.org/3/library/functions.html#iter (visited on 05/18/2020).

[10] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1994.

[11] James Gosling et al. *The Java® Language Specification Java SE 8 Edition*. Oracle, 2015.

[12] Jan Hannemann and Gregor Kiczales. "Design pattern implementation in Java and aspectJ." eng. In: *ACM SIGPLAN Notices* 37.11 (2002). ISSN: 03621340.

[13] John C. Mitchell. *Concepts in Programming Languages*. Cambridge: Cambridge University Press, 2002. ISBN: 9780511804175.

[14] Mozilla. *JavaScript Guide: Iterators and Generators*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Iterators_and_Generators (visited on 04/28/2020).

[15] *Node.js*. URL: https://nodejs.org/en/ (visited on 05/10/2020).

[16] Peter Norvig. *Design Patterns in Dynamic Languages*. 1996. URL: http://www.norvig.com/design-patterns/design-patterns.pdf (visited on 05/22/2019).

[17] *Object Oriented Design: Singleton Pattern*. URL: https://www.oodesign.com/singleton-pattern.html (visited on 12/10/2019).

[18] Martin Odersky. *The Scala Experiment*. URL: http://lampwww.epfl.ch/~odersky/talks/google06.pdf (visited on 03/10/2020).

[19] *OMG Unified Modeling Language™. Version 2.5.* 2015. URL: https://www.omg.org/spec/UML/2.5 (visited on 05/27/2019).

[20] Oracle. *Java™ Platform, Standard Edition 8 API Specification: Class Object - Clone*. URL: https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#clone-- (visited on 04/29/2020).

[21] Oracle. *Java™ Platform, Standard Edition 8 API Specification: Interface Cloneable*. URL: https://docs.oracle.com/javase/8/docs/api/java/lang/Cloneable.html (visited on 04/29/2020).

[22] Oracle. *Java™ Platform, Standard Edition 8 API Specification: Interface Iterable*. URL: https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html (visited on 04/22/2020).

[23] Oracle. *Java™ Platform, Standard Edition 8 API Specification: Interface Iterator*. URL: https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html (visited on 04/22/2020).

[24] Rk Raj et al. "EMERALD - A GENERAL-PURPOSE PROGRAMMING LANGUAGE." English. In: *Software-Practice & Experience* 21.1 (1991), pp. 91–118. ISSN: 0038-0644.

[25] Brandon Rhodes. *The Iterator Pattern*. URL: https://python-patterns.guide/gang-of-four/iterator/ (visited on 04/28/2020).

[26] Vebjørn Ring Sjølyst. "The Impact of Languages on Design Patterns." eng. MA thesis. 2017.

[27] *The Java™ Tutorials: Lambda expressions*. Oracle. URL: https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html.

[28] *The State of the Octoverse: Top languages over time*. GitHub. URL: https://octoverse.github.com/#top-languages (visited on 03/23/2020).

[29] *TOUR OF SCALA: SINGLETON OBJECTS*. URL: https://docs.scala-lang.org/tour/singleton-objects.html (visited on 10/02/2019).

[30] Joshua Webjørnsen. "Observation of interactions between Design Patterns and Programming Languages." eng. MA thesis. 2018.

[31] Wikipedia. *Anonymous function — Wikipedia, The Free Encyclopedia*. 2020. URL: http://en.wikipedia.org/w/index.php?title=Anonymous%5C%20function&oldid=957470612 (visited on 05/14/2020).

[32] Wikipedia. *Class-based programming — Wikipedia, The Free Encyclopedia*. 2019. URL: http://en.wikipedia.org/w/index.php?title=Class-based%5C%20programming&oldid=861963883 (visited on 05/15/2020).

[33] Wikipedia. *Duck typing — Wikipedia, The Free Encyclopedia*. 2020. URL: http://en.wikipedia.org/w/index.php?title=Duck%5C%20typing&oldid=959128884 (visited on 05/14/2020).

[34] Wikipedia. *JavaScript — Wikipedia, The Free Encyclopedia*. 2020. URL: http://en.wikipedia.org/w/index.php?title=JavaScript&oldid=947395565 (visited on 05/15/2020).

[35] Wikipedia. *Python (programming language) — Wikipedia, The Free Encyclopedia*. 2020. URL: http://en.wikipedia.org/w/index.php?title=Python%5C%20(programming%5C%20language)&oldid=961989641 (visited on 05/14/2020).