How Implementation Language Affects Design Patterns

A Comparison of Gang of Four Design Pattern Implementations in Different Languages

Kristian Pedersen

Thesis submitted for the degree of
Master in Programming and Networks
30 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2019

How Implementation Language Affects Design Patterns

A Comparison of Gang of Four Design Pattern Implementations in Different Languages

Kristian Pedersen

How Implementation Language Affects Design Patterns

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

Printed: Reprosentralen, University of Oslo

Abstract

In the present work the impact of implementation language on code quality for implementations based on the Gang of Four design patterns is examined. These are used by many developers and system architects to improve their development process.

The patterns Composite, Prototype, Adapter and Decorator are examined in the context of the programming languages Python, JavaScript, C#, Go and Smalltalk. A case was designed and implemented in these languages for each pattern and then analyzed based on a gauntlet of metrics.

It was found that there were differences between the languages. Having mechanics solving problems similar to the problem the design pattern is solving was positive. In addition it was positive for a language to have flexible typing schemes, little overhead and attribute visibility control. No language was found to be clearly best suited, but individual strengths and weaknesses are discussed.

Contents

List of Figures

List of Tables

Part I

Introduction

Chapter 1

Introduction

1.1 Motivation

Since design patterns for informatics was introduces in 1994 they have grown to be used in many areas of the technology development industry. [5] describes how they are used are used by some of the largest actors in the industry, like Siemens and IBM, to improve the architectural process. As design patterns are fairly abstract concepts they are used for a variety of domains. According to [37], this even includes the software for spacecrafts! They are however contentious, with some critics claiming they are overused [38, 69]. Learning more about their nature and about which contexts suits them well is therefore important. This is what I intend to do in this thesis.

Specifically I want to study the impact of choice of programming language on design patterns. Different programming languages have different strengths and weaknesses. How do these affect an implementation of a design pattern? Are some design patterns better suited for some kinds of languages? Answering these questions might help inform a system architect or developer about whether to employ design pattern based design in their project. Further, if the attributes of the languages causing these differences could be identified, it would help extend these answers to languages not explicitly tested.

[28] states that "Each [design] pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem". If this is true, then design patterns are closely related to solutions of common problems. Studying which languages solve these problems best would then be of value, as it relates to which of these languages are best equipped to solve a gauntlet of common problems. This thesis will therefore also explore the problem from another angle, and study the differences in programming languages based on their performance on design pattern cases.

There has been papers written on this subject before. Examples of these can be found in Section 2.2. However, as seen in the systematic mapping studies [43] and [3], it is not a major field within design pattern research. I therefore believe there are still interesting things to learn from

new experiments in the field and that I can find a different angle from the published papers.

1.2 Scope and Problem Statement

In this thesis we will generate and analyze data to learn more about the effect the choice of programming language has on the implementation of a design pattern in that language. We will limit the programming languages to object oriented programming languages, as these are quite popular and design patterns are most commonly used in the context of them. Further, we will limit design patterns to a subset of the ones found in [28], the Gang of Four's original book on the subject, since these are the most widely accepted design patterns. This leads to the problem statement:

> How does the implementation of Gang of Four design patterns differ in object oriented languages and which attributes of the languages cause these differences?

1.3 Goal

In the essay a smaller experiment was performed to help select which object oriented languages to study in the main experiment. On the basis of that discussion the languages C#, Python, JavaScript, Go and Smalltalk were selected.

For each of these languages and for a series of design patterns selected iteratively, as described in Section 3.5.9, the goal will be to learn:

Differences What are the differences between the implementations of the design patterns in the different languages?

Cause What is the cause of the difference? Do any aspects or mechanics in the language cause the different deviations?

Trends Are there any consistent trends for these differences across the design patterns?

Suitability for design patterns Do these differences cause any languages to suit a specific subset of the patterns better? Does any languages perform consistently well across all the patterns?

1.4 Approach

These questions are studied in the context of an empirical study containing several sub-studies. In each of these sub-studies a design pattern is chosen to be studied. Then a case for that design pattern is designed and implemented in the relevant languages. Data, in the form of metrics and code, is gathered and is analyzed. A conclusion is then drawn for the sub-study and the next design pattern is selected.

As many of these sub-studies are performed as time allows. Then the results from the studies are discussed in the context of each other and a more general conclusion is drawn.

For a more detailed explanation of the process see Section 3. Section 3.3 specifies the outline of the experiment and Section 3.5 discusses the validity of the approach and the steps taken to improve it.

1.5 Evaluation

In this section I will discuss how data is gathered and evaluated. In every sub-study an implementation is made for every relevant language. Data is gathered from these partly through direct qualitative observation, but mostly from applying a set of predefined metrics. The choice was made to apply many such metrics, to capture a larger set of potential differences.

The metrics used are NCLOC, cyclomatic complexity, speed of execution, depth of inheritance tree, number of children, class cohesion, number of inheritances and shallow references. Some of these are specialized or modified from the traditional versions found in literature and the last is defined from scratch here.

Further discussion on the reason behind these choices and definition of the individual metrics in the context of this thesis can be found in Section 3.4.

1.6 Work Done

In this section the work done to produce this thesis is briefly listed. The intent is partly to ensure no work goes overlooked and partly to give insight into the process behind the development of the thesis.

Essay Prior to starting the writing of the thesis an essay was created. As part of preparing for this essay the Gang of Four book [28] was read, a brief literature review was performed and a minor experiment was done. The essay is attached as Appendix A.

Literature Review Prior to working on the essay a literature review was performed. The main focus of the review was design patterns and experiments using a method similar to mine. Especial focus was put on finding papers where these two overlapped. Effort was also put into finding good sources on metrics, validity and attributes of programming languages. The review for these fields was more focused on find seminal, high quality works than looking at the breadth of the field. A list was compiled of the potentially relevant sources, with a two-sentence summary for each source. The initial list was about 60 sources long.

Further sourcing The above-mentioned list provided a good basis, but was far from complete, so during the further development of the thesis other sources were found. In total the great majority of sources

used were not found in the original list. The sources in the list did however serve as a good reference for finding these sources.

Planning The method was planned out in detail before being written down. A fair amount of work went into developing a methodology that led to fair implementations, as described in Section 3.5. Since the thesis is also somewhat based around metrics, time was devoted to researching and adapting these.

Designing cases The cases for each design pattern was designed. There are, for most design patterns, many different version fitted for different contexts. I had to selecting a version, specifying an example of it and note down discussion and reasoning about these choices. This took a surprising amount of time compared to the actual implementation. In total 4 cases were created.

Implementation For every case an implementation had to be made in each of the 5 languages. The implementations were then tested until they passed the test specified in 3.5.10.

Gathering Data The metrics were, for each pattern and language combination, calculated and listed in a spreadsheet. This took a surprising amount of time as there is 140 such data points in total and some of them were not trivial to compute.

Analysis Analysis of the data was performed both after every individual case and across the cases after all of them were implemented.

Writing This text was written concurrently with most of the above tasks being performed. Usually keyword based notes were created on a subject and then relatively shortly afterwards the corresponding section in the thesis was written in full. This was done to avoid having one long writing session at the end of the thesis work and to write while the subject was still fresh in my mind. Towards the end of the project all these sections were reworked to link them and make the text more cohesive.

1.7 Results

The code written as part of this experiment can be found at
https://github.com/KristianBPedersen/MasterThesis/
The tables of measurements are long and not easily summarized. They can be found in Section 4 in the result sections of the individual cases.

1.8 Conclusion

The following are the main findings of the thesis:

Differences There are differences between the implementations of design patterns in different languages.

JavaScript and Python Similarities Python and JavaScript are similar when it comes to implementing design patterns. Both performed well.

C#, Go and Smalltalk similarities C#, Go and Smalltalk are relatively similar when it comes to length, but differ in other quality measures.

Core Issues There are for many patterns a core issue for implementations of it related to the problem the pattern is addressing. Languages having a toolkit to easily solve these problems perform better when implementing the design patterns.

Mechanics The following mechanics were found to increase the quality of a language's design pattern implementations:

- Flexible typing and inheritance schemes.
- Low notation and/or definition overhead.
- Protected attribute visibility, or ideally a controllable mechanic.
- Having a toolkit solving the same issues as the design patterns are solving.

The full conclusion and a discussion of its limitations can be found in Chapter 7.

1.9 Thesis Outline

The thesis is structured based on *søk og skriv*'s recommendations. These can be found in [70].

Part I of the thesis contains the introduction. The intent is for this part to motivate the subject of the thesis and provide a high-level overview of the content. **Sections 1.1, 1.2** and **1.3** serve to motivate and outline the core questions of the thesis. **Sections 1.4** and **1.5** gives a brief introduction to the methodology of the thesis. **Sections 1.6, 1.7** and **1.8** gives an overview of what happened and what was discovered during the writing of the thesis.

Part II contains the theoretical background necessary for discussions in part IV.

Part III contains a description of what is done to produce the results in the thesis. **Chapter 3** contains a description of the overall methodology. Note the Sections 3.4 and 3.5 which are fairly long and are respectively about the metrics used and steps taken to increase validity.

Chapter 4 describes the cases implemented, as well as the results for the individual cases. Note that in this respect it breaks with the structure of the text by containing both method description and analysis. An explanation for why this was done can be found in the section's introduction.

Part IV examines the data gathered. First, in **Chapter 5** the data is analyzed, then in **Chapter 6** the data is discussed. A conclusion is extracted in **Chapter 7** before possible extensions are discussed in **Chapter 8**.

Part II

Theory

Chapter 2

Background

2.1 Introduction

In this chapter we present the background from scientific literature which we are basing the thesis on. This will be a relatively short chapter as background is also presented elsewhere. The background for design patterns and programming languages is presented in the essay and the theory related to metrics is found in Section 3.4, which is part of the method section. It was found that it made sense to present the metrics and their adaptation to the context of this experiment together, to avoid repetitions and provide more clarity. Effort was made to clearly separate my own additions to the metric from the theory from literature.

This leaves us with two main tasks in this chapter. To present related work, in order to place this work in the context of the scientific debate, and to present papers I will be comparing my results to in the discussion. The first of these is done in Section 2.2 and the second is done in Section 2.3.

2.2 Design Patterns Implementation

2.2.1 Introduction

In this section we summarize the scientific literature on the implementation of design patterns. The summary will start out general, but scope in toward the part of literature most relevant to this thesis: The effects of language on design pattern implementation.

2.2.2 Implementation Description

There are many sources showing implementation of design patterns. From the very start Gang of Four shows implementation examples in both Smalltalk and C++ in [28]. There does for most of the common object oriented languages exists a book containing examples of design pattern implementations in a that language. Examples are [8, 14, 27, 48, 53]. There is also several papers outlining implementation in more esoteric contexts or less object oriented languages. Examples of this are [15] which implements

design patterns in Fortran and [4] which describes them using first order logic.

2.2.3 Implementation Studies

There is a subfield within design pattern research studying implementations of design patterns through experiments with multiple developers. In these studies a set of developers is given the task to implement something using design patterns and data is gathered from the implementations. The goal can be to learn about the quality of the design pattern like in [59], learn about the developers like in [34], learn about the effects of using design patterns on developers like in [39] or learn about the effects of the developers on the design pattern like in [50]. An overview of the field can be found in [62]

There is also a tradition for researching design patterns through case studies of existing code. One could gather code from open source projects or public repositories and analyze it with respect to design patterns. A subfield of this is the automatic pattern detection field, which is centered around automatic detection of patterns in code. Examples of this are [31, 68]. There are also papers structured around seeking out design patterns in a real context and extracting quality metrics about them like [6, 17].

There is a field within design pattern implementation dedicated to studying the quality aspects of a single design pattern implementation or a collection of a few, implemented by an expert specifically for the study. Examples of this include [32, 33]. This approach is the closest of the major approaches to the one taken in this thesis.

2.2.4 Implementation and Language

This brings us to the studies most related to this the experiment performed in this one. The ones which try to study an implementation of a design pattern in relation to the language it was implemented in. This is, based on the data presented in [43], a smaller field than the ones discussed earlier in this section. It was therefore harder to find seminal works in the field.

The most common kind of study within this subfield is the one which examines one or more design pattern implementations in a single language in order to learn more about how that pattern works in said language. Examples of this is [9, 64]. One of these studies, [64], will be examined later in Section 2.3.2.

Another type of study is the kind which is comparative between a single case implemented in several languages. This kind of approach seemed even rarer than the above. Examples include [2] which implements a series of design patterns in both Java and ParaAJ in order to examine the differences in the language, [20] which attempts to isolate static and dynamic parts of a design pattern by implementing it in several languages and [51] which makes a point of heavily comparing its Scala implementation of visitor with other implementations.

No single paper was found that is directly overlapping with mine. This was ensured by using a boolean search for the languages used in the experiment, along with the keywords *metric* and *design pattern* and manually reviewing all the hits the Oria database. This was also done for all the four language combination of the five languages.

2.3 Individual Papers

2.3.1 Introduction

In this section some papers are briefly summarized and reviewed. These are the ones I am comparing my results to in the discussion chapter, so it is important to establish what they are prior to utilizing them. As discussed in Section 2.2.4 it was hard to find papers with directly comparable results. As such doing such comparisons is of limited value and will be limited to the most interesting papers.

2.3.2 GoHotDraw

In this section the paper [64], *GoHotDraw: Evaluating the Go Programming Language with Design Patterns*, is studied. It is a paper examining an implementation of JHotDraw in Go, a program designed by Erich Gamma, which has a design reliant on design patterns. It examines how the Singleton, Adapter and Template Method patterns are implemented in Go compared to the original Java implementation and a C++ implementation. It concludes that the implementations are fairly similar, but also notes a that composition was useful for the adapter pattern. In the future work section it states that the comparison is purely based on qualitative observations, which creates some issues with the validity of the experiment. However it is still so closely related to the experiment in this thesis that we will discuss and compare similarities with our findings.

2.3.3 Empirical Study of Github

There are many empirical studies gathering data about programming languages. Examples include [47, 58, 61]. In the discussion chapter the results of this thesis are compared with the results from [61], *A Large-Scale Study of Programming Languages and Code Quality in GitHub*. It gathers data from large projects on GitHub for several languages and then compares these based on the number of bug reports. It is worth noting that, unlike many of the other studies discussed in this chapter, it has no relation to design patterns.

It detects maintenance issues in the studied repositories by counting how many commits are bug-fixes and then using linear regression to account for control variables. It concludes that there is a consistent trend in the number of bugs found in in repositories in different languages, but that this difference is relatively small. They go on to note, in their *threats to validity* section, that there are flaws in this methodology. It only

detects the amount of fixed bugs, not bugs in total in the code, it does not take into account the severity of the bug and it only registers them when explicitly referred to as bugs. One might imagine that the development field would be relevant to the quality of the code, and that languages that were regularly used in certain fields would be affected by this. This is however investigated and discarded in the paper.

Overall it still seems a good and different measure for the maintainability, which it will be interesting to compare my findings to. It is however important to note that it is in no way considered an absolute source on the maintainability of different languages. As shown in [16, p. 111] this is still a hotly debated topic.

2.3.4 On the Issue of Language Support

In this section the paper [10], *Design Patterns & Frameworks: On the Issue of Language Support*, will be introduced. It is not a particularly well known paper, but is included as it brings up a subject quite close to one found in the discussion.

The paper is about what common issues with implementing design patterns are and suggests different ways that the languages can be supported by extensions to handle this better. As an example of this it shows an adapter implementation in a language known as LayOM. This language allows for easy reassignment of method names and it shows how the use of it makes the adapter pattern much easier to implement. In the discussion section we will mostly concern ourselves with this example and the observations surrounding it. The rest of the subjects discussed in the paper are not revisited, as most of them are irrelevant and the relevant portions are not backed up by references or data.

Part III

Method

Chapter 3

Methodology

3.1 Introduction

This section describes the methodology of the thesis' main experiment. Describing how the experiments are performed and the data is gathered is essential. The value of this thesis hinges on the validity of the data and a proper understanding of its biases. It is therefore important the reader is given as much insight as possible to the gathering process.

Section 3.2 describes the conditions surrounding the experiment. In Section 3.3 we give the general outline of the content of the experiment. Section 3.4 introduces the metrics used in the experiment and lastly section 3.5 explains the steps taken to make this data more meaningful.

3.2 Setting

In this section the environment surrounding the experiment is discussed. As seen in [35] the skill level and time limit for implementing a design pattern is relevant to the implementation's quality. Briefly discussing my own experience and the parameters for this experiment is therefore helpful for putting the data gathered from this experiment in the context of other experiments performed in the literature and will help when interpreting the data.

As mentioned in Chapter 1 this experiment is done in the context of a short master thesis, which has a lead time of about five months. This includes the development of methodology and writing the thesis itself. In addition an essay was written which, despite having a two semester deadline, took about three weeks of clock time to complete since it was done in addition to a full course load. This essay performed a similar experiment to the one done in the main thesis, but without the methodological rigor, with fewer design patterns and with some extra languages.

While I have unknowingly been using design patterns for most of my programming life this was my first introduction to them in a theoretical context. As part of the preparation for writing the essay I read [28], skimmed through a few papers referencing it, read some popularized texts

on applying it them like [67] and viewed some StackOverflow questions about things I found unclear. I then implemented design patterns as part of the experiment in the essay. Prior to starting the experiment I read or skimmed through many of the sources on design patterns referenced in Chapter 2. I would say I now know more about them than the average developer, but I am far from an expert in all facets of them.

My experience with the different programming languages is varied. I originally studied applied mathematics, where the main language used in courses was Python. It is also my go-to language to use when I am given a choice, so I have used it a fair amount later on as well. C# is my favorite of the Java-like languages. I have no formal training in it, but I've had many Java based courses and the two languages are fairly similar. I have a hobby of creating games in Unity, which is based on C# and have a fair amount of experience in it. JavaScript is a language I have some, but not a lot, of experience in. It was the main language in a single course I took and I have also used google script a bit, which is quite similar.

Go and Smalltalk I had never used before writing the essay. As part of the preparation for writing the essay I read a guide on the basics of both. I read the *GNU Tutorial for Smalltalk* [24] and *An introduction to programming in Go* [19].

3.3 Outline

In this section we present the outline of the experiment. That is: The main activities performed in the experiment, their subactivities and the order they are performed. The main focus is on giving an overview of what these activities are, rather than why I made them that way. For an explanation of the choices see the discussion in Section 3.5.

The experiment is performed in cycles. In each cycle a pattern is selected, a case is designed, the case is implemented and data is gathered. The general flow of a single cycle is found below. Sometimes some of the steps naturally happen out of order and sometimes one has to return to a previous step to fix a mistake. Both of these things are allowed. However, the intent is to separate pattern selection, case design, implementation and data gathering, so effort is put into avoiding that these occur out of order.

Pattern selection In this step the pattern to implement is selected. It is important to emphasize that this means the patterns are not all selected at the start of the experiment, like the languages, but rather the choices are adapted to the data already gathered. For a discussion on why see Section 3.5.9.

Brainstorm case ideas Several cases are envisioned for the chosen pattern. Simple, keyword based notes are created for each potential pattern.

Choice of case A case is chosen to be the one used in the experiment. It is chosen partially to be an intuitive application of the pattern and partially to be a case good at showcasing differences and similarities between languages. For a discussion on why see Section 3.5.4.

Specification of case The case is fully specified. It is the interesting choices made in this and the previous step which are found in Section 4's *Choices* subsections.

Design of diagram A UML class diagram is created for the case. It is modeled using UML 2 notation as described in [30]. For a discussion on how it is modeled and why see Section 3.5.5 and 3.5.6.

Pseudocode for test For every case a usage example is created. It functions as a test that the code is sufficiently defect free and, as discussed in Section 3.5.5, as an analyzable example of usage of the design pattern. Pseudocode is generated to fully specify this test.

Design of textual description A first draft for the textual description for the case found in the relevant subsection of Section 4 is written. This description may be altered later for clarity, but the content is considered to be finalized at this point.

Implementation In this step the chosen case is implemented for all the programming languages. The order of this is varied as the later implementations is likely to be of higher quality. See Section 3.5.8 for discussion of this.

Evaluation The data required for evaluation of the metrics discussed in Section 3.4 is gathered and added to a Google Sheets document.

Conclusion Conclusions pertaining to this cycle of the experiment is drawn based on the metrics gathered. Further conclusions will be drawn later, once all the experiments have been performed and the results can be compared.

3.4 Metrics

3.4.1 Introduction

This section discusses the metrics used when extracting data from the code generated in the experiment. Section 3.4.2 discusses the basis for the selection of these metrics. Section 3.4.3 introduces an example to help explain the metrics. The rest of the sections introduces and defines the metrics used in the experiment. Some sections focusing on a single metric, some discussing several related metrics.

3.4.2 Metric Selection

The differences in the implementations we are looking to measure can take many different forms. As such it is important to have metrics measuring many different aspects of the quality of the program. We will therefore want to use a large gauntlet of metrics, to cover as many of these aspects as possible.

To achieve this we want to select metrics that are cheap to evaluate and probably not more than one metric measuring the same thing. For instance are both the LOC metric and the Halstead's approach classified in [21, p. 345] as measures of size. We will therefore only be using one of them. Similarly there exists many measures for complexity, but we will only be using cyclomatic complexity.

We will also define a new metric in Section 3.4.7 and modify certain metrics from literature to better fit this use-case. This is done to increases our control over what is measured. The downside is that these new metrics lack a formal and empirical foundation. This makes questions about what constitutes a significant difference between two measured values and the validity of the metric harder to answer.

3.4.3 Example

Through the rest of this section we will be using an example to help illustrate how the metrics are measured. It is written in Java to not favor any of the languages used in the main experiment and found below.

```
1  //Example code used for examplifying the
       application of the metrics used in the thesis.
2
3  interface Animal {
4      public void speak();
5  }
6
7  abstract class Canine implements Animal {
8      protected String sound;
9      public void speak() {
10         System.out.println(this.sound);
11     }
12 }
13
14 class Dog extends Canine {
15     public Dog() {
16         sound = "Woof";
17     }
18 }
19
20 class Wolf extends Canine {
21     public Wolf() {
22         sound = "Howl";
23     }
24 }
25
26 class Pig implements Animal {
27     private String oldSound = "Oink";
28     private String youngSound = "Squee";
```

```
29      public int age;
30
31      public Pig(int age) {
32          this.age = age;
33      }
34
35      public void speak() {
36          if (age < 2) {
37              System.out.println(youngSound);
38          }
39          else {
40              System.out.println(oldSound);
41          }
42      }
43  }
44
45  class MetricExample {
46      public static void main(String[] args) {
47          Animal[] animals = new Animal[]{new Dog(),
                new Wolf(), new Pig(1), new Pig(2)};
48          for (Animal animal : animals) {
49              animal.speak();
50          }
51      }
52  }
```

3.4.4 Lines of code

Lines of code is a common metric used for measuring the size of a project. It simply counts the number of lines of code used in program. LOC is a relevant to the quality of the code since, according to [21, p. 336], it correlates with the implementation effort and maintainability of the code. Both of which are important in a development setting.

```
1  /*
2   *LOC = 11 and NCLOC = 8
3   */
4  if (i < 0)
5  {
6          i++;
7  }
8  else
9  {
10         i = 0;
11 }
```

Figure 3.1: Code snippet of lengthy Java code

```
1 if (i < 0) { i++;} else { i = 0;} //LOC = NCLOC = 1
```

Figure 3.2: Code snippet of compact Java code

Despite the simple definition there are several choices to be made when selecting how to apply the measure. Rules regarding what to count and/or how to structure the code has a major impact on the measure. For instance do the code snippets in Fig. 3.1 and Fig. 3.2 contain exactly the same code, but due to spacing they have very different lengths. Allowing both would have an impact on the validity of the measure, as clearly the code in Fig. 3.1 is not 11 times as hard to maintain as the code in Fig. 3.2. This is especially important when comparing code across multiple languages, as their natural coding styles would differ. Below is a set of choices made regarding the counting and spacing of the code. The first 5 of these are from the discussion in [21, pp. 339-344], while the last is created for this experiment.

Ignore comments A major choice is if lines containing only comments count as lines of code or not. Whether this makes sense or not depends on the usage of the metric. For instance if measuring amount of work it makes sense to count them, as it takes time writing them. However, if measuring maintainability they are generally considered a plus and could be skipped. For this experiment we will choose to ignore them as they are unlikely to be varying much between the languages. This variant of LOC is often referenced as NCLOC, Non Commented Lines Of Code, a convention we will be using in the rest of this thesis.

Ignore blank lines Blank lines are mainly there for human readability and generally do not vary between implementations in different languages. We will therefore be disregarding lines that are entirely blank from the count.

Separation of instructions In many languages an entire program could be written on a single line by using statement separators and never newline. There has to be some scheme for when to use newline. We will solve this by requiring a newline before every atomic statement, after every loop declaration and before every if/else block.

Consistent data declarations Sometimes data declarations will be split over several lines for readability. Since the examples are made to be simple it will seldom be necessary to use more than one line in this experiment. If there are any cases where splitting is necessary it will be done over the same number of lines in all of that case's implementations. Declaration of object oriented constructs will not be considered data declaration with respect to this rule, as how well the languages handle this is one of the things we wish to measure. For such constructs we will use the rules outlined in the *Separation of instructions* point above.

Including exception handling Sometimes exception handling is not included in the count. For this experiment, whenever we do exception handling, it will be because we are interested in measuring aspects of its implementation. Therefore we will not exclude data about this from the metric.

Bracketing As seen in Fig. 3.1 the placement of brackets has an impact on the measured length of the snippet. Removing the brackets from the snippet, which uses C#-style brackets, would halve its NCLOC score. Even for a less extreme example this would have a much greater impact on the measure than it would on maintainability or effort to create. On the other hand bracketing code does have some impact on these, so completely disregarding the brackets might be wrong too. The Java-style brackets of only giving the closing brackets its own line is therefore chosen as the standard for this experiment, as it might be a reasonable middle ground.

Example Evaluating the example from Section 3.4.3 we see that there is comments on one line, 7 blank lines and 52 lines total. Therefore the NCLOC for the example is $52 - 7 - 1 = 44$. The code in the example also follows the coding style guidelines set earlier in this section.

3.4.5 Cyclomatic complexity

The *cyclomatic number* is a measure for complexity first introduced in [44]. It is applied to a program's flowgraph and for a strongly connected graph it is formulated as:

$$v(G) = e - n + p, \text{where:}$$
$$\begin{aligned} v(G) &= \text{cyclomatic number for graph G} \\ e &= \#\text{edges} \\ n &= \#\text{nodes} \\ p &= \#\text{connectedComponents} \end{aligned}$$

For a non-connected graph one can simple create edges from all nodes without an out-edge back to the start node to make it strongly connected. [44] goes on to prove that for a strongly connected graph this number is equivalent to the number of linearly independent paths in the graph.

As pointed out in [21, p. 394] the cyclomatic number is not a full complexity measure. It is based on the number of edges, disregarding how chaotically they are connected. Despite this it is still absolutely sufficient for our needs as it is still a relevant metric. A high score on it indicates potential problems with maintainability or testability.

There are many programs capable of calculating the cyclomatic complexity automatically. For most of the code measured in this text SonarQube will be used. For details on how it calculates the cyclomatic number for the different languages see [66].

Example For the example in section 3.4.3 the cyclomatic number is 9. There is a total of 7 methods giving us $p = 7$, and two branch points (one if-sentence and one loop) giving us $e = n + 2$. Then $V(G) = n + 2 - n + 7 = 9$.

3.4.6 Object Oriented Measures

It is possible to define many different measures based on properties of objects and classes in a program. A seminal work in the field is [13], which defines six interesting metrics. More object oriented metrics are discussed in [7] and [21]. For us many of these metrics would not show anything interesting, because they measure some of the same things that the design pattern specifies. Those that are usable will however be interesting in the cases where they measure a difference, as that difference would often be closely related to the object orientation scheme of the implementation language.

Depth of Inheritance Tree The *Depth of Inheritance Tree* (DIT) measure from [13] which is defined in [21, p. 422] as the length of the longest path in the inheritance graph of a program. Ideally it would measure the maximum number of ancestor classes that could affect the implementation a class in the program. As noted in [21, p. 421] this is not necessarily the case when taking multiple inheritance into account, which some of our languages are reliant on. We will therefore, in addition to using the DIT, use another metric which more directly measures this value. This metric is discussed further below in the *Number of Inheritances* section.

A question when defining the DIT metric is what counts as an instance of inheritance. Should for example inheritance from an interface be counted? Since the different languages have so many different constructs we will count inheritance widely, to catch all the cases. We will therefore count inheritance between all kinds of object oriented constructs. We will however not count inheritance from pre-defined object oriented structures as inheritance, so for instance inheriting from the Object class is not counted.

Number of Inheritances As mention in the previous section we wish to measure the maximum number of classes affecting a class through inheritance. An alteration in a parent classes could make alterations in the class inheriting from them necessary. Being dependent on few other classes is therefore positive from a maintenance perspective.

Instead of counting the longest path in the inheritance graph, like in DIT, we will instead count the maximum number of grandparents any class in the program has. That is the maximum number of nodes which has a path down the inheritance graph to the same class. For cases where there is no multiple inheritance this will usually be the same as the DIT score. We still include both the metrics in the experiment as they are easy to measure and in the cases they differ it helps illuminate the effects

of allowing multiple inheritance. We will call this measure *Number Of Inheritances* and abbreviate it to NOI.

As for the DIT we will employ a broad definition of what constitutes inheritance.

Number of Children Number of Children (NOC) is another metric from [13]. For a given node (class/interface/etc) in an inheritance tree the NOC is measured as the total number of direct children. This measure only applies to a node, but can be limited to largest NOC score for any node in the program, giving us a single number to compare. We will rename this value to MNOC, *Maximum Number of Children*. The intent is that a class with many descendants would be hard to change, as a change would have larger ramifications. Therefore having a large MNOC number is negative.

For many of the experiments the MNOC score will be likely not be that relevant. We have still included it, as the cases where it is relevant there is no other metric in our gauntlet which can measuring the same thing.

Class Cohesion Class cohesion relates to every object representing a single, cohesive entity and not several disjoint things. For us it could indicate too many helper functions or the design pattern functionality and the case related functionality not meshing well.

[13] has a measure for this, the *Lack of Cohesion of Methods* metric. This metric is however floored to a value of 0 as long as there are more cohesive methods than non-cohesive methods. As we are comparing programs based on the same design pattern we can expect the differences between them to be relatively small and often be hidden by the flooring. We will therefore instead be using the Loose Class Cohesion defined in [54]:

$LCC(C) = NDC(C)/NP(C)$, where:

$$LCC(C) = \text{Loose Class Cohesion of a class C}$$
$$NC(C) = \text{\#method pairs in C reading/writing to shared variable}$$
$$NP(C) = \text{\#method pairs in C}$$

The reason this is denoted *loose cohesion*, rather than *tight cohesion*, is that it does not require the reading/writing to be happen explicitly in the method, but also allows for this to happen in an invoked function. That is: A pair of methods is in NC if they, or a function they invoke, read or write to the same, explicitly defined, variable.

This is also a metric defined for a single class. Here it makes less sense to look at the worst-case class than it did for the NOC measure, as having several somewhat uncohesive classes is not necessarily better than having one very uncohesive class. We will therefore use an aggregated version of the measure instead:

$$LCC(P) = \frac{\sum_{C \in P} LCC(C)}{Count(C \in P)}, \text{ where P is a program.}$$

Lastly we will extend the definition of class to be any of the object oriented constructs defining methods, to make the measure meaningful for all the languages used in the experiment.

Examples The example in Section 3.4.3 has $DIT = 2$, NOI = 2, $MNOC = 2$ and $TTC(P) = 1$. The DIT is 2 as the longest inheritance chain is $Animal \rightarrow Canine \rightarrow (Dog \text{ or } Wolf)$, which has two edges. The NOI = DIT = 3 as there are no multiple inheritance. The MNOC is 2 as no class or interface has more than two children. The LCC(P) is one as every method (including the constructors) read or write to the sound variable.

3.4.7 Shallow References

A difference between the implementations will be the number of referable entities created and how many of these are publicly accessible. There are positive and negative sides to this. Having too many globally accessible variables can lead to problems in larger projects regarding alterations to variables that should not be altered or usage of variables through an unintended interface. On the other hand it can be nice to have easy access to whichever variables are needed.

One measure for this would simply be the number of self-defined elements found in the global namespace. This would however not capture differences in visibility of attributes in these elements. We therefore include these in the namespace as well, but with a reduced weight. Similarly we also include the namespace of these objects with an even further reduced weight. While we could go deeper than this it would at this point get hard to count, so we stop at a depth of 2. We will call this measure *shallow references* and define it as:

$$SR = \sum_{i=0}^{2}(\frac{count(d_i)}{2^i}, \text{ where}$$

$$d_0 = \{a | a \textit{ in global namespace}\}$$
$$d1 = \{a | a \textit{ in namespace of } e \in d_{i-1}\}$$

We are using namespace a bit weird here, but it gives an intuitive feel to the metric. We will consider an element to be in the namespace of an object oriented construct if it is a user-defined method or attribute of that construct. We will also consider an object to be in the namespace of a collection if it is directly accessible from that collection, so for instance a[i] would be in the namespace of a. Lastly we have to define what we mean by global namespace. We will consider this to be all named entities accessible at the end of the program or the main method. We will, as mentioned earlier, only count objects defined by the user and not pre-generated methods or objects.

It is worth noting that this metric has no normalization for the length of the program. It is therefore not suited for comparison between the cases, only for comparison of the implementations within a case.

Example: We will apply this metric to the example from Section 3.4.3. $d_0 = 7$, as all the object oriented constructs are accessible in addition to the animal array. $d_1 = 8$ as the animals array contribute 4 and the only static methods in the classes are the three constructors and the main function. Lastly $d_2 = 16$, as there is one of each animal in the animal array and the total number of public methods/attributes for the animals is 4. The rest of the elements in d_1 are functions and therefore don't contribute anything to d_2. Combining this we gain $SR = 7 + \frac{8}{2} + \frac{16}{4} = 15$

3.4.8 NCLOC aggregation

It would for most of our metrics be possible to aggregate some of the values, to make the data easier to understand. We will however primarily do this for the NCLOC metric, as it handles such aggregation well. The adding the code length for a two programs would often give a comparable score to the code length of a larger program having the functionality of both programs. The same could not be said about, for instance, the depth of the inheritance tree. It would be the maximum of the two programs. The NCLOC metric is also the metric I expect will most often give a significant difference, which makes aggregating it even more tempting.

Portion of Language's NCLOC For a language like C# to consistently give longer implementations for a case than the Python implementations would not be unexpected. However, sometimes it will be much longer and sometimes only a little longer. To make this more easily readable we introduce the aggregate measure *Portion of Language's NCLOC*, which we will shorten to PLOC and define as:

$$PLOC(C, L) = \frac{NCLOC(C, L)}{\sum_{\hat{L}} NCLOC(C, \hat{L})}, \text{ where}$$

$$NCLOC(C, L) = \text{NCLOC score of case C in language L}$$

Portion of Case's NCLOC We can also aggregate the other way. How much of a case's total NCLOC is from a certain implementation? Given that we will usually compare this score between the languages after computing it there is not a lot of practical difference between this and the PLOC score. However, since it could give a different perspective to the data and is easy to compute we will still include it. We will shorten it to CLOC and define it as:

$$CLOC(C, L) = \frac{NCLOC(C, L)}{\sum_{\hat{C}} NCLOC(\hat{C}, L)}$$

Relative Size of Case We can also aggregate with respect to design patterns. Which cases require most code across all languages to implement?

We can call this *relative size* (RS), as it indicates the size of the code for implementing a case. The formula for this would be

$$RS(C) = \frac{\sum_{\hat{L}}(NCLOC(C, \hat{L})}{\sum_{\hat{L}} \sum_{\hat{C}} NCLOC(\hat{C}, \hat{L})}$$

Relative Verbosity of Language Similarly we can aggregate as above with respect to languages giving a single number representing how good the language is at efficiently expressing design patterns. This would of course be a simplification, but since it is so central to the core question of the thesis it would be wrong to not include it. We will abbreviate it RV and define it as:

$$RV(L) = \frac{\sum_{\hat{C}}(NCLOC(\hat{C}, L)}{\sum_{\hat{L}} \sum_{\hat{C}} NCLOC(C, L)}$$

Case/Language	Pascal	Java
Case 1	100	150
Case 2	200	250

Table 3.1: Mockup data for exemplifying aggregation

Example In this section we will show examples of the aggregates applied to the data from Table 3.4.8.

$$\begin{aligned}
PLOC(\text{Case 1}, \text{Pascal}) &= \frac{100}{100+200} = \frac{1}{3} \\
CLOC(\text{Case 1}, \text{Pascal}) &= \frac{100}{100+150} = \frac{2}{5} \\
RH(Case1) &= \frac{100+150}{100+150+200+250} = \frac{5}{14} \\
RV(Pascal) &= \frac{100+200}{100+150+200+250} = \frac{3}{7}
\end{aligned}$$

3.4.9 Code as Data

While the above metrics might be illuminating when discussing the quality attributes of the code, there is also a goal for this thesis to discuss the language constructs causing these differences. To do this without looking at which constructs are used in the code would be quite challenging. The code is therefore presented in full at https://github.com/KristianBPedersen/MasterThesis and referred to in the analysis. It is used in the context of the

other metrics to explain what causes their differences. Due to the similar nature of the implementations connecting the differences with the exact places they occur can be done relatively reliably, but one does have to note that this might be a cause for some biases.

3.5 Validity

3.5.1 Introduction

In this section we will discuss the choices made in the definition of the method with respect to the impact they have on the validity of the data produced. In Section 3.5.2 validity is defined. Then, in Section 3.5.3, the need for validity measures in this paper is motivated. In the following sections a series of such measures is discussed, before further threats to validity is discussed in Section 3.5.12.

3.5.2 Definition

A valid measure is defined in [21, p. 37] as a metric fulfilling the *Representation Condition*, which it defines as:

> A measurement mapping M must maps entities into numbers and empirical relations into numerical relation in such a way that the empirical relations preserve and are preserved by the numerical relations.

The validity of the measure is how close the measure is to being a valid measure.

3.5.3 Need For Validity

Bad data gives bad conclusions. There is therefore a need to put measures in place to ensure the data gathered from the experiment has value. If the code created has no relation to code created in a real-life setting, then no model or metric can extract information with real value from it. We would come no closer to finding answers to the problem statement.

As discussed in Section 3.5.12 there is a limit to what can reasonably be achieved. We will however, in this section, specify some measures to increase the validity of the experiment. The goal is to set some explicit rules and guidelines for how to select and specify cases and how and what to implement. By doing so one eliminates some of the bias caused by an implementer making these choices on the fly and some of the bias caused by the unnatural context.

3.5.4 Design and Implementation Philosophy

There are many ways to design a case for any given design pattern. By what criteria will the case used in the experiment be chosen? The goal will be two-fold. Firstly the goal will be to design an intuitive representation

of the design pattern, in order to make the discussion and analysis easier and more readable. Secondly the cases will be designed to investigate differences between the languages. The goal will be to have the languages show off their differences, which, as discussed in the essay, will not always occur if a minimalistic case is chosen.

Similarly there are many ways of implementing a given case in a language. Giving precise rules for everything that can be allowed and not is not a good idea. It would require a massive framework to be feasible and even then probably give some rulings that are worse than common sense. Later in this section some specific rules are set, but in general it would be better with an overarching philosophy. The main idea will be to, when an unanswered question arises, to think of it in the context of a real life situation. This is chosen situation is that of a small to medium sized project that is intended to be judged by others using both code review and the metrics described in Section 3.4. The goal is for this to strike a balance between a solution optimized with respects to the metrics and realistic code. It means very *hacky* solutions should be avoided and that, when considering several possible solutions, no combination of metric should be prioritized at the cost of all other things.

3.5.5 Well defined requirements

A problem, discussed in Section A.2.5.2 of the essay, was that the lack of clarity regarding which parts of the case are obligatory and which are flexible. It is a trade-off. Too rigid requirements might lead to many of the special aspects of the languages not getting used, while too flexible requirements might allow for shortcuts which leads to something fundamentally different than the intended case being implemented. Either way it is beneficial to define the limits of what is required explicitly, so that it does not vary from implementation to implementation.

For this experiment we chose to use UML class diagrams with core functionality annotated in psuedocode as the basis for the requirements. Only that which is explicitly modeled in the diagram is required. This was chosen over a description using pure pseudocode as it is more abstract and therefore is easier to make language independent. Another possibility would be to use a textual description as the basis, but natural language can be fairly ambiguous and a textual model could impart too much structure. For choices regarding modeling see Section 3.5.6.

In addition to the diagram there is for every case a usage example described in pseudocode with an associated description of output. The implementations of a case must contain an implementation of its usage example and it must generate the desired output. For choices regarding the pseudocode see Section 3.5.10

3.5.6 Model Design

Given that the UML class model is used for requirement specification rules should be set for how it is designed. I will start with the model for a given

design pattern from [28]. I will then extend it to describe with details for the chosen case and specify the implementation.

Conformance to an interface or abstract class will in general be implemented using the *realizes*-relation rather than for instance the *inheritance*-relation. This allows for the usage of structural typing and duck-typing, as discussed in section A.2.5.4 of the essay.

Collections can often be abstracted using the association-, composition- or aggregation relations with multiplicity. This gives a lot of flexibility to the implementation with respect to the choice of container structure.

3.5.7 Visibility Relaxation

The class model specifies the visibility of its fields and methods by plus and minus signs. As discussed in section A.3.2.7 of the essay, the languages used have very different handling of visibility. JavaScript, Go and Python does by default not support private values, while Smalltalk requires everything to be private. C# is flexible.

Since it is felt that using workarounds to circumvent these restrictions goes against the spirit of the languages we will only use the visibility restrictions allowed for in the default version of the standard OO construct in the language. In cases where private fields are not supported we will therefore make it public and in cases where public fields are unavailable we will use setter and getter functions.

3.5.8 Order of implementation

When performing several similar tasks in a row the order they are performed might have an impact on the performance of the task. For instance when writing the code for the experiment in the essay I always implemented in Python and C# first, as these were languages I knew well. When implementing for the other languages later it was tempting to reuse elements of these solutions even when they did not fully fit that language, leading to suboptimal solutions for these languages. On the other hand, when implementing the case for the first time, I did not necessarily find the optimal solutions to all problems.

As mentioned in section 3.3 the order which language to implement in first is varied between the cases. According to [62] a common technique to use in experiments with multiple developers is to assign different orders to different developers. This is not doable in this experiment, becasue there is only one developer. We will instead vary the implementation order between the cases.

3.5.9 Agile selection of patterns

The patterns studied will be selected iteratively. A case will be implemented, metrics from it will be gathered, analyzed and then the next pattern will be chosen. This is done to decrease the chance of pointless experiments being performed. As stated in Section 3.5.4 the goal is not to create

the simplest possible case, but rather to create cases where one could expect differences without exceeding the limits of the pattern. Selecting cases in this way hopefully increases the conformance to this goal.

3.5.10 Pseudocode

Pseudocode is code designed for human reading, rather than being interpreted by a computer. It allows for expressing functionality of a program compactly, ambiguously and/or language independent. There are several *dialects* of psuedocode. Due to their nature they are seldom a rigorously defined languages, but rather a collection of some fundamental constructs and an overall choice of style.

In this experiment the pseudocode is mainly used for describing usage examples and as supplementary notes in the UML diagram. For the first kind we will use the language specified for the pseudocode environment of the algorithmicx package for latex, as described in [36, p. 5]. We will extend it with allowing the creation of objects of a class with the new keyword, as well as invoking methods of objects using dot notation. Even with these modifications it is still a fairly simple language, with few allowed constructs. Since the goal of using pseudocode here is to have a language independent description, rather than making it more compact, this suits the usage examples well.

For the second case brevity is more important, as too much text might damage the readability of the model. It is also hard to get any markup on the language with most modeling tools. Therefore a simpler and more intuition based language is used. It uses Python like loop structures to reduce the amount of text, but unlike Python it is typed. This is done as specifying type is something I think I will often want to do.

3.5.11 Adapting Metrics to Go

For most of the languages studied in this text the application of the metrics from Section 3.4 is trivial. This is not the case for Go. There are two aspects of Go which makes it a bit more trickier: The built in constructors and the structural typing. In order to make how the metrics are evaluated more consistent and transparent we set rules for the measurement and implementation in the experiment with respect to both.

Constructors

Go does not support constructors functions natively as most other languages do. Instead any struct in Go can be initialized by the notation on the form StructName{attribute1, attribute2...,attributeN}. In practice it is however common to have as function named NewStructName which acts as a constructor function would in most other languages. Should this be done in the experiments? On one hand the simplicity of the structs is a feature of Go and the benefits of it should be showcased. On the other hand, when the object instantiated is complex, it can lead to long initialization

expressions that would never be used in practice. In order to get the best of both worlds the following rule for constructors were formulated:

> If a constructor function would consist of only a single statement returning a struct initialized using only its arguments, then that function can be skipped and a struct initializer can be used in its place.

Structural Typing

Many of the metrics defined in 3.4.6 are based on counting instances of inheritance. What does this entail with respect to Go's interfaces and its structural typing. Can a struct be said to inherit from an interface if it conforms to it? From a purely theoretical perspective, I suspect it would be a stretch to say that it does. However, in the context of the metrics using this I would say the opposite is true. The metrics are based on the idea that inheritance creates a dependency between the type and the subtype. An alteration in one might lead to requiring an alteration in the other. This is also true for conforming to an interface. If the interface changes, then the implementer will often require additional methods to comply to it. Therefore we will treat interface conformance for Go as inheritance.

3.5.12 Further threats to validity

There is fundamentally a human element in this kind of experiment. The cases have to be modeled by a human, the implementations has to be done by a human and a human has to measure and extract the data. Earlier in this section we have introduced measures to alleviate the problem, but it is unrealistic to attempt to fully eliminate it. Firstly due to the size of the framework that would be required and secondly due to fundamental trade-off between control of variables and maintaining a realistic setting [21]. In this section we will list the biggest factors that remain unanswered.

Suboptimal Solutions Every developer makes mistakes while coding. While one can be fairly certain one has found a solution to a problem, for instance by running tests, one can seldom be completely sure. To know that one has found an optimal solution is even harder.

Experience Difference As a developer I do not have equal skill in the languages used in the experiment. This might cause an unfair bias toward bad scores for Smalltalk and Go, the two languages I have the least experience with. My experience is discussed in Section 3.2 and as mentioned there I have spent some time educating myself on the language to remove bias. It is still a factor that might have some impact.

Case Design How the cases are designed is relevant is almost guaranteed to be relevant to the outcome of the experiment. Some guidelines on how to define the cases are laid out in Section 3.5.4. Even so,

finding cases that are both interesting and *completely fair* will be nearly impossible. It is not even clear what being *completely fair* would entail in this context.

Code As Data While the metrics form a basis for the data, one cannot expect to learn why languages behave like they do purely from studying the numbers. In the analysis section it will be necessary to also treat the code like data and correlate it with the metrics. This is not a fully rigorous process and it will probably entail some mistakes and biased selection from the code.

Metric Validity Do the metrics, in the measurements made as part of the experiment, measure what they are intended to? Do the setting of the experiment make them correspond worse to the quality attributes they are meant to represent? This question is best studied in the context of the data and the designed cases and is therefore relegated to be discussed later, in Section 6.4.

Chapter 4

Cases

4.1 Introduction

In this section the cases are presented. For every case the pattern is briefly described, a textual and modeled description of the case is presented, the results for the implementations of the case are presented and analysis for the case is performed. One can note that this breaks with the structure of the rest of the thesis, where analysis and method are kept strictly separated in different Parts. The cases represent mini experiments and this structure better reflects that. It was felt this structure would be more intuitive for the reader and that it would be easier to understand the analysis if presented in the context of the rest of the case.

4.2 Composite

4.2.1 Pattern

The composite pattern is applied in settings where it is natural for a client to both interact with objects composed of smaller parts and the individual parts composing these objects. It simplifies this by requiring the parts and compositions to have a uniform interface. It structures this as a tree of objects conforming to the interface, with the leaves being the atomic parts. Calls to methods are handled by composites by passing them down their subtrees, until it reaches the leaves, which handle them.

4.2.2 The case

In this case we are modeling a simplified design for calculating the maintenance cost of a subway network, its sub-networks or the individual structures composing it. As there is no need for realism this cost is measured in integers of an unspecified unit. The modeled objects will be:

NetworkElement Represents an element in the subway network. Either a network or a physical part. It plays the role of Component in the pattern

Network Represents a part of the subway network. Usually consists of several children, which are other networks or structures. It can add, remove or get children and it can calculate its maintenance by summing the cost of all children. It plays the role of Composite in the pattern.

Railroad Represents a section of railroad. Has a variable length and a maintenance cost of 2 times its length. It plays the role of Leaf in the pattern.

Tunnel Represents a section of tunnel. Has a variable length and a maintenance cost of 20 times its length. It plays the role of Leaf in the pattern.

Station Represents a subway station. Has a fixed maintenance cost. It plays the role of Leaf in the pattern.

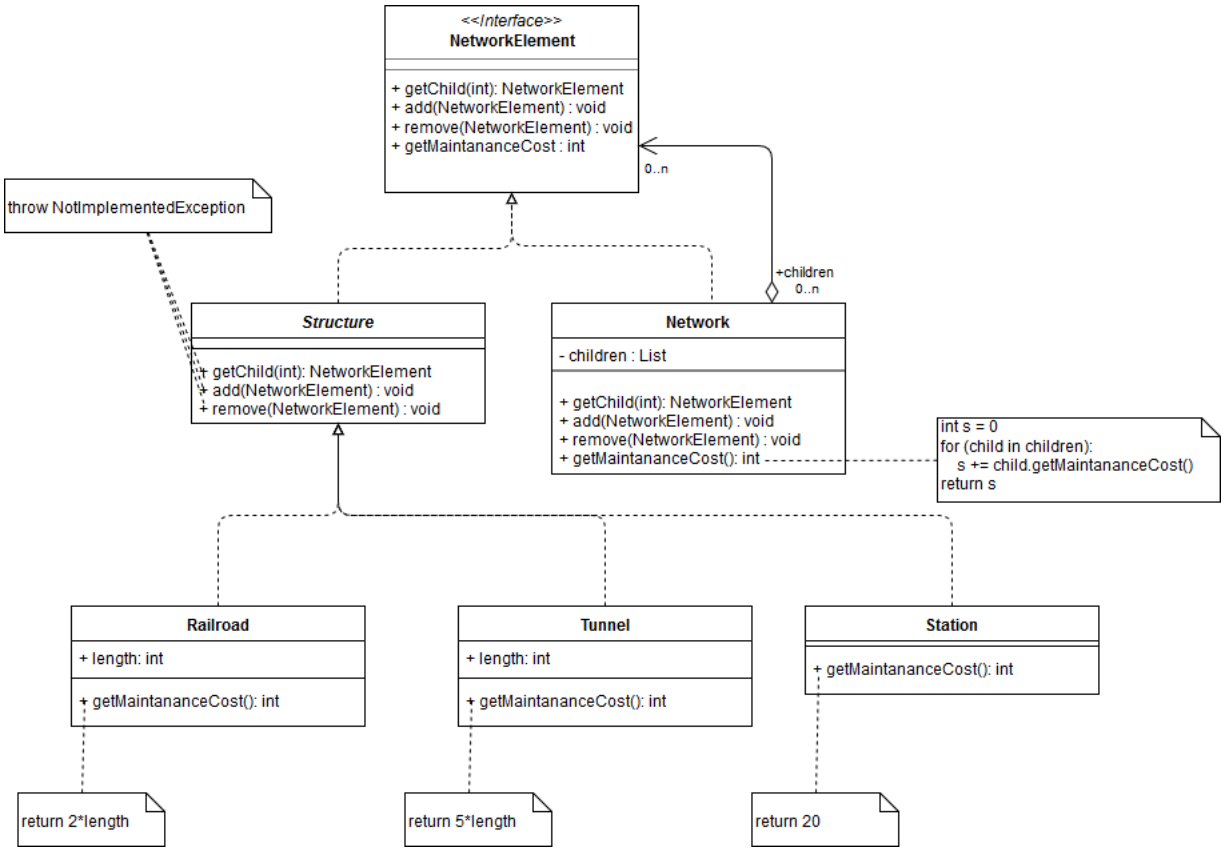

Figure 4.1: UML diagram for the composite case.

Algorithm 1 Usage example for Composite pattern

```
tunnelNetwork ← new Network()
tunnelNetwork.add(new Tunnel(5))
tunnelNetwork.add(new Railroad(5))
stationNetwork ← new Network()
stationNetwork.add(new Station())
stationNetwork.add(new Railroad(10))
bigNetwork ← new Network()
bigNetwork.add(tunnelNetwork)
bigNetwork.add(stationNetwork)
print(bigNetwork.getMaintananceCost())          ▷ Expecting: 75
```

4.2.3 Usage example

4.2.4 Choices

Allowing child management for leaves One of the major choices in the application of this pattern is whether to give the leaf elements the child management functions or not. For most applications it would not make sense to let the leaves be composed of other objects, so if they have the child management functions they would not be meaningful to invoke. [28] describes this as "a trade-off between safety and transparency". Giving them the management functions increases transparency as it allows all component to be treated uniformly by the client, but not doing so increases safety as the client is not allowed to invoke meaningless methods. In [28] the default pattern is described with children having the management functions. For this case we will model it in the same way. Partially because this was the choice in [28] and partially because it might allow for showcasing some differences between the languages.

Throwing errors What should happen when the leaf's child management function is called? [28] suggests several solutions. Nothing could happen, they could delegate the call to their parent or they could throw an error. For this case we will model it using the last choice, as it allows us to study how error throwing affects the implementation.

No explicit parent reference The case is designed without the need for sub-networks or structures to know which network they belong to. That is the children in the tree hierarchy have no parent references. Use of these are discussed in [28], with one of the main upsides being to facilitate for communication up the tree, allowing for things like combining it with the Chain of Responsibility pattern. We will not be requiring them for this case. They would increase the size without giving any obvious compensation.

Language/Metric	NCLOC	Cyclomatic	DIT	NOI	MNOC	LCC	SR
Python	41	15	2	2	3	0.28	18.5
C#	72	14	3	3	3	0.28	17.5
JavaScript	50	17	2	2	3	0.28	17
Go	80	15	2	3	4	0.24	20
Smalltalk	85	18	2	2	3	0.31	17

Table 4.1: Results for the composite case.

4.2.5 Results

4.2.6 Analysis

Non Comment Lines of Code

Python scores best for this metric, with JavaScript being a close second. Python scores well due to not having brackets and JavaScript does so due to being able to define all three methods of Structure in a single statement, since they are identical. The JavaScript implementation employs prototype notation, which makes the constructor statements more compact and the inheritance slightly longer compared to Python's class notation. Both benefit from duck-typing allowing for NetworkElement to not be explicitly defined. In both the languages Network.getMaintananceCost() is reduced to a one-line statement. JavaScript does this with a reduce statement, while Python uses its sum function on a generator statement.

C# scores 50% higher than JavaScript. Its code is fairly similar to the Python and JavaScript code with the exception of the above-mentioned tricks. The only one of them it is able to employ is a summing function to make Network.getMaintananceCost() a one-liner. To do so it does however have to import functional programming, so it is a little less efficient in this regard. It does also have to declare the types of all its class attributes, costing some extra lines.

For the most part, despite the two being structured relatively differently, the individual pieces of Go's code is recognizable in the C# implementation and are of comparable length. The exception to this is the remove and getMaintananceCost methods of Network, which are both longer than their c# one-liner counterparts. The first of these differences is due to Go not having a List.removeByIdentity method the second is due to Go not naturally supporting functional programming. In addition the import statement in Go is a bit longer than in C# and defining composition is done on a separate line from the struct definition. All of these small things add up to Go scoring worse than C#.

Smalltalk has the longest code. The most impactful reason for this is that it lacks a constructor statement. It has to be manually created from a static- and a dynamic method. This makes a two-line constructor in Python require 6 lines in SmallTalk. If not for this, and the need for a loop in Network.getMaintananceCost(), Smalltalk would probably score better than C# in the NCLOC metric. Like Python and JavaScript, Smalltalk

benefits from duck-typing to avoid explicitly defining NetworkElement explicitly and it does not need to define a main method like C# does.

Overall there is more than a 200% difference between the outlier scores for this metric. Concise formulation of basic statements and the availability of built-in functions are the most common reasons for a score difference.

Cyclomatic Complexity

There is, for all the implementations except Go's, only one branching point in the code: Whether to continue the loop in Network.getMaintananceCost() or whether to stop. For those implementations the cyclomatic complexity is reduced to the number of functions defined + 1.

For Python, C# and Go this number is fairly similar. C# does not need a constructor for Network as Python does, since it can initialize its children attribute without one. Go has the same advantage on Python, but due to having an extra loop and an if-test in its Network.remove method it still scores worse.

JavaScript scores highly because it must always define a function for every prototype it defines. In cases where the other languages require a constructor this evens out, but otherwise this increases its relative score by one.

Smalltalk scores the highest for this metric. This is due to it requiring two function definitions for its constructors. Since there are three constructors in the case this increases the score from 15 to 18, and this is therefore the only reason it scores higher than Python or Go.

Depth of Inheritance Tree

For all the languages one of the longest chains of inheritance is NetworkElement $\rightarrow$ Structure $\rightarrow$ Tunnel. Python, JavaScript and Go does not implement NetworkElement due to duck-typing and therefore scores only two. Go has a flatter structure, as Structure does not inherit from the NetworkElement interface, and therefore also scores 2.C# does not employ any such tricks and therefore scores 3.

Number of Inheritances

Due to there being no multiple inheritance the scores are mostly unchanged from the Depth of Inheritance Tree measure. The exception is Go, which, with its flatter inheritance structure, has a score of three instead of two.

Maximum Number of Children

For C#, Python, JavaScript and Smalltalk the Structure construct has the highest number of children: 3. For Go the NetworkElement interface is directly implemented by Network, Tunnel, Railroad and Station structs, and therefore scores one higher: 4.

Loose Class Cohesion

All of Structure's methods are non-cohesive with respect to each other or any functions defined in its children. Therefore the combined contribution to score from Structure and its children is fairly low. Network, on the other hand, is perfectly cohesive as every function reads or writes to the children variable. For C#, Python and JavaScript this means the cohesion score is $\frac{1+\frac{6}{15}}{5}$ as these are their only structures defining methods.

Smalltalk and Go scores a little different due to their constructors. Go avoids defining some, while Smalltalk has to define two constructor methods for every class with a constructor. Since the constructors are more cohesive than the average method this benefits Smalltalk's score and diminishes Go's score.

Shallow References

For this metric Smalltalk and JavaScript scores well. They both benefit from not having to define NetworkElement due to their typing rules. In addition JavaScript gains a bit from not having to define both constructor functions and classes, while Smalltalk gains a bit from having private variables. C# also has private variables, but scores a bit worse due to needing a class to wrap its main function in.

Python scores a bit higher. The only source of difference between its implementation and JavaScript's is the way classes are defined. Python does it the standard way, so for classes with an initializer JavaScript scores better on this metric.

Lastly there is Go. It has the advantage of only needing to define one constructor, but the fact that it is defined at top level rather than as a method hurts it. It also has the issue of not being able to define private methods like C# and Smalltalk, while still having to define an interface for NetworkElement and a main method.

4.2.7 Summary

There were no clear best language for this case. For many of the metrics the way classes were defined and how their constructor worked were key, due to the implementations needing to define of so many classes. Being able to avoid explicitly defining the NetworkElement interface and handling the Network's child mechanics were also regularly important. Interestingly there was little to no difference stemming from the handling of exceptions. All languages handled it equally well.

4.3 Prototype

4.3.1 Pattern

The Prototype pattern is a creational pattern. It gives objects the ability to clone themselves. Thus allowing a client to specify what they want to

create by instantiating an object and cloning it.

4.3.2 Case

In this case we are modeling the mass production of trains. Products can be either locomotives or trains, both having fairly different attributes. Once defined they can be cloned, to produce more (deep) copies of the product. The modeled objects are:

Product Has a getCost method and is cloneable. It plays the role of Prototype in the design pattern.

Locomotive Is a product. In addition it has a price, color and max speed. It plays the role of ConcretePrototype1 in the design patten.

Train Is a product. Consists of a locomotive and some wagons represented as strings. Its cost is equal to the locomotives cost plus 5 for each wagon. It plays the role of ConcretePrototype2 in the design pattern.

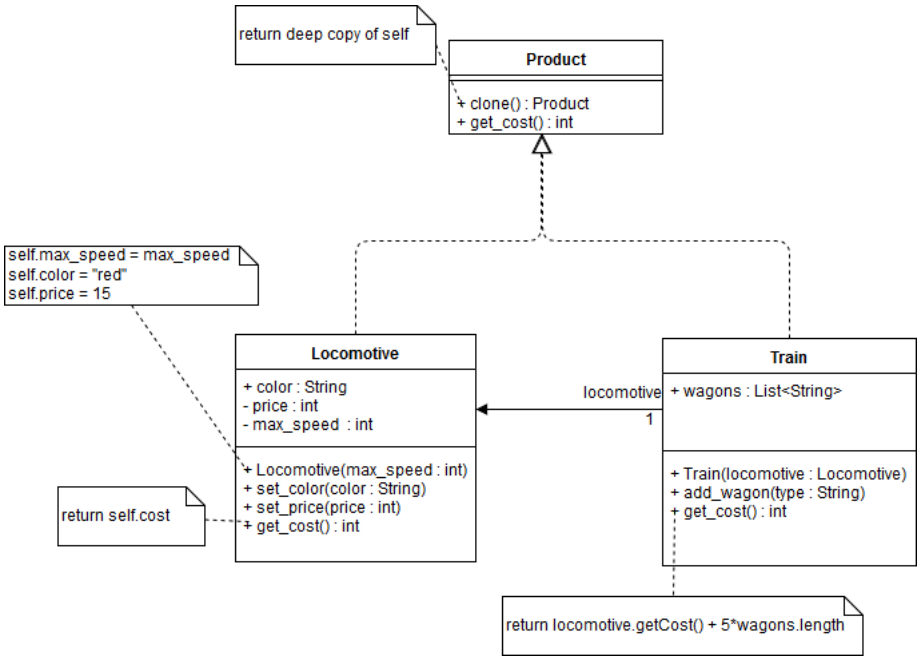

Figure 4.2: UML diagram for the prototype case.

4.3.3 Choices

Deep cloning The prototype pattern is applicable for both shallow- and deep cloning. That is when cloning an object with another object as an attribute one can either let the clone have the same object as an attribute, in which case it is shallow cloning, or one can create a copy of the object and let the clone have it as an attribute. In this case we

will elect to do the latter, deep cloning, as it is harder. The hope is that this will showcase more differences between the languages.

No client There is no specified client object in the case, as there is in [28], where it is defined as an object with an operation cloning a prototype. Since such an object can be structured as just about anything the choices related to specifying it probably overshadow any data related to the pattern. In a way the test serves as a client, but as we are modelling the test completely separately from the rest of the case it is not displayed in the diagram.

Weak constructors If one had constructors capable of perfectly creating any state of the prototype object, then creating a one-line clone operation would be as simple as invoking the constructor with the right input. This would be a one-line operation in most languages. While there might be some differences in the constructor the overall case would probably not yield much interesting data. Therefore the case requires the definition of an inflexible constructor and the test case requires the use of several non-constructor methods to specify the objects.

Choice of attributes The goal is to have attributes one could expect to be treated differently when cloning. Therefore we model one integer given in the constructor, one public string, one private integer, one public list of strings and one public object supporting the clone operation.

Locomotives are not trains From a pure modeling perspective it might make sense to structure the inheritance hierarchy differently. A locomotive is a train without any wagons. As such it would perhaps be cleaner to merge the classes into one *Train* class. However this version of the case allows us to study the differences caused by the inheritance scheme. It is also common for the prototype pattern to be applied in cases where merging the classes does not make sense. Therefore we will not be doing it in this case.

4.3.4 Usage example

4.3.5 Results

Language/Metric	NCLOC	Cyclomatic	DIT	NOI	MNOC	LCC	SR
Python	32	9	2	1	2	0.622	24.25
C#	59	10	2	1	2	0.883	18.5
JavaScript	41	10	1	0	0	0.933	22.25
Go	57	10	2	1	2	0.933	26.25
Smalltalk	56	13	2	1	2	0.619	18

Table 4.2: The results of the prototype case.

Algorithm 2 Usage example for the prototype pattern

```
oldLocomotive ← new Locomotive(10)
newLocomotive ← oldLocomotive.clone()
longTrain ← new Train(oldLocomotive)
longTrain.add_wagon('passenger')
shortTrain ← longTrain.clone()
longTrain.add_wagon('freight')
oldLocomotive.set_price(7)
print(newLocomotive.getCost() - oldLocomotive.getCost())
                                                    ▷ Expecting: 8
print(shortTrain.getCost() - longTrain.getCost())
                                                    ▷ Expecting: 3
```

4.3.6 Analysis

General

The requirement which makes this case special is the clone function, which creates a deep copy of the products. In Python and Smalltalk this is trivially implemented using built in deep copy functions. In JavaScript it is possible to do a mostly deep copy using a trick of converting to textual representation and back again. Unfortunately this does not copy functions, so some additional lines are required to add them. A similar trick is available in C# using data streams. It is however much more complex than its JavaScript one-liner equivalent, so I chose to not use it and instead use the built in shallow copy function *memberwiseClone* and manually creating a copy of the array and class in the Train class. Go has no deep copying functionality and is therefore also implemented using shallow copying and manually copying and setting attributes.

Non-Comment Lines of Code

Python's and JavaScript's code is the shortest again. Partially this is due to them, as discussed in Section 4.2.6, having a relatively space-efficient notation. Python scores lowest because its clone operation is so efficient. It is also general enough to implemented as a method for Product, so it only has to be defined one time, while JavaScript has to define it for both Locomotive and Train. Doing this does however mean that the Python implementation has to define a Product class, which the JavaScript implementation can use duck-typing to avoid.

C# and Go have similar length in this example. This is due to their implementations being very similar, only ordered a bit differently. They can be matched line for line, with the exception of the C# code being two lines longer due to requiring a class wrapper for its main function. Smalltalk is of about the same length as these, but is so despite its very efficient clone operation. This is due to its relatively lengthy constructor definitions, as discussed in Section 4.2.6, and it, by the rules from Section 3.5.7 needing to define a getter for color.

Cyclomatic Complexity

There are no choices or loops in the implementation. Therefore the cyclomatic complexity is reduced to being only the number of functions defined. For Python this is one less than C#, Go and JavaScript as it only defines the clone operation one time. Smalltalk has the same advantage, but due to its double constructors and the need for a getter function for color it still scores worse than the other languages.

Tree Related Metrics

For all the languages except JavaScript the inheritance tree looks the same. Product has two children: Train and Locomotive. Therefore they score the same in the tree related metrics. For JavaScript the structure is made completely flat due to duck-typing. There is a depth of 1 and no inheritance or children.

Loose Class Cohesion

For most the implementations the Train class is perfectly cohesive and the Locomotive class is almost perfectly cohesive (the set_color method not being cohesive with the price related methods). For C#, JavaScript and Go this is all the classes, so they therefore score very close to 1. C# scores slighly lower as it initialized the array in Train outside the constructor, making it non-cohesive with the add_wagon method.

Python and Smalltalk does also have a Product class. This reads or writes to no explicitly defined variables, so it is non-cohesive. Therefore these two classes score closer to two thirds.

Shallow References

C# and Smalltalk scores best here, due to having private variables. It is extra efficient in this case due to the large number of variables defined in the usage example, the large number of fields in Locomotive and the attributes of Train having more attributes in their namespace. Smalltalk scoring slightly higher than C# . It gains some from not having a main function or class, but loses most of it to having both a static- and dynamic method for its constructors.

JavaScript scores better than Python. This is due to its prototype notation making classes and constructors definable as the same thing and its use of duck-typing. Go scoring slightly worse than Python- This is due to its constructors being defined as functions at the top level and its reliance on a main function.

Summary

We see that, going by the metrics, there are two intuitive candidates for the best implementation: Python's and JavaScript's. Python's implementation scores very well on the complexity and size measure, while JavaScript

scores well on the object-oriented metrics. It is worth noting here the main reason for JavaScripts good scores is the use of duck-typing. Python could have done the same, but it was considered not worth the trade-offs.

Among the rest of the languages Smalltalk and C# both stand out due to their low Shallow References score. It is interesting that C# scores as well, if not better, than Smalltalk outside of this, despite Smalltalk having the best solution of any language to the core (deep cloning) aspects of the solution.

4.4 Adapter

4.4.1 Pattern

Adapter is a structural pattern. It applies to situations where you need to create an object that has the functionality of a class, but conforms to the interface of another.

4.4.2 Case

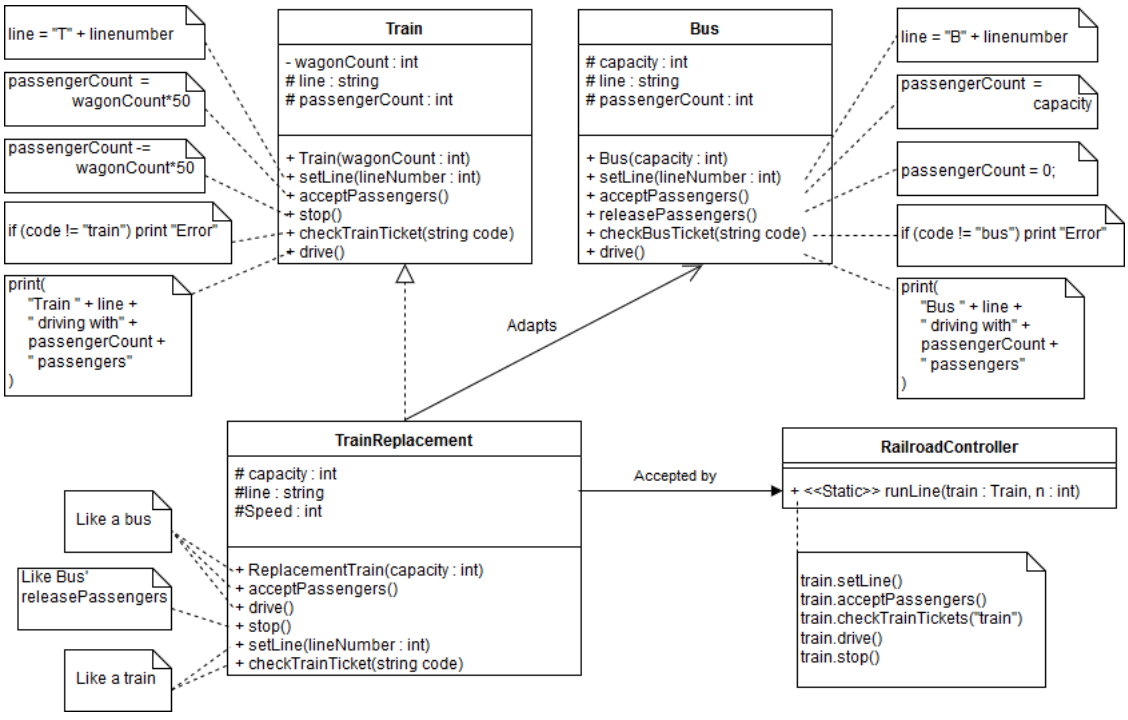

Figure 4.3: UML diagram for the adapter case.

For this case we are modeling a bus playing the role of a train. It is modeling the situation where, when a train or a part of the railroad is faulty, one temporarily uses a bus to transport the passengers. The replacement physically functions like a bus, while accepting train tickets and running the train's line. The objects involved are:

RailroadController Represents the railroad network. Has a method controlling the process where a train is running a line between two stations. It makes the train set its line name, accept passengers, register tickets, drive and then stop. It plays the role of Client in the pattern.

Train Does all a railroad controller expects. Uses wagon count to determine the capacity it has for accepting passengers. It plays the role of Target in the pattern.

Bus Does many of the same things as Train, but does them slightly differently. It uses the prefix B for line-names, directly accepts capacity for accepting passengers, has the function checkBusTicket() instead of checkTrainTicket() and has a differently named stop function. It plays the role of Adaptee in the pattern.

TrainReplacement Conforms to the Train interface, sets line-name and checks the ticket like a Train. For all other methods it has the same implementation as a bus. It plays the role of Adapter in the pattern.

4.4.3 Choices

Multiple inheritance focus The case is designed to be solvable using multiple inheritance. There are many shared function names between the interface of the adapter and the adaptee. Using multiple inheritance one could extend both Train and Bus and create the TrainReplacement mainly by smartly selecting which of their methods to inherit. There is however the issue of the stop method being differently named for Train and Bus and the fact that the inheritance scheme does not always favor methods from the Bus class over the Train class. This means the case is not completely optimal for multiple inheritance. One would however expect that if implementing using multiple inheritance yields little for this case, it would not often give good results in practice.

Many functions To see how the different languages handle different challenges I wanted at least one function that had to be renamed, one function that only existed for the adapter super-class and one function that is inherited from the adapter super-class, but also exists with the same name for the adaptee. To make sure the effects of the implementations handling of these special cases don't overshadow the handing of the base case (of inheriting directly from the adaptee) we add two methods corresponding to this case. The cost of this is that there are many defined methods, removing some of the simplicity of the pattern it is based on and making the case less of a representative for the boilerplate version of the adapter pattern.

No Vehicle class One could imagine modeling a shared interface for Bus and Train, which could for instance be named Vehicle. While this would make sense from a modeling perspective it is not obvious what

it would give the implementations except more complexity. Therefore it has not been done.

Short usage example Given that the client (RailroadController) is so fleshed out, there is little need for a long usage example. Initializing the replacement train and invoking the clients main method is sufficient.

Requiring calling of functions Given no restrictions the optimal solution, for many of the implementations, would be to copy the code from the bus or train class rather than calling a function containing the code. This would create a worse solution since a change in the code would have to be implemented in two places. In addition it would scale much worse with function length. This example purposefully has short functions in order to make it more readable and quicker to implement. In a real-world setting duplicating code would therefore not a good a solution. Because of this we are disallowing it by requiring the TrainReplacement class to call the functions defined in Train or Bus, rather than re-implementing them.

4.4.4 Usage example

Algorithm 3 Usage example for the adapter pattern

```
replacement ← new ReplacementTrain(40)
RailwayController.runLine(replacement, 2)
                          ▷ Expecting: Bus T2 driving with 40 passengers
```

4.4.5 Results

Language/Metric	NCLOC	Cyclomatic	DIT	NOI	MNOC	LCC	SR
Python	46	19	2	2	1	0.35	12.5
C#	97	22	2	2	2	0.39	13.5
JavaScript	58	17	2	2	1	0.35	11
Go	90	20	2	2	2	0.47	17.5
Smalltalk	85	22	2	1	1	0.41	12.5

Table 4.3: The results of the adapter case.

4.4.6 Analysis

General

There were two types of solutions among the implementations: Multiple inheritance from Train and Bus and inheritance from Bus using Train composition. The intuitive solution of inheriting from Train and using composition with a Bus was not optimal in any of the languages.

Multiple inheritance was used in the Python and JavaScript implementations. Both prioritized inheriting from Bus over Train and then used some additional tricks to solve the issues related to the stop and setLine functions. Since methods are simply function variables in JavaScript it could easily rearrange its methods to solve these issues, while Python had to override the relevant methods and explicitly invoke the relevant Train method in the context of the TrainReplacement object.

Inheritance from Bus and composition with Train was used by Go, C# and Smalltalk. It makes sense to inherit from Bus rather than Train, if it is technically possible, as most of the functionality of TrainReplacement is found in Bus. This was possible in Go due to Train's functionality already being specified using an interface, so all one had to do was conform to that interface to be a Train. In C# one had to define an interface ITrain and let both Train and TrainReplacment inherit from it. In Smalltalk this was not an issue, due to the typelessness of the language.

Lines of Code

The languages relying on multiple inheritance, Python and JavaScript, scored significantly better than the rest of the languages in lines of code. This is primarily due to not having the overhead of initializing the composition and the extra lines of code from manually invoking the composed object's methods. Both used 7 lines of code to define the TrainReplacement class, but Python does as usual score a bit lower due to its shorter function definition syntax.

Among the single inheritance implementations Smalltalk's implementation scored the lowest. One major reason for this is that it did not have to define an interface for train like functionality, but could rather rely on its duck-typing. It is also good at efficiently defining many class attributes, but bad at constructors. Since there are few classes and many attributes in this case its syntax was as efficient for this case as Go or C#'s.

Go scored a bit better than C#. This is due to a problem both Smalltalk and C# had, which Go did not have. One can note that the line attribute is used two places in the TrainReplacement class. Once in the *setLine* function and once in the *drive* function. For the implementations relying on composition these do however not reference the same line. The *setLine* function references the underlying Train object's line attribute. In Go one can simply set the TrainReplacement object's line attribute equal to this at the end of setLine. However, since the attribute is not globally visible in C# or Smalltalk , this is not possible in these languages and one has to extend the Train class with a getter of line to do this.

C# scores the worst here, for the reasons listed above. It also gains some extra overhead due to it not supporting implicit inheritance of constructors with more than one argument and the need for a class wrapper for the main method.

Cyclomatic Complexity

Also here the multiple inheritance based solutions scored better. Since they can inherit more they define fewer functions. JavaScript scores lowest because it only needs an initializer function for the TrainReplacement class, as it manages to do all the changes by manipulating function references.

Go scores the best of the rest. This is due to it not needing a getter for the *line* attribute, as discussed in the previous section. Smalltalk and Go share the last place. Smalltalk usually scores higher than C# due to its two-function constructors, but in this case they are equal since C# needs an extra constructor and a main function.

Depth of Inheritance Tree

For all the solutions TrainReplacement inherits from at least one class and there is no longer path in the tree. Therefore the depth is two for all the implementations.

Number of Inheritances

The Python and JavaScript solutions has TrainReplacement inheriting from two classes, therefore they score two here. So does the Go and C# as their TrainReplacement class relies on implementing an interfaces as well as inheriting form a class. Smalltalk does not have to do any of this and therefore scores only one.

Maximum Number of Children

For Python, Smalltalk and JavaScript no class has more than one child, therefore they only score one here. For the classes reliant on an interface both the TrainReplacement and Train class implement the *ITrain* interface. Therefore they score two for this metric.

Loose Class Cohesion

Here the multiple inheritance solutions scored worse than the single inheritance solutions. The main reason for this is that almost all the newly defined functions in the TrainReplacement class reads the composed train object. Thus being much more cohesive than their inherited counterparts.

Best of the single inheritance functions is Go. It does not have to define an extended Train class, thus the cohesive TrainReplacement class has a greater effect on the score, making it higher than for C# and Smalltalk. Go also has to explicitly define every attribute of a struct upon creation, meaning its constructor function writes to more objects and are therefore cohesive with more functions. Smalltalk scores slightly higher than C# due to the static and dynamic part of its constructors being cohesive with each other and an above average number of functions.

Python and JavaScript scores exactly the same in this metric, since despite the creation being a bit different, the finished classes contain exactly the same functions in the two implementations.

Shallow References

JavaScript scores the best for this metric. As usual this is due to its of defining classes through their constructors. If it had used class notation it would score the same as Python's implementation.

C# and Smalltalk gets an advantage from having no public attributes. However, since they have to introduce a class to pass along a needed attribute, they do not gain a large advantage from this. Also Smalltalk loses score due to its double constructors and C# loses score due to needing an extra interface and from having its main function wrapped in a class.

Go has the normal disadvantages of having top level defined constructor and having to define a main function. In addition it is the only one of the composition solutions to have a public composition attribute. This is quite costly as it refers to a class with many methods and attributes.

Summary

The multiple inheritance based solutions scores best in almost all the metrics. The exceptions to this is Number of Inheritances, where Smalltalk scores best and class cohesion . For the Cyclomatic Complexity and Lines of Code metric a major factor for multiple inheritance scoring best is that fewer things have to be explicitly defined.The tree based metrics mostly favor multiple inheritance as the advantages of inheriting from Bus is so great. This makes it worth introducing and inheriting from an interface to achieve it if it is not achieved by multiple inheritance of classes. The shallow references score is more even than usual due to the languages normally gaining an advantage from private visibility of attributes losing some of it in this case, due to needing a workaround to access an attribute. Class cohesion is better for the solutions relying on composition. This is because there in these implementations is an adaptee attribute in the adapter class, which is used in many of its adapter functions. This increases cohesion.

Among the multiple inheritance solutions Python scores better than JavaScript in LOC, but loses in all other metrics. The size advantage is due to Python's lack of brackets, while the rest is mostly due to JavaScript not having to define a single new function.

Among the single inheritance solutions Smalltalk not needing an interface and Go not needing a helper class makes them alternate in scoring best on the different metrics. C# consistently scores the worst, as it has both.

4.5 Decorator

4.5.1 Pattern

Decorator is a behavioral pattern. It is a way to add additional functionality to an object by extending its behavior. One creates a new *Decorator* object with the original object as a variable and redirects all function calls to the Decorator to the corresponding functions in the original object, thus duplicating its functionality. One can then extend the functionality of the original object by instead of only redirecting doing something before or after.

4.5.2 Case

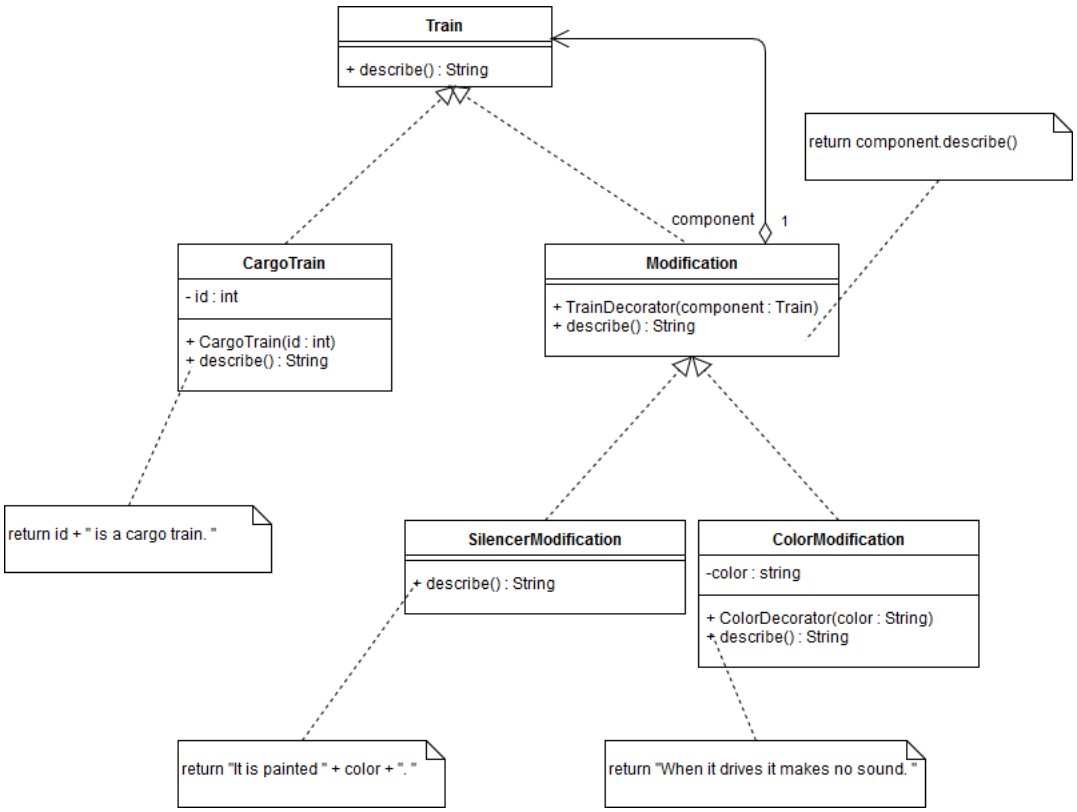

Figure 4.4: UML diagram for the decorator case.

In this case we are modeling trains. They are heavily simplified to have only one method: Describe. It returns a description of the train. We model only one kind of train: The Cargo train. In addition we model that trains can be modified to have a color and/or run silently. One can argue that it makes little sense to allow a train to be multiple colors or to be double silent, but in order to simplify the model we are allowing it here. The relevant classes are:

Train An interface, requiring a describe() method which returns a string.

It plays the role of Component in the design pattern.

CargoTrain A class implementing the Train interface, since it can describe itself. It plays the role of ConcreteComponent in the design pattern.

Modification A class implementing the Train interface. It is created with a train variable, representing the train it is modifying. It describes itself by requiring the train to describe itself. It plays the role of Decorator in the design pattern.

ColorModification A class implementing the Modification interface. It overrides the describe method to return a concatenation of its super class' description and a description of its color. It plays the role of ConcreteDecorator in the design pattern.

SilencerModification A class implementing the Modification interface. It overrides the describe method to return a concatenation of its super class' description and a description of its silent running. It plays the role of ConcreteDecorator in the design pattern.

4.5.3 Usage Example

Algorithm 4 Usage example for the decorator pattern

```
classicTrain ← new ColorModification(new CargoTrain(6), "red")
stealthTrain ← new SilencerModification(new ColorModification(new
CargoTrain(007), "black"))

print(classicTrain.describe())
                    ▷ Expected: 6 is a cargo train. It is painted red.

print(stealthTrain.describe())
        ▷ Expected: 7 is a cargo train. It is painted black. When it drives it
makes no sound.
```

4.5.4 Modeling choices

Simplicity focus Most of the other cases focus on difficult versions of the pattern. For this case the goal is to model a simple version of the pattern. A version, with few classes and variables, that has the minimum number of requirements necessary to implement a full version of the pattern. The goal is that this case may serve as a baseline for some of the others.

Allowing multiple modifications One could argue that it makes little sense for a train to be modified with the same modification twice. Especially a train being two or three times silent is not easily interpretable. The choice is however made to allow this, as

disallowing it would require a major increase in complexity and it is not a sufficient reason to choose a less intuitive case to model.

Obligatory Modification class Requiring a Modification class might seem weird in a case focused on simplicity, but we consider it necessary as it is a specified part of the pattern in [28]. The reason they have it as obligatory becomes evident if one considers more complex classes. There one might want to modify only one of many methods in that class several times, for different decorators. Then it makes sense to extract all the functionality which is default.

Two modifications The choice was made to have two classes playing the roles as concrete decorators. The reason for this was that the implementation of a stateless decorator is a bit different from those with a state, so we wanted one of each.

Impossible in Go Implementing the above class hierarchy directly in Go is impossible (or very nearly so). The reason for this is that there is no *super* mechanism in the language, that is: There is no simple way to reference a method from an anonymously composed struct in a struct if that struct contains a method with the same name. As described in the model this is necessary for the modification subclasses. Therefore the Go implementation will use named composition in addition to the anonymous composition. Letting this composition be done with pointers and letting both pointers point to the same object gives something quite close to having a *super* function. Since it is custom defined it is not quite what is modeled, but it is the closest thing I found.

4.5.5 Results

Language/Metric	NCLOC	Cyclomatic	DIT	NOI	MNOC	LCC	SR
Python	23	8	2	2	2	1	11.25
C#	45	9	3	3	2	1	10.5
JavaScript	32	8	2	2	2	1	10.75
Go	52	8	2	3	4	1	16
Smalltalk	45	11	2	2	2	1	11

Table 4.4: The results of the decorator case.

4.5.6 Analysis

General

There was, despite the attempt at simplicity, one complicating factor in the pattern. The describe method for ColorModification and SilencerModification requires access to Modification's describe method. This is very much part of the core of the pattern and not something one could simplify away.

The languages handled it to differing degree. Python, C# and Smalltalk all have a *super* object they can access and get the method from, so for them it was simple. JavaScript also has a *super* object, but it only works in the context of class definitions and object literals, not when defining an object through a function and the new keyword [46]. It is of course possible to do similar things using the prototype, but what is gained by the simpler notation is lost in the extra lines of code and added complexity from doing this. Therefore the class notation and super object is used for JavaScript. Go handles it as described in 4.5.4. This is the most clunky of the solutions by far as it requires additional variables and extra lines of code. It also is arguably not a complete solution to the case. This is because an implementer of Decorator has to know of this trick to extend it as intended.

Lines of code

Python scores best in this metric, as usual. Partially this is due to not using brackets, but it does in addition gain some from not having to define a train class and from having a native super method. JavaScript scores second best despite using the class notation for most of the definitions. Its main saving compared to the c# and Go is not having to define a Train class. SmallTalk also benefits from this, but due to the longer constructor statements and having to declare its class attributes it still scores a bit worse than JavaScript.

C# has the same score as Smalltalk. It has to define a train class and has some overhead from having to define its attributes. It does however have a native super method, so it scores better than Go. Go has all the same problems as C#, but also lacks the native *super* method.

Cyclomatic complexity

For such a simple example there are little differences regarding complexity. C# scores one extra as it cannot inherit constructors, so it has to create one for SilencerModification despite it being the exact same as Modification's constructor. Smalltalk scores some extra since it has to create two methods for each constructor.

Depth of Inheritance Tree

C# is the only one scoring above two. Python, Smalltalk and JavaScript avoids having to define the train class using duck-typing. Go defines it, but has a flatter structure due to structural typing.

Number of Inheritances

The languages which define train scores three here. Since it is the root of the deepest tree, the ones without it scores one less.

Maximum Number of Children

For all the languages, except Go, Modification has the most direct children, two. For Go the train interface is implemented by four structs and therefore it scores 4.

Loose Class Cohesion

All the implementations have perfect cohesion. The describe methods and the initializes always share at least one attribute.

Shallow References

C# scores the lowest. This is due to having private variables. This is extra impactful in this case since it means it does not have the train variable in the public namespace of its decorators, so it avoids counting both them and their attributes. JavaScript scores second best. As usual it has some advantage due to its object defining functions, but since it mostly relies on class syntax this matters less than normal. It also makes some savings from not defining a Train interface.

Smalltalk comes at a close third. It scores well for the same reasons as C#, but loses some due to its double constructors. Python follows Smalltalk, scoring high because of its public variables. In a clear last place we have Go with 16. It scores this high due to the top-level constructor definition and due to explicitly setting the super reference. This increases the score by that much because referencing an object with many attributes/methods is costly with regard to this metric. An interesting mechanic is that Smalltalk's the use of the Train interface in the Modification class is slightly positive, compared to the non-interface solutions, since it hides the attributes associated with a specific kind of train. This does not nearly do enough to cancel out any of the other negative effects, but is still worth noting as it could have a bigger effect in other settings.

Summary

Most of the metrics were fairly even. The most common factors for differences were languages avoiding defining the Train interface, how well the languages managed to extend a method and general notation differences.

Part IV

Discussion and Conclusion

Chapter 5

Analysis

5.1 Introduction

In this chapter the remaining analysis is performed. For the individual cases the data was presented and analyzed in that case's analysis section. The focus of this chapter is therefore to analyze the data from the different cases in relation to the other cases. For the data relating to the NCLOC metric this is done using aggregates as discussed in Section 3.4.8. This is presented in Section 5.2. The comparison for the other metrics is presented with less rigor in Section 5.3.

5.2 Aggregates

5.2.1 Introduction

Language/Case	Composite	Prototype	Adapter	Decorator
Python	41	32	46	23
C#	72	59	97	45
JavaScript	50	41	58	32
Go	80	57	90	52
Smalltalk	85	56	85	45

Table 5.1: The NCLOC score of combinations of languages and cases.

Table 5.1 collates the NCLOC scores for the different languages and cases. Using the aggregates discussed in Section 3.4.8 we wish to aggregate the information from this table and present it in this section. The intent is for this to give numbers which are more intuitively meaningful than the raw NCLOC values. Obviously no new information is presented in this section, therefore it is presented in the analysis chapter rather than a separate results chapter.

Language/Case	Composite	Prototype	Adapter	Decorator
Python	0.13	0.13	0.12	0.12
C#	0.22	0.24	0.26	0.23
JavaScript	0.15	0.17	0.15	0.16
Go	0.24	0.23	0.24	0.26
Smalltalk	0.26	0.23	0.23	0.23

Table 5.2: The *Proportion of Case's NCLOC* score of combinations of languages and cases.

5.2.2 Proportion of Case's NCLOC

The values for this aggregate is presented in Table 5.2.2. This metric shows, given a case and a language, the proportion of the total lines of code for the case used for the implementation in the language. We can see that there are clear and fairly consistent trends in the values. Python consistently scores the lowest and JavaScript the second lowest. The remaining languages score relatively similarly, with values around 23%.

5.2.3 Proportion of Language's NCLOC

Language/Case	Composite	Prototype	Adapter	Decorator
Python	0.29	0.23	0.32	0.16
C#	0.26	0.22	0.36	0.16
JavaScript	0.28	0.23	0.32	0.18
Go	0.29	0.20	0.32	0.19
Smalltalk	0.31	0.21	0.31	0.17

Table 5.3: The *Proportion of Language's NCLOC* score of combinations of languages and cases.

The values for this aggregate is presented in Table 5.3. This metric shows, given a case and a language, the proportion of the total lines of code for the language which is used on the case. There is also here a clear trend. Certain cases require more code to implement in all the languages than other cases. Here the trend is less consistent than in Section 5.2, though.

We see for the composite pattern that C# scores relatively better than it usually does, while Smalltalk scores worse. For the prototype case Python and JavaScript scores proportionally worse than usual, while Go and Smalltalk score proportionally better. For the adapter case C# scores proportionally worse, while the rest score fairly equal. And for the Decorator case Go and JavaScript scores worse while the rest score fairly equal.

5.2.4 Relative Size

The values for this aggregate is presented in Table 5.4. It represent the lines of code used on this case compared to the other cases. It shows that the

Case	Composite	Prototype	Adapter	Decorator
Score	0.29	0.21	0.33	0.17

Table 5.4: The *Relative Size* score for the different cases.

decorator case was the shortest, the prototype case was the second shortest, the adapter case was the second longest and the composite case was the longest. The first two and last two are of comparatively equal size. We can observe that these number relate fairly well to the trends observed in Section 5.2.3.

5.2.5 Relative Verbosity of Language

Language	Python	C#	JavaScript	Go	Smalltalk
Score	0.12	0.24	0.16	0.24	0.24

Table 5.5: The *Relative Verbosity of Language* score for the different languages.

The values for this aggregate is presented in Table 5.5. It gives a single value to the size of implementations in the given language across all cases. We see that these represent the same as the trends in Section 5.2. Python scores best, JavaScript second and then the rest is similar. It is important to note that, even though, these numbers are closely related to the research question of the thesis, they are in no way the final conclusion of the experiment. They are only directly related to size of the implementations and even then only serves as a jump of point for further discussion.

5.2.6 Summary

In this section we presented aggregated data for the NCLOC metrics. As expected, we saw a clear link between the values in Table 5.2.2 and in Table 5.4 and between the values in Table 5.3 and in Table 5.5. In both of the large tables there is some variance and outliers.

5.3 Cross case comparison

5.3.1 Introduction

In this section we see the analysis done for the individual cases in Section 4 in the context of each other. That means we, for every metric, compare the values for that metric between the cases and extract common trends in the underlying reasons.

5.3.2 NCLOC

A comparison of the data for this metric is done in Section 5.2. We will therefore, in this section, be focusing on common reasons for the size

differences.

In most of the cases there were a one or two issues which were not trivially solvable in every language. Being one of the languages not solving these issues well were a common reason for higher NCLOC score. In addition it was often possible to use the object oriented mechanics of the languages to avoid declaring certain classes or interfaces. Syntax and basic mechanics of the language also had some impact. In particular Python gained a lot from not using brackets in its declarations and Smalltalk often had a significantly increased score due to its lack of a built in constructor. The languages required to declare class attributes also often required some extra lines to do this.

5.3.3 Cyclomatic complexity

Internally in a case there would usually be some variance between the languages. There is not a consistent ordering of the languages' scores but usually one would find Python and JavaScript near the top, while Go and C# somewhat worse and Smalltalk to be the lowest.

For all the languages the loops and test statements are very similar. It is therefore no shock that the main source of difference in cyclomatic complexity is the number of defined functions functions.

A common source of difference were the constructors. Smalltalk scored significantly worse due requiring two methods for every constructor. JavaScript would also consistently get a worse score from its mechanic of defining classes with functions, which leads to an extra function definition when the other languages don't need a constructor. C# sometimes gained and sometimes lost from its constructor. On one hand it had to explicitly pass the constructors arguments to the base class' constructor, while most of the other languages automated it if the constructor was unchanged. On the other hand it sometimes didn't need a constructor as it could initialize its variables to default values without needing a constructor.

Smalltalk and C# also scored a bit worse due to needing helper functions to get or set private attributes in some classes.

5.3.4 Tree Related Metrics

With the exception of the multiple inheritance case the tree related metrics followed a pattern. The *Depth of Inheritance Tree* (DIT) scores and *Number Of Inheritances* scores (NOI) were mostly equal, except for Go often scoring one higher in NOI. Sometimes Python, JavaScript and/or Smalltalk would score lower than the other languages in both. Also the *Maximum Number Of Children metric* (MNOC) mostly remained unchanged between the implementations of a case, with the exception of the Go implementation often scoring higher.

The reason Go often scored higher is that there often were inheritance from both interfaces and structs. Therefore there were more often multiple inheritance, affecting the NOI metric and there were often an interface with many children, which increases the MNOC score.

The reason Python, JavaScript and Smalltalk often were able to score one lower is due to duck-typing. They could thus often skip defining an interface at the top of the inheritance tree and therefore decrease the depth of the tree with one.

5.3.5 Shallow References

For Shallow References there were also some patterns. Overall C# and JavaScript scored best. With Smalltalk scoring slightly worse, Python somewhat worse than that again and Go clearly worst.

The reason C# scored so well was the use of private variables. On the other hand it often had to define objects or helper functions the other classes could avoid, which made it slighly worse again. Smalltalk also benefited greatly from private variables, but due to having two methods in the namespace of an object for every constructor made it worse overall.

JavaScript gained a bit here by not having to differentiate between a class and the initializer function to that class, if employing its prototype notation.

Go scored as bad is it did primarily because it defines a lot of items at top level. It also employs none of the tricks to avoid defining extra interfaces the scripting languages does, while it also only has public variables. It is the worst of both worlds.

5.3.6 Summary

In this section we saw that there, for every metric, were some patterns across cases and some common reasons, related to the mechanics of the languages, for the existence of these patterns.

5.4 Summary

In this chapter we have seen the analysis from the individual cases in light of each other. We have from that extracted some common reasons for differences, shared between the cases. We will in the next chapter build on this analysis and the analysis from the individual cases to discuss the implications of the data, the applicability of the data and try to answer the questions posed in Section 1.3.

Chapter 6

Discussion

6.1 Introduction

In this chapter the analysis from Chapter 5 and 4 is re-examined with the goal of finding answers to the reasearch question. Section 6.2 starts by identifying aspects of the patterns creating large differences in the code. It calls these *core issues* and discusses their significance. In the next section, Section 6.3, the analysis is examined from the context of the languages' attributes. It examines how the languages performs, what mechanics are contributing to this performance and attempts to generalize the findings. Lastly there is the Metric Validity section, Section 6.4. It discusses the validity of the metrics in light of the case design and data gathered.

6.2 Core Issues

6.2.1 Introduction

A pattern in the implemented cases was that there was a challenges for the implementations which were handled well by languages having a certain feature, but which the other languages struggled with. For instance for the Prototype case this challenge was the hardcopying of objects. If a language supported this functionality the clone method of the prototype object became trivial, otherwise it could take many lines.

One could argue that this is not the most interesting aspect of the results. Whether a language supports a certain trick in its standard library is not particularly generalizable or something one could learn from. In this section I will argue the opposite. A case could be made that these challenges are heavily related to the core of the design pattern and that a language's handling of them is relevant for that language's applicability to use the design pattern.

6.2.2 The Issues

In this section the core issues of the different cases are listed.

Composite

Challenge Easily creating the managing functions for nodes in a tree of objects and efficiently calling and aggregating children methods.

Solution Having access to a good list-like collection and reduce or summation functions.

Prototype

Challenge Creating deep copies of objects.

Solution Having a automated deep copying method.

Adapter

Challenge Conforming to one interface, while having the functionality of another, similarly defined, object.

Solution Multiple inheritance and easily manipulating method assignments.

Decorator

Challenge A child object being able to invoke the parent class' method in the context of the child.

Solution Having access to a *super* object.

6.2.3 Relevance

One can note that the solutions described in 6.2.2 are closely related to the problem the associated design pattern sets out to solve. For instance:

- The decorator pattern relates to reusing and adding to methods of a class, the super object is a mechanic allowing calling parent functions when one wants to reuse them.
- The composite pattern is at its core about representing many objects with one and allowing for a single interface to interact with all of them. Similarly, a list is an object composed of many smaller objects and the reduce and sum functions are related to interacting with many objects in a list in one call.
- Prototype is about the creation of objects similar to a run-time object, while copy functionality is about creating a copies of objects at run-time.
- Adapter is about inheriting interface from one class and functionality from another, while multiple inheritance allows for inheritance from multiple sources and manipulation of method assignments allows for altering the interface of a class while keeping the functionality.

In a way it is not surprising that languages with built-in mechanics to solve the issues a design pattern sets out to solve would also be well suited for implementing the design pattern. It seems reasonable that they could often use the mechanic to simplify the implementation of the pattern.

6.2.4 Validity

There are certain issues with blindly trusting these seeming correlations. Firstly the cases studied are not the only versions of the pattern, as discussed in Section 3.5.4, the choice of case has a major effect on the data. This is also true here. If one were to create a case for the prototype pattern only requiring soft copying, then the hard copying functionality would no longer be necessary. While one might say it would only be replaced with the requirement for a soft copying function, this is something all the tested languages would have, and thus we would not be observing a *core issue* for that prototype case. Since some of the cases were designed to present challenges for some of the languages, one would expect this to increase the number of differing solutions to core issues.

There is also the problem of these issues not being rigorously selected. They are selected using qualitative criteria, as places in the code *feeling* like they are places of great variance. Then, also in qualitative way, the place is connected to a challenge and a mechanic giving a solution. The first of these could be conceived measured quantitatively by, for instance, listing functions with the greatest difference with respect to certain metrics. The second is harder to find a way to set up rigorously, as it relates to ideas rather than observable phenomena. Connecting multiple inheritance to the adapter challenge is, for instance, a bit of a stretch, though I think it is ultimately reasonable, at least for this case of the design pattern.

Despite this, I think the observation has value. It would be a surprisingly strong coincidence for all four of the studied cases to present such a correlation without there being some kind of underlying phenomena. To argue that this will be present for every case related to design patterns would be a fallacy. However, to conclude, like I did the argument in Section 6.2.1, that these differences are irrelevant would probably also be a mistake.

6.2.5 Implications

Applying this theory to other patterns or other languages than the ones tested seems reasonable, within the limits discussed in Section 6.2.4. If one can identify a mechanic in the language that is closely related to the solution of the design pattern, one could expect a higher chance of a good implementation of the design pattern in that language. For instance one could expect a good implementation of the memento pattern in languages having a mechanic related to saving objects states in a variable, like Python's pickle or JavaScript's JSON.stringify.

Does this mean that patterns are better used in the context of these languages? Not necessarily. The fact that the language has no mechanic related to doing what the pattern is trying to achieve, means that implementing the pattern is all the more important. One would, for example, seldom fully implement the prototype pattern in Python, since its copy function is so accessible and directly applicable.

6.3 Languages

6.3.1 Introduction

In this section we discuss the applicability of languages to the tested design patterns In the first few sections this is discussed for the individual languages used in the experiment. Then, in Section 6.3.7 these findings are discussed in the context of each other. Lastly, in Section 6.3.8, an attempt is made at generalizing these results by extracting common themes in what makes a language well suited for design patterns.

6.3.2 Python

Python scored among the best in most of the cases and for most metrics. The exception were Shallow References and Loose Class Cohesion, where it sometimes scored a bit below average.

Lines of Code

The primary reason Python scores as well as it does is that it often avoids defining objects or functions the other languages have to define. It often avoid defining interfaces or functions in abstract classes due to duck-typing, it never needs any helper functions to deal with things like attribute visibility or sort out kinks with inheritance and it can sometimes avoid defining a constructor when other languages have to.

There are no brackets to indicate code blocks in Python. This means that almost every definition of a function or class is shorter than their counterparts in the other languages. For all the cases it is an important factor in Python's low NCLOC score, as there are quite a few such definitions in all of them. However, as noted in Section 6.4.2, it probably has more impact on the score than it does on the quality attributes of the implementations.

Core Issues

Python also has a good solution for each of the tested design pattern's *core issues*. It sometimes has to make an import statement more than the most elegant solution, but otherwise it is solves the problem as well as any other solution.

Loose Class Cohesion

The sometimes lower score in Loose Class Cohesion for Python mainly stems from two factors. Firstly it sometimes does not need to define helper functions other languages use. This can be a detriment if those methods are cohesive with the other methods of that class. Secondly, mainly in the prototype case, it creates a parent class that abstracts away sufficiently much of the context that it loses cohesion from it. It states it clones all its variables, however, since it has no variables this does not read or write to

any variable and is therefore not cohesive with anything. Not even itself. While the first of these reasons seem like something one would expect to occur occasionally the second seems so narrow it would be wrong to judge the language on that basis.

Shallow References

For Shallow References Python scores middling. Sometimes a bit above average and sometimes a bit below. Most of the time it has a low score this is because it has no way to set the visibility of attributes. It cannot hide these attributes from the global namespace, in any way. As discussed in the essay there are conventions for how to designate variables as private in Python, which mitigates the problem somewhat. It is still one of the few measured weaknesses for Python, though.

Total Assessment

Collecting all of this we see that Python is, compared to the other languages discussed in this section, quite good at implementing the tested design patterns. It has some weaknesses related to the cohesion of the classes and the number of objects defined in its global namespace, but is otherwise scoring great on metrics related to maintainability, conciseness and complexity. The main reasons for this is its use of duck-typing, its flexibility to find good solutions to the core isses and its efficient definitions avoiding a lot of the overhead of the other languages.

6.3.3 JavaScript

JavaScript scores well with respect to NCLOC, Shallow References and the tree related metrics.

Lines of Code

JavaScript scores well in the NCLOC metric for the same reasons as Python. It seldom needs helper functions, it uses duck-typing to avoid class definitions and mostly have good solutions to the *core issues* of the patterns. One case where it deviates from this is the Prototype case. In this case it does not have a good, built in hard copy function, so it has to resort to manually create copies of the attributes which are not basic data types. This makes it score significantly different than Python for this case, for the details see Section 4.3.6.

Shallow References & Cyclomatic Complexity

JavaScript generally scores well in the shallow references metric, but worse than most in the cyclomatic complexity metric. The reason for both of these phenomena are connected to its unique way of defining classes through functions. On one hand it means at least one function for each class, so it increases cyclomatic complexity. On the other hand it means that the

class definition and the constructor for that class is referenced only by a single variable, so it decreases Shallow References. Since JavaScript ES5 and forward support class notation it is not mandatory to use the function notation. As seen in the decorator case in Section 4.5 the implementations deviate less with respect to these two metrics when class notation is used.

Tree Based Metrics

With the exception of the earlier discussed Prototype case the tree based metrics for JavaScript and Python are the same. They both employ duck-typing and multiple inheritance and usually have the exact same inheritance graph.

Loose Class Cohesion

Lastly there is the Loose Class Cohesion metric. For this JavaScript also scores very similar to Python, with the exception of the prototype case, where JavaScript scores among the best.

Total Assessment

Overall JavaScript scores well. Many of the language's mechanics causing this is shared by Python. They differ primarily when it comes to their way of defining their object oriented constructs and due to the differences related to the prototype case.

6.3.4 C#

C# does, for this experiment, play the role of the classical, fully typed, object oriented language. Like C++ or Java. It performs a lot like one would expect such a language to perform. It has a relatively high NCLOC score, it usually scores well when it comes to Shallow References and it often has relatively large class hierarchies.

Lines of Code

The size of the implementation being consistently this high can be partially traced back to C# not having the right tools for some of the *core issues* of the different cases. Looking at the relative score from Section 5.2 we see that the two relatively longest implementations in C# is the Prototype and Adapter. These are the cases where C# is considered to have failed at finding a good solution to the core issues, which fits with this theory. However C# scores high in the measures where it has a good solution to these issues too, so there has to be more factors in play. In both these cases there is an interface which has to be explicitly defined by C# and not for most of the other languages. This adds quite a bit extra to C#'s score.

In addition C# has to explicitly declare its class attributes. While this is sometimes beneficial for the NCLOC score, due to avoiding the need of some constructors it is overall a negative. Lastly there is some overhead

due to needing a main function. Generally 3 extra lines for all the cases, compared to most of the other tested languages.

Cyclomatic Complexity

C# scores ok in the cyclomatic complexity metric. In the adapter case where it has to define many more functions than the ones reliant on multiple inheritance it scores a bit higher than normal, but in the other cases it is usually relatively close to the best scoring one.

Sometimes it scores a bit lower or higher due to its constructor. For some classes it can avoid defining one, as it can set its initial value when declaring the attribute, while for other classes it scores a bit lower as it has to explicitly pass arguments to its parents constructor regardless of whether it introduces something new.

Shallow References

The Shallow References score of the C# implementations benefits from its visibility control of its class attributes. Especially in cases, like the prototype case, where some of the attributes are complex objects this can create a significant difference.

In the Adapter case there was an issue of a variable being protected, and therefore not accessible from another class. This meant the C# implementation had to define an extra class and method to make it accessible, which for that case more than outweighed the gain from having control of the variables. One could argue that this would be less of a problem in practice, as one in cases like that sometimes would have the option of making the variable accessible or have an already a defined getter variable for the value.

C# also loses some score in this metric from needing to define a main function and a class wrapper for it and due to generally defining more objects than the other languages.

Tree Based Metrics

When it comes to the inheritance trees the data shows that C# generally has long trees and the expected amount of children. This is because it takes no shortcuts like many of the other languages tested here does. If it wants to reference two unrelated classes with one variable it has to define an interface of their shared functionality. It is hard to imagine a situation where Python would have to define a class and C# could avoid it.

Loose Class Cohesion

Class cohesion for C# has the median value in class cohesion in all the experiments. The reasons for this varies. For the adapter and prototype case, it benefits in cohesion from not solving the *core issues* of the cases, so it scores better than the two that does, but is the worst among the rest. This

is due to it introducing helper functions and removing responsibility from its constructor functions.

For the other two cases it defines the normal amount of functions and interacts with the normal amount of variables, and is therefore only different from the languages which themselves does something extra.

Total Assessment

Overall C# scores worse than one might intuitively expect. One would think that since design patterns were conceived with C++ in mind they would be easy to implement in C#. This is however, based on the data gatered in this experiment, not the case as it scores relatively high in many metrics. A common theme is that it is too inflexible and requires the explicit definition of too many entities to be the best at anything except shallow references.

6.3.5 Go

Go scores bad in NCLOC and Shallow references and ok in cyclomatic complexity and class cohesion. It scores differently in the tree related metrics.

Lines of Code & Cyclomatic Complexity

For the NCLOC and cyclomatic complexity metrics I often noted in the individual sections that one could map the C# and Go implementations line for line. There sometimes were minor differences, like one failing to solve a *core issue* or needing a helper function for something, but overall it was very similar. It is therefore not surprising that they score very similar in these two metrics as well.

Shallow References

The Shallow References score of the Go implementation are the worst, often by a decent margin, for all of the cases. The biggest reason for this is that in Go one often uses functions in the global scope instead of constructors. Go also sometimes defines both an interface and a struct when many other languages could rely on only a single class. In addition to this it has a main method in the global scope which not all other languages have. All these factors combine to make Go's implementations score bad here.

Loose Class Cohesion

Because Go cannot create an object without explicitly setting each of its attributes, Go's constructors are cohesive with more methods. This increases cohesiveness. Go also relied on the composition solution over multiple inheritance for the adapter case, which is positive for cohesion. On the other hand, due to the rules set in Section 3.5.11, Go does not always have to define a constructor. Since constructor functions are generally more

cohesive than the average function, this can be non-benefical. In the one case where this mechanic came into play, the composite case, Go scores the lowest.

Tree Based Metrics

When it comes to tree metrics Go scores differently than the others, due to its structurally typed interfaces. Given how we defined this kind of inheritance to affect the inheritance graph in Section 3.5.11 we can conclude that the resulting tree is often flatter. This is especially applicable in the cases where a normal class implementation would have a depth of 3, a class inheriting from a class which again inherits from an interface. In these cases Go's inheritance scheme instead gives a struct inheriting from both an interface and a struct. The flip-side of this is that the interface in Go generally has more direct children and therefore scores higher in the maximum number of children metric. Whether this is a positive trade-off or not is hard to say. Firstly because it is a quite context dependent questions. Secondly because it is hard to tell whether the definitions from 3.5.11 are meaningful and what they do to the validity of the metric. It is however certainly interesting that there is so consistent differences between the two inheritance schemes.

Total Assessment

A take away is that it is interesting that despite how closely related C# and Go are in the size and complexity metrics, they are still quite different in some of the other quality metrics. This both indicates that there are limits to what these two metrics show and that C# and Go are not interchangeable as tools.

6.3.6 Smalltalk

Smalltalk scores as the other static languages in NCLOC, the worst in complexity, among the best in Shallow References, mostly as the dynamic languages in the tree based metrics and a bit differently in class cohesion.

Core Issues

Smalltalk mostly solves the *core issues* of the cases well. It lacks an aggregation function (but has a great collection for managing the elements) for the composite case and multiple inheritance for the adapter case. Otherwise it solves them as well, or better than any other language, with respect to all the metrics.

Lines of Code

Compared to the other static languages Smalltalk has an advantage in its better solution of core issues and its lack of interface definitions. One would imagine this to lead to a quite low NCLOC score. However, the

data in Table 5.5 suggests that it is about equal to C# and Go in this respect. This is in a large part due to the extra lines of code needed for defining both a static and a dynamic constructor. In also needs to declare variables in its functions and classes, which makes it have some lines the dynamic languages does not. One would expect that for larger classes the constructor problem would be less of an issue, as their constructor would be a smaller part of the total. None of the classes used in the experiments were particular big, so the impact in a more realistic setting might be smaller. On the other hand there is few design patterns which independently of the context requires large classes, which might mitigate this effect.

Cyclomatic Complexity

Smalltalk's constructor mechanic is is also the cause for its high score on the cyclomatic complexity metric. Defining extra functions increases the score for the metric. On can however note, as discussed in Section 6.4.3, that this difference is maybe not measuring complexity in a meaningful way. The difference in the number of paths through the program would not be greatly affected by this addition. There is also, as for the NCLOC metric, the issue of these classes being relatively small. Making the constructor more relevant than it might be in many real settings.

Shallow References

The visibility of attributes of objects in Smalltalk not being visible outside the object they are defined in makes Smalltalk score well in the Shallow References metric. It scores about the same as C#. Also here the double constructors have a negative effect and the duck-typing has a positive effect.

Tree Based Metrics

Smalltalk scores the same as Python in all the tree based metrics in all the experiments, except the adapter experiment. This is due to all three languages supporting duck-typing. Smalltalk does however not support multiple inheritance, making it different for the Adapter case. It does in this case score better than Python and JavaScript in the Number of Inheritance metric, as it only requires inheritance from a single source. Overall it therefore scores best among all the languages in this metric.

Class Cohesion

The score for this metric is also affected by the constructor definitions. Having two functions related to constructors means the cohesion of the constructors account for more of the total cohesion. In the examples where the constructors are above average cohesive Smalltalk scores slightly better than the average language. This is the case for the composite and adapter

cases. For the Prototype case we do however have the opposite mechanic. It and Python scores about the same since both introduce a non-cohesive class to abstract cloning functionality. However, since the constructor in this case is less than average cohesive, it has a negative impact on the score.

Summary

Smalltalk scores quite similar to C# for the non-tree based metrics and more like a dynamic language for those. This corresponds with the intuitive feeling of Smalltalk as a less restrictive language than C#. It is worth noting that a lot of the scores are affected by Smalltalk's unique constructor definitions. As the size of the classes increase the relative impact of this would decrease and, from the observations in this section, it seems reasonable that Smalltalk's relative score would get better.

6.3.7 Comparison

In Table 5.5 we saw a clear divide between JavaScript and Python and the rest of the languages. This divide is clear in this section as well, but is perhaps less pronounced.

It is observed in Section 6.3.3 that Python and JavaScript is scoring similarly in many metrics. Cyclomatic complexity and shallow references are the exceptions. Overall they both still gain a lot from their flexibility, especially with regard to inheritance schemes. Python might be performing slightly better if one does not put a lot of weight on the cyclomatic complexity scores, due to it solving all the core issues and having a regularly lower NCLOC score.

C#, Go and Smalltalk scores differently than Python and JavaScript. They are however, for the non-NCLOC metrics, less uniform. Their inheritance graphs are all different. Go is flatter, but with more inheritance. Smalltalk is more like a dynamic language, while C# is always defining everything modeled. The visibility rules of Smalltalk and C# makes them score better than Go, which has all attributes public. Their constructor mechanics and other details lead to differences in cohesion and cyclomatic complexity. In addition there is differences in which of the core issues they are able to solve, and some other individual problems for the different languages. There is overall many more differences between the static languages than there is between the dynamic languages.

Declaring a single language *best performing* based on the gathered data would be somewhat meaningless. No language scores better than all others in all the metrics, which means that what is the best language would depend on the context. Python might be good candidate, due to it being the only language to solve all the core issues, it having the consistently best NCLOC score and overall decent scores otherwise. It does however score worse than the other languages in the shallow references score and not strictly better in cyclomatic complexity or class cohesion. If one were to weigh these metrics more, then one could easily end up with JavaScript or even Smalltalk as a good candidate for the *best* language. If one also were

to weigh the tree based metrics very little, maybe C#? It is hard to find a realistic configuration that would lead to Go being the optimal language for the task, but as it is not strictly worse than the others in all metrics one cannot discount that such a configuration exists.

6.3.8 Generalization

In this section the observations from the earlier sections is attempted generalized to attributes of the languages, and in that way extended to be applicable to other languages and design patterns than the ones tested.

Inheritance Schemes and Duck-Typing

A common theme in the above discussion was that mechanics which allowed languages to not explicitly define things made the languages score well. Duck-typing was one such mechanic and probably the one with the greatest overall impact. It allows for not explicitly defining interfaces, which gives an advantage in the metrics NCLOC, the tree based metrics and shallow references. For every case tested here there was one such interface, although for the Adapter case it was only viable for the Smalltalk solution and in the Prototype case only for the JavaScript solution. It seems likely that this would be the case for other patterns as well. One would therefore expect other language with duck-typing or other similar mechanics to be good at implementing design patterns.

It is interesting to note that the Structural Typing of Go, though in a sense very similar to duck-typing, gave little advantage for most metrics. In Go one still had to write the interfaces, one just did not have to declare one was implementing them. This gave a much smaller gain. As discussed in Section 6.3.5 it gave a flatter tree structure, but whether this is a good thing can be debated.

Multiple inheritance was relevant in the adapter case. As described in Section 4.4.3 the case was designed to make multiple inheritance applicable. The tested languages using multiple inheritance, Python and JavaScript, scored significantly better than the other languages in most metrics for the case. One could to some degree expect this to be extended to other languages with multiple inheritance. However, it was also noted in the analysis that a major part of the success was due to easily choosing which of the inherited methods to apply. Go does for instance support multiple inheritance, but has to explicitly when calling the function choose which method to apply, making using its multiple inheritance mechanic unsuitable for the adapter case. This point is elaborated in Section 6.5.4

Could one expect other design patterns to also benefit from multiple inheritance? For patterns related to combining functionality of classes, then maybe. So cases based on the Bridge pattern or other cases based on the adapter pattern would be candidates. Most patterns are however not based around combining functionality from two sources an are thus probably not benefiting from multiple inheritance. Furthermore the case studied was particularly designed to be solvable by multiple inheritance. One therefore

cannot expect the gain to be as good (or a gain at all) in all other cases for the patterns.

Low Overhead

It was also a cost, especially to the cyclomatic complexity and NCLOC metric, associated with languages explicitly declaring things that in other languages were assumed. For instance Smalltalk has its explicit static and dynamic constructors, the static languages have to declare attributes, C# has to explicitly pass arguments to inherited constructors, Go and C# has to declare main methods, etc. These things may individually be small, but having many of them and applying them several times in an implementation could cause significantly increased scores. Some of these extra declarations scale on the number of classes, some scale on the content of the objects and some don't scale at all. For these cases, since they defined many but small classes, the ones scaling on the number of classes were the ones affecting the case the most. JavaScript gained a fair amount of score from its slighly more efficient prototype definitions and Smalltalk lost a lot of score from its double constructors.

In a more real world setting one might expect this to change, as the cases studied here were often defined with small objects to avoid noise and increase ease of implementation. On the other hand, most design patterns in [28] have relatively many modeled classes, compared to defined functions or attributes. One would therefore expect this ratio to somewhat carry over to actual cases as well, at least if the classes used are relatively cohesive. This might make overhead related to class definitions more expensive than other overhead in the language for design patterns than other coding problems. It is however important to keep in mind that any extra definition, regardless of what it scales on, is ultimately a detriment to the implementation if it does not also have some kind of a positive effect.

Attribute Visibility

Having attributes only visible within a class can be positive with respect to the shallow references metric, as seen for the C# and Smalltalk implementations. It is however not purely positive, as it might require extra work to pass them between classes, as certain patterns require. For the studied cases this was mainly a problem for the adapter implementations that relied on composition. For this case even the shallow references metric was affected, as it required the introduction of additional functions, which had to be publicly visible. Therefore, having protected variables are not without cost. For C#, unlike Smalltalk, it is however a choice. It could in cases where it is beneficial with a public attribute use one. Having this mechanic would be harder outside a language with explicit attribute declaration, so there is a cost associated with the flexibility of C#, although it is hidden.

There seems little reason for this to be less true for other languages or patterns than the ones tested. Private variables are positive with respect

to polluting the namespaces, but sometimes comes with costs in other measures. Languages with flexible visibility are better, but might have hidden costs associated with extra definitions.

Flexibility

As seen in Section 6.2.5 there is value in having tools or mechanics in the language related to the core of the design pattern. In this section we saw the value of having an unrestrictive inheritance and typing policy. Both of these attributes are related to the flexibility of the language. One might therefore conclude that flexibility is important for languages implementing design patterns. Is this reasonable also in real world settings and for other design patterns?

Intuitively it makes sense that a language that can solve a problem in multiple ways is more likely to find a good solution to an individual problem. On the other hand, there are reasons that not every language is as flexible as possible. A language with less functionality might be easier to learn as there are fewer mechanics one needs to understand. Also a restrictive language with the right restrictions might restrict the code to a more uniform and healthy implementation. The first of these phenomena is not measured in any of our metrics, and is therefore not accounted for in the data. The second should to some degree be measured by the maintenance and code health related metrics used in this study, but it is of course possible that that they don't measure the full extent of it or that it would be more pronounced in a larger or more complex context.

From the discussion in Section 6.2 it seems that the existence of interesting *core issues* are more likely in the context of design patterns than otherwise. From the discussion for inheritance schemes and typing earlier in this section it seems the same is true for flexibility related to this. Therefore it seems reasonable that for design patterns, more so than for other coding problems, the flexibility with respect to these attributes is important.

6.4 Metric Validity

6.4.1 Introduction

In this section the validity of the metrics are examined in the context of the data gathered. Do the observed differences correspond to a difference in the attributes the metrics are attempting to measure? In the analysis sections there are observed several reasons for discrepancies. Which of these are real problems and which are merely artificially inflating the metrics values? Answering these questions are relevant as the data support many of the rest of the conclusions in this chapter and gives an insight into which quality attributes a good score for a metric can be associated with.

6.4.2 Lines of Code

In Section 3.4.4 we introduced this metric with the intent that it would measure size and thus also measure implementation effort and maintainability. In Section 5.3.2 the common reasons for a good score in this metric was identified as: Avoiding definitions of classes or interfaces, syntax differences and solving the *core issues* described in Section 6.2 well.

Avoiding definitions is a boon both when it comes to implementation effort and maintainability. Anything you don't have to implement or maintain saves effort. A case could be made that without explicitly defined interfaces the code is less maintainable, as it is not as self documenting. Documenting these things with comments and mainting those comments would still probably be considerably less work.

Syntax differences can be a valid reason for an increase in maintainability- or implementation effort. For instance does it seem reasonable that Smalltalk needing to define two functions for every constructor increases these metrics, as implementing and maintaining two things is more work than one. On the other hand, it seems a stretch to claim that Python's lack of brackets gives a comparable gain in maintainability and implementation effort to the impact it has on the data.

Lastly there is the effect from the *core issues*. A short implementation of these seems like it would have the intended positive effect, as it indicates a simple solution is at the core of the problem. In some cases one could argue that the use of these uncommon attributes or functions in the language is confusing and therefore creates more of a maintainability problem than the longer, but simple, solution.

Summarizing the above points we see that, while there are some issues, the metric is mostly valid. It is also known in the literature as a fairly robust metric [21, p. 340], so the data based on it probably indicates real issues with the quality of the code.

6.4.3 Cyclomatic Complexity

This metric is supposed to measure the complexity of an implementation. As discussed in Section 3.4.5 cyclomatic complexity has some, fundamental issues regarding validity as a complexity measurement. In this section we will discuss further potential issues with the data.

For most of the cases the main difference in cyclomatic complexity was the function count. While this is an interesting metric, for well designed functions one could argue it is measures a kind of size, it lacks some of the foundation of cyclomatic complexity. The core idea of cyclomatic complexity is that it measures the number of possible paths through the code. Functions are added under the assumption that they can be connected together in any order, thus having an edge between each other in the program's graph [44]. If this is not a reasonable flow for a case this assumption breaks. So, for instance, Smalltalk having two constructors breaks with this idea because one of them will almost always be called in the context of the other.

Overall it is still a quality measure with a lot of value to it, but one has to be aware that it does not show exactly what one would expect cyclomatic complexity to show

6.4.4 Tree Metrics

These are simple to measure and therefore very clear in what they show. This also makes them more reliable as sorting out false positives becomes fairly simple on a case to case basis.

The one challenge with them were how to define inheritance in the languages without standard class inheritance. Especially defining inheritance for Go proved a bit tricky and is discusses in Section 3.5.11. For the results related to Go one therefore has to view them in the context of this definition.

Otherwise the metrics seem fairly well founded. As discussed in Section 3.4.6 the idea is that a simpler tree leads to changes having less of a cascading effect, with the change spreading to fewer other classes. One could argue that not implementing a interface due to duck-typing does not actually remove the constraints of the interface if the object still has to conform to said interface. And that thus a change in the interface will still be as impactful as before to the maintenance cost of the system, if not more, since the information about the change is hidden. While this to some degree is a fair point it should not be over valued, as, on the other hand, a change in the interface might not need to affect every class implementing the interface, maybe just the one which prompted the change.

6.4.5 Loose Class Cohesion

Class cohesion is supposed to relate to a single object not have too many responsibilities [54]. It should not contain too many functions not related to each other, since then it would be better to spit it into two functions.

Class cohesion is a bit weird when applied to the classes of these experiments, as they all are supposed to have the same functionality. So what does non-cohesion mean in this context? Mostly, as discussed in Section 3.4.6, it means the class is requiring extra functions which are less related to the case and more related to the technical aspects of implementation. It also is often increased or decreased by the need for defining extra classes. If these classes makes sense on their own they increase cohesion and if not they decrease it. Both of these factors correspond relatively well to the intent of the metric, therefore one can expect differences in it to be meaningful.

There is however a problem with the aggregation, in that adding a non-cohesive class can have a huge impact on the score, which might be disproportionate to the actual effect on the actual cohesiveness. Adding a small class can overshadow several highly un-cohesive methods.

Overall it is still a good tool for detecting indicating differences, but one has to be careful when comparing raw values in this metric.

6.4.6 Shallow References

The intent of this metric was for it to measure the pollution of the global namespace with variables. This is considered damaging since it means more of the defined objects are visible to interact with. This is a problem, since it means there is more objects for a developer to keep track of at all times and because it could allow alteration of variables through unintended interfaces. On the other hand, as pointed out in Section 3.4.7, it is not all bad, because sometimes it might lead to having easy access to a variable. It is interesting to note that, as we saw in the Adapter example, this is sometimes reflected in the metric, since additional public methods has to be defined to set or get a hidden variable.

In the composite case one can note that the choice of depth was relevant. By setting a depth of two the intent was for the residual tail of the sum to be irrelevant. For the composite case this was however not the case, since there was a array containing custom defined objects at the bottom level. If this had been counted it would have had a significant impact on the score. Apart from this the metric seemed to perform as intended, though as pointed out in Section 3.4.7 one has to be careful with comparing the scores between the cases as it is not normalized in any way.

6.4.7 Metric Coverage

In this section we explore whether the chosen metrics give a good indicator of the quality of the program. Both by summarizing the earlier sections on the individual metrics and by looking at what is lacking.

The used metrics give a good indicator of things related to the code's robustness to change, effort to implement, compactness and how readable the code is. Some of the individual metrics is best studied in the context of the code and some don't give indicators of everything they potentially could. Still, it seems all of the metrics indicate real quality issues and that when studying the differences in the context of the other measured values and the code it gives meaningful results for these quality attributes.

One thing related to the code's robustness to change this experiment does not measure is the amount done in the individual statements. When a one-line statement is not defining a function, class or interface, its content does not matter to most of the measures. A bracket holds the same value as a statement containing multiple function invocations. While this might be a somewhat tricky thing to measure properly, one could for instance have used a metric like Halstead's Approach [21, p. 344] to gain some insight into it.

There is also no measure relating to program efficiency. That is how fast the programs run and how much resources they use. Studying the run time of the different implementations could be interesting as it might, for instance, show the cost of using the scripting languages. Averaged runtimes were originally part of this experiment. It was however abandoned due to the focus of the experiment altering to be more about identifying the cause for differences in the languages, than a pure

comparison of applicability. There was also some issues with measuring accurately due to the programs being short, making the overhead more significant and GNU Smalltalk only running in a Linux environment.

There are many other possibilities for interesting metrics to apply to in this experiment. Some examples be found in [21] and it is of course also possible to invent more, like we did with Shallow References. These could show different perspectives on the quality of the code. This is however left as future work.

6.5 Comparing to Literature

6.5.1 Introduction

In this section the conclusions drawn so far are discussed in the light of the papers introduced in Section 2.3.

6.5.2 GoHotDraw

The paper[64], discussed in 2.3.2, compares the Adapter design pattern implemented in Go with similar implementations in Java and C++. While we have not used Java or C++ in our adapter experiment, we have tested them in the essay and concluded they gave very similar implementations to C#. It is therefore reasonable to compare our findings for C# and Go to the papers findings for Java/C++ and Go.

It was concluded in [64] that there were relatively few differences between the implementations in Java and Go. A similar phenomena was noted in Section 6.3.5, with respect to size and complexity measure. It does however also note that in other respects the languages were not as similar, since there were differences in the tree based metrics, shallow references and (some) in class cohesion. Given that Go always have public variables and that Go has a fundamentally different inheritance structure it is likely that these differences were present here as well, just not deemed relevant in the qualitative approach. This is probably the reason the paper reaches the conclusion that the languages are interchangeable, while my analysis does not.

Another interesting difference is that the paper states that Go's composition feature was mildly relevant for the adapter case. It used anonymous composition to have the adapter have easy access to the methods from the adaptee. In our case this was not a reasonable solution due to all the shared method names between the adapter and adaptee. It is however interesting to note the that the anonymous composition mechanic could have been relevant with only minor tweaks to one of the cases.

6.5.3 Empirical Study of Github

In this section we are comparing the results from our analysis to those of [61]. The paper is introduced in Section 2.3.3. It primarily measures the maintainability of projects in a language, but critically uses none of the

standard metrics to do so. It will therefore be instructive to see how it compares to our findings with respect to maintainability.

Of the languages studied in this text only Python, JavaScript, Go and C# are studied in [61]. It orders them, in order of decreasing maintainability as: Go, C#, JavaScript and Python. All the differences are relatively small, but the jump from Go to C# is relatively large compared to the other differences. What is interesting is that the results are, more or less, flipped from our conclusion where Python scored best on maintainability.

Does this mean that the metrics used in this text are invalid? By no means. [61] is not performed in the same context as the experiment in this thesis and the differences it shows are small. In addition, as discussed in 2.3.3, it has some validity issues and is not considered perfect authority on the issue . It does however show that the context of design patterns and the way of measuring might be relevant.

6.5.4 On the Issue of Language Support

In this section we discuss the adapter example from [10] presented in Section 2.3.4. The LayOM language used in the paper allows specifying inheritance from one source and renaming the inherited methods. The paper shows how this could be employed to create an efficient implementation of the adapter design pattern.

There are several aspects of this that are interesting. Firstly, that the paper addresses the same issue as Section 6.2 describes as the core issue of the adapter case, indicates that this might indeed be a central aspect of the pattern. Secondly, the fact that multiple inheritance is not used in [10]'s adapter implementation indicates that it might be the method manipulation which is the important part of the solution described for adapter in Section 6.2.2. Lastly the LayOM solution is also applicable in cases where no method names are shared between the adapter and adaptee, something which is not true for the multiple inheritance solutions. Maybe having layer mechanics like LayOM is the true solution to the language's core issues, but none of the languages tested had the required mechanic?

6.5.5 Conclusion

In this section the discussion was enhanced by viewing it in the context of some selected papers from the scientific field. While no completely new conclusions were reached, it adds to the picture for some of the earlier conclusions. The first paper indicates that the similarities between Go and C# with respect to implementing design patterns is no fluke, the second paper raises questions with the validity of the metrics with regard to maintainability and the last gives a better understanding of what the *core issues* of the adapter pattern are.

6.6 Summary

In this chapter the analysis from Chapter 5 was examined and conclusions were extracted. These conclusions are summarized in Chapter 7, where they are presented in the context of the research goals.

Chapter 7

Conclusion

7.1 Introduction

In this section the thesis is concluded. The most important points of the discussion are extracted and emphasized to give a clearer view of what has been achieved.

Section 7.2 contains a list of the most important parts of the conclusion. Section 7.3 expands on these conclusions and presents them in the setting of the research question and goals. Then Section 7.4 summarizes the limitations and validity of these conclusions.

7.2 Highlights

In this section the highlights of the conclusion is presented. These points are expanded on in Section 7.3, but is presented here to make my core claims more visible.

Differences There are differences between the implementations of design patterns in different languages.

JavaScript and Python Similarities Python and JavaScript are similar when it comes to implementing design patterns. Both performed well.

C#, Go and Smalltalk similarities C#, Go and Smalltalk are relatively similar when it comes to length, but differ in other quality measures.

Core Issues There are for many patterns a core issue for implementations of it related to the problem the pattern is addressing. Languages having a toolkit to easily solve these problems perform better when implementing the design patterns.

Mechanics The following mechanics were found to increase the quality of a language's design pattern implementations:

- Flexible typing and inheritance schemes.
- Low notation and/or definition overhead.

- Protected attribute visibility, or ideally a controllable mechanic.
- Having a toolkit solving the same issues as the design patterns are solving.

7.3 Revisiting the Problem Statement

The problem statement, as presented in Section 1.2, is

> How does the implementation of Gang of Four design patterns differ in object oriented languages and which attributes of the languages cause these differences?

In section 1.3 this was specified to four goals: To learn what the differences are, to examine the cause of the differences, to look for trends in these reasons and to discuss the overall suitability of different object oriented languages for design patterns. In this section we will examine possible answers to these questions extracted from the discussion in Chapter 6. Further discussion on the reasoning behind these answers, as well as a discussion of their validity and other potential answers, can be found in that chapter, but will be referenced here.

7.3.1 Differences

There were, for every case, an observable difference in the data for the different metrics and langauges. The exact nature of the differences can be found in the analysis section of Chapter 4, while trends in the data can be found in Section 6.3 and summarized in 7.3.3.

In some cases these differences corresponded to a meaningful difference in the quality of the implementation and in some cases it is best explained as a result of the metric not measuring what it is intended to measure. Separating between these two phenomena is at the core of making the other conclusions worthwhile, as the rest of the conclusions builds on these observations. Section 6.4 discusses this for the different metrics.

Section 6.4.7 concludes that these metrics are a good measure for code quality with respect to the code's robustness to change, effort to implement and how healthy the code would be to develop further. It does however also warn that, due to the context, some of the metrics don't accurately reflect all the quality metrics it could do in other cases, so a direct comparison with the same metrics in other contexts are not encouraged.

7.3.2 Causes

There were varying causes for the observed differences. A low level mapping of differences to fundamental causes can be found in the analysis section for the individual cases in Chapter 4. The analysis in Chapter 5 correlates these and studies common differences across the cases.

Section 6.2 attempts to map some of these causes back to the intended functionality of the design pattern. It concludes that for all the studied

cases there is differences caused by the languages support for solving the problems of the same nature as the design pattern was intended for. While Section 6.2.4 concludes that it is not to be expected to find such differences for every pattern and case, it states that it is likely that there are other cases where the mechanic comes into play.

7.3.3 Trends

Looking at the consistency in Table 5.3 it is clear that there are trends in how well the languages perform on a given case. These trends are explored in the 6.3 section, with each of the studied languages getting their own conclusion:

Python

The Python code for the studied cases tended toward being concise, maintainable and healthy. This was supported by a relatively good score in most metrics. The noted exceptions were shallow references and class cohesion. Some of this stems from it being the only language to have a good solution to all of the *core issues* discussed in Section 6.2. Other reasons listed are its use of duck-typing and it lacking a lot of the overhead from definitions or notation the other languages are losing score from. Its low score in shallow references are attributed to its lack of visibility control of attributes, while its score in class cohesion is linked with it using less (cohesive) helper functions.

JavaScript

The JavaScript code scores well with respect to being concise, maintainable and healthy. There were noted similarity with the Python scores. With the two being similar for all the cases except the Prototype case, which was explained by JavaScript lacking good cloning utilities. In the cases where it relied on function based prototype definitions, it generally had less noise pollution than Python, but this came at the cost of higher cyclomatic complexity. Overall though, it is still concluded that the two languages performed fairly similarly on the tested cases, with Python perhaps edging a slight lead due to its low NCLOC score and it solving all the core issues of the cases.

C#

C# scored well with respect to noise pollution and complexity, but were worse in other quality attributes. It had deep inheritance hierarchies and many lines of code. As discussed in Section 6.4 these attributes are often associated with high maintenance- and development costs. The reasons listed in Section 6.3.4 for these trends are that it lacked the toolkit to solve all the core issues properly, had rigid typing/inheritance rules and some definition overhead.

Go

Go scored bad to ok in all the metrics. It mirrored C# when it came to complexity and size. Both when it came to causes and when it came to score. Unlike C# it did however score bad in shallow references and its tree based metrics were regularly different than for all the other languages. The shallow references score is attributed to its public visibility of attributes and its constructors. The difference in tree score reflects that Go has a fundamentally different inheritance scheme. Go's inheritance tree is generally lower than C#'s, but has interfaces with more children than the other languages. Compared to the other languages, the tree related metrics were usually strictly worse. Overall this low score indicates there are issues with implementations with respect to the maintainability and overall health of the code.

Smalltalk

Smalltalk also scored similar to C# in most metrics. The exceptions here being cyclomatic complexity and the tree related metrics. The similarities in score with C# were not found to be from the same causes. For Smalltalk the main cause was its constructors. In most of the cases there are relatively many classes, making the cost of defining two constructors for each of them expensive in many of the metrics.

It does however, as discussed in Section 6.3.6 not have many of the weaknesses C# has: It solves the *core issues* relatively well, it has little overhead except for the constructors and it saves a bit from not having to define interfaces. The result is that they score fairly similar, for different reasons. In settings with larger classes it is therefore to expect that it performs better, as the constructors account for a smaller part of the total, while for smaller classes one would expect the opposite effect.

7.3.4 Suitability

Based on these trends it was, in Section 6.3.7 concluded that no single language was best. Python might be the best candidate, but since it is not strictly better in any score it can hardly be said to be best in all contexts. Similarly Go scored the worst, but also had metrics where other languages performed worse. The remaining three all had individual strengths and weaknesses. If one were to put import on shallow references one could end up with Smalltalk or JavaScript being the best. If one in addition were to ignore the tree based metrics then C# would be a good candidate.

In Section 6.3.8 there were identified four core themes in good attributes for languages to score well in the studied metrics:

- Smart inheritance or typing schemes
- Low notation or definition overhead
- Protected, or ideally controllable, visibility of attributes

- Flexibility. Especially with regard to having tools and functionality related to the problems the implemented design patterns are attempting to solve.

Of these attributes, except the visibility one, it was argued that they are more relevant in the setting of design pattern implementation than they would normally be. It is concluded that a language with all these attributes, without any other detrimental attributes, would be well suited for implementing design patterns.

7.4 Limitations

In this section the validity of the conclusion is summarized. How certain are the conclusions and what are the reasons for potential uncertainty?

As discussed in section 3.5 there were many steps taken to improve the validity of the experiment. Despite this it is still, in its very nature, gathering imperfect data. It relies on the study of a single developer's code and is analyzed by the same developer. Section 3.5.12 identifies several issues with this: That a single developer might find suboptimal solution, that the data is volatile with respect to the developers skill, that there might be bias in the case design and that when treating code as data subjective choices are made.

There is also the issue of subjective choices. As discussed in Section 3.5.4 several choices made during the implementation of code which cannot be done according to some rule. How these are solved would vary based on the taste of the person implementing and therefore makes the data have a personal and subjective tint. Similar issues also exists for the case design.

In Section 6.4 the validity of the metrics are examined in the light of the chosen cases and the gathered data. Do they have a relation to the things they are intended to measure? The section concludes that all the metrics are measuring the quality of the code. And that together they indicate something about the health of the code with respect to its robustness to change, cost of implementation and how readable the code is. It does however note that not all metrics show exactly what they do in the literature, so one has to be careful when comparing them to the same measures outside this context.

Lastly the metrics do not measure every aspect of the quality of the code. This means some of the conclusions had to be under-built using qualitative data and probably that some of the interactions between design patterns and implementation language were not fully captured. This is discussed in detail in Section 6.4.7.

Overall I would still say that the conclusions are relatively well supported given the resources and time limitations of a short master thesis.

Chapter 8

Future work

8.1 Future work

There are several ways one could extend this thesis.

One option is to simply repeat the experiment for other languages, design patterns and/or metrics. There are still untested object oriented languages, interesting design patterns, both in [28] and in other sources like [65] and metrics that could be used. Some such metrics are suggested at the end of Section 6.4.7. The potential combinations are near endless.

It would also be interesting to see the experiment performed with a larger sample size. Several developers and/or more cases. That way the trustworthiness of the data would increase.

Lastly one could alter the context of the experiment. It would be interesting to see the data for cases designed based on a different setting or design philosophy. Would Smalltalk, as we speculate in Section 6.3.6, perform better in a context with larger classes? Would the findings of this thesis still hold in a more realistic setting? It could help clear up some of the lingering doubt in the validity of the experiment, as it would to a greater degree allow for the isolation of the context.

Appendix A

Essay

A.1 Design Patterns

A.1.1 Scope

Design pattern is a term that originates in [1], an article on architecture. It referred to the concept of a reusable core solution to a set of architectural problems. In [28] this was expanded to also concern problems in software engineering, which are the ones I study in this text. After the publication of [28] many other design patterns for software engineering has been defined. Some, like the Servant Pattern found in [57] are new patterns for object orientation in general and some, like the concurrency ones described in [65], are for specialized fields. I do not review those in this essay as the goal of the thesis is to take an in-depth look at a few and I think there is enough interesting cases among the design patterns in [28].

A.1.2 Definition

In [1] design patterns are defined as:

> Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it in the same way twice. ([1, p. x])

In [28, p. 2] they reuse this definition in the setting of computer science, but also specifies that a pattern should describe a name, a problem, a solution and consequences. They then limit their scope to design patterns that are "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context".

Some consider this definition to be too imprecise. Because [28] is my main source for this part of the text I reuse its definition and scope.

A.1.3 Usage

In [28, p. 1] the goal of introducing design patterns is given as:

> Design Patterns solve specific design problems and make object-oriented designs more flexible, elegant and ultimately reusable. They help designers reuse successful designs by basing new designs on prior experience. ([28, p. 1])

The intent is for patterns to provide designers with good solutions to design problems and describes the context in which these solutions are applicable.

A.1.4 Relation to Implementation

It is important to differentiate a design pattern from its implementation. A design pattern is a set of concepts that can be used in implementations,

but they are not the implementation itself. Neither are they exact recipes for implementation of working code. For some design patterns there exists variants which differ greatly in implementation. For instance can an iterator be merely a cursor, which only keeps track of a position within an aggregate, or it can contain the full code of the algorithm using the aggregate only as a collection. For every pattern there is a section in [28] dedicated to discussing the implementation of the pattern, in which many such variants are discussed.

Another question, one which will be essential in this thesis, is how the programming language affects the implementation. What can be easily expressed varies greatly between object oriented programming languages. Whether the language supports static classes, multiple inheritance, dynamic class definitions, etc has an impact on how the pattern is best implemented and how it performs. For instance implementing the prototype pattern is almost trivial in Javascript as its object orientation centers around prototypes. For further discussion see Section A.3 and the thesis itself.

A.2 A Preliminary Experiment

A.2.1 Motivation

The first step of the thesis will be to plan the use of programming languages, design patterns and implementation rules for the thesis' experiments. To prepare for this I am therefore performing a smaller scale experiment as part of writing this essay. By implementing a few simple design patterns for all the languages I consider using in the thesis I will hopefully gain a better understanding of the relevant differences between these languages and notice some potential problems for the main experiment while they are still preventable.

A.2.2 Goal

The goal of the experiment is to gather qualitative data which will assist in formulating a good problem statement and methodology for the thesis' main experiment. Two cases based around design pattern are invented, designed and implemented. Afterwards impressions about both the code itself and the process of generating it is gathered.

A.2.3 Method

A.2.3.1 General

I will implement an example for the design patterns Factory and Observer for the languages Python, Javascript, Java, C++, C#, Smalltalk and Go. These examples will be implemented in the setting of a subway network, as this is the likely setting of the thesis itself.

For these experiments I will follow some basic intuition on what will give consistent and comparable programs, rather than basing it on strict

rules like I plan to do in the thesis' experiments. I do this partly because I want the development of this methodology to be part of the main thesis and partly because I believe trying many different solutions now will give better insight into developing this methodology. Similarly I will not use any metrics or rigorous methods in the analysis of the results, but rather just point out differences I find interesting.

A.2.3.2 The Factory Case

For this case I chose a minimalist approach. I tried to create a case containing the bare minimum necessary to exemplify of the Factory design pattern. The case is based around the trains used in the subway network. For this example the trains to have two relevant attributes: id and color. The subway net will contain trains of the types: Modern train and Classic train. Where the only difference we consider is that a modern train is white and a classic train is red.

These trains will be created by *TrainCreator* factories, one type of factory for each kind of train. These factories will have different schema for assigning id to trains, but to keep with our minimalist goal we will simplify these to the modern factory giving id = 1 and the classic factory giving id = 2.

The main part of the program will make a train of each kind using the factories and then have the trains print a description of themselves.

A.2.3.3 The Observer Case

For this example I chose to be a bit more liberal with adding context details. Partly this is necessary since the design pattern is more open and partly this is a conscious choice as I want to have tested both before the main experiments.

I will be looking at the case of passengers in a train. The passengers are observing the train and react when the train enters a new station. If they are at their destination they will leave the train. We will considered two types of passengers: Those that remembers a station name and exits the train when they see that name and those that count the stations until they've reached the number of stations they are traveling.

The main loop will create a train, add passengers of both kinds to it, run the train until all passengers have left and then print the final station for verification.

A.2.4 Results

How the code for the main experiment is to be attached to the thesis is not yet decided. The code for this experiment will be attached in the same way.

A.2.5 Discussion of Implementation Process

A.2.5.1 Scope

In this section I discuss interesting aspects of the implementation of the two examples. The goal is to point out some classes and examples of problems so that I can discuss solutions to these in the main thesis.

A.2.5.2 Deciding What Is Core

While implementing the examples I regularly found myself questioning what the essence of the example was. Which parts of it can I alter to fit the language's capabilities and which parts must remain unaltered. Obviously the final result must be correct and the structure of the design pattern must be unchanged. It is however not always as clear cut, as exemplified by the discussion in section A.2.5.4 and A.2.5.6. I will have to be very careful about specifying what is mandatory and what isn't for the cases in the main thesis.

A.2.5.3 Use of Uncommon Functionality

Some of the languages I have tested are fairly flexible and support some functionality that is seldom used. Should this be used if it gives a better implementation of the example? For instance both Python and Java support abstract classes, but they are seldom used in Python and does not fit Python's philosophy of putting the responsibility for correctness on the user. It feels wrong in Java to implement Factory without it and wrong in Python to implement with it, despite the two languages having the same capabilities. A more extreme case is the abstract class in Smalltalk and JavaScript where one can simulate it by throwing errors. Should one do this or just put the onus on the user?

A.2.5.4 Use of Structural- and Duck Typing

Python and JavaScript supports duck typing and Go supports structural typing. I will have to make a choice regarding the degree I can use this. Do I for instance, in the factory case for Python, need to implement a *TrainCreator* class and then extend it or could I just create *ClassicTrainCreator* and *ModernTrainCreator* directly with the appropriate methods? For this experiment I chose not to, but it is something I will need to consider in the main experiment.

A.2.5.5 Oversimplification of the Factory Case

I chose to have the color of the trains be the only distinguishing factor between modern- and classic trains in the factory example. In hindsight I wish I had made them have some different attributes or methods as well. Differences between languages were obscured due to this simplification.

For instance in JavaScript I only had to clone the prototype *Train* to create *ModernTrain*, hiding the differences in how methods are added.

A.2.5.6 Collections for Observers

One surprising source of difference between the implementations were how they handled the train's collection of passengers. The fact that passengers might leave the train during the notification loop in which they are updated puts requirements on how they are stored and iterated. For many languages this could be solved by using a list-like structure and iterating backwards. How simple this was varied a lot. For instance does Go not support generic lists and C++ lacks routines for *removeByValue*. I often felt myself questioning how radically I could change the storage solution or iteration algorithm. In the future I should be clear about this.

A.2.5.7 Skill difference

I know some of the languages used in the experiment better than others. For instance I have used Python for years, but for Smalltalk and Go these were my first non-HelloWorld programs. This manifests in two ways. Firstly I will, due to experience, more often find good solutions in Python than in Smalltalk. Secondly my coding style is formed by years of coding in the languages I know well so I subconsciously structure my code in ways which favor these languages. This is a bias I will have to handle or account for in my thesis.

A.2.6 Conclusion

As seen in Section A.2.5 choosing rules for how to implement code in a uniform manner in the different languages will not be a trivial issue. It will be important to spend time researching the methodology of similar experiments, define my approach well and to have a strategy for amending it after the experiment has begun.

A.3 Programming Languages

A.3.1 Motivation

In this section I discuss the languages I am considering using in the thesis. These languages are a combination of languages I know well and languages I think would be interesting to compare these to. The goal is to create a basis for selecting a subset of these to use in the coding experiments in the main thesis. I pay special attention to the language's handling of object orientation because this is a central concept for the design patterns in [28].

A.3.2 Concepts

In this section I define some useful concepts for analyzing programming languages and discuss how the languages we are using fit with these.

A.3.2.1 Static- and Dynamic Typing

[12, p. 4] offers two definitions for static typing. A strict one: "the type of every expression can be determined " and one less strict: "all expressions are guaranteed to be type consistent although the type itself may be statically unknown ". In this text we will employ the latter.

Of the languages we are considering Java, C#, C++ and Go are statically typed. They are compiled and you get type errors from the compiler. Python, SmallTalk and JavaScript are dynamically typed. If you try, for instance, to call an integer as a function in these languages you are not told of any problems before you reach the relevant part of the code during run-time execution.

A.3.2.2 Strong- and Weak Typing

These concepts were originally defined in [40, p. 52] as "Strong type checking means that whenever an object is passed from a calling function to a called function, its type must be compatible with the type declared in the called function". This is however not a universally accepted definition. For instance in [41] it is classified by the amount of information carried by the type of an expression.

Both definitions have the issue of no common language being completely compliant with their definition of strong typing, so they are often relaxed to languages that mostly fulfill it. Leading to both having large gray-zones in which the typing is ill defined. In this text I have employed the definition of [40], as this is the closest to what I want to express: The flexibility of the language with regard to type.

According to this definition most of the languages we are considering are strongly typed, since they mostly require defined input and do little automatic conversion. They all have small exceptions, like Python automatically converting to boolean or Java converting to String before performing addition with another String.

The only language on our list which is weakly typed is JavaScript, since it famously does a lot of automatic type conversion. It is also worth noting that the common use of pointer arithmetic and conversion in C++ and Go makes them less type safe than other languages it is natural to compare them to, like Java or C#. However, they are still so strict I would consider them strongly typed.

A.3.2.3 Explicit Pointers

Most object oriented languages have pointers in the sense that they allow reference to a value. For instance in Java one can view all objects variables as pointers. They do not contain the value of the object, but rather a reference to it. By explicit pointers we therefore mean something more specific, namely that it is possible to directly manipulate these pointer values by doing actions like dereferencing, referencing or pointer arithmetic.

Python, SmallTalk, JavaScript and Java does not support these kind of pointers. C# technically supports C++ like pointers, but it requires the *unsafe* flag to be turned on, so it is seldom used in practice [45]. C++ and Go both support explicit pointers, but, of the two, only C++ allows for pointer arithmetic

A.3.2.4 Main Object Oriented Construct

For all the languages discussed here there is a type construct which one would naturally associate with objects and inheritance. For Python, Smalltalk, Java, C# and C++ this is the class: A template for creating objects describing both member variables and methods. Go uses structs, which specify the member variables, along with interfaces to specify the methods.

JavaScript is different in that it relies on prototype inheritance. [55] defined it as "[A language where,] to provide inheritance, objects can have a prototype object, which acts as a template object that it inherits methods and properties from. An object's prototype object may also have a prototype object, which it inherits methods and properties from, and so on.". ES5 and onward also has class syntax, but this is only top level syntactic sugar. The underlying construct is still prototype based.

A.3.2.5 Multiple Inheritance

Multiple inheritance means one type of object can inherit behavior from more than one parent. Python and C++ support this by simply listing more classes in the extend statement. Java and C# only allow extend staments for one class, but also for implementation of an arbitrary number of interfaces. Javascript allows for combining prototypes to effectively inherit from both.

Go supports multiple inheritance through allowing multiple embedding. The difference between this and subclassing is that an embedding only forwards the call to an embedding, making the chance of interference smaller. A case could be made that this is composition and not multiple inheritance, but for this purpose I think referring to it as inheritance and noting the difference makes the most sense.

A.3.2.6 Abstract typing

Abstract typing is the language's capability for creating types which cannot be instantiated directly. Java, C++ and C# support this through abstract classes and interfaces. Go only has interfaces, but supports implementing parts of an interface and then inheriting.

As discussed in A.2.5.3, Python can import an abstract class from the standard library, making it capable of abstract typing. JavaScript and Smalltalk have no direct abstract typing capabilities, but there exist conventions for throwing errors in uncallable constructors.

A.3.2.7 Visibility

Visibility is the language's support for controlling the accessibility of an object's member variables and methods. In Java, C# and C++ this is can be done by adding private/public/protected modifiers. In Go and Python everything is always public. However [63], the official Python style guide, recommends using underscores to designate variables to be treated as if they were private.

In Smalltalk an object's member variables is not accessible from the outside. GNU Smalltalk, the variant we are using in essay, does however allow accessing them through reflection [23].

In JavaScript everything is visible by default, but it allows for visibility control by defining, but not binding to this, variables and methods in the scope of its constructors.

A.3.2.8 Duck Typing

The name of duck typing comes from the saying "If it looks like a duck and quacks like a duck, it must be a duck". [26] defines it as "A programming style which does not look at an object's type to determine if it has the right interface; instead, the method or attribute is simply called or used ". Python and JavaScript does this, in the sense that you get no object type errors until you try calling an undefined method.

A.3.2.9 Structural Typing

Structural typing can be defined as "[a type system where] a type U is a subtype of T if its methods and fields are a superset of T's methods and fields" ([42]). Go is the only language we look at here with structural typing. In Go any structure that confirms to an interface is of the type the interface defines. For Java, C++, Smalltalk and C# this is not the case, as they define subtyping explicitly.

Duck typing and structural typing are related concepts, but not entirely the same. Structural typing is about subtyping, while duck typing is about allowing method calls without knowing the type. Neither Python or JavaScript has structural typing.

A.3.3 Tables

In this section I tabulate some the conclusions from A.3.2 for easy reference.

A.3.3.1 General

Language	Typing strength	Static typing	Explicit Pointers
Python	Strong	Dynamic	No
JavaScript	Weak	Dynamic	No
Java	Strong	Static	No
C#	Strong	Static	Not recommended
C++	Strong	Static	Yes
Go	Strong	Static	Yes
Smalltalk	Strong	Dynamic	No

A.3.3.2 Object Orientation Constructs

Language	Main OO Construct	Multiple inheritance	Abstract typing
Python	Class	Yes	Importable
JavaScript	Prototype	Yes	By returning error
Java	Class	Only from interfaces	Yes
C#	Class	Only from interfaces	Yes
C++	Class	Yes	Yes
Go	Struct	By composition	Yes
Smalltalk	Class	No	By returning error

A.3.3.3 Object Orientation Aspects

Language	Visibility	Duck Typing	Structural Typing
Python	By convention	Yes	No
JavaScript	By scope wrapping	Yes	No
Java	By modifiers	No	No
C#	By modifiers	No	No
C++	By modifiers	No	No
Go	Always public	No	Yes
Smalltalk	Protected except by reflection	No	No

A.3.4 Python

A.3.4.1 History

Accoding to [25] Python was created by Guido van Rossum in the early 1990s. It is currently being developed by the Python Software Foundation as an open source project supporting two major versions: 3 and 2.7. By [11] Python is one of the most used languages in the industry and it is growing in popularity.

A.3.4.2 Special Aspects

As mentioned in A.2.5.3 one of the defining aspects of Python is its extensibility and modifiability. Its object orientation capabilities therefore varies based on the context it is used in, both from conventions and import of extensions.

A.3.4.3 Similarity

Based solely the table one would expect Python and JavaScript to be fairly similar. While they certainly have more in common with each other than with the other languages mentioned, the fact that object orientation in JavaScript is based on prototypes makes for some major differences.

A.3.5 JavaScript

A.3.5.1 History

JavaScript was first created in 1995 by Brandan Eich for the Netscape browser under the name Mocha. As explained in [22] there were many competing versions of JavaScript making development of multi-compatible software hard. Now the ES dialect has grown to be the most used one and according to [71] the major browsers are mostly compliant with it. By [11] JavaScript one of the most used languages in the industry, but currently its popularity is falling. Originally JavaScript was mostly used for web design, but with the growing number of node packages it is currently being used in a larger variety of contexts.

A.3.5.2 Special aspects

For versions of JavaScript newer than ES5 there are two ways to create prototypes: Defining them directly and using the class syntax. Both styles are used in practice, the first being used partially to support legacy code and partially due to preference. For my thesis I will have to choose one such style to use.

A.3.5.3 Similarity

See Section A.3.4.3.

A.3.6 Java

A.3.6.1 History

According to [52] Java was originally developed by Sun in 1995. It is currently being developed by Oracle. The monthly TIOBE index [11] has consistently ranked it as one of the month's two most used languages since 2001. It is currently ranked as the most used language.

A.3.6.2 Special Aspects

Historically Java has relied on the Command design pattern to pass functions as arguments. Java 8 has introduced method references making it easier to use functional programming techniques in Java. It could be interesting to see if this has any impact on any design patterns.

A.3.6.3 Similarity

Java, C# and C++ are very similar languages. They are capable of many of the same things and use a similar syntax. Most similar of the three are Java and C#. They seem so similar that using both in the main experiment would yield little extra information.

A.3.7 C#

A.3.7.1 History

C# was created by Microsoft in the late 1990s as part of developing the .NET framework. According to [18] it was very similar to Java. It has since then grown both in scope and in popularity. [11] ranks it as one of the world's most popular languages, but still less popular than its direct competitors Java and C++.

A.3.7.2 Similarity

See Section A.3.6.3.

A.3.8 C++

A.3.8.1 History

C++ is one of the earliest object oriented languages. According to [49] its development began in 1979 and was heavily inspired by the Simula language. Since then it has been a very popular language and even in 2018 [11] rates it as one of its most popular languages. It is also one of the languages used in [28], making it a language with special interest for this thesis.

A.3.8.2 Similarity

C++ has some major differences from Java and C#. For instance, as shown in the tables of Section A.3.3, it allows for explicit pointers and multiple inheritance. While this is likely to have some impact on the implementation of design patterns it might not be sufficient to warrant including two of the three languages in the experiment.

A.3.9 Go

A.3.9.1 History

Go is the newest language of the ones considered here, since according to [60] its development started in 2007. Intially it was developed internally by employees of Google, but in 2009 it became fully open source and quickly grew in popularity according to [11]. As of 2018, this growth has stopped.

A.3.9.2 Special Aspects

Go is designed to be more concise and simple than the other statically typed languages. To achieve this it does not feature some of the standard functionality like generics, implicit type conversion and implementation inheritance.

A.3.9.3 Similarity

The lack of classes and the structural typing makes Go's object orientation significantly different from the other languages. I expect this will be reflected in the design pattern implementations.

A.3.10 Smalltalk

A.3.10.1 History

According to [29] the development of Smalltalk began in the early 70s at the Xerox Palo Alto Research Center. While it is still a fairly popular language, winning second place at [56], Stack Overflow's *Most Loved Language Developer Survey*, it is now rarely used in the industry. According to [11] it is currently among the 50th to 100th most used language. It is of special relevance to this text since it was one of the languages used in [28]'s implementation of design patterns.

A.3.10.2 Special Aspects

There are several versions of Smalltalk. In my experiments I chose to use GNU Smalltalk as this was easy to use directly from the command line. Many of the other major variants were made to be used in a virtual environment, which would make sterile testing harder.

A.3.10.3 Similarity

Since Smalltalk is based around defining messages for objects it has a significantly different syntax than the other languages. Also the fact that it is strongly- and dynamic typed makes it an interesting middle point between the script-like languages such as Python and the languages like Java.

Bibliography

[1] Christopher Alexander. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Rress, 1977.

[2] Khalid Aljasser. "Implementing design patterns as parametric aspects using ParaAJ: the case of the singleton, observer, and decorator design patterns." In: *Computer Languages, Systems & Structures* 45 (2016), pp. 1–15.

[3] Apostolos Ampatzoglou, Sofia Charalampidou, and Ioannis Stamelos. "Research state of the art on GoF design patterns: A mapping study." In: *Journal of Systems and Software* 86.7 (2013), pp. 1945–1964.

[4] Ian Bayley and Hong Zhu. "Specifying behavioural features of design patterns in first order logic." In: *2008 32nd Annual IEEE International Computer Software and Applications Conference*. IEEE. 2008, pp. 203–210.

[5] Kent Beck et al. "Industrial experience with design patterns." eng. In: *Proceedings of the 18th international conference on software engineering*. ICSE '96. IEEE Computer Society, 1996, pp. 103–114. ISBN: 0818672463.

[6] J.M Bieman, D Jain, and H.J Yang. "OO design patterns, design structure, and program changes: an industrial case study." eng. In: *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*. IEEE, 2001, pp. 580–589. ISBN: 0769511899.

[7] Aaron B Binkley and Stephen R Schach. "A comparison of sixteen quality metrics for object-oriented design." In: *Information Processing Letters* 58.6 (1996), pp. 271–275.

[8] Judith Bishop. *C# 3.0 design patterns*. eng. Beijing.

[9] Judith Bishop. "Language features meet design patterns: raising the abstraction bar." eng. In: *Proceedings of the 2nd international workshop on the role of abstraction in software engineering*. Vol. 2008. ROA '08 11. ACM, 2008, pp. 1–7. ISBN: 9781605580289.

[10] Jan Bosch. "Design Pattern & Frameworks: On the Issue of Language Support." In: *Proceedings of LSDF'97: Workshop on language support for design patterns and object-oriented framworks*. Högskolan i Karlskrona/Ronneby. 1997.

[11] TIOBE software BV. *TIOBE Index — TIOBE - The Software Quality Company*. 2018. URL: http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm (visited on 11/16/2018).

[12] Luca Cardelli and Peter Wegner. "On Understanding Types, Data Abstraction, and Polymorphism." In: *ACM Comput. Surv.* 17.4 (Dec. 1985), pp. 471–523. ISSN: 0360-0300. DOI: 10.1145/6041.6042. URL: http://doi.acm.org/10.1145/6041.6042.

[13] Shyam R Chidamber and Chris F Kemerer. "A metrics suite for object oriented design." In: *IEEE Transactions on Software Engineering* 20.6 (1994), pp. 476–493.

[14] James William Cooper. *Java Design Patterns: A Tutorial*. Addison-Wesley Professional, 2000.

[15] Viktor K. Decyk and Henry J. Gardner. "Object-oriented design patterns in Fortran 90/95: mazev1, mazev2 and mazev3." eng. In: *Computer Physics Communications* 178.8 (2008), pp. 611–620. ISSN: 0010-4655.

[16] Premkumar Devanbu, Thomas Zimmermann, and Christian Bird. "Belief & Evidence in Empirical Software Engineering." eng. In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. Vol. 14-22-. ACM, 2016, pp. 108–119. ISBN: 9781450339001.

[17] M Di Penta et al. "An empirical study of the relationships between design pattern roles and class change proneness." eng. In: *2008 IEEE International Conference on Software Maintenance*. IEEE, 2008, pp. 217–226. ISBN: 9781424426133.

[18] Erik Dietrich and Patrick Smacchia. *The History of C#*. 2017. URL: https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-version-history (visited on 12/17/2018).

[19] Caleb Doxsey. *An introduction to programming in Go*. 2012. URL: http://www.golang-book.com/books/intro (visited on 03/05/2019).

[20] E. Eide et al. "Static and dynamic structure in design patterns." In: 2002, pp. 208–218.

[21] Norman Fenton and James Bieman. *Software metrics: a rigorous and practical approach*. CRC press, 2014.

[22] David Flanagan. *JavaScript: The Definitive Guide*. " O'Reilly Media, Inc.", 2006.

[23] Free Software Foundation. *GNU Smalltalk Library Reference: Behavior*. 2018. URL: https://www.gnu.org/software/smalltalk/manual-base/html_node/Behavior_002daccessing-instances-and-variables.html (visited on 12/17/2018).

[24] Free Software Foundation. *GNU Smalltalk User's Guide: Tutorial*. 2018. URL: https://www.gnu.org/software/smalltalk/manual/html_node/Tutorial.html (visited on 03/05/2018).

[25] Python Software Foundation. *A.1 History of the software*. 2018. URL: https://docs.python.org/2.0/ref/node92.html (visited on 12/17/2018).

[26] Python Software Foundation. *Glossary — Python 3.7.2rc1 documentation*. 2018. URL: https://docs.python.org/3/glossary.html#term-duck-typing (visited on 12/17/2018).

[27] Adam Freeman. *Pro Design Patterns in Swift*. Apress, 2015.

[28] Erich Gamma et al. *Design patterns: Elements of reusable software components*. Addison-Wesley Professional, 1995.

[29] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[30] Object Management Group. *About the Unified Modeling Language Specification Version 2.5.1*. 2019. URL: https://www.omg.org/spec/UML/ (visited on 05/21/2019).

[31] D. Heuzeroth et al. "Automatic design pattern detection." In: vol. 2003-. IEEE Computer Society, 2003, pp. 94–103. ISBN: 0769518834.

[32] Nien-Lin Hsueh, Peng-Hua Chu, and William Chu. "A quantitative approach for evaluating the quality of design patterns." eng. In: *The Journal of Systems & Software* 81.8 (2008), pp. 1430–1439. ISSN: 0164-1212.

[33] Brian Huston. "The effects of design pattern application on metric scores." eng. In: *The Journal of Systems & Software* 58.3 (2001), pp. 261–269. ISSN: 0164-1212.

[34] M.A. Jalil and S.A.M. Noah. "The difficulties of using design patterns among novices: An exploratory study." In: *Proceedings - The 2007 International Conference on Computational Science and its Applications, ICCSA 2007*. 2007, pp. 97–103. ISBN: 0769529453.

[35] M.A. Jalil and S.A.M. Noah. "The difficulties of using design patterns among novices: An exploratory study." In: *Proceedings - The 2007 International Conference on Computational Science and its Applications, ICCSA 2007*. 2007, pp. 97–103. ISBN: 0769529453.

[36] Szasz Janos. *The algorithmicx package*. 2012. URL: http://tug.ctan.org/macros/latex/contrib/algorithmicx/algorithmicx.pdf (visited on 03/06/2019).

[37] Barry Keepence and Mike Mannion. "Using Patterns to Model Variability in Product Families." In: *IEEE software* 16.4 (1999), pp. 102–108.

[38] F Khomh and Y.-G Gueheneuc. "Do Design Patterns Impact Software Quality Positively?" eng. In: *2008 12th European Conference on Software Maintenance and Reengineering*. IEEE, 2008, pp. 274–278. ISBN: 9781424421572.

[39] Gwendolyn Kolfschoten et al. "Cognitive Learning Efficiency through the Use of Design Patterns in Teaching." eng. In: *Computers & Education* 54.3 (2010), pp. 652–660. ISSN: 0360-1315.

[40] Barbara Liskov and Stephen Zilles. "Programming with Abstract Data Types." In: *SIGPLAN Not.* 9.4 (Mar. 1974), pp. 50–59. ISSN: 0362-1340. DOI: 10.1145/942572.807045. URL: http://doi.acm.org.ezproxy.uio.no/10.1145/942572.807045.

[41] Ole Lehrmann Madsen, Boris Magnusson, and Birger Møller-Pedersen. "Strong Typing of Object-oriented Languages Revisited." In: *SIGPLAN Not.* 25.10 (Sept. 1990), pp. 140–150. ISSN: 0362-1340. DOI: 10.1145/97946.97964. URL: http://doi.acm.org/10.1145/97946.97964.

[42] Donna Malayeri and Jonathan Aldrich. "Integrating nominal and structural subtyping." In: *European Conference on Object-Oriented Programming*. Springer. 2008, pp. 260–284.

[43] B Bafandeh Mayvan, Abbas Rasoolzadegan, and Z Ghavidel Yazdi. "The state of the art on design patterns: A systematic mapping of the literature." In: *Journal of Systems and Software* 125 (2017), pp. 93–118.

[44] Thomas J McCabe. "A Complexity Measure." In: *IEEE Transactions on Software Engineering* 4 (1976), pp. 308–320.

[45] Microsoft. *Unsafe Code and Pointers - C# Programming Guide*. 2018. URL: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/unsafe-code-pointers/index (visited on 12/17/2018).

[46] Mozilla. *super - JavaScript — MDN*. 2019. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/super (visited on 05/21/2019).

[47] Sebastian Nanz and Carlo A Furia. "A Comparative Study of Programming Languages in Rosetta Code." eng. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. IEEE, 2015, pp. 778–788. ISBN: 9781479919345.

[48] Dmitri Nesteruk. *Design Patterns in Modern C++: Reusable Approaches for Object-Oriented Software Design*. eng. 1st ed. Apress, 2018. ISBN: 9781484236031.

[49] The C++ Resource Network. *History of C++*. 2016. URL: http://www.cplusplus.com/info/history/ (visited on 12/17/2018).

[50] T.H. Ng et al. "Human and program factors affecting the maintenance of programs with deployed design patterns." eng. In: *Information and Software Technology* 54.1 (2011). ISSN: 0950-5849.

[51] Bcds Oliveira, M Wang, and J Gibbons. "The VISITOR Pattern as a Reusable, Generic, Type-Safe Component." English. In: *Acm Sigplan Notices* 43.10 (2008), pp. 439–456. ISSN: 0362-1340.

[52] Oracle. *The History of Java Technology*. 2018. URL: https://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html (visited on 12/17/2018).

[53] Addy Osmani. *Learning JavaScript design patterns*. eng. Sebastopol, California, 2012.

[54] Linda Ott et al. "Developing measures of class cohesion for object-oriented software." In: *Proc. Annual Oregon Workshop on Software Merics (AOWSM'95)*. Vol. 11. 1995.

[55] Stack Overflow. *Object Prototypes — MDN*. 2018. URL: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes (visited on 12/17/2018).

[56] Stack Overflow. *Stack Overflow Developer Survey Results 2017*. 2017. URL: https://insights.stackoverflow.com/survey/2017#technology-most-loved-dreaded-and-wanted-languages (visited on 12/17/2018).

[57] Rudolf Pecinovsky, Jarmila Pavličková, and Luboš Pavliček. "Let's modify the objects-first approach intosign-patterns-first." In: *ACM Sigcse Bulletin* 38.3 (2006), pp. 188–192.

[58] L Prechelt. "An empirical comparison of seven programming languages." eng. In: *Computer* 33.10 (2000), pp. 23–29. ISSN: 0018-9162.

[59] L Prechelt et al. "A controlled experiment in maintenance: comparing design patterns to simpler solutions." eng. In: *IEEE Transactions on Software Engineering* 27.12 (2001), pp. 1134–1144. ISSN: 0098-5589.

[60] Go Project. *Frequently Asked Questions (FAQ) - The Go Programming Language*. 2018. URL: https://golang.org/doc/faq#history (visited on 12/17/2018).

[61] Baishakhi Ray et al. "A Large-Scale Study of Programming Languages and Code Quality in GitHub." eng. In: *Communications of the ACM* 60.10 (2017), pp. 91–100. ISSN: 1557-7317.

[62] Maria Riaz, Travis Breaux, and Laurie Williams. "How have we evaluated software pattern application? A systematic mapping study of research design practices." eng. In: *Information and Software Technology* 65.C (2015), pp. 14–38. ISSN: 0950-5849.

[63] Guido van Rossum. *PEP 8 – Style Guide for Python Code*. 2013. URL: https://www.python.org/dev/peps/pep-0008/ (visited on 12/17/2018).

[64] Frank Schmager, Nicholas Cameron, and James Noble. "GoHotDraw: evaluating the Go programming language with design patterns." eng. In: *Evaluation and Usability of Programming Languages and Tools*. PLATEAU '10. ACM, 2010, pp. 1–6. ISBN: 9781450305471.

[65] Douglas C Schmidt et al. *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*. Vol. 2. John Wiley & Sons, 2000.

[66] SonarSource. *Metric Definitions*. 2019. URL: https://docs.sonarqube.org/latest/user-guide/metric-definitions/ (visited on 03/12/2019).

[67] SourceMaking. *Source Making: Design Patterns*. 2019. URL: https://sourcemaking.com/design%5C_patterns (visited on 05/21/2019).

[68] N Tsantalis et al. "Design Pattern Detection Using Similarity Scoring." eng. In: *IEEE Transactions on Software Engineering* 32.11 (2006), pp. 896–909. ISSN: 0098-5589.

[69] P Wendorff. "Assessment of design patterns during software reengineering: lessons learned from a large commercial project." eng. In: *Proceedings Fifth European Conference on Software Maintenance and Reengineering*. IEEE, 2001, pp. 77–84. ISBN: 0769510280.

[70] Search & write. *Structuring a thesis*. 2019. URL: https://sokogskriv.no/en/writing/structure-and-argumentation/structuring-a-thesis/ (visited on 03/20/2019).

[71] Juriy Zaytsev. *ECMA 6 Compability Table*. 2018. URL: https://kangax.github.io/compat-table/es6/ (visited on 12/17/2018).