Improving Object-Oriented Programming by Integrating Language Features to Support Immutability

William Flageol

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of
Doctor of Philosophy (Software Engineering) at
Concordia University
Montréal, Québec, Canada

March 2023

Abstract

Improving Object-Oriented Programming by Integrating Language Features to Support Immutability

William Flageol, Ph.D.
Concordia University, 2023

Nowadays Object-Oriented Programming (OOP) is widely considered as the de-facto general programming paradigm. While successful, OOP is not without problems. In 1994, a book was published with a set of 23 design patterns addressing recurring problems found in OOP software. These patterns are well-known in the industry and are taught in universities as part of software engineering curricula. Despite their usefulness in solving recurring problems, these design patterns bring a certain complexity in their implementation. That complexity, however, is influenced by the features available in the implementation language. In this work, we choose to look at design patterns by focusing on the problems they attempt to solve and the language features that can be used to solve them. We aim to contribute a guideline for designing extensions for better support of immutability in OOP languages.

Our main goal is to investigate the impact of specific language features on OOP and make recommendations to improve OOP languages in general. We first perform a mapping study to catalogue the language features that have been proposed in the literature to improve design pattern implementations. From those features, we focus on investigating the impact of immutability-related features on OOP.

We then perform an exploratory study measuring the impact of introducing immutability in OOP software with the objective of establishing the advantages and drawbacks of using immutability in the context of OOP. Results indicate that immutability may produce more granular and easier-to-understand programs.

We also perform an experiment to measure the impact of new language features added into the C# language for better immutability support. Results show that these specific language features improve the quality of life for developers aiming to implement immutability in OOP.

We finally present a new design pattern aimed at solving a problem with method overriding in the context of immutable hierarchies of objects. We discuss the impact of language features on the implementation of this pattern by comparing it in different programming languages, including Clojure, Java, and Kotlin.

Finally, we implement these language features as a language extension to Common Lisp and discuss their usage.

Acknowledgments

Contributions of Author

William Flageol, Yann-Gaël Guéhéneuc, Mourad Badri, and Stefan Monnier, "A Mapping Study of Language Features Improving Object-Oriented Design Patterns", submitted to Information and Software Technology, under review (minor revision sent on March 6th). This work is discussed in Chapter 3.

William Flageol, Yann-Gaël Guéhéneuc, Mourad Badri, and Stefan Monnier, "A Multi-method Exploratory Study on the Impact of Immutability on Object-oriented Software Development", submitted to Empirical Software Engineering, under review. This work is discussed in Chapter 4.

William Flageol, Yann-Gaël Guéhéneuc, Mourad Badri, and Stefan Monnier, "A Multi-method Empirical Study on New Features for Immutability Support in C#", submitted to Journal of Systems and Softare, under review. This work is discussed in Chapter 5.

William Flageol, Yann-Gaël Guéhéneuc, Mourad Badri, and Stefan Monnier, "Design Pattern for Reusing Immutable Methods in Object-Oriented Languages", In EuroPLoP '23: 28th European Conference on Pattern Languages of Programs, July 05–09, 2023, Koslter Irsee, Germany.

Contents

List of Figures **xiii**

List of Tables **xiv**

1 Introduction **1**

1.1 Research Hypothesis 4

1.2 Thesis Overview and Contributions 4

1.2.1 Chapter 2: Background 4

1.2.2 Chapter 3: Mapping Suggested Language Features for Improving Object-Oriented Design Patterns 4

1.2.3 Chapter 4: Exploring the Impact of Immutability on Object-oriented Software Development 5

1.2.4 Chapter 5: Studying New Features for Immutability Support in C# 6

1.2.5 Chapter 6: Solving Immutable Method Reusability Problems with a New Design Pattern 6

1.2.6 Chapter 7: Conclusion and Future Work 7

2 Background **8**

2.1 On Object-Oriented Programming 8

2.1.1 SOLID and GRASP Principles 10

2.2 On Design Patterns 13

2.2.1 Creational Patterns 15

2.2.2 Structural Patterns 17
2.2.3 Behavioural Patterns 19
2.3 On Design Pattern Improvements 23
2.4 On Mutability and Immutability 24
2.4.1 Impact of Immutability 25
2.4.2 Immutability in Programming Languages 26

3 Mapping Suggested Language Features for Improving Object-Oriented Design Patterns 31

3.1 Introduction 31
3.2 Background 33
3.3 Related Studies 34
3.3.1 Keywords for the Search Query 38
3.3.2 Databases for the Search Query 38
3.3.3 Query Building and Execution 38
3.3.4 Filtering 39
3.3.5 Snowballing 41
3.3.6 Compilation Form 41
3.4 Results 42
3.4.1 Meta-Programming 45
3.4.2 Reactive Programming 52
3.4.3 Functional Programming 53
3.4.4 Object-Oriented Programming 56
3.4.5 Design Patterns 61
3.4.6 Measures 61
3.4.7 Types of Papers 65
3.4.8 Timeline 66
3.5 Discussion 67
3.5.1 Language Features 67

3.5.2 Design Patterns 72
3.5.3 Measures 73
3.5.4 Empirical Studies 74
3.6 Recommendations 75
3.7 Threats to Validity 77
3.7.1 Interval Validity 77
3.7.2 External Validity 77
3.7.3 Construct Validity 78
3.7.4 Conclusion Validity 78
3.8 Conclusion 78

4 Exploring the Impact of Immutability on Object-oriented Software Development 81

4.1 Introduction 81
4.2 Background 83
4.2.1 This Study 83
4.2.2 Motivating Example 84
4.2.3 Related Studies 86
4.3 Method 87
4.4 Quantitative Measures 89
4.4.1 Data Collection 89
4.4.2 Statistical Tests 93
4.4.3 Research Question 4.1 95
4.5 Qualitative Measures 97
4.5.1 Data Collection 98
4.5.2 Statistical Tests 99
4.5.3 Research Question 4.2 100
4.5.4 Research Question 4.3 102
4.6 Discussion 103

4.6.1 Main Research Question 103

4.7 Threats to Validity 109

4.7.1 Internal Validity 110

4.7.2 External Validity 112

4.7.3 Construct Validity 112

4.7.4 Conclusion Validity 112

4.8 Conclusion 113

5 Studying New Features for Immutability Support in C# 114

5.1 Introduction 114

5.2 Background 116

5.3 Method 117

5.3.1 Language Features 117

5.3.2 Pilot Study 121

5.3.3 Participants 122

5.3.4 The Base Programs 123

5.3.5 Survey 125

5.4 Results 126

5.4.1 Program Analysis 127

5.4.2 Survey Analysis 131

5.4.3 Interview 135

5.5 Discussion 136

5.5.1 Research Question 5.1 136

5.5.2 Research Question 5.2 138

5.5.3 Research Question 5.3 139

5.5.4 Main Research Question 140

5.6 Threats to Validity 141

5.6.1 Interval Validity 141

5.6.2 External Validity 142

5.6.3 Construct Validity 142

5.6.4 Conclusion Validity 143

5.7 Conclusion 143

6 Solving Immutable Method Reusability Problems with a New Design Pattern 146

6.1 Introduction 146

6.2 Problem outline 147

6.2.1 Specifications 148

6.2.2 Problem 1: Inheritance and immutability 149

6.2.3 Problem 2: Return-value Polymorphism 150

6.3 Solution: The Immutable Factory Method 150

6.3.1 Addressing Problem 1: Inheritance 151

6.3.2 Addressing Problem 2: Return-type Polymorphism 152

6.4 Variants in other languages 154

6.4.1 Clojure 154

6.4.2 Kotlin 155

6.5 Discussion 157

6.5.1 General Form 157

6.5.2 Disadvantages and Limitations 158

6.5.3 Language Feature: Functional Updating 159

6.5.4 Dynamic vs Static Typing 160

6.6 Common Lisp Implementations 161

6.6.1 Implementation in base Common Lisp 161

6.6.2 Extending Common Lisp 163

6.7 Conclusion 165

7 Conclusion and Future Work 167

7.1 Contributions and Findings 167

7.2 Future Work 170

Bibliography **172**

List of Figures

1 Overall flow of the thesis . . . 3

2 Implementation example of Factory Method . . . 35

3 Flowchart of the steps in our approach . . . 37

4 Mind map of paradigms, languages, and features for improving design patterns 47

5 Number of papers by design pattern . . . 63

6 Papers distribution by year . . . 67

7 Meta-programming papers distribution by year . . . 68

8 AOP papers distribution by year . . . 68

9 Non-AOP papers distribution by year . . . 69

10 Google Trends results on Meta-programming, Aspect-oriented Programming and Functional Programming . . . 69

11 Mutable representation of a Command invoker . . . 85

12 Immutable representation of a Command invoker . . . 86

13 Simple class diagram of the geometry library specifications . . . 148

14 Specifications . . . 151

15 General Form . . . 158

List of Tables

1 Database Query 40

2 Query Results by Database 41

3 Publication Venues 44

4 Summary of papers on language features for improving design patterns implementations 46

5 Language features per design patterns 62

6 Measures by usage in papers 64

7 Study types 66

8 Programs Summary 89

9 Program measures 93

10 Measures on participants 99

11 Mann-Whitney U test results between groups and product measures, no corrections done for multiple hypothesis tests 100

12 Survey answers averages 101

13 Mann-Whitney U test results between groups and survey answers 102

14 Summary data on survey answers by each team 105

15 Summary data on survey answers by grades 106

16 Program Measures 131

17 Base Program Measures 131

18 Mann Whitney U test results 131

19 Student Answers for Appraisal of the Base Program 132

20 Professional Developers Answers for Appraisal of the Base Program 133
21 Student Answers for the Appraisal of the Implementation of the New Functionalities 133
22 Professional Developers Answers for Appraisal of the Implementation of the New Functionalities 134
23 Student Answers for NASA Task Load Index 134
24 Developer Answers for NASA Task Load Index 135

Chapter 1

Introduction

For the last three decades, we have widely considered object-oriented programming (OOP) the de-facto general programming paradigm. The paradigm is often used in industry and is taught in universities as the basis of computer science and software engineering. Most mainstream programming languages, such as C++, C#, Java, JavaScript, Python, and many others, use OOP as their main paradigm.

OOP is not the only programming paradigm, however. Functional programming has gained in popularity in recent years, with many mainstream languages, like C# and Java, introducing new functional programming features, and new languages, like Kotlin and Rust, also having many features from the paradigm. Meta-programming is another paradigm which has seen some use throughout the years, for example with the high interest in aspect-oriented programming in research between 2002 and 2016.

Modern programming languages are adopting features from multiple paradigms and combining them with OOP. Is this only a matter of preference of programmers, or are there situations where OOP struggles to provide clear solutions? We believe we can find the answer to this by looking at OOP design patterns.

Design patterns are generic solutions to recurring problems in software engineering. The most popular set of design patterns was published in 1994 by Gamma et al. [33]. In their book, they describe a set of 23 design patterns, each a solution to a problem found in OOP software. In this thesis, we study these design patterns but choose to focus on their underlying problems instead of their solution. These underlying problems show situations where OOP does not have a clear solution: these situations require the use of design patterns to solve efficiently.

However, design pattern implementations vary depending on the programming language used. Some language features might make a design pattern trivial or obsolete [67]. **We are interested in studying which language features have the most potential to improve OOP by solving the underlying problems of various design patterns.**

Initially, we considered making the combination of functional and object-oriented programming the main focus of this thesis. However, there also seemed to be promising trends with meta-programming and aspect-oriented programming. To gain a better understanding of the current knowledge of language features to improve OOP, we opted to start with a mapping study.

We perform a mapping study of the suggested language features to improve design pattern implementations and catalogued 18 language features with the potential to improve OOP. These features were divided into functional programming, meta-programming, and pure OOP features. Because our initial idea was the combination of functional programming and OOP, and we already had some insight as to how to make this work, we decided to take a deeper look into functional programming features. One feature, in particular, becomes the main focus: immutability.

Immutability is a property of functional programming where programs are considered as chains of functions with no side effects. In an immutable program, a function may not modify (or mutate) the value of any of its arguments, or modify its environment in any way. While many advantages of immutability have been claimed [1, 13], combining OOP with immutability poses interesting challenges. In particular, we were interested in knowing if there would be significant disadvantages, such as an increase in complexity or a decrease in the understandability of programs.

We performed an exploratory study on the impact of immutability on OOP software development. In this study, we performed an experiment with a group of 67 undergraduate students. The goal was to determine whether any disadvantages of using immutability

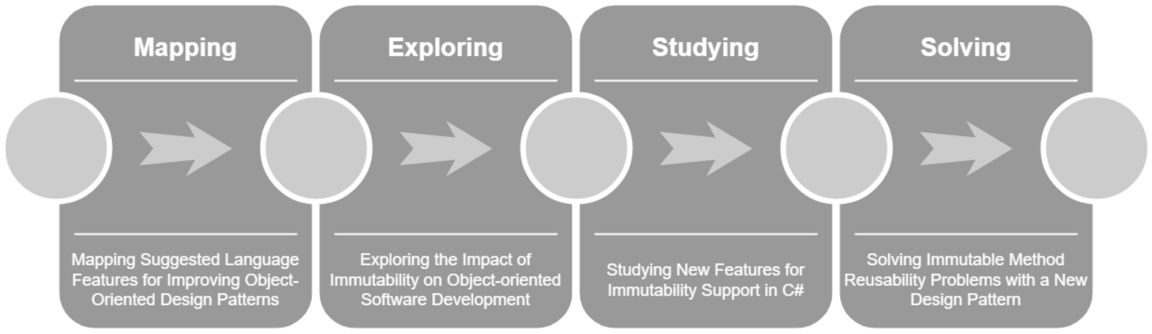
Mapping
Exploring
Studying
Solving
Mapping Suggested Language Features for Improving Object-Oriented Design Patterns
Exploring the Impact of Immutability on Object-oriented Software Development
Studying New Features for Immutability Support in C#
Solving Immutable Method Reusability Problems with a New Design Pattern

Figure 1: Overall flow of the thesis

outweighed the possible advantages. Our results showed the opposite: we could not find any concrete disadvantage in using immutability, while we found some advantages with team communication and code granularity.

We then studied specific language features related to immutability support recently added to C#. We performed an empirical study to determine the impact of adding these features to the language. The experiment involved 10 graduate students and two professional developers. We found that the immutability-related language features had a positive impact when used to implement immutable programs. In particular, immutable programs implemented using the new features had a higher maintainability index, lower cyclomatic complexity, and lower code size.

Finally, we wanted to present a solution to a problem by adding immutability features to an existing language. We examined a situation where combining immutability with OOP subtyping leads to code duplication and scalability issues. We first proposed a new design pattern to solve this problem in Java. While the code duplication and scalability issues were mitigated, they were not eliminated completely, and the solution had some added complexity. By studying how other languages would solve this situation, in particular Clojure and Kotlin, we concluded that dynamic typing and functional updating were key features in solving the problem. We then created an extension to add functional updating to Common Lisp and implemented a solution with no scalability or code duplication issues.

Figure 1 shows the flow of the studies in this thesis discussed in Chapters 3, 4, 5, and 6.

1.1 Research Hypothesis

Our thesis statement is: **Object-Oriented Programming can be improved by adding language features which solve the underlying issues behind design patterns.** We show this by mapping language features to improve design pattern implementations, exploring the usage of immutability in OOP, studying specific language features related to immutability, and finally presenting a new design pattern specific to immutable OOP and solving its underlying issues by integrating language features into an existing language.

1.2 Thesis Overview and Contributions

This section presents an overview of the structure of this thesis and its contributions. We summarize each chapter of the thesis in what follows.

1.2.1 Chapter 2: Background

In this chapter, we present an overview of object-oriented programming. We discuss the basic principles of the paradigm (SOLID, GRASP) and present the 23 design patterns which will be the focus of much of the next chapters. We also discuss our initial ideas for improvement on these design patterns using language features. Finally, we present an overview of the concept behind immutability and the research that has been done on the subject in the literature.

1.2.2 Chapter 3: Mapping Suggested Language Features for Improving Object-Oriented Design Patterns

In this chapter, we present our mapping study on the suggested language features to improve OOP design patterns. We perform a systematic mapping study to catalogue language features in the literature claiming to improve object-oriented design pattern implementations, as well as how primary studies measure these improvements. We perform a search in three databases, yielding a total of 874 papers, from which we obtained 34 relevant papers. We extract and study data about the language features claiming to improve design patterns implementations, the most often cited design patterns, the measures used to assess the improvements, and the case studies and experiments with which these improvements were studied.

Using the results, we catalogue 18 language features claimed in the literature to improve design patterns and categorize them into paradigms. We find that some design patterns are more prevalent than others, such as Observer and Visitor. Measures related to code size, code scattering and understandability are preferred. Case studies are done in-vitro, and experiments are rare.

1.2.3 Chapter 4: Exploring the Impact of Immutability on Object-oriented Software Development

In this chapter, we present the first multi-method exploratory study on the impact of immutability in OOP software development. We perform an experiment using a class of 67 undergraduate software-engineering students. We divide the participants into two groups: one (treatment group) developing software following the rules established by Bloch [13] to achieve transitive immutability and the other (control group) developing the same software using mutable objects. We collect data from the developed software and survey the participants.

Results show that the treatment group had a lighter workload, less difficulty in implementing the specifications, and less complex code than the control group. They show that the negative impact of immutability on software development is outweighed by its positive impact. They support implementing immutability in object-oriented programming languages and using it in software development.

1.2.4 Chapter 5: Studying New Features for Immutability Support in C#

In this chapter, we present a multi-method empirical study on the impact of a set of new immutability-related features recently added to C#. We perform a multi-method empirical experiment with 10 students and two professional developers. The experiment consists of inspecting and understanding a base program written in immutable OOP and extending it by adding new functionalities. The participants were separated between a treatment and a control group. Only the treatment group would use the new features to develop their program. We collect qualitative and quantitative data from the program using manual analysis and static code analysis. We also survey the participants to collect subjective data on their experience. We finally perform an interview with one of the professional developers to obtain their opinion on the new features.

Results indicate that the new features may have a positive impact on software maintainability. Quantitative data shows that the Maintainability Index of the programs submitted by the treatment group is generally higher. Qualitative analysis shows that the control group, which did not have access to the new features, ended up using ad-hoc implementations of the new features to solve the problem, instead of going for the more traditional design patterns-based based approach (i.e., the Visitor pattern).

1.2.5 Chapter 6: Solving Immutable Method Reusability Problems with a New Design Pattern

In this chapter, we present a problem with method overriding when combining immutability and OOP. Non-destructive mutators (methods used on immutable objects which return a new object instead of modifying the receiver) cannot easily be reused through inheritance. A naive approach creates code duplication and has scalability issues. We analyse an example of this overriding problem and propose a solution in a new design pattern based on the factory method pattern. We also discuss the advantages and limitations of this pattern, as well as implementations in Clojure, Java, and Kotlin. We also identify and discuss the language features that mostly affect the implementation of this pattern.

Our proposed design pattern helps mitigate some of the code duplication and scalability problems of a naive approach. However, the inclusion of a functional updating language feature is required to completely remove the scalability issues. We demonstrate this by extending Common Lisp by integrating functional updating as a new feature and solving the code duplication and scalability issues.

1.2.6 Chapter 7: Conclusion and Future Work

In this chapter, we summarize the results and contributions of this thesis. We then discuss potential avenues for future research.

Chapter 2

Background

2.1 On Object-Oriented Programming

Object-oriented programming is a programming paradigm that was first implemented in the Simula, Smalltalk and Lisp languages in the 1970s. The idea behind the paradigm is to divide data and behaviours of a program into smaller, understandable pieces called “objects”. This can allow the programmer to get rid of global data and free functions and encapsulate both of them into objects. Objects can have “responsibilities”, a concept to which we will come back later when we take a look at the SOLID principles, and can communicate with other objects through “messages”, or “methods“ (to avoid confusion, and because the distinction is not relevant to the subject of this thesis, we will use the term “method“ from now on).

Objects can be created through a special kind of object, called a ”class”. Classes are objects that define and create other objects. In early OOP languages, classes are full-fledged objects and are manipulated as any other objects, without specialized keywords. In later OOP implementations, such as C++ and Java, classes are a special kind of construct, outside of the object system, that require special keywords, such as “new”. This may in part be due to the addition of static typing, which was not present in earlier OOP languages. This gives classes a special status in these languages and makes it more difficult for a developer to control how objects are created from them (we will discuss this when we take a look at design patterns). Because of this special status given to classes, perhaps these later OOP implementations would be better called "class-oriented languages", or "class-based OOP".

In Smalltalk, every operation is executed by method calling. For example, arithmetic operations are methods called on objects representing numbers. Creating an object is a method called on a class object. A programmer can create new object types (i.e., classes) and extend existing ones by adding new methods. Most OOP languages use “single dispatch” method calling. This means that each method is associated to only one object type (i.e. the method is declared as part of the type) and will typically only have access to the internals of objects of this particular type (sometimes through a special keyword such as “this“). This is not a mandatory property of OOP; some OOP implementations (e.g., CLOS) support “multimethods”, with which a method can be associated to any number of object types. These methods "dispatch" the method calls towards multiple arguments, instead of a single target object.

This method calling style used by OOP may lend itself more naturally to imperative-style programming, where each line of code is executed one after the other (as opposed to declarative-style where the order of execute does not matter). This is likely why the vast majority of OOP languages are imperative languages. This also leads to another significant property of OOP languages: mutability.

Mutability is a property present in imperative languages. A program can modify (or “mutate”) data as it executes. A line of code can change the value of a binding (a value given a specific name, often called "variables", although they do not always vary) at any time. When a method does this in OOP, we call it a “side-effect”, and the method is also said to “mutate” the object(s). In contrast, a method that only returns a value and does no mutation can be said to be “side-effect free” or “referentially transparent”. This distinction is important because referentially transparent, or “immutable”, objects have some desirable properties, such as being easier to compose and easier to persist in databases, as well as sometimes making code easier to understand. We will discuss immutability in depth in Section 2.4.

The last concept we will look into in OOP is inheritance. When defining an object type, OOP languages allow it to “inherit” another type. This inheritance relationship makes it so that objects of the inheriting, or child, type has access to (some of) the methods and data of its parent type. This allows code reuse between object types and is one of the main reusability mechanisms in OOP. There are many different implementations of inheritance in OOP languages. Most modern languages only allow single inheritance, where an object type can only inherit from one other type. Some languages, such as C++ and Common Lisp, allow for multiple inheritance, where object types can inherit from as many types as wanted.

Single inheritance is often preferred by programming languages because of its simplicity, but multiple inheritance can model certain situations much more efficiently. This is why single inheritance languages also usually introduce the concept of “interfaces”, special object types that contain method definitions, usually allowing no implementation. In these languages, objects can be subtypes of, or “implement”, as many interfaces as they like, but can still only inherit from one class. Subtyping helps alleviate the limitations of single inheritance, but can sometimes lead to code duplication when multiple objects have the same method implementation (which interfaces typically cannot hold, so they must be replicated in each implementing object).

Code reuse in OOP is not limited to only sharing method implementation. Inheritance and subtyping also introduce the concept of object polymorphism, where a method passed to an object will look-up a different implementation depending on the class of the target object. This allows a piece of controlling code to be reused in multiple ways by changing the type of its target objects (for instance, by passing different object types as method parameters). Object polymorphism is perhaps the most important concept in OOP, as it gives the paradigm much of its flexibility and power.

2.1.1 SOLID and GRASP Principles

Over the years, principles have been established to help develop software efficiently using OOP. These guidelines have been typically taught in schools and referred to in books about

OOP under the form of the “GRASP” and “SOLID” principles.

GRASP principles, sometimes referred to as “GRASP design patterns”, are a set of 9 specific guidelines on how to design object-oriented programs. These principles are interesting to us because they describe what programmers aim for when developing programs and can give an idea of which metrics to use to measure object-oriented code. The following is a short summary of the 9 GRASP principles:

Information Expert. Suggests that related methods and data should be kept together in the same object types.

Creator. A set of rules suggesting where objects should be created (e.g. by its parent object in a hierarchy, or by the object already containing the data required to initialize it).

Low coupling. An object X is said to be “coupled” to an object Y when X contains an attribute of type Y, when X has a method using Y or when X is a subtype of Y. This guideline suggests minimizing these kind of relationship as much as possible.

Controller. Suggests separating the internal workings of a software from its external interface (e.g. a web site or a graphical interface). In this context, the controller object will mediate communication between the interface and the business code.

High cohesion. A highly cohesive object possesses responsibilities that are closely related to each other. There is no one measure of cohesion, but it is often calculated by counting the number of shared attributes used by the methods of an object.

Polymorphism. Suggests using the most generic version of an object possible, as to allow the caller to replace it by other similar objects. This usually takes the form, in statically typed languages, of declaring objects by their most generic type (e.g. declaring an ArrayList as a generic List instead, if the specific functionalities of ArrayList are not important).

Pure fabrication. Sometimes, business code can be improved by adding objects that are not tied to any physical concept. The controller object mentionned earlier is an example of this. Many specific design patterns, which we'll see later in this document, could be said to fall into this category.

Indirection. Objects can sometimes be decoupled by introducing an intermediary object to mediate between them. Again, this is used in the controller pattern to separate (i.e. decouple) the business logic from the user interface.

Protected variations. To make software evolution easier, it is possible to protect a subsystem by wrapping some of its functionality so that various implementations can be created, without having to depend on a single specific one.

Some of these principles are simple guidelines on how to design programs (such as low coupling, high cohesion) while others are more akin to design patterns (such as indirection and protected variations).

Another set of principles often cited in software engineering are the SOLID principles. These are 5 principles introduced in a 2000 paper about object-oriented design [64]. Although sharing no specific link with GRASP principles, they address very similar concepts. Following is a description of each of the SOLID principles:

Single responsibility. This principle states that a given object should only ever have one responsibility. If multiple responsibilities are required, multiple objects are needed. This principle can be related to the high cohesion GRASP principle.

Open-closed. This principles states that objects should keep their internal data closed to modification. This is usually achieved in languages by access protection modifiers, such as "private". To allow for flexibility, instead of allowing itself to be modified, an object should offer a way to be extended, typically by either composition or inheritance.

Liskov substitution. This principle is based upon a rule found in a paper published by Liskov and Wing [58]. Here is the rule: "Let $\phi(z)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of

T". In other words, every object in a program should be able to be replaced by an object of a subtype without the program breaking.

Interface segregation. This principle states that a large class should not implement multiple interfaces, each with its own sub-responsibility. User of the class can instead reference the required interface. This principle helps achieve the low coupling GRASP principle.

Dependency injection. This principle states that instead of creating its own dependency by instantiating objects of a specific type, an object can instead defer this instantiation to its creator (usually by way of constructor parameters). This allows more flexibility in minimizing coupling and controlling which object calls which interface.

The SOLID principles take for granted certain language features when compared to other principles and design patterns. For example, the "open-closed" principle requires a language to have some form of access protection for attributes and methods (although, it could arguably still be used in other languages by relying on the "good will" of the programmers). The interface segregation principle is only possible in languages that either support multiple inheritance or subtyping via interfaces (like Java and C#). Finally, these principles strongly imply that the language is statically typed.

From the GRASP and SOLID principles, we can already see some hints as to what "good" object-oriented code looks like: it has low coupling, high cohesion, objects have few responsibilities and code is kept as generic and reusable as possible. These will be good starting criteria in determining how to measure improvements later on.

2.2 On Design Patterns

In 1994, Design Patterns: Elements of Reusable Object-Oriented Software was published [33]. It has been cited in numerous papers and has been the basis for many courses in software engineering curricula. It contains an analysis and description of 23 "design patterns" commonly found in object-oriented programs. A design pattern is defined as a solution to a recurring problem. Based on their experience and analysis of software source code, the authors came up with these solutions, which they explain in detail and illustrate with examples.

There are two main parts to the book. The first part is an example of an object-oriented WYSIWYG text editor, going somewhat in-depth in the source code and showing some examples of the design patterns used and their advantages. The second part, which makes up most of the book, is a pattern-by-pattern description of the 23 design patterns, with some running examples, and the advantages and disadvantages of each (main focus being on the advantages).

These examples are mostly made using the C++ language. The authors often make references to Smalltalk and sometimes some examples are given in that language too, but they are few and far between. While design patterns typically strive to be language agnostic, some of them are particular to specific programming languages or features. The Smalltalk examples are interesting because they often are much simpler and require much less "boilerplate code" than the C++ examples, which hints that language features are an important factor in simplifying and improving object-oriented implementations.

The design patterns are divided into three sections: creational patterns, structural patterns, and behavioral patterns. Creational patterns govern object creation. They provide solutions to problems that can arise when generating new objects dynamically, while minimizing coupling between classes. For example, the user of a creational pattern will be able to request new objects, without having to know what exact class will be instanced or how exactly the object will be built. Structural patterns are about how objects are composed with one another. They allow low-coupling composition between different classes and can be used as a form of data structure to model trees, lists, and other useful structures. Finally, behavioral patterns allow the user to dynamically change the behavior of an object without having to resort to changing its class at run-time (which is impossible in many programming languages).

This book has been a major influence in shaping modern object-oriented programming, be it in the industry, where design patterns are often seen as an ideal way to code, or in academia, where they are often taught as a more advanced topic for software engineering. We will see in the systematic literature review in the next section that the design patterns from this book, often called the “Gang of Four” (GoF) design patterns, are the basis against which many improvements to object-oriented programming are compared.

Design patterns can be seen as being the final expression of object-oriented programming: uses of major concepts in the paradigm to solve, in generic ways, recurring problems. The way we choose to look at them in this thesis is examine the problems that they try to solve: these are problems that OOP cannot solve simply; they require these design patterns to solve. Design patterns can vary in complexity, depending on many factors, such as the program itself, or the language features available. When trying to find ways to improve object-oriented programming, problems solved by design patterns are an interesting starting point.

We now go through each of the 23 design patterns presented in the book, giving a short summary of the problem, how they solve it, and some ideas about language features that could influence their implementation.

2.2.1 Creational Patterns

Abstract Factory. An Abstract Factory allows the client to request new objects without having to know their implementation class. The client then receives an object conforming to a specific type. This pattern allows decoupling of responsibilities between the client and the factory. The client no longer must be coupled to a specific class and defers instantiation to the factory. In recent languages, this pattern has been implemented by “dependency injection” frameworks, where it allows specifying in advance which concrete implementations of interface a program will use, while the framework acts as an Abstract Factory, generating the concrete instances. As shown in the GoF book, Abstract Factory can be greatly simplified when using a language that supports classes as first-class objects (such as CLOS or Smalltalk).

Builder. A Builder allows the user to delegate object initialization to another object, which allows initialization to be done in steps and makes it simpler to understand and require less boilerplate code. In the Kotlin programming language, the Builder pattern can be made very simple by taking advantage of the special lambda syntax[1]. A functional version of the Builder pattern also exists, sometimes named “fluent API” and often seen in JavaScript programs.

Factory Method. The Factory Method provides an abstract type with a main method to create objects that will use (or defer to) one or more sub-methods. These sub-methods are able to be overridden by any type inheriting this creator type, thus allowing customization of the construction process. This pattern is mostly an application of the behavioral Template Method pattern (see below) in a creational context. It is a simple pattern which makes use of polymorphism. In functional programming, the underlying goal behind this pattern is usually achieved using higher order functions to create constructor functions that receive the appropriate “sub-functions” as closures.

Prototype. Prototype stores a partially initialized object, which is copied/completed to create new objects. As the prototype uses a run-time object instead of a class to generate its objects, it can be modified dynamically to change what kind of object it generates, allowing dynamic object construction in languages that do not support classes as first-class elements. The pattern can be difficult to implement in languages that do not support object cloning (requiring it to be implemented manually for each prototype). JavaScript, being a prototype-based language, uses this pattern to instantiate all objects.

Singleton. The Singleton pattern is rather unique in that it does not create new objects but ensures that only one instance of a certain class will exist in a given system for a given run. It is somewhat controversial as it can increase coupling between objects in a program. Every object that references the Singleton object effectively becomes coupled to it. Combining it with other patterns, such as Abstract Factory, can mitigate the issue. As with the Abstract Factory, the Singleton pattern is often

[1]This is the approach used by the UI framework TornadoFX: `https://tornadofx.io/`

implemented using dependency injection, where the framework can guarantee that a single instance will exist, while the Abstract Factory feature of the framework can be used to retrieve that instance.

2.2.2 Structural Patterns

Adapter. Adapter is used to make two otherwise incompatible types able to interact with each other. One type will be “wrapped” by the adapter to conform to a specific interface required by the other type. The pattern achieves this by having the adapter subtype the target interface and delegate to the appropriate methods of the wrapped object. However, its overuse may clutter the program with many “small” adapter classes with no business logic of their own. When using a language that supports closures, only one adapter class is needed by target interface (the adapter class can be built using closures to specify the behaviour of the adapter methods). In a functional language, this problem may be solved by using records of functions.

Bridge. Bridge is a counterpart to Adapter. It is used when multiple subtypes of tightly coupled objects must communicate with one another. This pattern allows separating the communication between objects and changing them dynamically at run-time. It can also be used to split a large class into multiple parts to enhance software cohesion. As with Adapter, a Bridge can be implemented using functional records and closures.

Composite. The Composite pattern is used to create a tree hierarchy. The pattern achieves this by having each element be one of two types: a ”leaf” type, representing a single element; or a ”node” type, a container for other elements. These two types will be subtypes of the parent ”element” type, which describes every element of the hierarchy. This is a useful pattern when designing hierarchic taxonomies, manipulated by a program, such as UI frameworks, in which every element can be either a container for more elements or a single element. One disadvantage of the pattern is the difficulty to control the hierarchy. For example, disallowing a certain type of element to be set as children of a specific node may be difficult to enforce. An alternative to this pattern (more often used in functional programming) is creating a strongly typed hierarchy of records. This allows more control over the hierarchy while sacrificing abstraction (to use the hierarchy, you must know each element that goes into it). This alternative approach is more useful in languages which support pattern matching because it makes traversing the hierarchy simple.

Decorator. The Decorator is, in essence, a wrapper object that can add behaviour before delegating to its wrapped object. This is done by having the wrapper be a subtype of the wrapped object, thus sharing the same interface. This pattern is a simple way to dynamically add behaviour to objects while sacrificing object identity (the decorator object and the wrapped object are not the same). The GoF book mentions it is possible to remove decorators, but if you have multiple Decorators on a single object, it becomes difficult to find or remove a specific one without combining the Decorator with another pattern, such as Chain of Responsibility. In a language like CLOS, the Decorator pattern may be replaced by a combination of method qualifiers ("after", "before", "around") and dynamic method creation.

Facade. Facade creates a single interface for operating a system or sub-system, simplifying its usage. In other words, Facade creates an API for a system. This pattern has perhaps the fewest ties with OOP. Facade is effectively the equivalent of functional programming "modules". The notion that a Facade is often also a Singleton reinforces this observation. In a functional language, you would simply create a module that would, in turn, use other modules in its functions.

Flyweight. The Flyweight is perhaps a less commonly used and more complex design pattern. It allows the creation of small objects with shared state. The main goal is reducing memory consumption in programs in which a great number of objects exist at once. The implementation of `String` in languages like Java and C# may be considered a Flyweight. Flyweights can be used whenever data is immutable to avoid having multiple copies of the same data. The Haskell programming language, being an immutable language, makes use of memoization, a similar concept where function results are calculated only once, and every subsequent use of that function with the same arguments will return that calculated result.

Proxy. Proxy is used to separate an object from its implementation. This can be useful when an object's implementation must be executed on a server, but the user of the object is on the client. It can also be used when object creation would require too much memory. In that case, the Proxy partially loads only what is necessary whenever operations are called on it. In languages with flexibility on how methods are called, such as Smalltalk with its `doesNotUnderstand` method, creating proxies is as trivial as overriding that method.

2.2.3 Behavioural Patterns

Chain of Responsibility. A Chain of Responsibility is used to create a system where requests can be sent without having to know exactly which object will handle them. Each handler object has a way of signalling whether it can handle a specific request and the first object to be able to handle it stops the chain. This is reminiscent of CLOS method combination system and Chain of Responsibility can be implemented by combining that system and dynamic methods.

Command. A Command essentially represents a functional programming closure to which functionality can be added (e.g. "undo", "redo", monitoring, etc.). This pattern is most useful in OOP languages without functional programming support, such as older versions of Java, which do not support closures. The GoF book mentions that this pattern is not about making closures or functors (objects acting as functions), but about the binding between the receiver and the function. However, most of the complexity of this pattern stems from wrapping executable code inside an object (i.e. a closure). Indeed, in functional programming languages, this pattern becomes a lot less useful or complex. Furthermore, if state is immutable, undo/redo mechanisms can be done by simply preserving states at a specific moment in time and replacing them when the operation is needed, further trivializing this pattern.

Interpreter. Interpreter is perhaps the most situational pattern. This pattern is an approach to designing a programming language interpreter in an OOP fashion. Interpreter is somewhat similar to the Composite pattern in its implementation. However, there are arguably easier and shorter ways to implement interpreters than with objects, notably using functional programming [72]. As with the Composite, records can be used to create a strongly typed hierarchy of the language's expressions. Building the interpreter becomes a matter of traversing this hierarchy using match statements. An interesting functional programming design pattern called "parser combinators" can also be used to implement BNF grammars, which is the approach used in the Haskell Parsec[2] library and its F# equivalent, FParsec [3].

Iterator. Iterator is used to traverse sequences of objects. The iterator object holds a reference to a position in the sequence and can give its user the next object in the sequence until every object has been traversed. In more recent languages and programs, the Iterator pattern is replaced by `foreach` statements, which provide similar functionality. In Smalltalk, this pattern is replaced by the `do` method, which allows iterating over any collection. This is similar to the functional programming `map` in its usage, applying a function to each element of a sequence. `Map` is a different design pattern which uses recursivity. While there are differences between the Iterator and the functional `map` (e.g., having access to the current iteration state in Iterator), some `map` implementations bridge this gap considerably (notably Common Lisp, which implementation allows iterating multiple lists in parallel). We also compare `map` to Visitor below.

Mediator. Mediator is a generalization of the controller element in the "Model-View-Controller" design pattern (non-GoF). It separates controlling behaviours among multiple elements. This pattern does not use subtyping, instead using composition to keep references to all the components it manages. Whenever an action is required

[2]https://hackage.haskell.org/package/parsec/
[3]https://www.quanttec.com/fparsec/

by a component, that component signals it to the Mediator, which will execute operations on other components. There exist architectural patterns other than MVC that do not use a controller (or Mediator), notably the MVVM pattern (Model-View-ViewModel). Modern functional programming UI frameworks often use a completely different approach, called reactive functional programming.

Memento. Memento allows saving or transferring an object's state without breaking its encapsulation. An originator object will store part or all of an object's state in the Memento, which in turn can be used by other parts of the system to save that state. There is however a trade-off, because a crosscutting concern is added to the originator object: it must know that it is going to be saved and must provide a method to create the Memento object. In some programming languages, such as CLOS, it is possible to use reflection to traverse objects and save them as-is. This breaks encapsulation but does not add cross-cutting concerns to objects.

Observer. Observer allows dynamically setting an object to notify another whenever a specific event happens. It is very useful in UI frameworks in which elements must notify the controller (or equivalent) whenever the user interacts with them. There are many modern implementations of this pattern. Most of them make use of closures to avoid creating many different Observer types. This pattern shares some of the same issues as Memento: it imposes a cross-cutting concern on the observed object. The observed object must know that it can be observed and must manage this behaviour. In CLOS, it is possible, by making use of dynamically created methods and method qualifiers, to remove this cross-cutting concern.

State. State allows changing the behaviour of an object at run-time. The behaviour is described by a State type that can then have any number of subtypes (states). At run-time, an object can change behaviour by modifying its State instance to the different State subtypes. The naive way to achieve this without this design pattern would be with `switch` statements. Both approaches offer different benefits and drawbacks. The State pattern makes it easy to add or remove states (you only need to create a new state class), while `switch` statements make this harder (you must change every switch in your code to add the new branch). However, the `switch` statements approach makes it easier to add new state-based operations (you only need to write a single `switch` statement), while the State pattern requires adding a new method to every existing state. The State design pattern centralizes state, while `switch` statements centralize behaviour[4].

Strategy. Strategy offers a different approach to change the behaviour of objects at run-time. The idea is to build an object while providing it with a concrete Strategy object, which will define its behaviour. This pattern is useful when you cannot know in advance what exact implementation is needed at run-time. In functional languages, this pattern is implemented easily using higher-order functions or records of functions, avoiding the need to create many concrete strategy classes.

Template Method. Template Method is an application of object polymorphism to create "hooks" within a method for overriding. The Template Method calls other methods within the same object. These other methods will then be overridden in subtypes. This pattern allows subtypes to redefine some of the behaviour of the Template Method, without exposing internal details, which could break the method if they were changed (for instance, the call order). A Template Method can also be created without resorting to subtyping at all if the language supports functional programming. The method then simply becomes a higher-order function.

Visitor. Visitor allows traversing a hierarchy of objects while executing actions on each element. This is somewhat reminiscent of the Iterator pattern (and the `map` function in functional programming), with the difference that a Visitor can traverse hierarchies of heterogeneous classes (while Iterator traverses sequences of homogeneous classes). Visitor requires creating a new class for each operation that could be executed on

[4]Also see the Expression Problem: `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`

the hierarchy, although this requirement can be mitigated using closures and higher-order functions, if available. It is possible to achieve the same goal as the Visitor in a functional language using a hierarchy of records and higher-order functions to traverse and apply changes to it. This is what is typically done in compilers written in functional languages.

2.3 On Design Pattern Improvements

There are a few recurring themes of note in these patterns' descriptions and analyses. The creational patterns are often implemented in dependency injection (DI) frameworks nowadays. The functionality that they bring is useful enough to consider making it part of a language or framework, instead of relying on the designers to use them or not. Having a framework (or a language feature) handle the boilerplate of writing the Abstract Factory, Singleton, or Prototype patterns would help programmers focus on the business logic. In DI frameworks, this usually takes the form of a configuration file run at the start of a program. This file names various classes that will become factories, singletons, and such. In programs that use these frameworks, they will often replace usage of the language explicit constructor (e.g., the `new` keyword) entirely, instead calling upon the framework to generate objects. This hints that explicit language constructors might not be necessary for a programming language and should instead be flexible (as in Smalltalk). However, usage of a DI framework comes with costs: learning curve and extra processing. Programmers may have a harder time understanding and maintaining the program efficiently, which could be mitigated by having the language itself implement these patterns, instead of a third-party tool.

Many of the structural patterns have alternative implementations when using a language supporting functional programming features. In a language that supports higher-order functions, Adapter and Bridge can be implemented without using inheritance. While the use of switch statements is often frowned upon in object-oriented programming, the match statement, its functional programming (and slightly more sophisticated) equivalent, can be used as an alternative to a combination of the Composite and Visitor patterns, with varying drawbacks and benefits. The other patterns in this category all have some language feature that can be used to alter (and possibly improve) them, such as CLOS' method qualifiers (a concept similar to aspect-oriented programming) for Decorator, Smalltalk's `doesNotUnderstand` method for Proxy or simply functional modules for Facade.

Behavioural patterns are varied and draw many similarities to features present in some languages. For instance, Chain of Responsibility is reminiscent of CLOS' method combination system and Mediator has the same basis as the MVC architecture adopted by modern web languages. Some of these patterns are integrated into languages in one form or another, such as `foreach` statements in replacement of the Iterator pattern, or events (e.g. C#) replacing Observer by using an approach similar to closures. In some cases, immutability can trivialize a pattern, such as with Command and Memento, by removing all notions of mutation. In fact, as with structural patterns, many behavioural patterns can be shortened using functional features such as closures (e.g. Observer, Strategy, Visitor). Finally, Interpreter can be implemented using a whole different approach when dealing with immutable functional languages by instead making use of the "parser combinators" functional pattern.

2.4 On Mutability and Immutability

Many (perhaps most?) popular programming languages nowadays can trace their roots to C, and even earlier in time, Algol. Before the OOP paradigm became mainstream, much programming languages were procedural and imperative. In procedural languages, a procedure is a function with no return value that is executed for its side effects. A procedure changes, or *mutates*, the state of the program. The OOP paradigm, being an offspring of procedural languages, is no different.

The OOP paradigm is about the encapsulation of data and behaviour. Objects are expected to have methods that mutate the state of the object, by changing the values of its internal attributes. Most OOP programs work by mutating objects and changing their behaviour.

However, programs do not have to work by mutation. Functional programming languages, such as Haskell, are typically not based on mutation. These languages do not trace their roots to procedural imperative languages, but to lambda calculus instead. The lambda calculus, like the Turing machine, is a mathematical model used to represent computation. Unlike the Turing machine, the lambda calculus has no concept of state. It uses function application and variable binding (often implemented by closures and lambda expressions in programming languages) to achieve the desired results. Because of the absence of any kind of mutation mechanism, functional programs are immutable. **Immutability is the absence of mutation in a program, function, or object [1, 13].**

2.4.1 Impact of Immutability

Immutability is used and enforced only in some programming languages. Yet, developers' lore [1, 13] state that immutability has advantages in software development. The natural question to ask is: should we enforce immutability in general?

Abelson et al. mention in their book, *Structure and Interpretation of Computer Programs* [1], that avoiding mutable data structures can help to reason about the behaviour of programs. Mutations often lead to changes in behaviour in a program (which is often the goal of the mutation). By limiting state changes, and especially global state changes (e.g., global variables), the behaviour of the programs is more consistent and easier to understand.

Helland [47] discusses advantages in distributed database systems. An immutable database, also called an "append-only" database, automatically performs changes logging, by virtue of the fact that no piece of data can be modified in the database. Thus, records are kept indefinitely (or until someone decides to delete them).

Bloch states in *Effective Java* [13] that mutations should be avoided whenever possible. Like Abelson et al., they mention the advantage of having simpler state management, but also safety when working with concurrent programming. Mutations are a source of errors in multi-threaded programs because the state of a shared object could change at unexpected moments and, in turn, could cause an expected change in the behaviour of the program.

Most advantages stated are, however, authoritative arguments. Empirical studies on the effects of mutability or immutability are few and far between.

Coblenz et al. [23] conducted interviews with professional developers and found that participants stated that incorrect state mutations are a major source of bugs in programs. For example, an object thought to be immutable would be changed by some other object, perhaps because of optimization by reusing references instead of using immutable copies.

Dolado et al. [26] investigated the impact of side effects on program comprehension. While side effects (or the lack thereof) are indeed related to immutability, the results of their study were oriented toward syntax recommendations, such as specific side-effect operators like increment operations.

Stylos and Clarke [85] studied the impact of constructor parameters on program comprehension. They found that programmers preferred and were more effective at using parameter-less constructors, mutating the objects after instantiation to set their desired states. This finding indicates that there may be a usability trade-off in enforcing immutability.

Gordon and al. [40] presented an extension to the C# type system to better control immutability. The system was used by a Microsoft team and their impressions were collected. Anecdotal evidence showed that many bugs related to mutation were found and corrected using reference immutability.

2.4.2 Immutability in Programming Languages

While functional languages, such as Haskell or Scheme, were designed with immutability at their inception, for others immutability support was added later in their life cycle. We divide immutability support into two categories: language features *enforcing* immutability and language features *improving the use* of immutability.

An example of the former would be "read-only" keywords (such as `final` or `const`), which prevents a value from changing after initialization. An example of the latter would be "record updating" (also called non-destructing mutation), which allows changing some values of some attributes while cloning an object, effectively "mutating" the object while preserving immutability.

Immutability Enforcement

Functional languages typically support both enforcing and using immutability, but that is not necessarily the case for OOP languages. While “read-only” keywords have been available in older imperative languages such as Pascal [89], they do not constitute “full” immutability enforcement.

Boyland et al. [15] discuss the semantics of different access annotations on references and pointers, in particular different types of access rights capabilities of references, such as limiting reading, writing, and ownership. They show how these semantics can be applied to extend type systems using annotations.

Coblenz et al. [23] describe properties of immutability enforcement systems in programming languages. They make a difference between systems with *read-only* references, which disallows mutation through that reference, and assignability restrictions (e.g., Java `final`), which disallows assigning to a given reference. Neither is true immutability because in the first case, there is the possibility of mutating the referenced object through another reference and in the second case the referenced object itself can be mutated. Both these features enforce *non-transitive* immutability.

For *transitive* immutability, a language must disallow mutations of a given object from all references. For example, given a list containing references to some objects, for the list to be *transitively* immutable, the list should disallow mutation on itself and the objects contained within should also disallow mutation. Very few OOP languages enforce transitive immutability, and those that do usually are hybrid functional/OOP languages (e.g., OCaml, F#, Scala).

Research exists to enforce transitive immutability in existing OOP languages, such as Java and C#. Zibin et al. [95] present Immutability Generic Java (IGJ), a language extension to Java that enforces reference immutability and object immutability. They define object immutability as disallowing mutation on an object, even if other instances of the same class could be mutable. Reference immutability is the equivalent of the “read-only” restriction discussed above. Similarly, Tschantz and Ernst [91] present Javari, a type extension to Java to enforce reference immutability constraints.

Kniesel and Theisen [52] present Java with Access Control (JAC), which extends Java with *read-only types*, which are described as having "transitive propagation read-only access protection". The types enforce immutability at the class, object or parameter level.

Coblenz et al. [22] also present Glacier, a type annotation system for Java to specify different types of immutability for classes. In particular, this system can enforce transitive immutability, among other properties discussed in their previous work [23]. They evaluate the performance of their system in an experiment with 20 professional Java programmers. They report that participants who used their system were more successful at implementing a correct immutable system than those who did not.

Unkel and Lam [92] introduce the concept of stationary fields, which are similar to Java `final` fields, but allow unlimited mutation if they all happen before the first read. This concept effectively enforces immutability and solves some issues discussed by Stylos and Clarke [85] with argument-less constructors.

Microsoft added immutable collections to their .NET Framework around 2017[5], inspired by their functional F# language and, by extension, OCaml. These immutable collections guarantee that their structure is preserved at all times and disallow element-level assignation. They also implement an API inspired by the Builder design pattern [33] to allow copy and mutation of the structures.

The Rust programming language[6] uses ownership mechanisms to manage state and mutation. Rust uses the "read-only" type of restriction but ensures that only one "mutable" reference exists for a given structure at any time. Every other reference must be read-only. This mechanism brings some of the same advantages as immutability, such as safer state management and safer concurrent programming.

[5]https://learn.microsoft.com/en-us/archive/msdn-magazine/2017/march/net-framework-immutable-collections

[6]`https://www.rust-lang.org/`

Immutability Usage

While most of the research focuses on enforcing immutability, other language features facilitate immutability or use it to bring certain advantages. An example of an advantage would be how the Haskell language optimizes calculations. By using memoization [68] and immutable functions (i.e., functions that have no side-effect on their arguments or the program), Haskell can store the results of a function so that result is returned instead of being recalculated when the same function is called with the same arguments. Memoization uses the property that an immutable (or pure) function always returns the same result when given the same arguments.

OOP languages have been embracing immutability support lately, with new features being added and new languages using more immutability.

Microsoft C# received functional features, such as pattern matching[7], record types, and record updating[8]. The default type for collections in the .NET Framework is `IEnumerable`, which is immutable and allows usage of *LINQ*, a functional-inspired library for collection manipulation, implementing many functional concepts, such as map, reduce, fold, etc.

Java also received some functional features in Java 14, such as record types[9] and pattern matching[10] but not some of the usability features that C# did, such as record updating and immutable collections. Java 9 introduced the `List.of` method for the creation of immutable lists, but support is limited (e.g., no API to mutate list non-destructively).

Kotlin, a more recent JVM language with full interoperability with Java, while not embracing full immutability, makes a distinction between mutable and immutable references (i.e., read-only restrictions) with its `var` and `val` keywords for declaring variables. Immutable references are usually used by default. It also includes many functional programming features such as pattern matching and record types.

Microsoft F# language is a functional language that also supports OOP. Based on OCaml, it supports immutability but has the `mutable` keyword[11]. While the combination

[7]https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/patterns
[8]https://learn.microsoft.com/en-us/dotnet/csharp/whats-new/tutorials/records
[9]https://docs.oracle.com/en/java/javase/14/language/records.html
[10]https://docs.oracle.com/en/java/javase/14/language/switch-expressions.html
[11]https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/values/

of functional programming and OOP is interesting, the integration between the two can be lacking at times. For example, methods cannot be easily converted into closures, as functions can, which limits the usage of some functional programming concepts with objects.

Chapter 3

Mapping Suggested Language Features for Improving Object-Oriented Design Patterns

3.1 Introduction

In OOP, design patterns are presented as “templates” that can be adapted to specific problems for which using “naive” OOP solutions would not be optimal. However, they can introduce new problems, such as raising complexity, requiring that programmers be familiar with these design patterns to get the most out of their advantages. We know that overuse of design patterns, as well as “over-engineering” [49], should be avoided.

One secondary study by Zhang and Budgen [93] considered empirical studies on the effects of applying design patterns in software design. They showed that there was a lack of empirical studies on most design patterns and that empirically founded advantages and disadvantages are difficult to find in the literature. It also discussed design patterns implementations sometimes being harmful to program understanding, while they do help with maintenance and evolution.

Primary studies tried to find other object-oriented patterns for situations not covered by the initial 23 design patterns. Others tried to use object-oriented approaches to improve existing patterns [25]. Yet others tried to leverage language features from other programming paradigms to improve specific measures of design patterns [2, 3, 27, 44, 84]. Yet, there has not been much focus on the relationship between programming language features and design pattern implementations.

To gain a better understanding of this relationship between design pattern implementations and language features, we propose to study the literature on design pattern improvements. We perform a systematic mapping study to identify language features claimed to improve design pattern implementations. The goal is to identify which language features have been suggested to improve OOP design pattern implementations, which design pattern implementations are being improved upon, which measures are used to evaluate these improvements, and what empirical data was collected to assess these improvements.

We performed a search query in three databases (Ei Compendex, Inspec, GEOBASE) using the Elsevier search engine Engineering Village. Our initial query yielded 874 papers, which we assessed for quality and used for snowballing, resulting in a total of 34 primary studies. We then extracted relevant data from these 34 studies and then discuss and catalogue our observations and their meaning.

We catalogue 18 language features claimed in the primary studies to improve design pattern implementations and categorize them into paradigms.

This catalogue is useful to identify trends and create a road map for research on language features to improve object-oriented design patterns. Considering the popularity of design patterns, improving their implementation and adding language features to better solve their underlying concerns is an efficient way to improve Object-Oriented Programming. We intend to use this in the future as a basis to research specific language features that may help in improving Object-Oriented Programming.

The rest of this chapter is organized as follows: in Section 3.2, we further explain how design patterns can be improved by language features and give a running example. In Section 3.3, we present an overview of other related secondary studies on the topic. In Section 3.3, we shortly present the concept of the systematic mapping study, we pose our research questions and present the methodology to perform our mapping study. In Section 3.5, we discuss these results and their implications. In Section 3.6, we list notable or interesting findings and our recommendations for future work. Section 3.7 lists threats to the validity of this study. In Section 3.8, we conclude with the main findings of this chapter and further possible research.

This chapter is the basis of a paper sent to the Information and Software Technology journal. We sent a minor revision on March 6th.

3.2 Background

We illustrate how a design pattern can be improved by specific language features using the Factory Method design pattern, as discussed in the GoF book [33]. Figure 2 shows the first example of an implementation of Factory Method in the book. We present this example because of how directly the Factory Method design pattern maps to object instantiation language features. The example was designed with the C++ language in mind, but would be applicable to most other modern OOP languages, such as C# and Java.

The goal of this pattern is to decouple an application from concrete implementations of abstract classes or interfaces. To achieve this goal, we must encapsulate object instantiation. Factory Method suggests creating an abstraction of this creation process and using this abstraction throughout the software instead of the normal object creation feature of the language (e.g., the `new` keyword). In Figure 2, the class `Application` represents the abstraction of the creation process of documents. If we wanted the application to instantiate documents of type `MyDocument`, we would use an application of type `MyApplication`.

This pattern encapsulates the normal object instantiation mechanism provided by the language. The solution of the design pattern is to encapsulate this mechanism because the instantiation mechanism in C++ (and similar languages) does not allow for decoupling an abstraction from its concrete implementations (i.e., the `new` keyword only allows for the instantiation of concrete objects).

A variant to this solution is given a few pages later [33] with Smalltalk. In Smalltalk, there is no keyword for object creation. Object creation is accomplished using a method declared in the meta-class of the class (also called `new`). Because classes are also objects in Smalltalk, it is possible to call this method on a variable, effectively the Factory Method pattern. Thus, the document example could be implemented in Smalltalk with a method in `Application` that returns the class to use to create document objects. The `MyApplication` class could implement this method as returning `MyDocument`, as such:

```
"In class MyApplication"
documentClass
    ^ MyDocument
createDocument
    ^ documentClass new
```

In this implementation, there is no need to supersede the normal object creation mechanism of the language. We instead use it normally on the class returned by `documentClass`[1] By using Smalltalk's meta-class and object instantiation language features, the pattern becomes part of the language and is more easily combined with other language features.

There are other examples in the GoF book [33] where design patterns have implementations that are simpler in Smalltalk than C++. This discrepancy between implementations suggests that design pattern implementations depend on what features are supported by the programming language.

In some languages, design patterns can even become obsolete and disappear entirely when the concept is fully supported by the language [67].

In this chapter, we are interested in cataloguing language features that impact the 23 GoF design patterns, and see what this impact is and how it was measured.

3.3 Related Studies

Multiple secondary studies exist about object-oriented design patterns.

[1]Using this, we could restructure the model in different ways. For example, we could remove the `MyApplication` class and replace it with a collection holding the different concrete classes used to create objects for the application.

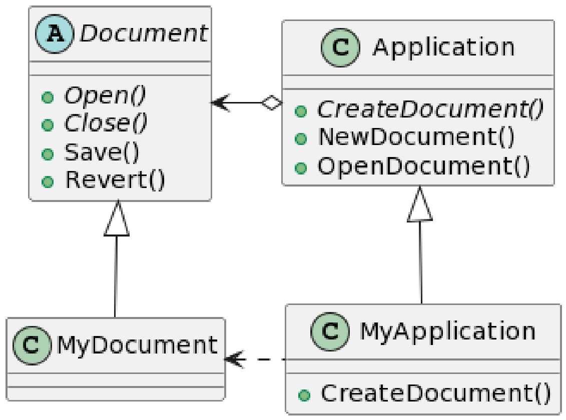

Figure 2: Implementation example of Factory Method

In 2012, Zhang et al. [93] published a systematic literature review about the effectiveness of software design patterns. They targeted empirical studies made on the GoF design patterns. They concluded that there is a lack of empirical studies about design patterns and concrete advantages and disadvantages are difficult to find in the literature. They also concluded that design patterns may not help with understandability, sometimes decreasing it dramatically.

In 2013, Ampatzoglou et al. [6] performed a mapping study on the impact of design patterns on software quality. They focused on the GoF design patterns and categorized research into different categories (formalization, detection, and application). They found that research on the detection of design patterns and their impact was the most active. In another paper, the same authors proposed a catalogue [5] of alternative designs for the GoF patterns. These alternative designs usually add new functionality to existing patterns, either to make them more flexible [31] or to adapt them to specific situations [31].

In 2016, Mayvan et al. [9] published a state of the art on research on design patterns with the goal of presenting an entry-level summary to those who would seek to enter the research field. They perform a systematic mapping study to identify popular research trends, such as commonly used keywords, most active researchers and venues, and the distribution of publications by topics. They identify Pattern Development as the most popular topic, which groups the introduction of new patterns or pattern variants and categorization of design patterns by field (e.g., mobile applications, security, etc.).

These studies, and the studies they review, focus mainly on evaluating the impact of design patterns on object-oriented software. Many of them compare code that was developed without the usage of design patterns against code that was developed with. Others explore alternative design patterns to respond to specific problems. Our study catalogues language features used in the literature claiming to improve object-oriented design pattern implementations, without modifying their functionality, as well as how these improvements are measured and the empirical data available.

sectionMethodology

To reach our goal and answer our research questions, we perform a systematic mapping study following similar practices as introduced by Peterson et al. [71] in their systematic mapping studies guidelines for software engineering and executed by Ampatzoglou et al. [6]. Our methodology also draws from general guidelines on systematic literature reviews [50, 51].

Our objective is to find language features in research done to improve OOP design patterns. In particular, we want to survey the techniques and tools developed to make these improvements, as well as the measures used to assess them. We are also interested in understanding the kind of experiments done to evaluate these tools and techniques. We want to answer the following research questions:

(1) RQ3.1: What language features have been suggested to improve design patterns implementations?

(2) RQ3.2: Which design patterns have the most associated language features suggestions?

(3) RQ3.3: What measures have been used to evaluate the impact of these language features on design patterns implementations?

(4) RQ3.4: What experiments have been done on these language features?

To answer these questions, we perform a search to find and select the papers that will

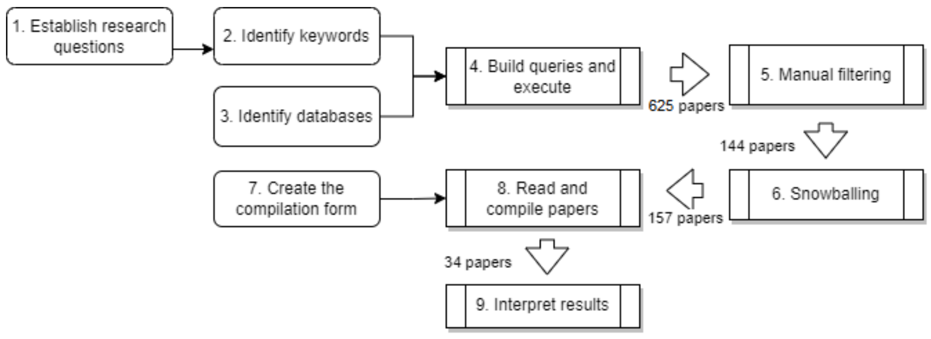

Figure 3: Flowchart of the steps in our approach

be studied in this study. Then, we ensure that the identified papers are related and offer answers to the research questions by performing a quality assessment. Finally, we extract data from the papers and synthesize it to reach our goal.

To reduce bias as much as possible, we perform the literature search in multiple databases and gather a large number of papers, which we then systematically study. We also perform a snowballing step to include all papers citing or cited by any paper which is not rejected in the manual sorting, followed by other snowballing iterations on the resulting set until reaching a fixed point.

Consequently, we follow these nine steps (as shown in Figure 3):

(1) Identify keywords for searching for papers.

(2) Identify databases for searching for papers.

(3) Build a search query and execute it based on the keywords on each database.

(4) Filter the results manually based on the abstract and keywords.

(5) Perform a forward and backward snowballing step where each paper's references and papers that reference it are also considered (manually).

(6) Create a form for compiling the data provided by the papers.

(7) Read and compile each paper, removing any paper that does not provide data for at least one of our research questions.

(8) Interpret the compiled data and answer the research questions.

3.3.1 Keywords for the Search Query

Based on our goal, we identify keywords to find papers potentially providing data to answer our research questions. These keywords do not encompass all possible synonyms and are general. For example, we include "object-oriented programming" as a keyword, but do not add "OOP", "object-oriented", or other synonyms. We add synonyms in Step 4 when we build the database query. We are investigating improvements in design patterns, thus we added some keywords related to weaknesses and anti-patterns. While these could be seen as unrelated, there is no disadvantage in having a broader query and unrelated papers will be removed in later steps of the study.

We identify the following keywords to use for building the search query: *object-oriented programming, design patterns, anti-patterns, weakness, disadvantage, improvement, enhancement, refinement, better, expand.*

3.3.2 Databases for the Search Query

To perform as large a search as possible, we use multiple online databases for scientific papers. These databases are part of an online tool called Engineering Village, hosted by Elsevier. They contain data from a large number of scientific journal databases, such as ACM and IEEE. Given the focus of this study on the GoF design patterns, we consider papers published between 1995 and 2022.

3.3.3 Query Building and Execution

We are using only one search engine, thus only one query is needed. All three of the databases are accessed from Elsevier Engineering Village Web site[2]. We restrict the scope of the query to the title, abstract, and keywords of the papers. We opt to use the more general term "object" (and then filter manually) instead of the various ways of writing

[2]https://www.engineeringvillage.com/

"object-oriented programming" to avoid accidentally excluding relevant studies that used a slightly different expression (e.g., object oriented software).

Through several iterations, we add a large number of keyword exclusions ("NOT" keywords) to increase the relevance of the studies and reduce the number of unrelated studies. We exclude these keywords specifically from the "keywords" field in the database (meaning the paper should be about these topics), and not from the abstract (e.g., we are not excluding a paper because "test" is found in its abstract). The final query is shown in Table 1. This query yields **874 studies**.

The final query results are shown in Table 2. Removing duplicates is done via a script, as we found the Engineering Village search engine functionality to remove duplicates was not accurate. After removing duplicates, we retain 625 studies for manual filtering.

3.3.4 Filtering

We manually read the title and abstract of each of these studies and determine whether the study has the potential to provide data to answer one of our four research questions. We choose to be conservative in this step and include any study that could provide data for a research question, even if remotely. Following quality assessment guidelines [71], inclusion criteria were applied to the titles and abstracts:

(1) Study is in the field of software engineering.

(2) Study is related to the improvement of design patterns and must propose a language feature to do so. (Studies relating exclusively to measuring the impact of existing design patterns against non-pattern code are not kept.)

(3) Study was published between 1995 and 2022.

Similarly, criteria were used to exclude papers:

(1) Study is not related to one or more specific design patterns.

(2) Study is not peer-reviewed.

Query	Comments
((oop OR object) AND (design pattern OR design patterns OR anti-pattern OR anti-patterns OR paradigm) AND (weakness* OR disadvantage* OR improve* OR enhance* OR refine* OR help OR better OR expand*) WN AB)	Must be linked to design patterns Initially, we intended the scope to be broader, but this shouldn't impact the final results Must be about improving or finding weaknesses These keywords are located in abstract text
NOT ((teach* OR learn* OR test* OR modeling OR automated OR automation OR tool* OR mobile OR optimiz* OR simulation OR mining OR medical OR bio* OR hardware OR hdl OR parallel* OR api OR find* OR sql) WN KY)	These keywords are excluded to avoid papers about learning, automating, optimizing tools, or other non-related subjects encountered during the first passes of the query. Excluded keywords must not appear in the keyword field.
NOT ((pattern recognition OR pattern identification OR pattern detection OR object recognition OR object detection OR feature extraction OR computer vision OR pattern clustering OR learning (artificial intelligence) OR image segmentation OR pattern classification OR data mining OR computer aided design OR formal specification OR image classification OR user interfaces OR computer simulation OR internet OR image processing OR codes (symbols) OR image enhancement OR embedded systems OR image reconstruction OR cameras OR distributed computer systems OR product design OR artificial intelligence OR iterative methods OR neural nets OR neural networks OR object tracking OR genetic algorithms OR classification (of information) OR learning systems OR mathematical models OR middleware OR internet of things OR cloud computing OR decision making OR electroencephalography OR virtual reality OR risk management OR health care OR distributed object management OR query processing OR knowledge based systems) WN KY)	Finally, a large list of expressions was excluded from the keywords because they tended to be attached to papers about concepts unrelated to this research.

Table 1: Database Query

Database	Results
Compendex	316
Inspec	371
Geobase	13
All databases	700
After de-duplication	625

Table 2: Query Results by Database

(3) Study is not written in English.

(4) Study has no accessible full-text online.

(5) Documents that are books or gray literature.

(6) Study is a duplicate. (We kept the longer version.)

After this filtering step, **144 papers** remain.

3.3.5 Snowballing

We consider both the references of each study and any study referencing the said study for each of the 144 studies yielded by the previous step.

To search for forward references, we use the Scopus search engine. We apply the same year restriction (1995–2022) and the same criteria for filtering. After the third iteration of snowballing, we notice that all of the results are duplicates of already included studies, so we stop the snowballing process. Both the first and second authors of this study independently perform this step. The second author was not privy to the research questions being assessed until the process was done. We consolidate the results afterwards and address any discrepancies. After eliminating all duplicates, we obtain a total of **157 studies** that could provide data to answer our research questions.

3.3.6 Compilation Form

To compile the information from the obtained studies systematically and consistently, we build a form that we fill out for each study. Our form has the following information:

(1) The title of the study.

(2) The type of the study (experiment, case study, conceptual analysis, literature review, or survey).

(3) The study publication venue.

(4) The study publisher (or digital source).

(5) A list of all design patterns mentioned by the study.

(6) A list of all language features proposed by the study to improve the specified design patterns.

(7) A list of all measures used by the study to evaluate the suggested improvements.

(8) In the case of experiments, information about the participants (number and expertise).

(9) In the case of case studies, information about the studied cases (number and whether or not they are in-vivo or in-vitro).

After using the form to extract data from the 157 compiled studies, we remove 123 as they did not provide an answer to any of our research questions. Many of the studies were evaluating the impact of existing design patterns, rather than any improvements, and this does not fit the goal of this study. (Thus, we remove 123 of the obtained studies, confirming that we were conservative in our process and likely did not miss any relevant paper.) We finally obtain **34 studies**.

3.4 Results

We classify the 34 remaining studies in various ways to answer our research questions. We first perform a general analysis of the sources (publication venues and publishers) the papers came from. Finally, we categorize each study into a type and take a more detailed look at the experiments and case studies.

Table 3 shows the number of papers for each publication venue. As we can see, there are a large variety of venues in our dataset. Some venues show up multiple times, such as PLoP, ECOOP, OOPSLA and SAC, but the rest of the papers all come from different venues. The large variety of publication venues could indicate that the subject of using language features to improve object-oriented design patterns is rather niche and that there is no specific venue for this.

Out of the 34 papers, there are 18 conference papers, 11 journal papers, and 5 workshop papers. The full data collected on each paper, including publishers, is available in our replication package[3].

We extract from each paper any language feature used to improve object-oriented design patterns. We use a broad definition of language features. The extracted data is, as expected, very diverse. Some features are closely related to specific programming paradigms (e.g., aspect-oriented pointcuts). Other features are more general and could be applied in many ways (e.g., mixins). To make sense of this data and obtain a global view, we create a mind map of these improvements that can be seen in Figure 4.

In this figure, nodes represent paradigms, paradigm applications and language features. Colours represent the type of node. This mind map helps navigate and make sense of the data. We classified language features into the paradigms to which they relate the most. However, some features could fit into multiple categories. When this happened, we classified the feature according to how it was presented in the study where it appeared. For example, we found different studies pertaining to Mixins in the context of both Aspect-Oriented Programming and Object-Oriented Programming, thus we added the feature to both paradigms. We explain our categorization in the following.

Purple nodes represent programming paradigms. We define a programming paradigm as a set of cohesive features for linking domain concepts with programming concepts and for organizing these programming concepts. We identified in our dataset four main paradigms: Object-oriented Programming, Functional Programming, Meta-programming and Reactive Programming.

[3]https://www.ptidej.net/downloads/replications/ist22/

Publication Venue	**# Papers**
Conference on Pattern Languages of Programs (PLoP)	4
European Conference on Object-Oriented Programming (ECOOP)	2
Special Interest Group on Programming Languages (SIGPLAN)	2
Symposium on Applied Computing (SAC)	2
Computer Languages, Systems & Structures	1
International Conference on Advanced Communication Control and Computing Technologies (ICACCCT)	1
International Conference on Computer Science and Information Technology (CSIT)	1
International Conference on Informatics and Systems (INFOS)	1
International Conference on Objects, Components, Models and Patterns	1
International Conference on Software Engineering Advances (ICSEA)	1
International Conference on Software Technology and Engineering (ICSE)	1
International Conference on Systems, Programming, Languages and Applications: Software for Humanity	1
International Journal of Software Engineering and Knowledge Engineering	1
International Journal of Software Innovation	1
International Symposium on Foundations of Software Engineering (FSE)	1
International Workshop on Context-Oriented Programming	1
Journal of Object Technology	1
Journal of Software	1
Journal of Systems and Software	1
Journal of the Brazilian Computer Society	1
Lecture Notes in Computer Science	1
Second Eastern European Regional Conference on the Engineering of Computer Based Systems	1
Software Engineering and Knowledge Engineering (SEKE)	1
Software Technology and Engineering Practice (STEP)	1
Software: Practice and Experience	1
Workshop on Aspects, components, and patterns for infrastructure software (ACP4IS)	1
Workshop on Generic Programming (WGP)	1
Workshop on Reuse in Object-Oriented Information Systems Design	1

Table 3: Publication Venues

Green nodes in the mind map represent language features. We define a language feature as an element of the syntax or grammar of a programming language that can be used to express a solution to a problem. For example, classes are a language feature of object-oriented languages used to represent real-life categorization of objects and create and manipulate instances of those objects. In our mind map, we assigned each language feature found in the studies to its underlying paradigm.

Table 4 offers a detailed view of the information discussed in this section. The table contains references to every paper in the study and can be used to find papers on a particular language feature or paradigm. We also specify the implementation languages presented in the papers for each of the features where available.

3.4.1 Meta-Programming

Meta-programming is a paradigm allowing programs to generate or modify their structure (either at compile, load-time, or run-time). It is possible to effectively implement new paradigms using Meta-programming, its classification is slightly different from that of the other paradigms. Directly under Meta-programming, we classify applications of Meta-Programming (as yellow nodes). These applications are specific usage of Meta-programming to create new languages. For example, we classify Aspect-Oriented Programming as an application of Meta-programming because it modifies the structure of the target program at compile-time. Meta-programming applications are paradigm extensions to a target language, rather than stand-alone paradigms. For example, Krishnamurthi et al. [53] use the MzScheme language's meta-programming capabilities to implement Zodiac, an extension to the language combining features from Object-Oriented Programming and Functional Programming to implement the *Visitor* pattern.

Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) is a paradigm extension to procedural programming that was introduced in 1997 by Kiczales et al. [2]. The goal of AOP is to increase

Paradigm	Language Feature	Papers	# papers
Functional Programming	Case Classes	[69]	1
Functional Programming	Closures	[11, 36]	2
Functional Programming	Immutability	[76]	1
Meta-Programming	AOP Annotations	[38, 59]	2
Meta-Programming	AOP Mixins	[7, 55]	2
Meta-Programming	AOP Join-point	[4, 8, 10, 12, 14, 18, 27, 28, 30, 34, 37, 39, 41, 45, 80, 87, 88]	17
Meta-Programming	Layer Objects	[84]	1
Meta-Programming	Pattern Keywords	[35, 94]	2
Meta-programming	Reflection	[32, 48, 61]	3
Object-Oriented Programming	Chameleon Objects	[11]	1
Object-Oriented Programming	Class Extension	[45]	1
Object-Oriented Programming	Default Implementation	[11]	1
Object-Oriented Programming	Extended Initialization	[11]	1
Object-Oriented Programming	Mixins	[11, 17, 76]	3
Object-Oriented Programming	Multiple Inheritance	[76]	1
Object-Oriented Programming	Object Interaction Styles	[11]	1
Object-Oriented Programming	Subclassing members in a subclass	[11]	1
Reactive Programming	Signals	[77]	1

Table 4: Summary of papers on language features for improving design patterns implementations

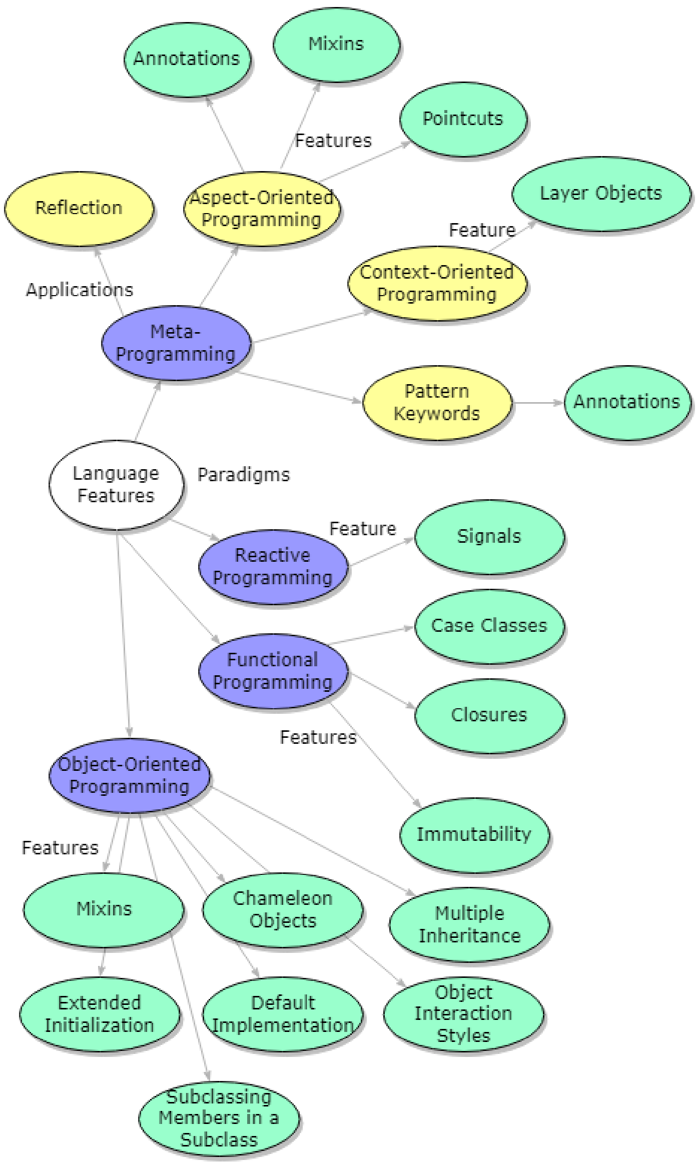

Figure 4: Mind map of paradigms, languages, and features for improving design patterns

modularity by encapsulating cross-cutting concerns into code units called Aspects. Aspects are a language construct similar to classes in OOP. OOP/AOP implementations, like AspectJ[4], are the most popular in the literature, even though the original study did not associate AOP directly to OOP.

AOP works by making changes to the structure of the program at specific points. For example, it is possible to create an aspect which adds logging functionality to every method of a given class. Thus, with AOP, it is possible to extract recurring functionality to encapsulate them in aspects and avoid cluttering classes with unrelated, or cross-cutting, concerns.

Among the primary studies, we found three main implementations of AOP within OOP. The most common implementation [4, 8, 10, 12, 14, 18, 27, 28, 30, 34, 37, 39, 41, 45, 80, 87, 88] uses pointcuts and advices (also called the join-point model) to modify existing classes. Consider the following pointcut[5]:

```
pointcut setter(): target(Point) &&
                   (call(void setX(int)) ||
                    call(void setY(int)));
```

This pointcut designates every location where a method named `setX` or `setY`, with a single argument of type `int` and a return value of `void`, is called on an instance of the class `Point`.

The second part of this model is the advice, which designates the code inserted at every location referenced by the pointcut. Advices typically take the form of a regular method of the extended language, with the exception that it is usually decorated with a keyword such as `after`, `before`, or `around`, which dictates where the advice should be inserted relative to the pointcut.

The example shown is unrelated to design patterns implementations, since using AOP to implement design patterns typically involve multiple files and many lines of code, which would make it more difficult to see the important features of the paradigm.

[4]https://www.eclipse.org/aspectj/
[5]https://www.eclipse.org/aspectj/doc/released/progguide/language-joinPoints.html

Hannemann and Kiczales [45] present a full implementation of every design pattern using AspectJ and compare them to Java implementations. They reported that they could remove many code duplications. For example, in the case of *Observer*, they extract the record-keeping of the observer list, as well as the `add` and `detach` methods, into an abstract aspect named `ObserverProtocol`.

AOP can be implemented in a different way using code annotations [38, 59]. This approach is similar to the join-point model, replacing the pointcuts with annotations. This approach is used by the C# framework PostSharp[6]. Using this framework, one could implement Microsoft WPF `INotifyPropertyChanged` interface, which is an implementation of the *Observer* pattern, with the following annotation[7]:

```
[NotifyPropertyChanged]
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public Address Address { get; set; }
}
```

Using this approach, Giunta et al. [38] and Jicheng et al. [59] create annotations to represent the *Factory Method*, *Observer*, and *Singleton* design patterns.

Yet another implementation of AOP uses mixins or an equivalent feature. Kuhlemann et al. [55] use Jak, a Java language extension that combines classes and mixins, to achieve similar functionality to AspectJ. They implement every design pattern in Jak and compare their implementations to AspectJ implementations [45]. They conclude that Jak has better support for modularizing cross-cutting concerns, but that the extension should be complementary to AspectJ. Axelsen et al. [7] propose the concept of Package Templates to implement the *Observer* pattern and compare it to an AspectJ implementation. Package Templates function like module mixins (at the package level) in that they can introduce

[6]https://www.postsharp.net/

[7]https://www.postsharp.net/postsharp

(generate) new classes, methods and attributes when used. They showed that their approach provides a sufficiently powerful framework to implement design patterns with minimal AOP mechanism compared to the join-point approach.

Layer Objects

Springer et al. [84] present the concept of Layer Objects to implement the *Decorator*, *Observer*, and *Visitor* design patterns using Context-Oriented Programming, which allows changing the behaviour of objects based on scope (or context). A Layer Object can be activated (e.g., with the `with` keyword) within a specific scope, and will override the behaviour of specified classes within that scope. They present examples implemented in a Java-like pseudo-code.

With this feature, it is possible to add behaviour to an existing object (as per the *Decorator* pattern). For example, if we had a `Field` class, representing a tile in a game and wanted to add a “burning” by which a player takes damage when entering the field, the `BurningFieldDecorator` could be implemented as such:

```
class Field {
  def enter(player) {
    // Do something when a player enters the field.
  }
}


class BurningFieldDecorator {
  def damage = 15;


  def Field.enter(player) {
    player.health -= thisLayer.damage;
    proceed(player);
  }
}

// Usage example
def decorator = new BurningFieldDecorator();
field.activate(decorator);
```

Pattern Keywords

Some studies opted for introducing design patterns directly. Ghaleb et al. [35] propose integrating *Decorator*, *Observer*, and *Singleton* as keywords. For example, a *Singleton* could be declared like this:

```
public singleton class A
{
    instantiate A as s1;

    public static void main(String[] args)
    {
        instantiate A as s2, s3;
    }
}
```

The `singleton` keyword marks the class `A` as a *Singleton*, of which its unique instance can be obtained using the `instantiate` keyword.

Zhang and Oliveira [94] do something similar with the *Visitor* pattern by introducing EVF. EVF is a framework to implement *Visitors* in Java. It introduces new keywords in the form of annotations (such as `@Visitor`) to create data structures with internal or external visitors.

Adding keywords to a language to directly support design patterns requires either the use of Meta-programming or the creation of language extensions or domain-specific language. Unlike the approach presented by Krishnamurthi et al. [53], where they used the macro feature of MzScheme to implement Zodiac, an extension to the language for implementing *Visitor*, the studies presented under this feature used third-party tools to add keywords to Java. The effect, however, is the same as if they had used macros to create these keywords (which Java does not support). Whether design patterns should be language features is however debatable [20].

Reflection

Reflection is an application of meta-programming. It allows developers to manipulate the data structures at run-time. For example, one could add a new method to an existing class, change the type of an attribute, etc. Unlike macros or Aspect-Oriented Programming, Reflection typically does not allow modifying code directly (e.g., inserting lines of code in the middle of a method). Reflection is available in mainstream programming languages like Java and C#.

Fortuin [32] uses Reflection to implement the *Abstract Factory* pattern in Java. They use a static method that, based on the given arguments, constructs an object by accessing the proper class constructors and passing the arguments using a hashtable.

Mai and Champlain [61] and Hussein et al. [48] use Reflection to implement the *Visitor* pattern to overcome the limitation of Java of only supporting single dispatch (as discussed in Section 3.4.3 about Case Classes). Mai and Champlain [61] defined a `findMethod` method that gets the correct `Visit` method on the visitor object. Reflection could also be used to automatically implement `accept` methods on the target, removing the cross-cutting concern from the visitable object.

3.4.2 Reactive Programming

Reactive Programming [77] is a design paradigm which addresses asynchronous programming logic, especially concerning value updating and change propagation. It can be combined with Functional and Object-Oriented Programming to facilitate dealing with problems related to data updating (the same kind of problems targeted by the *Observer* design pattern). Scala offers libraries such as REScala [78] and Scala.react [62] to enable reactive programming in the language.

Signals

Signals are a feature of Reactive Programming that allows expressing dependencies among values [77]. When a dependency of a signal changes, the expression defined by the signal is automatically recomputed from the new values of the dependencies. This functionality is similar to the *Observer* pattern. For example, take the following Scala code:

```
val a = Var(1)
val b = Var(2)
val s = Signal{ a() + b() }

println(s.getVal()) // 3
a() = 4
println(s.getVal()) // 6
```

When the value of variable `a` changes, so does the value of signal `s`. Thus, support of Signals and Reactive Programming in a language renders the *Observer* pattern obsolete.

3.4.3 Functional Programming

Functional Programming is a paradigm that uses pure functions to assemble higher-order functions (i.e., functions that receive other functions as arguments, such as map, reduce, etc.) to create programs. We classified studies that used features related to this paradigm: case classes [69] (also called pattern matching), closures [11, 36], and immutability [53]. We admit that immutability is less of a feature and more of a property, but it is a property of pure function and functional languages usually support and sometimes enforce immutability. Immutability can also affect how design patterns are implemented [53], which is why we chose to include it here.

Case Classes

Case Classes, also known as pattern matching, are a feature of functional programming languages. In Scala, Case Classes are used to create sets of classes that can be distinguished using the `match` keyword. Oliveira et al. [69] use Scala to present many variations of the *Visitor* design patterns, as well as a library to use them. They use Case Classes to implement the accepting mechanism of this design pattern. They then use pattern matching to distinguish the different types to visit. Using their library, it is possible to implement the *Visitor* design pattern:

```
// Visitor structure for a Binary Tree.
trait Tree {
    def accept[R] (v :TreeVisitor[R]):R
}
case class Empty extends Tree {
    def accept[R] (v :TreeVisitor[R]):R = v.empty
}
case class Fork (x :int,l : Tree,r: Tree) extends Tree {
    def accept[R] (v :TreeVisitor[R]):R =
        v.fork (x,l,r)
}

trait TreeVisitor[R] {
    def empty :R
    def fork (x : int,l :Tree,r: Tree):R
}

// Concrete implementation of visitor to calculate
// the depth of the Tree.
def depth  = new CaseTree [External,int] {
    def Empty = 0
    def Fork (x : int,l :R[TreeVisitor],r:R[TreeVisitor]) =
        1+max (l.accept (this),r.accept (this))
}
```

This implementation does not require both `accept` and `visit` methods usually present in the *Visitor* implementation. They only implement an `accept` method.

Closures

Closures are the main feature of the Functional Programming paradigm. A Closure is an encapsulation of behaviour and data, much like Classes are in OOP. Unlike Classes, a Closure only encapsulates one function and its data cannot be directly accessed from outside the closure (with some exceptions). Closures are supported by most modern languages, such as Java, C#, C++, Python (with some caveats), JavaScript, Kotlin, etc. In most languages, Closures are created using some kind of lambda syntax such as :

```
var i = 42
var closure = (argument) => {
  // Some code which can use the i variable.
}
```

Gibbons [36] present advanced uses of the `map`, `fold`, and other standard FP methods in Haskell. They discuss functional implementations of the *Composite* (using recursive data structures), *Iterator* (using `map`), *Visitor* (using `fold`), and *Builder* (using `unfold`) design patterns.

Compared to Closures, the *Command* design pattern has the advantage of being able to expose its data to the outside world if needed, allowing some degree of control over the behaviour of the object. Batdalov and Nikiforova [11] propose to replace conventional functions with a generalization of the *Command* design pattern. Every function or method would become a Functor (an object with a single public method) and would make usage of the *Command* pattern obsolete.

Immutability

Immutability is a property of many Functional Programming languages. The Haskell language enforces immutability, which in turn makes the use of certain functional design patterns, such as monads, ubiquitous in the language.

Immutability has an effect on design pattern implementations. For example, we stated above that the *Command* design pattern is similar to Closures, with the exception that it can expose its internal data and mutators. In an immutable context, this advantage becomes unconsequential as the exposed data could not be interacted with in any significant way. In an immutable context, Closures make the *Command* pattern obsolete.

Sakamoto et al. [76] present two variations on the *State* design pattern implemented using Scala. One of these is the Deeply Immutable State pattern, which makes use of the immutability features of Scala (such as the ability to easily clone Type Classes while modifying certain attributes, similar to Record Updating in OCaml[8]) for its implementation.

3.4.4 Object-Oriented Programming

Object-Oriented Programming is a paradigm that uses four general features to organize code: encapsulation, abstraction, inheritance, and polymorphism. Features classified under this paradigm directly affect OOP or require it to function. For example, the Chameleon Objects [11] feature allows objects to dynamically change class. This requires and affects classes and objects, features which are directly related to OOP.

Chameleon Objects

Chameleon Objects is a feature proposed by Batdalov and Nikiforova [11] to help with the implementation of the *State* and *Factory Method* design patterns. The feature allows an object to change class at runtime. While no language was presented in Batdalov and Nikiforova's paper, some languages allow such change already (e.g., Common Lisp, Perl). With this feature, it is possible to implement the *State* pattern by changing the class of an object when it changes state. The same feature could be used to allow a class constructor to

[8]https://dev.realworldocaml.org/records.html

change the instantiated object class depending on its arguments, effectively implementing a *Factory Method*.

In Common Lisp, a class can be changed using the following simple line:

```
(change-class target-object target-class)
```

The same can be achieved in Perl with the `bless` keyword:

```
bless $targetObject, 'Package::TargetClass';
```

Class Extension

Hanneman and Kiczales [45] use AOP to implement design patterns using different features of AspectJ for different patterns. For *Adapter, Bridge, Builder, Factory Method, Template Method*, and *Visitor*, they specifically use the open class feature of AspectJ, which allows classes to be extended with new methods and attributes outside of their declaration (at compile-time).

With class extensions, it becomes possible to encapsulate duplicated code from a design pattern implementation into an extension. For example, we could extract the `accept` method of the *Visitor* pattern into a class extension, effectively allowing a *Visitor*-free implementation to exist independently of its pattern implementation.

Default Implementation

Default Implementation is a feature proposed by Batdalov and Nikiforova [11]. It allows abstract classes to specify a default concrete implementation. This feature is similar to how the *Abstract Factory* pattern is implemented in Dependency Injection frameworks, such as Spring[9]. Beyond *Abstract Factory*, the authors argue this feature would help implementations of the *Builder*, *Bridge*, *Command*, and *Strategy* patterns.

[9]https://spring.io/

Extended Initialization

Extended Initialization is a feature proposed by Batdalov and Nikiforova [11] to help with the implementation of the *Builder* and *Fatory Method* patterns. This feature would allow object creation to be divided into multiple steps (effectively achieving the same functionality as the *Builder* pattern and rendering it obsolete).

Mixins

While we already introduced Mixins in the context of Aspect-Oriented Programming in Section 3.4.1, this feature can be used separately from AOP. Mixins are an extension of the OOP paradigm to combine classes together. Unlike inheritance, Mixins do not enforce an "is-a" relationship between the individual classes.

Burton and Sekirinski [17] present implementations of the *Decorator*, *Proxy*, *Chain of Responsibility*, and *Strategy* design patterns using Mixins in Java. For example, a Decorator may be implemented by combining the Decorator with the target class, as such:

```
class Component { Operation(); }

class ConcreteComponent implements Component { ... };

class DecoratorMixA implements Component
                needs Component
    Operation() { ... Component.Operation() };

class DecoratorMixB implements Component
                needs Component
    Operation() { ... Component.Operation() };

// Usage
class Client {
    main() {
        ConcreteComponent cc =
            new ConcreteComponent with DecoratorMixA;
        extend cc with DecoratorMixB;

        cc.Operation();
    }
}
```

The authors declared that both decorators are of type Component but also require a concrete instance of Component as a Mixin. In the usage section, they mix the concrete instance `ConcreteComponent` with the `DecoratorMixA` class, and then extend it with `DecoratorMixB`, effectively creating an object which is a combination of all three classes.

Sakamoto et al. also use Mixins in their implementation of the *State* pattern in Scala. Batdalov and Nikiforova [11] propose a concept of Responsibility Delegation, which would allow a class to delegate a part or all of its responsibility to another class (perhaps similar to the `DoesNotUnderstand` method in Smalltalk), achieving a behaviour similar to Mixins.

Multiple Inheritance

Many OOP languages only support single inheritance. This led to the introduction of interfaces to circumvent the limitation of an object only being able to be part of one hierarchy. Interfaces, however, do not typically allow reuse.

Some design patterns have duplicated code that we could factor into another class, to reduce code duplication and cross-cutting concerns. For example, one could extract the bookkeeping logic of the *Observer* pattern into another class and have observable objects inherit that class. However, in a single-inheritance context, this would “hijack” the only inheritance possibility for the observable object.

Sakamoto et al. [76] present an implementation of the State design pattern in Java (using single inheritance) and C++ (using multiple inheritance). They evaluated that the Java implementation has code duplication, while the C++ implementation does not.

Object Interaction Styles

Object Interaction Styles is a feature proposed by Batdalov and Nikiforova [11] to help with the implementation of the *Proxy*, *Observer*, and *Facade* patterns. They would allow different ways of interacting with objects through method calls. The default interaction style is synchronous calls, where a caller awaits the result of the method before continuing to the next instruction. Other interaction styles include asynchronous request/response, broadcast, and publish/subscribe. An example of this would be the async syntax in C# and JavaScript.

Subclassing Members in a Subclass

Batdalov and Nikiforova [11] propose that OOP languages allow subclassing a member in a subclass to help with the implementation of the *Template Method* and *Visitor* patterns. A subclass may redefine one of its members by restricting its class.

To understand this feature, let us take the example from [11] about Android development. Take the following pseudocode:

```
class Activity { ... }
class Fragment {
    ...
    Activity getActivity() { ... }
}
```

If we wanted to subclass `Activity` into a new class `MyActivity` and `Fragment` into `MyFragment`, we would end with the following subclasses:

```
class MyActivity extends Activity { ... }
class MyFragment extends Fragment {
    ...
```

```
        Activity getActivity() { ... }
    }
```

However, if we knew the `getActivity` method of `MyFragment` could ever only return instances of `MyActivity`, we could not make the change, as the interface of the parent class `Fragment` requires this method to return the type `Activity`.

The proposed feature would allow us to make that change. The feature does not violate covariance or contravariance but could be considered unsafe in some cases [60].

3.4.5 Design Patterns

For each primary study, we extracted the design patterns for which improvements were suggested. Figure 5 shows the distribution of the number of studies by design pattern. As discussed in Section 3.3, we only considered patterns from the GoF book [33].

Figure 5 shows researchers' interest in design patterns. We notice that some patterns are more prevalent than others in our dataset. In particular, *Observer*, *Visitor*, and *Decorator* receive the most attention from researchers.

If we look at the number of features associated with each pattern, *Observer* and *Visitor* also come on top, but there is an interesting relationship at play here. While *Observer* has received overwhelmingly more attention from papers (20 papers against *Visitor* 13), they have the same number of associated features. Also, they have a similar amount of papers focusing solely on them ([7, 14, 27, 30, 77, 88] for *Observer* and [41, 48, 53, 61, 69, 94] for *Visitor*). Other patterns have very few papers that address only them. The only other two are *Abstract Factory* [32] and *State* [76].

3.4.6 Measures

We extracted the different measures used in the studies to compare the suggested improvements to design patterns to the classical implementations. Some studies did not measure or compare their implementation and were demonstrations of new language features or of a language extension. We observed 39 different measure names. Table 6 shows the usage

Design Pattern	# Features	Papers	# Papers
Abstract Factory	4	[8, 11, 32, 39, 45, 55, 80]	7
Adapter	4	[11, 39, 45, 55, 87]	5
Bridge	5	[11, 39, 45, 55]	4
Builder	6	[11, 36, 39, 45]	4
Chain of Responsibility	4	[11, 12, 17, 28, 39, 45, 55]	7
Command	4	[11, 12, 39, 45, 55]	5
Composite	4	[10, 11, 36, 37, 39, 45, 55]	7
Decorator	6	[4, 10, 11, 17, 35, 39, 45, 55, 84]	9
Facade	4	[11, 39, 45]	3
Factory Method	6	[11, 37, 38, 39, 45, 55]	6
Flyweight	3	[8, 11, 37, 39, 45, 55]	6
Interpreter	2	[39, 45, 55]	3
Iterator	3	[36, 39, 45, 55]	4
Mediator	3	[11, 39, 45, 55, 80]	5
Memento	2	[39, 45, 55]	3
Observer	9	[4, 7, 10, 11, 12, 14, 27, 30, 35, 37, 38, 39, 45, 55, 59, 77, 80, 84, 87, 88]	20
Prototype	2	[39, 45, 55, 80]	4
Proxy	7	[8, 11, 17, 37, 38, 39, 45, 55]	8
Singleton	4	[4, 8, 12, 35, 38, 39, 45, 55]	8
State	6	[11, 12, 39, 45, 55, 76, 80]	7
Strategy	4	[11, 17, 28, 39, 45, 55, 80, 87]	8
Template Method	4	[11, 39, 45, 55]	4
Visitor	9	[10, 11, 36, 39, 41, 45, 48, 53, 55, 61, 69, 84, 94]	13

Table 5: Language features per design patterns

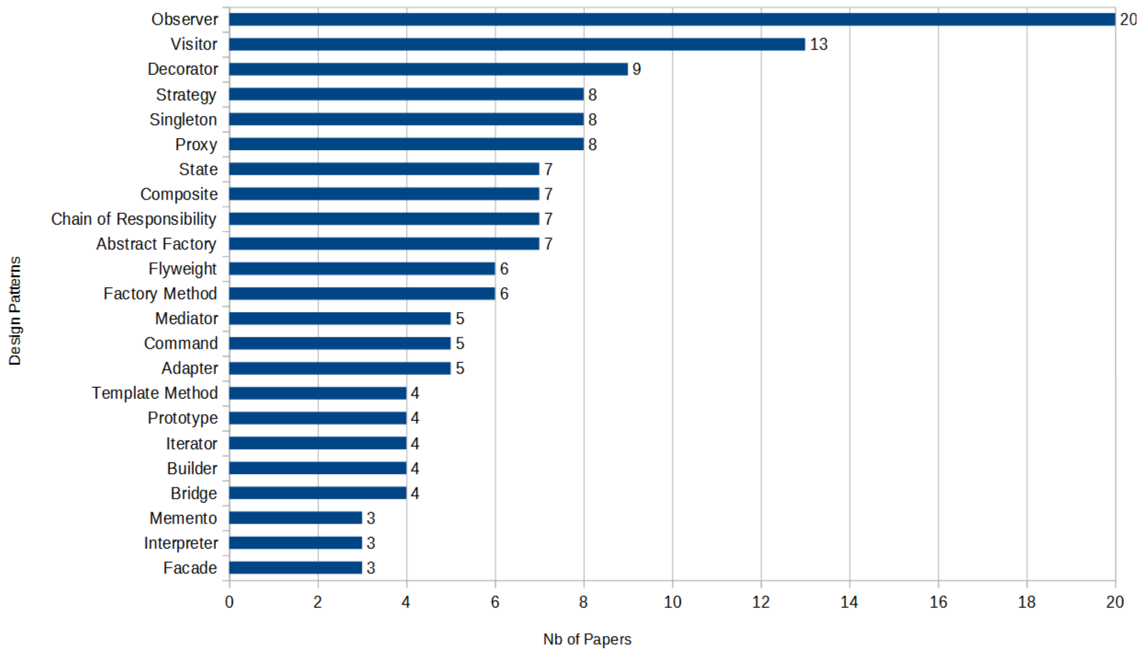

Figure 5: Number of papers by design pattern

of these measures by number of papers.

Some measures are more prevalent than others to measure improvements. The top ten measures are related to the usage of inheritance (DIT), coupling and cohesion (CBC, LCOO), concern diffusion (CDC, CDLOC, CDO), code size (LOC, NOA, WOC), and reusability. This matches the stated goal of design patterns [33].

Most of these measures (DIT, CBC, LCOO, LOC, WOC) come from the Chidamber and Kemerer suite of measures [21]. Concern diffusion measures relate to measuring improvements from Aspect-Oriented Programming regarding cross-cutting concerns [80].

Many of the other measures are unique to only one paper. For example, Teebiga and Velan [87] use a measure they call lines of class code (LOCC), which they attribute to Ceccato and Tonella [19]. LOCC measures the number of lines of code within classes (as opposed to within aspects). The measure is similar to the number of lines of code (LOC), but the context in which it is used (i.e., to differentiate between aspect and class code) is different enough that we opted to classify it as its own entry.

Most of the measures presented in Table 6 are used in AOP studies. In fact, only three

Measures	Type	# Papers
Depth of Inheritance Tree (DIT)	Quantitative	6
Coupling Between Components (CBC)	Quantitative	5
Lack of Cohesion in Operations (LCOO)	Quantitative	5
Lines of Code (LOC)	Quantitative	5
Concern Diffusion over Components (CDC)	Quantitative	4
Concern Diffusion over LOC (CDLOC)	Quantitative	4
Concern Diffusion over Operations (CDO)	Quantitative	4
Number of Attributes (NOA)	Quantitative	4
Weighted Operations per Component (WOC)	Quantitative	4
Reusability	Qualitative	3
Composition Transparency	Qualitative	2
Coupling on Intercepted Modules (CIM)	Quantitative	2
Duplicated Code (DC)	Quantitative	2
Locality	Qualitative	2
Modularity	Qualitative	2
Unpluggability	Qualitative	2
Weighted Operations in Module (WOM)	Quantitative	2
Abstract-Pattern-Reusability	Qualitative	1
Binding-Reusability	Qualitative	1
Cohesion	Qualitative	1
Comprehension	Quantitative	1
Correctness	Quantitative	1
Coupling Between Modules (CBM)	Quantitative	1
Coupling on Advice Execution (CAE)	Quantitative	1
Coupling on Field Access (CFA)	Quantitative	1
Coupling on Method Call (CMC)	Quantitative	1
Cross-cutting concerns	Qualitative	1
Crosscutting Degree of an Aspect (CDA)	Quantitative	1
Encapsulation	Qualitative	1
Execution time	Quantitative	1
Generalization (Inheritance)	Qualitative	1
Indirection	Qualitative	1
Lines of Class Code (LOCC)	Quantitative	1
Number of Children (NoC)	Quantitative	1
Number of classes and relations (NOCR)	Quantitative	1
Response For a Module (RFM)	Quantitative	1
Separation of concerns	Qualitative	1
Skill floor	Quantitative	1

Table 6: Measures by usage in papers

non-AOP papers make use of any measure at all [76, 77, 94]. Zhang and Oliveira [94] use lines of code to compare various Visitor implementations. Salvaneschi et al. [77] asked participants to answer specific questions to test their comprehension of Reactive Programming. They then measure their correctness in answering the questions (score of the participants on the task), their comprehension of the code (time required to complete the task) and the skill floor (a correlation between measured programming skill and performance during the task). Sakamoto et al. [76] compared the amount of duplicated code and the number of classes and coupling between implementations of the State patterns.

The other non-AOP papers were descriptive studies presenting a new feature without any comparison. They argued improvements in an informal way.

3.4.7 Types of Papers

We categorize each paper based on the type of study. We are particularly interested in case studies and experiments, to answer our fourth research question: What experiments have been done on these language features?.

We use the same categories as those of Ampatzoglou et al. [6]: case studies, experiments, descriptive studies, conceptual analysis, and literature review. Table 7 summarises our data.

Most of the studies are descriptive studies: 23 out of the 34 studies. They presented a new tool or feature by describing its functionalities and advantages but did not necessarily compare it to other approaches (as discussed in Section 3.4.6).

There are also 10 case studies out of the 34 studies. These case studies compare a new feature against established OOP implementations. Case studies can be further divided based on the source of their subjects: either in-vivo or in-vitro. In-vivo case studies are done on already-existing software, often available online as open-source projects. In-vitro case studies are done on software made specifically for the case study. Every case study in our analyzed studies was in-vitro because there are not many (if any) opportunities to find projects developed by third parties using new tools or methods proposed by a paper.

We found only one study that performed a controlled experiment. Salvaneschi et al. [77] performed an empirical experiment on program comprehension of design patterns using

Study type	# papers
Case Study	10
Experiment	1
Descriptive Study	23

Table 7: Study types

reactive programming with 38 undergraduate students (which we discuss in Section 3.4.6).

We interpret the fact that every case study is in-vitro and that there is only one experiment to mean that most of the new features proposed have not been extensively tested.

3.4.8 Timeline

We also studied the distributions of our data over time, to assess any trends or irregularities. Figure 6 shows the distribution of the publication years of the 34 studies. Interest in the subject seems to have peaked around 2010 and it waned in recent years.

We take a look at the distribution of the other two major categories of studies in our study: Meta-programming papers and Aspect-Oriented papers. The other categories had too few studies to obtain a meaningful distribution.

Figure 7 is the distribution of papers related to Meta-programming over their publication year, and Figure 8 is the same considering only papers related to Aspect-oriented Programming. The Meta-programming papers follow a similar distribution to that of the general distributions of the papers.

The AOP distribution shows a peak around 2010 and no AOP paper has been published on improving object-oriented design patterns implementations since 2016.

When we filter out AOP papers from the timeline, we obtain Figure 9, which shows a more evened-out distribution over the years.

However, our dataset is small so we compare our findings to the popularity of each paradigm in general over the years. We used the Google Trends website[10] to obtain data about the proportion of searches for keywords relating to each paradigm. We used "metaprogramming", "aspect oriented programming" and "functional programming" as keywords

[10]https://trends.google.com/

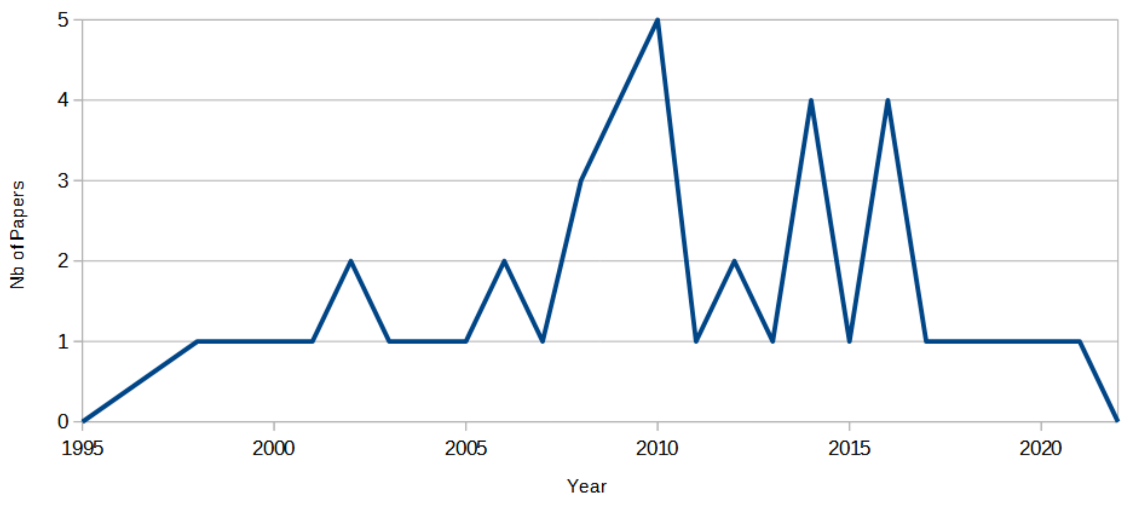

Figure 6: Papers distribution by year

and performed the search in the United States. We summarize the results of those queries in Figure 10. These represent how many searches were made for each term over the years.

In this figure, we observe the same trends as in our distributions of studies. There is a peak around 2010 for Meta-Programming and Aspect-Oriented Programming and they both seem to wane in recent years, although AOP more so than Meta-Programming. Functional Programming appears to be gaining popularity in recent years, which also matches the resurgence of new functional programming features in modern languages (e.g., the addition of Closures and lambda syntax in Java, immutability features in C#, etc.).

3.5 Discussion

We answer and discuss our research questions.

3.5.1 Language Features

Our first research question is: *RQ3.1: What language features have been suggested to improve design patterns implementations?*

We extracted 18 features from 34 selected papers. These features range across a variety of programming paradigms and improve design patterns implementations in various ways.

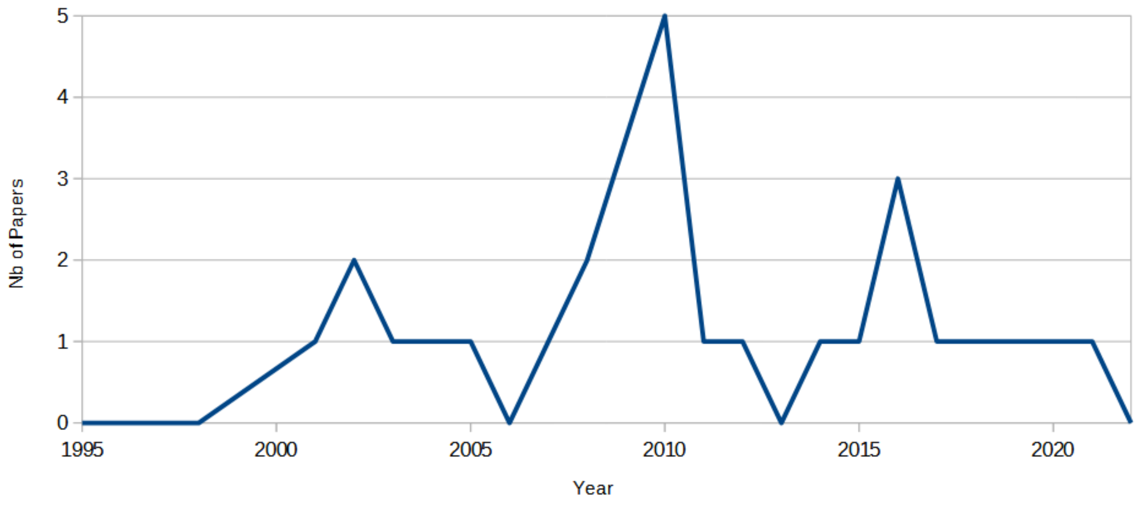

Figure 7: Meta-programming papers distribution by year

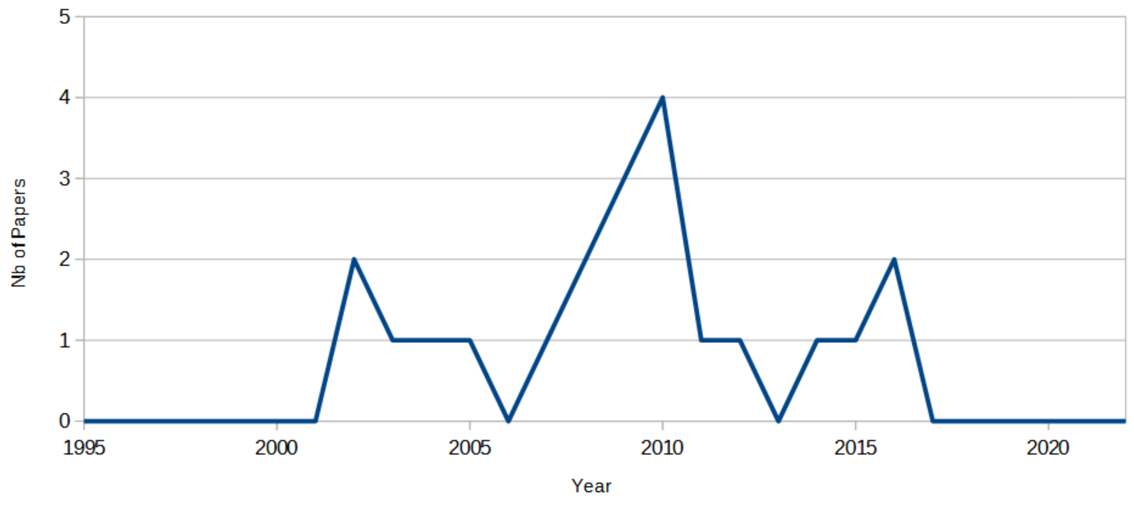

Figure 8: AOP papers distribution by year

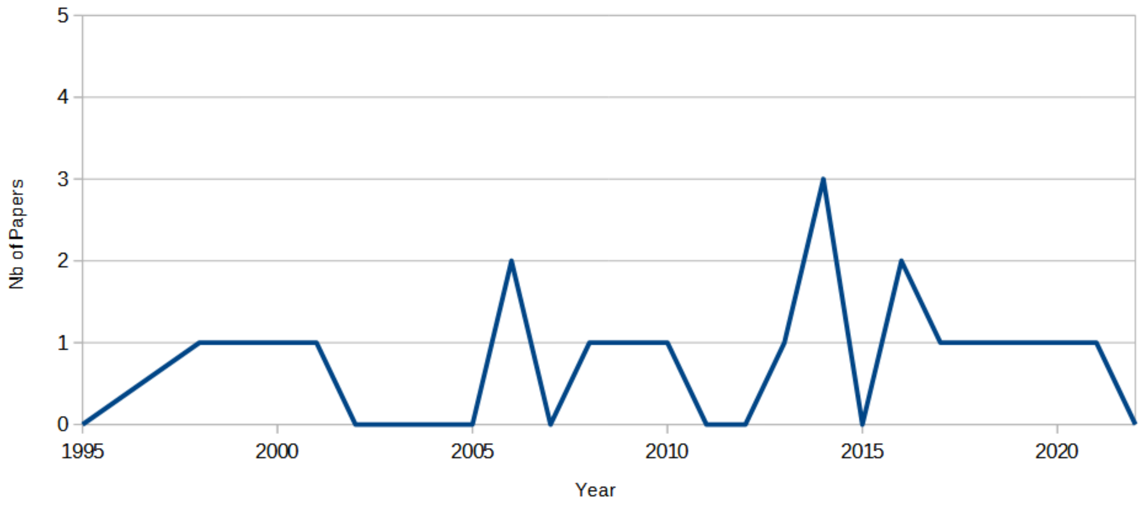

Figure 9: Non-AOP papers distribution by year

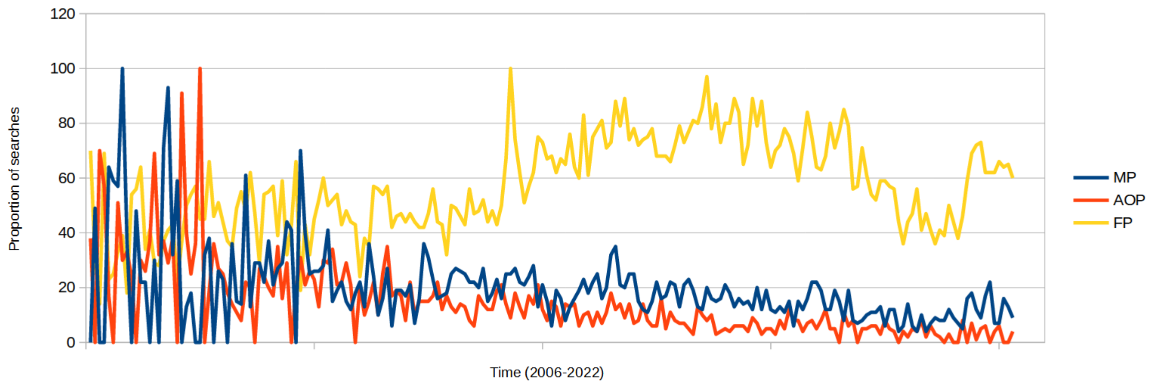

Figure 10: Google Trends results on Meta-programming, Aspect-oriented Programming and Functional Programming

Many of these features (7 out of 18) were implemented as extensions to Java. In general, Java seems to be the most popular language to discuss improvements in design patterns. The next most popular language seems to be Scala, which is another JVM language, most often used to discuss functional programming features. Even when using dynamically-typed pseudo-code, Springer et al. [84] describe it as ”Java-like”, even though to us it looks more like Python with brackets. There seems to be a prevalence of Java in design patterns and OOP research, which may be a concern because Java does not represent every OOP language, and many features available in other OOP languages, such as C#, Common Lisp, Kotlin, and Smalltalk, are not available in Java.

Most features stem from Meta-programming applications, which modify or insert code into OOP software. Features like Reflection and Pattern Keywords are used to reduce the amount of code duplication and code size of design patterns. It is not surprising that Meta-programming is a popular tool to address code duplication and size, as one of its main strengths is the ability to factor out code and remove duplication.

Aspect-Oriented Programming is a major part of the Meta-programming features. AOP and Layer Objects, while also reducing code duplication, focus on reducing cross-cutting concerns in design patterns [45]. Despite AOP being a language and paradigm-agnostic feature (the original paper was presented using Common Lisp [2]), every paper in our dataset discusses it using various Java extensions. While there are solutions which borrows from AOP in other languages, such as the PostSharp library in C#, the paradigm itself does not seem to have gained much traction in other languages.

Some studies also presented domain-specific languages added on top of an existing language, such as Zodiac, an extension of MzScheme used to implement the *Visitor* pattern [53].

Many features were also suggested to extend OOP. Chameleon Objects allow changing the class of an object at runtime. Mixins allow combining multiple classes together to facilitate reuse through composition. Multiple Inheritance solves some issues with code duplication by allowing multiple sources of reuse. Code reuse in pure OOP is usually limited to inheritance and subtyping mechanisms, so naturally, the features suggested interact with these mechanisms.

Some features come from Functional Programming. Case Classes in Scala are an implementation of pattern matching, which can be used to circumvent the issues of single dispatch languages and facilitate the implementation of the *Visitor* pattern. The feature has been gaining popularity of late, being added to existing languages such as C# and Java in the form of Records.

Closures are another functional feature that has been implemented in many OOP languages, such as Java, C++, and C#, but also part of older languages, such as Smalltalk (code blocks) and Common Lisp. They can make the implementation of many design patterns easier by storing behaviour as values. It is interesting, and telling, that the feature was already included in Smalltalk, one of the originators of OOP, but was then excluded from later implementations of the paradigm in Java and C++, only to be added back to these languages later.

Immutability, when combined with other features such as Closures, can render certain patterns obsolete (e.g., *Command*) and simplify the implementation of others (e.g., *State*). Patterns that are built around the idea of encapsulating changing state are made much simpler when state changes are eliminated entirely.

Finally, Signals and Reactive Programming have been created to solve the same underlying problem as the *Observer* pattern. Managing events and changes in a program is a common concern. Especially in GUI development, many platforms propose different approaches to this concern. Some, like Java Swing, make use of the *Observer* pattern. Others, like Microsoft WPF, propose a system based on the *Command* pattern. Reactive Programming incorporates change management at the language level.

What language features have been suggested to improve design patterns implementations?

We catalogued 18 language features used to improve design pattern implementations. Meta-programming features are the most suggested, with many studies written about Aspect-oriented Programming. However, our timeline analysis suggests that AOP is not as prevalent nowadays and that Functional Programming features are becoming more and more popular.

3.5.2 Design Patterns

The second research question is *RQ3.2: Which design patterns have the most associated language features suggestions?*

Our data indicate that some design patterns are more prevalent than others as targets for improvement. Some papers include every 23 GoF design patterns in their study (3 papers), but most analyze specific patterns.

We observe that the most prevalent patterns to improve are the *Observer*, *Visitor*, and *Decorator* design patterns. We could interpret pattern prevalence in different ways. Perhaps these patterns are those that have the most obvious flaws, and that is why many papers propose features to improve them. We thought perhaps the most prevalent patterns in research would also be the most used in practice, but that does not seem to be the case. *Observer* may well be the most used pattern of all, but it would be difficult to argue that *Visitor* is.

The fact that *Observer* and *Visitor* are the most prevalent patterns, that they have the most associated language features, and that they have the most papers focusing solely on them would indicate that the underlying problems these patterns are solving are of great interest to OOP.

In the case of *Observer*, there is a need in software systems for controlling data flow and change propagation. The *Observer* pattern as described in the GoF book [33], however, comes with some limitations. As discussed in Section 3.4.4 about Multiple Inheritance, the *Observer* implies duplication that cannot be easily abstracted using inheritance. It also has cross-cutting concerns [2, 45] because it requires the observed object to know and manage its observers. The proposed language features improve on these aspects.

In the case of *Visitor*, modern programming languages (such as Java, C++, and C#) lack support for dynamic dispatch on function parameters. In OOP, dynamic dispatch is only supported on the object on which a method is invoked (allowing the use of polymorphism).

Some languages, such as the Common Lisp Object System (CLOS), support both OOP and dynamic dispatch on function arguments, effectively allowing multiple dispatch. In such languages, the visitor pattern becomes trivial as there is no longer any need for the accept/visit mechanism.

Most of the solutions proposed a way to circumvent the lack of dynamic dispatch on function arguments through the use of Reflection, Case Classes (pattern matching), or Aspects. Many studies propose ways to make the implementation of the *Visitor* easier by integrating it into the language using aspects [45], factoring its complexity using reflection [48, 61], or simplifying it using pattern matching [69].

Which design patterns have the most associated language features suggestions?

Research done to improve design patterns seems to favour specific patterns. Observer, Visitor, and Decorator (in order of appearances) are most often studied. In particular, Observer and Visitor appear to stem from a lack of language support of certain features (such as propagation of change and dynamic argument dispatch).

3.5.3 Measures

For the third research question, *RQ3.3: What measures have been used to evaluate the impact of these language features on design patterns implementations?*, we extracted the measures used by each paper and looked at their distribution.

The measures most often used were those related to reusability. In particular, two sets of measures were prevalent: the CK measures [21] and AOP-related measures [80].

Most of the papers with measurements and comparisons were related to AOP. Most non-AOP papers were descriptive studies and offered no measurement or comparison of the performance of their proposed features.

For AOP papers, measure results were mixed. While many AOP implementations showed improvements with regard to concern diffusion, cohesion, and code size, some studies reported increased code size and higher complexity. Non-AOP papers showed more straightforward results, with reduced code size for Pattern Keywords, reduced code duplication for Multiple Inheritance and Mixins, and improved understandability for Reactive Programming.

The measures most used indicate that design patterns may be a source of complexity and that maintainability and understandability could be improved by making their implementation simpler using certain language features. There is also a concern about concern diffusion in design patterns. For example, the objects involved in the *Visitor* pattern have the added concern of being part of the pattern (i.e., having Visit or Accept methods). Some studies propose to extract this concern from the involved objects and abstract it into its own unit (e.g., an aspect).

What measures have been used to evaluate the impact of these language features on design patterns implementations?

The measures most often used in the literature to evaluate improvements to object-oriented design patterns are measures related to maintainability and understandability, especially those proposed by Chidamber and Kemerer [21]. AOP-related papers also add measures related to concern diffusion, or cross-cutting concerns.

3.5.4 Empirical Studies

Our final research question concerns empirical studies: *RQ3.4: What experiments have been done on these language features?*

We compiled the types of studies in our dataset in Table 7. Most of the studies (23 papers) are descriptive studies proposing or explaining a new feature or language extension. These typically do not offer concrete data to compare the new features with existing approaches.

The second category is case studies (10 papers). Every case study on the subject is done in-vitro, with the studied subjects created specifically for the case study. It would be difficult to find software "in the wild" using new features just proposed by a scientific paper. Because in-vivo case studies are impractical, controlled experiments are important to evaluate the effectiveness of new technology.

We only found one experiment done on evaluating improvement to design pattern implementations with 38 undergraduate students. As discussed above and in Section 3.4.7, there is a clear need for more empirical experiments to evaluate the effects of proposed new features on design pattern implementations and on object-oriented development in general. While experiments on students have their advantages, experiments should also be done on professional developers to gain better insight into the impact that the proposed features have on development.

What experiments have been done on these language features?

Most of the studies are descriptive studies without comparison data. There are also many in-vitro case studies, with a lack of studies done on professional projects developed independently. We found only one experiment, which used student participants.

3.6 Recommendations

We list below notable or interesting findings and add our recommendations for future work.

Meta-programming features seem the most prevalent to implement new features to potentially improve OOP design patterns implementations. The past prevalence of the AOP paradigm indicates that it had good potential for improving OOP design patterns. Perhaps the introduction of another layer of complexity (i.e., aspects and the popular join-point mechanism used by AspectJ) on top of the OOP rebuked developers from adopting the paradigm. While AOP popularity has faded in recent years, it may still be of some use as a comparison point for new methods or improvements.

Meta-programming in general offers the possibility to extend languages by adding new features without the need for external tools. While some features have been prevalent in mainstream languages, such as Reflection in Java and C#, many Meta-programming features are absent in mainstream languages. For example, Lisp-style macros, allowing the effective addition of new keywords within a language, are seldom seen in modern languages.

While we found few works that used Functional Programming to improve OOP design patterns, the popularity of functional programming appears to be on the rise. As more functional programming features are added to OOP languages, it would be interesting to study how we can use those features to improve OOP design patterns. There certainly seems to be a trend in recent years to try to fuse OOP and FP together. We believe the impact of this is worth studying.

Most existing research seems to favour specific design patterns. Observer, Visitor, and Decorator were studied much more often than others. Thus, we have more data on approaches to improve the implementation of these specific patterns. Some design patterns only appear in studies concerned with all 23 GoF patterns (most likely for the sake of completeness). It would be interesting to study the less prevalent patterns to understand what kind of approaches would specifically improve those. Each design pattern solves a recurring design problem, so the community should strive to study these problems and, perhaps ideally, find features to solve them.

For reproducibility and comparability with existing research, when evaluating or comparing improvement to OOP design patterns, it would be beneficial to include measures pertaining to maintainability and understandability. In particular, the CK measures [21] are particularly prevalent in this kind of research. These include depth of inheritance tree, lack of cohesion in operations, coupling between components, number of lines of code, and weighted operations per component. AOP-related measures might be interesting to use also, for comparison with the many papers published proposing improvement using AOP. AOP measures are mainly focused on cross-cutting concerns.

However, these measures do not cover every possible improvement. Some improvements are difficult to measure using quantitative data. It would be important to explore less prevalent measures, or survey developers' opinions on code complexity and reusability and perform qualitative reviews.

Most papers we found were descriptive studies. More empirical studies on real-world case studies and experiments using professional developers would yield a deeper understanding of the techniques, tools and features that can be used to improve OOP design patterns, and OOP in general.

3.7 Threats to Validity

In this section, we discuss potential threats to the validity of this study. While we tried to keep the study and its results as objective as possible, there are some threats that need to be addressed. We divided the threats into Internal, External, Construct, and Conclusion Validity.

3.7.1 Interval Validity

The manual sorting, filtering, and compilation steps of the review were done by the first author, which increases the consistency of the results. Yet, it also introduces a threat to reliability and trustworthiness, so we added a second reviewer to help with the snowballing step and add some redundancy and a second viewpoint.

3.7.2 External Validity

Our results come from papers from a very diverse set of publication venues. The papers are not concentrated in popular venues concerning programming languages and design patterns, such as TOPLAS and POPL. We attribute this to the fact that this particular subject, proposing language features to improve OOP design patterns implementation, is more of a niche subject and less popular in prominent publications, and pertains more to software engineering than programming languages. Most of the results in our query that were from major publication venues tended to be about measuring the impact of design patterns, which was not within the scope of this study (and already has a mapping study [6] discussed in our related studies section).

3.7.3 Construct Validity

We defined our methodology as precisely as possible to make our results reproducible. The dataset used for this study is available online[11]. We took some decisions that could affect the results of our study. The query used has many exclusion keywords to obtain more precise results. We tried queries with fewer exclusion keywords, but they did not add more relevant results which made the manual filtering more difficult. We believe any missing relevant results that were lost because of this decision would have been caught during the snowballing step.

The manual sorting, filtering, and compilation steps of the review were done by the first author, which increases the consistency of the results. Yet, it also introduces a threat to reliability and trustworthiness, so we added a second reviewer to help with the snowballing step and add some redundancy and a second viewpoint.

3.7.4 Conclusion Validity

We believe that the previous threats are acceptable and that different choices would not significantly alter the results of this mapping study, whose goal is to identify trends in research on the subject of language features to improve OOP design pattern implementations. As such, we limit our conclusions to that goal and propose recommendations based on our findings.

3.8 Conclusion

In this chapter, we presented a mapping study of the primary studies on language features to improve Object-Oriented Programming design pattern implementations. Our objective was to catalogue the language features that were used in the literature to improve OOP design patterns. We asked the four following research questions:

(1) RQ3.1: What language features have been suggested to improve design patterns implementations?

[11]https://www.ptidej.net/downloads/replications/ist22/

(2) RQ3.2: Which design patterns have the most associated language features suggestions?

(3) RQ3.3: What measures have been used to evaluate the impact of these language features on design patterns implementations?

(4) RQ3.4: What experiments have been done on these language features?

We devised and followed a methodology (which we describe in Section 3.3) that would yield the data needed to answer these questions and the main objective of the mapping study and create a catalogue of 18 language features claiming to improve design patterns implementations.

We performed a search query in three databases which yielded 874 papers, which we then assessed for quality and used for snowballing, resulting in a total of 34 primary studies. We then extracted relevant data from these 34 studies and discussed and catalogued our observations and their meaning.

We categorized the language features found in the papers into categories based on programming paradigms. We presented a summary of this map with every paper in our study, as well as a catalogue describing every feature extracted. We found that most of the research on the topic suggested approaches related to Meta-programming, with Aspect-oriented Programming often considered, although interest in it seems to have faded in recent years. The design patterns most often cited in papers on the topic were the Observer, Visitor, and Decorator. We also extracted every measure used in each paper into a table to find the most frequently used. We found that measures related to maintainability and understandability were most often used, with the Chidamber and Kemerer [21] measures being particularly prevalent.

Finally, we categorized every paper based on the type of study performed. Most papers were descriptive studies intended to introduce new ideas or approaches. There were also case studies, which were done in-vitro. We found only one experiment, done with student participants.

With these findings, we contribute a catalogue of 18 language features proposed in the literature to improve Object-Oriented design patterns. These features mostly aim to improve maintainability and understandability by reducing concern diffusion and code duplication.

Chapter 4

Exploring the Impact of Immutability on Object-oriented Software Development

4.1 Introduction

Among the language features presented in the last chapter, we chose to focus on one particular feature: immutability. As discussed in Chapter 2, research on immutability focuses on immutability enforcement and correctness. While developers' lore claims many advantages to writing immutable code, there are few studies that back this up.

These advantages include easier management of race conditions, better reasoning about code, and simplified testing and maintenance. Immutability eliminates many mutation-related bugs (e.g., when two objects have references to the same data and one object modifies that data, unexpectedly altering the behaviour of the other object).

However, immutability also imposes restrictions on the developers by removing their ability to modify objects after their creation. Thus, developers must use specific concepts, such as functional lenses[1] or monads[2], to achieve behaviours that would otherwise be simple

[1]https://hackage.haskell.org/package/lens
[2]https://hackage.haskell.org/package/base-4.16.1.0/docs/Control-Monad.html

to implement with mutable objects.

To the best of our knowledge, there exists no empirical study on the impact of immutability on object-oriented software development. Programming language designers introduce immutability based on (their) personal anecdotes and developers' lore.

Consequently, in this chapter, we propose the first empirical study on the impact of immutability on object-oriented software development. Through this chapter, we want to assess the advantages and disadvantages of using immutability in object-oriented programming languages.

We perform a multi-method exploratory empirical study on the impact of applying immutability to object-oriented development. We ask the following main research question: *What is the impact of immutability on object-oriented development?*.

To answer this question, we divide a set of 67 third-year undergraduate software engineering students, into two groups. We refer to these groups as the treatment group and the control group. The groups of participants are also divided into teams of 4 or 5 students. Each team is tasked with developing the same program using OOP. The teams in the treatment group must use immutable objects while the teams in the control group use mutable objects.

We perform a multi-method analysis of quantitative and qualitative aspects of the resulting programs. For the quantitative analysis, we collect measures from the resulting programs. We use these measures to compare factually the programs developed with mutable or immutable objects.

For the qualitative analysis, we collect subjective assessments with a survey of the participants on their experience with the project. We ask each participant to answer the survey and we compare the answers of the treatment and control groups. We collect 12 measures related to the workload, difficulty of implementing the specifications, and the complexity of the resulting program.

Despite some differences between the measures of the treatment and control groups, we do not observe a significant negative impact on the treatment group. On the contrary, we observe a slightly positive impact on function size and a general trend for participants in the treatment group to find the workload, difficulty, and complexity lower than in the control group.

Thus, we show that the use of immutability in OOP languages does not have a negative impact on quantitative and qualitative measures and may have a positive impact on developers' performance and on code granularity and readability.

The rest of this chapter is as follows: Section 4.2 situates the study with regards to existing literature and presents a motivating example of implementing immutability with the Command design pattern [33]. Section 4.3 presents our research questions and method to assess the impact of immutability. Sections 4.4, 4.5, and 4.6 discuss the results and findings of the study. Section 4.7 discusses potential threats to the validity of the study. Finally, Section 4.8 summarizes and concludes the chapter.

This chapter is the basis of a paper sent to the Empirical Software Engineering journal. It is currently under review.

4.2 Background

4.2.1 This Study

The majority of research works on immutability focus on enforcement. While usability features for immutability have been introduced in programming languages, little research exists on the impact of these features. We could not find any paper discussing these features in the context of OOP.

The impact of immutability on code has also not been researched in depth. While we listed the advantages of using immutability in Chapter 2, most of the arguments are authoritative and without supporting empirical data.

In this study, we investigate the impact of adopting immutability in object-oriented development and retroactively justify the research on language support for immutability. We do not limit our study to any particular OOP language. We consider immutability enforced using the set of rules suggested by Bloch [13]:

(1) **"Don't provide methods that modify the object's state."** No method can

change the attribute of an object once it has been constructed to ensure object immutability.

(2) **“Ensure that the class can’t be extended.”** Future users of the code cannot break immutability by subclassing and adding mutating methods.

(3) **“Make all fields final.”** Attributes cannot be mutated after object creation.

(4) **“Make all fields private.”** Attributes cannot be mutated by clients of the object, which is usually a good practice even without considering immutability.

(5) **“Ensure exclusive access to any mutable components.”** Mutable variables are allowed when they are narrowly scoped and protected from exterior influence. For example, a method could have a mutable variable (e.g., a loop index) as long as it does not expose it to clients.

These rules, when followed by developers, ensure *transitive immutability* in any object-oriented language. We evaluate the impact of following these rules in an actual software development project and find trends to perform further research on the subject.

4.2.2 Motivating Example

To illustrate the impact of immutability on code, we use an example based on the Command design pattern and a “Undo” functionality. This example was taught in the software engineering class in which this study took place.

Figure 11 shows the class diagram of a subset of a mutable implementation: a mutable database (e.g., a list or array) on which to execute commands. One such command involves updating some record in the database, implemented by the `UpdateCommand` class, which implements the `ICommand` interface with the `Execute()` and `Undo()` methods. The commands can execute and undo their changes, as the database has no such functionality.

`CommandInvoker` is the controller class executing commands and tracking their exeuction to undo them in the correct order.

In contrast, Figure 12 shows the class diagram of a subset of an immutable database. The database creates a copy of itself for every change, never modifying existing copies. To support undo, we no longer require the commands to implement `Undo()`. Instead, the `CommandInvoker` stores previous versions of the database upon calling `Execute` and can simply go back to previous versions when `Undo()` is called.

The immutable implementation requires fewer methods implemented throughout the classes, which could have a positive impact on maintainability and evolution. [3]

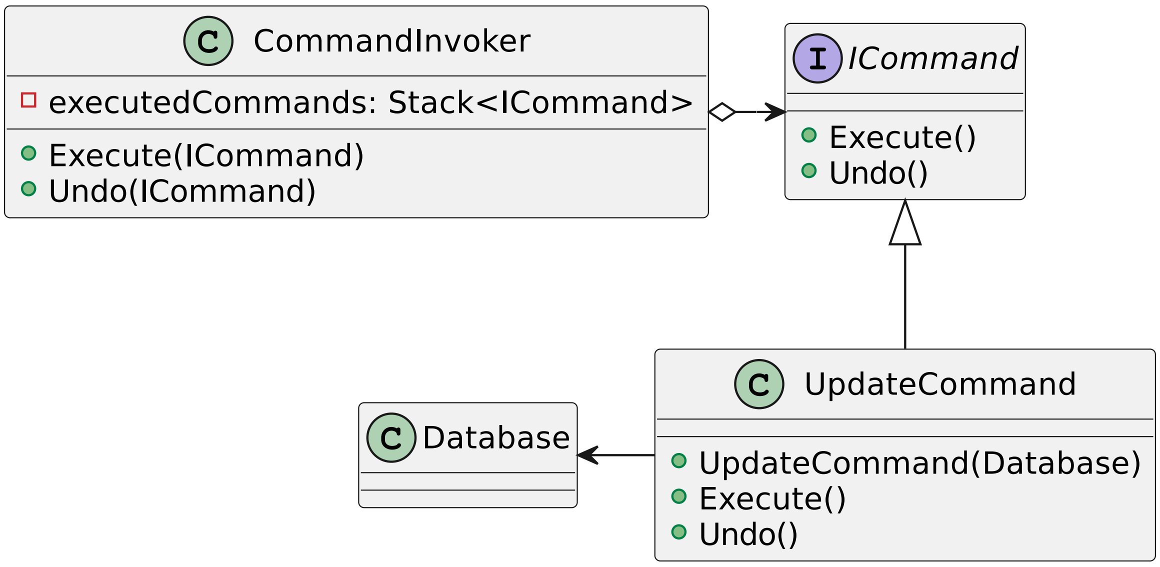

Figure 11: Mutable representation of a Command invoker

However, with immutable objects, programs requiring mutations must implement special workarounds or patterns. For example, complex nested data structures (i.e., objects containing other objects) require the creation of new objects for the child and the parent up to the top of the hierarchy when changing the child. In functional languages, the design pattern Lens[4] exists to implement this requirement.

[3]The database would be implemented differently in both implementations and there could be some performance concerns with having an immutable database. In this example, the database is a placeholder for any data structure that can change over time. Using append-only data structures, it is possible to have memory and performance-efficient immutable data structures, as in Haskell and Clojure. Datomic is an example of a Clojure implementation of a fully immutable database, cf. https://www.datomic.com/.

[4]https://hackage.haskell.org/package/lens

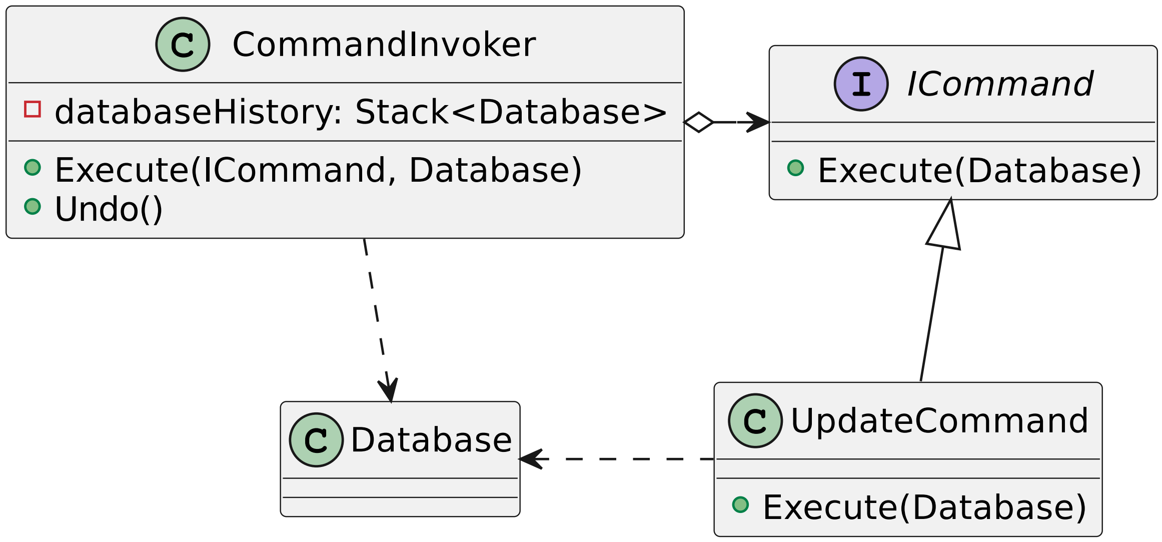

Figure 12: Immutable representation of a Command invoker

4.2.3 Related Studies

Immutability in OOP development has been researched since the very inception of OOP [42, 57, 73, 86].

In 2010, McCaffrey and Bonar [66] investigated the use of functional programming for writing software tests. Professional test engineers were given training in developing F# tests and given a survey following the training. While the study was informal and limited in scope, the results indicated that the engineers saw value in using functional programming features to design tests.

In 2019, Eyolfson and Lam [29] published an empirical study investigating how C++ developers use immutability. They evaluated seven open-source projects and estimated the prevalence of immutability using their own static-analysis tool and `const` annotations on methods. They found few fully immutable classes, but many immutable methods (between 46% and 53%, depending on inclusion criteria).

In 2020, Bugayenko and al. [16] performed an empirical study to measure the impact of immutability on Java classes. They examined classes in 240 GitHub repositories and classified each of them as immutable or mutable. They calculated the number of non-comment lines of code and found that immutable classes tended to be smaller, with fewer methods and attributes.

Although previous work tends to support developers' lore regarding the benefits of immutability, no previous work performed a multi-method exploratory study on the impact of immutability on OOP development.

4.3 Method

We perform an experiment with a class of B.Sc. software-engineering students taking a course on advanced OOP, focusing on the 23 design patterns by Gamma and al. [33] (GoF patterns). The students all had a similar academic background, with experience mainly in object-oriented programming using Java and C#. The course in which this experiment took place exposed them to advanced concepts in C# OOP, such as the application of design patterns and unit testing techniques.

The main focus of the class was the classical implementation of the various design patterns and other OOP concepts, with about a fifth (9 hours out of 45) of the course dedicated to immutability, including immutable variants of the various patterns. No particular emphasis was put on immutability. All students were taught the same material. We asked the students at the beginning of the semester and none of them reported having any prior experience with immutability.

Participation in the experiment was voluntary. We gave an incentive in the form of extra credits (10% of the total grade of the assignment), also to compensate for the additional constraints the study put on the students' assignments. Out of 84 students in the class, 67 chose to participate, in teams of 4 to 5 students. We randomly divided the participants into two groups. The control group used classical OOP (including mutation), while the treatment group used only immutable objects. The treatment group was required to follow the set of rules suggested by Bloch [13] (see Section 4.2.1). Table 10 summarises information on the participants and their teams[5].

The object of the experiment, i.e., the project, was the development of a Sudoku solver,

[5]There are more participants in the treatment group because one team decided to opt in after the project had begun. We randomly assigned them to a group, which was by chance the treatment group.

with a simple user interface to visualize and solve Sudoku puzzles. The interface had to support various notation styles (corner notation, center notation, colouring) and basic functionalities (e.g., undo–redo). The project was partially based on existing software for solving Sudoku, such as Cracking the Cryptic's SodukuPad[6].

This project was in the context of a software engineering class and some additional requirements had to be added with regard to the class curriculum. We divided the project into two phases. In the first phase, the students developed the core of the solver using five SOLID principles [64] and three GRASP design patterns [56]. In the second phase, we required additional features (e.g., the colouring and undo-redo features) and the use of three GoF patterns of their choice. These requirements were identical for both treatment and control groups and thus had little influence on the results of the study.

Most teams write their programs in C#, an object-oriented language without support for immutability. Table 8 summarizes the programs created by the participants. We collect various measures on the programs and participants, which we divide into quantitative (objective) and qualitative (subjective) measures.

We based the grading of the project on two factors: the correctness of the developed program and the proper use and implementation of the principles and patterns. The students had to demonstrate that their program was fully functional, with every feature working correctly. They also had to write a report describing how they interpreted and implemented the SOLID/GRASP principles (in Phase 1) and the GoF design patterns (in Phase 2).

To answer our main research question, we use these measures to answer these three secondary questions:

(1) RQ4.1: What insight can we draw from the quantitative data collected from the developed software?

(2) RQ4.2: What insight can we draw from the qualitative data collected from the participants?

(3) RQ4.3: What recurring concerns or comments were left by the participants?

[6]https://app.crackingthecryptic.com/

Team #	Group	Language	Grade
1	Immutable	TypeScript	100
2	Immutable	C#	100
3	Immutable	C#	93
4	Immutable	C#	100
6	Immutable	C#	100
7	Immutable	C#	96
9	Immutable	C#	67
11	Immutable	C#	100
5	Mutable	Unity C#	90
8	Mutable	C#	100
10	Mutable	C#	88
12	Mutable	C#	100
13	Mutable	Java	85
14	Mutable	C#	56
Average (All)			91.07
Average (Mutable)			86.5
Average (Immutable)			94.5

Table 8: Programs Summary

4.4 Quantitative Measures

In this section, we present the method used for data collection of the quantitative measures and the results of our statistical analysis. We then discuss these results and answer our first research question.

4.4.1 Data Collection

To obtain quantitative, objective data for each program, we need to select a suite of measures to calculate. Since this is an exploratory study, we want to be as broad as possible so as to not miss potential trends for future research. However, this also means that we need to use many measures, which will make it unlikely that any single measure will be statistically significant [74]. We also need an automated tool, because manual measurement is prone to error and subjectivity. The tool must support the programming languages used by the teams (C#, Java and TypeScript).

Consequently, we chose SciTools Understand[7] to collect the quantitative measures on

[7] https://www.scitools.com/features

the programs. Understand allows the calculation of a variety of object-oriented measures in a large number of programming languages. We chose the product measures described recently by Majumder et al. [63] in their analysis of process and product measures. A description of each individual measure is available on SciTools Website[8]. These correspond in most parts to the measures suggested by Chidamber and Kemerer in [21].

While every program included unit tests as part of their code, we excluded all tests from our analysis because unit tests seem to have different quality characteristics to application code [24]. For example, while size-related measures (e.g., number of lines of code, number of instance attributes, etc.) should be minimized for application code; it may be different for unit tests (larger unit testing suites may have more tests, cover more cases, have better readability, etc.).

Table 9 summarizes the measures collected on each program[9]. For each measure, we calculated the average for all programs and the average for the programs of each group (immutable and mutable). Whenever a measure name is followed by the tags `Avg` or `Max` in parenthesis, the value of this measure was aggregated for every class in the program using either the average across classes or the maximum value of any class. We summarise the comparison of the results between the two groups in the following.

We first compare measures related to the average class complexity. The measures in this category are: Average Cyclomatic Complexity (AvgCycl); Average Modified Cyclomatic Complexity (AvgCycl*); Average Strict Cyclomatic Complexity (AvgCyclStrict); and, Average Essential Complexity (AvgEss). We also calculated the sum and maximum values of these measures (e.g., SumCycl, MaxCycl, etc.) for every class.

The Cyclomatic Complexity is calculated using McCabe's formula [65], which counts the number of branching paths in a method (e.g., if statements). Modified Cyclomatic Complexity is a variant (sometimes referred to as CCN3) in which multi-decision statements (i.e., switch statements) are considered a single path. Strict Cyclomatic Complexity counts

[8]https://support.scitools.com/support/solutions/articles/70000582223-what-metrics-does-understand-have-

[9]This table is a summary of the measures we collected. The complete dataset is available in our replication package.

ANDs and ORs in a conditional statement as additional paths. Essential Complexity reduces the control flow graph by considering any statement with a single point of entry and a single point of exit as a single path.

There is not much difference between the immutable and mutable groups regarding complexity. The immutable group has higher normal (Cycl) and strict Cyclomatic Complexities (CyclStrict), but similar or lower Modified (Cycl*) and Essential Complexities (Ess).

We then consider measures related to the size of methods (and functions). These measures are: Average Number of Lines in Functions (FnLines); Average Number of Lines of Code in Functions (FnLinesCode); and, Average Number of Lines of Comments in Functions (FnLinesComm).

We observe that immutable programs tend to have on average slightly shorter methods/functions[10] (FnLines) than mutable programs, but the longest ones were also found in immutable programs. Immutable programs tend to have a lower count of comment lines (FnLinesComm).

We also consider measures related to class interactions, such as coupling and inheritance. These measures are: Number of Base Classes (NbClassBase); Coupling Between Objects (CBO); Modified Coupling Between Objects (CBO*); and, Depth of Inheritance Tree (DIT). Modified Coupling Between Objects is a variant of CBO that does not count calls to library classes (e.g., the .NET Framework classes in C#).

Mutable programs have a slightly lower overall coupling (CBO), while immutable programs have a comparatively lower modified coupling (CBO*): immutable programs have a lower coupling with user-created classes, but a higher coupling to library classes. Immutable programs have somewhat shallower inheritance trees (DIT).

We also consider measures related to methods and attributes of classes. These measures are: Number of Class Methods (NbClassMethods); Number of Class Attributes (NbClassAttr); Number of Instance Methods (NbInstMethods); Number of Instance Attributes (NbInstAttr); Number of Methods (NbMethods); Number of Methods including inherited (NbMethodsAll); Number of Protected Methods (NbMethodsProt); and, Number of Public

[10]We use “methods” in the following to denote both methods and functions for the sake of simplicity.

Methods (NbMethodsPub).

While most of these measures are generally similar between the groups, there is a higher number of static methods (NbClassMethods) in mutable programs, while immutable ones have a higher number of static attributes (NbClassAttr). On the contrary, there are more instance methods (NbInstMethods) in immutable programs and more instance attributes (NbInstAttr) in mutable ones. However, when combining all types of methods (NbMethod, NbMethodsAll), immutable programs have a higher number of methods than mutable ones.

We also consider measures related to class cohesion and scope nesting. These measures are: Maximum Nesting Level (MaxNesting); Percentage of Lack of Cohesion in Methods (LCOM%); and, Percentage of Modified Lack of Cohesion in Methods (LCOM*%).

The Percentage of Lack of Cohesion in Methods is calculated as the ratio between the total number of attributes of a class and their usage in each method of that class. The modified variant excludes getters and setters (i.e., methods interacting with a single attribute). The measure values are very similar between the two groups, with mutable programs having a slightly higher normal cohesion (LCOM%) than immutable ones, but immutable ones having a slightly higher modified cohesion (LCOM*%). Mutable programs have slightly higher nesting (MaxNesting) than immutable ones.

We finally consider measures related to the programs as wholes. These measures are: Number of Lines (LoC); Number of Blank Lines of Code (LoCBlank); Number of Lines of Code (LocCode); Number of Lines containing Declarations (LoCDecl); Number of Lines containing Executable Statements (LoCExec); Number of Lines containing Comments (LoCComm); Number of Semicolons (NbSemicolons); Number of Statements (NbStmt); Number of Declarations (NbDecl); Number of Executable Statements (NbExe); and, Ratio of Comment to Code (RatioComm).

Sizes are slightly lower for immutable programs in general. Immutable programs also have fewer lines of code (LoC) and significantly fewer comments (LoCComm). Yet, the ratio of lines of comments to lines of code (RatioComm) is similar between the two groups.

Measure	Avg	Avg (I)	Avg (M)
AvgCycl (Avg)	1.4038	1.4624	1.3257
AvgCycl (Max)	5.2143	5.5000	4.8333
AvgCycl* (Avg)	1.3454	1.3864	1.2908
AvgCycl* (Max)	4.4286	4.3750	4.5000
AvgCyclStrict (Avg)	1.6905	1.9074	1.4014
AvgCyclStrict (Max)	7.5000	9.0000	5.5000
AvgEss (Avg)	0.9466	0.9699	0.9156
AvgEss (Max)	1.7143	1.5000	2.0000
NbClassBase (Avg)	1.2545	1.2736	1.2290
NbClassBase (Max)	2.0000	2.0000	2.0000
CBO (Avg)	6.9152	7.2760	6.4943
CBO (Max)	27.3846	26.5714	28.3333
CBO* (Avg)	3.0608	2.5215	3.6899
CBO* (Max)	12.1538	9.5714	15.1667
NOC (Avg)	0.1297	0.1292	0.1303
NOC (Max)	2.0714	2.1250	2.0000
NbClassMethods (Avg)	0.4382	0.4003	0.4886
NbClassMethods (Max)	4.7143	4.8750	4.5000
NbClassAttr (Avg)	0.3465	0.4195	0.2492
NbClassAttr (Max)	2.2143	2.5000	1.8333
NbInstMethods (Avg)	6.1109	6.2317	5.9499
NbInstMethods (Max)	25.5714	27.1250	23.5000
NbInstAttr (Avg)	1.6174	1.3972	1.9111
NbInstAttr (Max)	7.1429	7.3750	6.8333
NbMethods (Avg)	6.5654	6.6605	6.4385
NbMethods (Max)	27.0714	28.7500	24.8333
NbMethodsAll (Avg)	94.5030	112.6830	70.2629
NbMethodsAll (Max)	414.9286	513.2500	283.8333
NbMethodsProt (Avg)	0.1027	0.0936	0.1148
NbMethodsProt (Max)	1.7143	1.5000	2.0000
NbMethodsPub (Avg)	5.2550	5.1922	5.3388
NbMethodsPub (Max)	22.0714	24.0000	19.5000
SumEss (Avg)	6.4359	6.4865	6.3685
SumEss (Max)	31.2143	31.0000	31.5000
MaxCycl (Avg)	2.8764	2.8819	2.8689
MaxCycl (Max)	10.8571	11.1250	10.5000
MaxCycl* (Avg)	2.6594	2.6064	2.7301
MaxCycl* (Max)	9.5714	9.6250	9.5000
MaxCyclStrict (Avg)	3.4061	3.5976	3.1826
MaxCyclStrict (Max)	14.9231	16.4286	13.1667
MaxEss (Avg)	1.3097	1.3222	1.2931
MaxEss (Max)	6.0714	5.7500	6.5000
DIT (Avg)	0.3526	0.2785	0.4513
DIT (Max)	1.3571	1.2500	1.5000
MaxNesting (Avg)	1.0591	0.9046	1.2652
MaxNesting (Max)	3.7857	3.6250	4.0000
LCOM% (Avg)	31.3982	31.4723	31.3118
LCOM% (Max)	84.0769	84.8571	83.1667
LCOM*% (Avg)	27.5726	26.4119	28.9269
LCOM*% (Max)	79.6923	78.1429	81.5000
SumCycl (Avg)	10.4729	10.2071	10.8273
SumCycl (Max)	54.4286	51.1250	58.8333
SumCycl* (Avg)	10.0892	9.7026	10.6047
SumCycl* (Max)	53.3571	50.6250	57.0000
SumCyclStrict (Avg)	11.4045	11.1294	11.7714
SumCyclStrict (Max)	59.4286	55.0000	65.3333
FnLines (Avg)	6.4768	6.0209	7.0846
FnLines (Max)	76.5714	79.5000	72.6667
FnLinesCode (Avg)	5.8576	5.6029	6.1973
FnLinesCode (Max)	62.2857	65.3750	58.1667
FnLinesComm (Avg)	0.2342	0.1670	0.3237
FnLinesComm (Max)	10.1429	11.3750	8.5000
LoC	2029.2858	1945.7500	2140.6667
LoCBlank	265.8571	226.0000	319.0000
LoCCode	1555.6428	1536.7500	1580.8334
LoCDecl	469.3077	448.1429	494.0000
LoCExec	583.6923	606.1429	557.5000
LoCComm	167.0000	118.7500	231.3333
NbSemicolons	672.0769	675.0000	668.6667
NbStmt	958.4286	943.7500	978.0000
NbDecl	483.3571	465.3750	507.3333
NbExe	521.5000	527.8750	513.0000
RatioComm	0.1044	0.1078	0.0998

Table 9: Program measures

4.4.2 Statistical Tests

To get more insight into the impact of the groups on the quantitative measures, we perform some further statistical analyses. We do not perform any hypothesis testing, as the goal of this work is to explore the general impact of immutability on object-oriented programming and make suggestions for future research. This approach is incompatible with precise hypothesis testing and statistical analysis [74].

We establish the independent variable as being part of the treatment or control group. Projects in the treatment group are assigned the value 1, while the control group receives the value 0. We then calculate whether the independent variable has an impact on the various measures.

To use the proper statistical test, we first establish whether each measure follows a normal distribution using the Shapiro-Wilk test [82]. For most measures, we cannot verify the normality hypothesis, implying that the data is not normally distributed.

We choose the *Mann-Whitney U* test, which is a non-parametric test on two sets of ordinal variables to determine if they were sampled from the same populations. If they are not, we can assume that the treatment and control populations are different and have influenced the measures.

We can then use *Cliff's delta* to obtain the effect size of the independent variable on the measures. Because our data is not normally distributed, we cannot get a clear grasp of the effect size by using average or median deltas.

Because of the number of measures included in this study, we cannot compare the p-value to the typical threshold of 0.05 [74]. While this is not the goal of our approach, after Bonferroni correction, the p-value threshold for our data would be 0.0006.

This methodology is similar to the one used by Salvaneschi [79] in their paper measuring the impact of Reactive Programming on program comprehension. Their study has a similar structure to ours and their data is also non-normally distributed, which is why we opted to adopt a similar statistical analysis methodology.

Table 11 summarizes the results of each test. Our objective here is not to find a significant impact on any individual measure. , which none of the results in Table 11 are reaching. Instead, we take the p-value for its literal meaning: the probability that the measured effect is a coincidence. We can identify interesting results in the data and set up future research focusing on these particular measures.

The Cliff's delta column in Table 11 measures the effect size and direction. An effect size close to zero (i.e., with an absolute value lower than 0.15) is usually considered insignificant.

We notice that the sizes of methods seem lower in immutable programs. The average number of lines (FnLines) is less than in mutable programs, especially when considering lines of comment (FnLinesComm). It may be the case that immutable programs have fewer lines of comment per method/function on average than mutable ones. Although likely not significant, we observe a similar relationship with overall program size, where immutable programs have a lower number of lines of code (LoC) and comments (LoCComm).

There seems to be a relationship between immutability and the number of class (or static) attributes (NbClassAttr). Immutable programs in our study have more class attributes than mutable programs. They also have more methods (NbMethods, NbMethodsAll). Put together with the observations about the sizes of methods, it appears immutable programs are structured as many smaller methods instead of fewer larger ones in mutable programs.

Interestingly, we do not observe any relationship related to complexity, despite the observations on the averages in the previous sub-section. Immutable programs in our study have slightly lower modified cyclomatic complexity (AvgCycl*, MaxCycl*) than mutable ones.

4.4.3 Research Question 4.1

Our first research question is *RQ4.1: What insight can we draw from the quantitative data collected from the developed software?*

We collected measures data from the 14 programs developed by the participants of the project. The results presented in Section 4.4 show some differences between the treatment and control groups. We discuss these differences in this subsection.

We observed that the Cyclomatic Complexity (Cycl) was slightly higher for immutable programs but that the Essential Complexity (Ess) and the Modified Cyclomatic Complexity (Cycl*) were similar in both groups. We attribute these results to the prevalence of switch statements in functional-style code pattern matching. Pattern matching is a feature supported by C# 8.0, which we taught during the course of this study and illustrated on some design patterns. For example, we showed how it can be used to traverse composite objects (Composite pattern) without requiring the use of the Visitor pattern. Modified and Essential Complexities do not count switch statements, which explains why these measures are not significantly higher in the immutable programs. **We conclude that immutable and mutable programs are of similar complexity.**

We also observed differences in the sizes and numbers of methods in the programs. Immutable programs had more methods, but these were shorter. Without any state to modify, immutable methods are simpler, mathematical expressions. With shorter methods, our results showed that immutable programs had to implement more methods, especially when counting inherited methods (NbMethodAll). Thus, immutable programs followed recommended practices in software development to have more numerous smaller methods over fewer large methods. Large methods are considered a code smell [81, 90]. These results are consistent with the findings of Bugayenko and al. [16] in their empirical study on the impact of immutability on Java classes. **We conclude that immutable programs may have shorter, more granular methods, following software development recommended practices.**

Immutable programs had a higher number of class attributes (or static fields) possibly because attributes were immutable (by definition) and, thus, with less negative impact than in mutable programs[11]. Additionally, all immutable programs used a single, mutable class attribute to store the state of the programs. This mutable class attribute was necessary because languages like C# and Java do not allow the creation of a fully immutable program with a graphical user interface. Thus, each immutable program had at least one class attribute, while the mutable programs could have none. **We conclude that there is no difference between the number of attributes of immutable and mutable programs.**

There are also some differences in Coupling Between Objects values. The mutable programs had overall lower coupling (CBO) while immutable programs had lower coupling between user-created objects (CBO*). Coupling to library classes has a lower negative effect on maintainability than coupling to user-created classes because library classes usually offer well-tested, comprehensible classes and methods, which are not maintained by the developers themselves. Furthermore, some mainstream object-oriented libraries are written in the immutable style (e.g., LINQ in C#) and were used by the treatment group, which

[11]Using class attributes, or global attributes, creates a common coupling, with possibly unforeseen side effects.

explains the higher number of calls to library classes. **We conclude that immutable programs are built on firmer grounds because they rely more on well-tested libraries.**

Immutable programs were smaller in number of lines of code (LoC) in general, although the difference with mutable programs was small. Two immutable programs were outliers with large sizes: Program 4 had over 4,000 lines of code while program 6 was below 700. Yet, both programs received a grade of 100%. Their difference in size is due to teams' particular development styles. Team 4 favoured a granular style with many classes, few methods per class, and their own exception types. Team 6 used a "monolithic" approach, with few classes requiring fewer object interactions. **We conclude that program sizes do not reflect their styles or quality.**

We observed a large difference between the number of lines of comments between groups, with immutable programs having significantly fewer comments than mutable ones. With the fact that immutable programs had more and smaller methods, we argue that immutable programs may have needed fewer comments to be understood and read than their mutable counterparts and, thus, that they had more readable code. **We conclude that functions in immutable programs may be easier to understand than their mutable counterparts. We suggest further research on the impact of immutability on program comprehension.**

We could not observe any significant negative impact of immutability. Our results indicate that the impact of immutability may be positive, specifically due to shorter, more readable methods. We suggest investigating the impact of immutability on code granularity and understandability in future research.

4.5 Qualitative Measures

In this section, we present the method used for data collection of the qualitative measures and the results of our statistical analysis. We then discuss these results and answer our second and third research questions.

4.5.1 Data Collection

The students who participated in the study were invited, at the end of the semester, to complete a survey to assess their experience with the project. With the survey, we asked each participant:

(1) To auto-evaluate their expertise in OOP (1 to 5).

(2) To auto-evaluate their participation in the project within their team (1 to 5).

(3) For their impression on the workload, difficulty and complexity of the resulting program for each of the following: Phase 1 implementation, Phase 2 implementation, SOLID/GRASP principles implementations, GoF patterns implementations (1 to 5).

(4) Whether they would consider using immutability for future projects (yes or no).

(5) To leave some open-ended comments.

The goal of this survey was to establish the relationships between participants in each group and their personal assessment of the project. If immutability has more disadvantages than advantages, we expected the treatment group to rate the workload, difficulty, and–or complexity higher than the control group.

Table 12 summarizes the answers to the survey given by the participants by providing the averages of the 50 participants who answered the survey. The full dataset, including the translated survey questions, is available in the replication package[12].

We observe that the averages for every single category are slightly lower for the treatment group than the control group: the treatment group felt that the workload, difficulty, and complexity of their programs were lower than what the control group felt.

The participants in general had a high self-evaluation of their expertise in OOP, with the treatment group having a somewhat higher self-evaluation. Lastly, 58% of the participants (both groups combined) answered that they would consider using immutability in the future. Interestingly, more participants from the control group (65%) answered positively than from

[12] https://www.ptidej.net/downloads/replications/emse22b/

the immutable group (53%). Perhaps the treatment group was slightly put off by the initial learning curve of introducing immutability.

Participants	67
Teams	14
Participants (mutable group)	29
Teams (mutable group)	6
Participants (immutable group)	37
Teams (immutable group)	8
Survey respondents	50
Survey respondents (mutable group)	20
Survey respondents (immutable group)	30

Table 10: Measures on participants

4.5.2 Statistical Tests

We used the same approach for testing the impact of the treatment on the survey answers as for the quantitative measures in Section 4.4.2. We are interested in the answers in Section 3 of the survey (the workload, difficulty, and complexity). We tested for normality using the Shapiro-Wilk test and found that every test resulted in a p-value lower than 0.0002, which shows non-normal distributions.

We assigned participants in the control group an independent variable value of 0, while the treatment group was assigned 1. Table 13 shows the results of the *Mann Whitney U* tests between the independent variable and each survey measure.

These results show that every relationship for every single answer is negative: participants within the treatment group consistently found every part of the assignment shorter, easier, and their resulting program less complex. As the p-values are not below the usual threshold ($\alpha = 0.004$, with Bonferroni adjustment), the individual measures are not significant. However, every effect size is positive, which hints at the immutable group having an easier time (which is supported by the averages computed in Section 4.4).

Measure	P-Value	Cliff's Delta	Measure	P-Value	Cliff's Delta
AvgCycl (Avg)	0.8465	-0.0833	MaxEss (Avg)	0.8465	-0.0833
AvgCycl (Max)	1.0000	-0.0208	MaxEss (Max)	0.3972	0.2917
AvgCycl* (Avg)	0.5613	0.2083	MaxNesting (Avg)	0.1752	0.4583
AvgCycl* (Max)	0.7399	0.1250	MaxNesting (Max)	1.0000	0.0208
AvgCyclStrict (Avg)	0.8972	-0.0625	NOC (Avg)	0.9482	0.0417
AvgCyclStrict (Max)	0.8902	-0.0625	NOC (Max)	0.8950	0.0625
AvgEss (Avg)	0.8401	-0.0833	NbClassAttr (Avg)	0.0926	-0.5625
AvgEss (Max)	0.6637	0.1458	NbClassAttr (Max)	0.0789	-0.5625
CBO (Avg)	0.5203	-0.2381	NbClassBase (Avg)	0.6510	-0.1667
CBO (Max)	0.7172	0.1429	NbClassBase (Max)	1.0000	0.0000
CBO* (Avg)	0.8303	0.0952	NbClassMethods (Avg)	1.0000	0.0000
CBO* (Max)	0.3848	0.3095	NbClassMethods (Max)	1.0000	0.0208
DIT (Avg)	0.7466	0.1250	NbDecl	0.4772	0.2500
DIT (Max)	0.5813	0.1875	NbExe	0.7469	0.1250
FnLines (Avg)	0.3329	0.3333	NbInstAttr (Avg)	0.8465	-0.0833
FnLines (Max)	0.9485	0.0417	NbInstAttr (Max)	1.0000	-0.0208
FnLinesCode (Avg)	0.7469	0.1250	NbInstMethods (Avg)	0.6514	-0.1667
FnLinesCode (Max)	0.8463	0.0833	NbInstMethods (Max)	0.5181	0.2292
FnLinesComm (Avg)	0.0612	0.6250	NbMethods (Avg)	0.8465	-0.0833
FnLinesComm (Max)	0.4772	0.2500	NbMethods (Max)	0.8465	0.0833
LCOM*% (Avg)	0.9431	0.0476	NbMethodsAll (Avg)	0.8465	-0.0833
LCOM*% (Max)	0.7730	0.1190	NbMethodsAll (Max)	0.4777	-0.2500
LCOM% (Avg)	0.9431	-0.0476	NbMethodsProtect (Avg)	0.9450	0.0417
LCOM% (Max)	0.8296	-0.0952	NbMethodsProtect (Max)	0.9448	0.0417
LoC	0.5613	0.2083	NbMethodsPublic (Avg)	0.9485	0.0417
LoCBlank	0.1962	0.4375	NbMethodsPublic (Max)	0.9483	0.0417
LoCCode	0.7469	0.1250	NbSemicolons	0.7210	0.1429
LoCComm	0.4777	0.2500	NbStmt	0.4777	0.2500
LoCDecl	0.7210	0.1429	RatioComm	0.8465	0.0833
LoCExec	1.0000	0.0000	SumCycl (Avg)	1.0000	0.0000
MaxCycl (Avg)	0.7960	0.1042	SumCycl (Max)	0.3656	0.3125
MaxCycl (Max)	0.2369	0.3958	SumCycl* (Avg)	0.9485	0.0417
MaxCycl* (Avg)	0.4381	0.2708	SumCycl* (Max)	0.4772	0.2500
MaxCycl* (Max)	0.3254	0.3333	SumCyclStrict (Avg)	1.0000	0.0000
MaxCyclStrict (Avg)	1.0000	-0.0000	SumCyclStrict (Max)	0.2195	0.4167
MaxCyclStrict (Max)	0.7188	0.1429	SumEss (Avg)	0.6514	-0.1667
MaxEss (Avg)	0.8465	-0.0833	SumEss (Max)	0.6982	0.1458

Table 11: Mann-Whitney U test results between groups and product measures, no corrections done for multiple hypothesis tests

4.5.3 Research Question 4.2

Our second research question is *RQ4.2: What insight can we draw from the qualitative data collected from the participants?*

The survey asked participants for their subjective assessment of the workload, difficulty, and complexity of the project. It is similar to the Nasa Task Load Index (NASA-TLX) [46], adapted to our specific needs. The translated version of the survey is available in our replication package[12].

Table 13 shows that every single survey answer follows the same trend: the values for the treatment group are lower. While high p-values indicate that any difference between

measure	Average (All)	Average (Mutable)	Average (Immutable)
Expertise (self-evaluation)	3.78	3.55	3.93
Contribution (self-evaluation)	3.92	3.8	4
Workload (Phase 1)	3.14	3.25	3.07
Workload (SOLID/GRASP)	3.2	3.3	3.13
Workload (Phase 2)	2.68	2.7	2.67
Workload (GoF)	2.82	2.95	2.73
Difficulty (Phase 1)	3.06	3.3	2.9
Difficulty (SOLID)	3.18	3.3	3.1
Difficulty (Phase 2)	2.64	2.85	2.5
Difficulty (GoF)	2.84	2.95	2.77
Complexity (Phase 1)	3.16	3.3	3.07
Complexity (SOLID/GRASP)	3.04	3.15	2.97
Complexity (Phase 2)	2.96	3.15	2.83
Complexity (GoF)	2.88	3.15	2.7
Would use immutability?	58%	65%	53.33%

Table 12: Survey answers averages

the treatment and control groups may be coincidental, it showed that, for each phase and criteria, the participants in the treatment group found the project easier: the workload lower, the difficulty lower, and their resulting programs less complex.

We argue that this common trend hints at a relationship between being part of the treatment group and workload, difficulty, and complexity. This relationship is further evidenced by the students in the treatment group having overall higher grades than the other students as shown in Table 8.

We believe further research is needed on the subject of development effort when comparing immutable and classical object-oriented development. In particular, further studies could attempt to measure the workload in terms of time spent on each task.

Participants in the treatment group consistently found the workload lower, the difficulty easier and their resulting programs less complex than participants in the control group. This observation indicates that the subjective impact of immutability may be positive, rather than negative. We suggest further experiments measuring the workload of software development for immutable software.

measure	P-Value	Cliff's Delta
Workload (Phase 1)	0.3065	0.16
Workload (SOLID/GRASP)	0.2708	0.17
Workload (Phase 2)	0.8242	0.04
Workload (GoF)	0.4589	0.12
Difficulty (Phase 1)	0.1386	0.24
Difficulty (SOLID)	0.4891	0.11
Difficulty (Phase 2)	0.3112	0.17
Difficulty (GoF)	0.5834	0.09
Complexity (Phase 1)	0.5137	0.11
Complexity (SOLID/GRASP)	0.5509	0.1
Complexity (Phase 2)	0.3145	0.16
Complexity (GoF)	0.1571	0.23

Table 13: Mann-Whitney U test results between groups and survey answers

4.5.4 Research Question 4.3

Our third research question is *RQ4.3: What recurring concerns or comments were left by the participants?*

At the end of the survey, we asked participants to comment on their experience and further discuss whether or not they would consider using immutability in the future. There were many comments on a wide variety of topics. Using a grounded theory method, we classified each comment into recurring themes. What follows is a summary of our findings.

The participants were divided among those who thought immutability simplified their code and those who thought it made it more complex.

On the simplification side, participants mentioned how some functionalities, such as undo/redo, were much easier to implement because they had the guarantee that an object would never be modified after its creation. In general, there were many comments on how Phase 2 (during which refactoring and evolution took place) was easier with immutability.

Some participants commented on how immutability eased communication among teammates. They had to adopt a similar style to ensure that their program used only immutable objects and, thus, reading others' code was easier. Readability may have also been improved by immutable programs having shorter, more granular classes and methods. Interestingly, some participants in the control group commented that if they had used immutable objects, coordinating and communicating with their teammates would have been easier.

On the complexity side, participants in the treatment group mentioned a steep learning curve when first learning to use immutability in their programs. Others commented that the GoF design patterns were harder to implement using immutability.

Participants also mentioned that creating new objects for every change can become cumbersome, especially with nested objects. Some also mentioned that building user interfaces with immutable objects was difficult because available frameworks expected mutable objects. Most projects used the WPF framework[13] developed for classical OOP.

Other participants mentioned that they thought complexity could be reduced by using a language supporting proper immutability. Some participants in the mutable group commented that using immutability would have made their code more complex.

Participants were divided about immutability. Negative comments mainly concerned the learning curve and lack of language support. Positive comments pertained to communication among teammates and more understandable programs. We suggest future experiments measuring the impact of immutability on communication between developers. We also suggest improving support for immutability in OOP languages.

4.6 Discussion

We now discuss questions concerning the experiment.

4.6.1 Main Research Question

The main research question of this study is: What is the impact of immutability on object-oriented development?. With the results discussed in Sections 4.4 and 4.5, we can now answer this question.

While some data reflects an increase in complexity (a few quantitative measures and some participants' comments), generally, the quantitative data showed that the immutable

[13]https://docs.microsoft.com/en-us/visualstudio/designers/getting-started-with-wpf?view=vs-2022

programs were more granular and had better readability while the qualitative data showed that participants found the workload, difficulty, and complexity of the project lower.

The relationship between the group, grades, and survey answers is interesting. We observed that the participants in the treatment group generally had higher grades and also found the workload, difficulty, and complexity lower than the other participants. The results also showed that students with full marks on average found the project harder than the other students, which we explain by students working harder to get a better grade, independently of being in the control or treatment group.

We did not observe any significant disadvantage of using immutability in the data collected for this study, both quantitative and qualitative and both internal (measures, survey) and external (grades). We conclude that the disadvantages of using immutability in the context of OOP are outweighed by the advantages.

Were there significant differences in the survey answers among members of the same team?

We studied the answers of individual teams to see if members of the same team had different opinions on the workload, difficulty, and complexity of the project. Table 14 summarizes the average of the survey answers for workload, difficulty, and complexity (with standard deviations in parentheses). We notice that complexity had some of the highest standard deviations (e.g., 1.4087 for team 5). We picked the two teams with the highest complexity standard deviation (teams 5 and 9) and looked at their results in further detail.

For Team 5, part of the control group, only two members answered the survey. One member rated complexity at 1 for both phases of the project while the other rated complexity at 4 for both phases. The first participant self-evaluated their expertise at 5, while the second at 4. The first member was possibly more expert in OOP than the other member, which could explain their different assessment of the complexity.

For Team 9, part of the treatment group, there were three respondents. One rated the complexity of Phase 1 with a 1 and Phase 2 with a 3. The other two gave 4 and 5. The team members all self-evaluated their expertise at 3 or 4. Again, expertise seems to impact the perception of complexity.

The members of Teams 5 and 9 were most likely at different levels in terms of programming expertise despite their self-assessment. They possibly overestimated their capacity due to the Dunning-Kruger effect [54], yielding a high self-assessment average of 3.93 in Table 12).

We conclude that any difference between survey answers among members of the same team was due to differences between programming expertise among team members, not membership to a group. Also, our analysis did not yield any discrepancy that would indicate that the respondents were not honest or answered randomly.

Team #	Workload Average (SD)	Difficulty Average (SD)	Complexity Average (SD)
1	3.0625 (0.6585)	3.5000 (0.6124)	3.6250 (0.6960)
2	3.0000 (0.4082)	2.8333 (0.6872)	2.8333 (0.6872)
3	2.9167 (0.4930)	2.5000 (0.7638)	2.4167 (0.9538)
4	2.6500 (1.0137)	2.5000 (0.8660)	3.1000 (0.9950)
5	2.5000 (0.8660)	2.5000 (1.3229)	2.6250 (1.4087)
6	2.4375 (0.6092)	2.5000 (0.7906)	3.0000 (0.9354)
7	2.2000 (0.7483)	2.1000 (0.7000)	2.1000 (0.8307)
8	2.6250 (0.8570)	2.8125 (0.9499)	2.8750 (0.9922)
9	4.2500 (0.7217)	3.7500 (0.5951)	3.5000 (1.0408)
10	3.3750 (0.7806)	3.2500 (0.6614)	3.3125 (0.5830)
11	3.0000 (0.5477)	3.2500 (0.9421)	3.0500 (0.6690)
12	3.7500 (0.5590)	3.8125 (0.5266)	3.8125 (0.6343)
13	3.7500 (0.4330)	4.5000 (0.5000)	4.0000 (0.0000)
14	2.8333 (0.5528)	2.3333 (0.7454)	2.4167 (0.9538)

Table 14: Summary data on survey answers by each team

Were the answers to the survey influenced by the grade each student obtained?

The participants completed the survey before receiving their final grades for the project. However, they had access to their grades for Phase 1 and other course assignments. They could infer how well they were doing in the course. We divided the participants into three categories: those that had full marks, those who had marks higher than or equal to 85%, and those who had marks lower than 85%. Table 15 summarises the average per category for the answers on workload, difficulty, and complexity (with standard deviations in parentheses).

Students who obtained full marks reported on average a higher workload, higher difficulty, and higher complexity than the other students. The students in the middle grade category rated the lowest, with the low category rating a bit higher. We explain this observation by the fact that the students who got full marks may have worked harder and implemented more complex code to obtain better grades. We explain the difference between the middle and lower categories by the lower category struggling to apply the concepts taught in the course.

In Section 4.5.3, we concluded that the participants in the treatment group found the project generally easier. We also concluded that teams in the treatment group had higher grades on average. In this section, we now concluded that teams with higher grades usually found the workload heavier. We argue that, while the treatment group had higher grades than the control group, they still found the work easier even though they had to work harder to obtain their higher grades. Thus, **we conclude that immutability influences the participants' experience rather than their grades.**

Grades	Workload Average (SD)	Difficulty Average (SD)	Complexity Average (SD)
100	3.2250 (0.8212)	3.4250 (0.9189)	3.5000 (0.9747)
]85-100[	2.8864 (0.8974)	2.6932 (0.9577)	2.7273 (1.0305)
[0-85]	2.9028 (0.8686)	2.9444 (0.9263)	3.0833 (0.8457)

Table 15: Summary data on survey answers by grades

Were there qualitative differences in grading between the two groups?

Since immutability was not a major part of the course's curriculum, and not every student had to use it in their project, grading did not consider immutability for the most part. After phase 1, we had to intervene with three treatment group teams because they did not implement immutability correctly. After that intervention, no further concerns arouse regarding immutability correctness.

However, we still took a look at if there were any differences in recurring mistakes and comments left for the students in each group. There were more full marks in the treatment group (four) than in the control group (two). In general, both groups implemented the

GRASP and SOLID principles in phase one successfully, with one exception in the treatment group that did struggle with both the course content and immutability at first. Phase 2 design patterns were also generally well understood and implemented. Some teams in both groups struggled with patterns, but no significant difference between groups could be identified on that point.

We notice a difference between groups when we look at unit test implementations. In both phases, groups had to implement unit tests for the various functions required in their software. There were more teams that struggled with unit testing in the control group than in the treatment group. A common mistake was unit tests that were dependent on external systems (e.g., loaded saved files or required external GUI manipulation). In general, teams in the control group had worse unit test coverage than teams in the treatment group.

While the relationship between using immutability and unit testing cannot be generalized from these results, we argue that the immutable systems were most likely easier to test. Because of the lack of state change, it is easier to create independent test scenarios. Furthermore, if the immutable projects also had more numerous shorter methods, as reported in Section 4.4.3, that would also explain why they had an easier time creating unit tests for their software.

We conclude that there was a qualitative difference between the treatment and control group regarding unit testing. The control group wrote unit tests with worse coverage and some teams struggled with external dependencies. For aspects other than unit tests, both groups experienced similar difficulties and no significant difference could be found.

Why was performance not measured?

One issue raised by participants during the course concerned performance: creating new objects with every change may become a performance bottleneck.

We believe that performance is impacted by technology more so than by the development methodology. Measuring the performance of immutable programs written with an imperative, mutable OOP language would be like measuring the performance of an object-oriented program written in Haskell.

Most of the programs were developed in C#, as shown in Table 8, which is not particularly well suited for immutability. Microsoft added features to support immutability in recent updates[14] but we did not teach these features during the course and none of the participants used them on their own. C# includes a library providing immutable collections[15], which was used by some participants, but the language itself lacks many of the optimization available in languages favouring immutability, such as F# and Haskell.

Haskell in particular is a relatively high-performance language[16] that embraces full immutability. It achieves good performance in part by optimizations possible because of full referential transparency. Thus, we argue that performance issues related to immutability could be managed at the language and compiler levels.

We conclude that there are other significant advantages than performance to using immutability and there should be an effort made in OOP languages to improve the performance of immutability features.

Was the treatment group hampered by the lack of language support for immutability?

One restriction imposed on the participants was that they use a programming language supporting object-oriented programming. While there exist languages supporting both OOP and immutability features (such as OCaml and F#), the participants chose mainstream OOP languages such as C#, Java, and Typescript. Some used older versions of C# such as Unity C#, which have even less support for immutability. However, the Java and Unity C# teams were part of the control group, which did not have to introduce immutability in their code. Perhaps they would have selected a different language had they been part of the treatment group.

Out of the chosen languages, C# has most likely the best support for immutability.

[14]https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-9

[15]https://docs.microsoft.com/en-us/archive/msdn-magazine/2017/march/net-framework-immutable-collections

[16]https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/haskell.html

In fact, some of these features were presented to the students as part of the class content regarding immutability. In particular, pattern matching was presented as an alternative to the visitor pattern [69]. Some of the teams in the treatment group made use of this feature.

There were, however, some comments about the lack of language support for immutability making the learning curve more difficult for the treatment group. Some teams mentioned they had to spend more time at the start of the project establishing how components were going to interact with each other without the possibility of using mutators. While this contributes to a steeper learning curve, we believe this may explain in part why these same teams reported having an easier time with team communication in the later phase of the project. Having been forced to plan ahead early on made them develop a stronger structure for their code, which made communication easier in the long run.

While having more support for immutability features would reduce this initial learning curve by giving developers clear tools and techniques to use for achieving immutability, we believe this initial structural discussion would still need to take place. As such, the communication improvements would remain even with stronger immutability features.

We conclude that the lack of immutability support in languages contributed to a steeper learning curve. However, the initial more lengthy discussion about the project structure improves team communication in the long run. We believe the addition of better support for immutability in OOP languages would reduce this initial learning curve, without removing the positive impact on communication.

4.7 Threats to Validity

While we tried to keep the study and its results as objective and reproducible as possible, there are some threats to their validity.

4.7.1 Internal Validity

The study was done on a convenience sampling of 3rd-year undergraduate students in a B.Sc. of computer science. The participants were not selected at random but volunteered. There was also an incentive in the form of bonus grades (equivalent to a maximum of 10% bonus for the project assignment). This bonus was to compensate the participants for the added constraints to their semester assignment (the project).

While using student participants in studies tends to limit the generalization of any results, we believe there are advantages in this particular case. Students have limited experience in software development and thus, have fewer biases towards particular technologies or coding styles. They also can dedicate more time and focus to the given tasks, as professional developers would have to complete them during their personal spare time. Also, these students are the professionals of tomorrow. Yet, replicating this study with experienced professionals would grant significant additional insight.

We tried to avoid biasing the participants towards immutability. We mostly taught in the class classical mutable object-oriented programming. Over the 15 classes given in the course, less than three classes were spent on immutability. These classes introduced the basic concepts required to use immutability in the assignment, but not enough to create a significant bias towards immutability. We accept this threat.

The project was part of a software engineering class and some of the requirements were part of the curriculum and not necessary for the experiment. For example, the requirements of implementing GRASP and SOLID principles, as well as the GoF design patterns, were not strictly necessary for the experiment and would not necessarily represent how software would be made in the industry. However, we believe that in particular for the GoF design patterns, the added complexity prevents the project from being too trivial. The project was limited in terms of scope and total lines of code, and it would not be reasonable to expect a large-scale industrial-level project out of either a software engineering class or a controlled experiment.

There is also the question as to whether the usage of the GoF design patterns would skew the results in favour of mutability. After all, classical object-oriented programming makes heavy use of mutability. However, as we discussed in Section 4.2.2, some design patterns can be simplified when used in the context of immutability. While there was probably overall some bias towards mutable style, our results mostly favour immutable style. We believe this lends more credibility to our results and not less.

We left as much decision-making as possible to the students during the project. Most participants chose to develop their program in the programming language presented during the course, C#. Table 8 shows that only three projects were not done in pure C#.

The tools available to extract measuress from the programs were limited because not every program was developed in the same language. For instance, were every project developed in C#, we could've used Microsoft's own Maintainability Index[17] measure to compare maintainability between programs. The programming languages chosen were all available in SciTools Understand, which offers the usual Chidamber and Kemerer measures [21]. Even then, some measures were not available for TypeScript (e.g., CBO and LCOM). While this difference likely did not influence our results, future experiments should limit the participants to using the same programming language and make use of more specialized tools.

The fact that we measured over 80 measures over the programs makes it so that none of the individual measures are statistically significant. The usual p-value threshold of 0.05 does not apply (Bonferroni correction would make this value closer to 0.0006 instead). However, since this is an exploratory study aiming to find potential trends for future research, we instead took a look at the data as a whole and attempted to find patterns indicating potential advantages or disadvantages. We do not claim to have found significant differences between mutable and immutable approaches, but rather potential future research trends. Furthermore, it is unlikely that every quantitative result would coincidentally follow the same trend, which showed that participants in the treatment group found the project easier.

[17]https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-maintainability-index-range-and-meaning?view=vs-2022

4.7.2 External Validity

The Hawthorne effect may have influenced participants to perform better by knowing they were participating in this study. Participants knew about the subject of the study, which could have yielded a positive attitude towards the project in general and immutability in particular. This threat would be very difficult to avoid, however, so we chose to accept it.

There is also a concern regarding the students training in immutability. The results of the experiment would be influenced by the participants' skill and knowledge with immutability. As discussed at the beginning of Section 3, the participants had no prior experience with immutability and received around 10 hours (1/5 of the course) of training on the subject. Thus, their training on immutability was minimal. The course and experiment did not attempt to create a bias toward immutability, yet the results led to positive results towards immutability. Thus, the results suggest that replicating the study with participants with further experience with immutability should yield an effect even more favourable.

4.7.3 Construct Validity

We used a survey to obtain subjective data from the participants about the project. We then used their answers to compare the two groups. As discussed in Section 4.6.1, the participants seem to have in general overestimated their expertise. We mitigated the use of subjective data using multiple methods, including objective quantitative data collected from the developed programs.

For the quantitative data, we chose to measure as many measures as available in the tool to avoid any selection bias. Using an automated tool also allowed us to avoid any threats related to manual data collection.

4.7.4 Conclusion Validity

We avoided drawing strong conclusions as to the effect of immutability on specific measures because our results do not show significant correlations. Instead, we looked at and discussed overall trends. We concluded that the disadvantages of immutability in OOP programming are outweighed by its advantages.

4.8 Conclusion

In this chapter, we reported on an exploratory experiment on the impact of immutability on object-oriented software development. We divided a set of 67 participants into a treatment and a control group, composed of 8 and 6 teams of 4 or 5 students. Both groups developed the same project. We followed a multi-method approach to assess the impact of using immutability. We collected quantitative measures on the programs as well as qualitative assessments of the participants using a survey. We analysed the varied data to assess the impact of using immutable objects on software development.

We observed no significant negative impact related to using immutability on programs or participants. In the quantitative data, for the treatment group, we observed an increase in number of methods and a decrease in method size as well as number of comments. Programs developed by teams using immutability were more granular and required fewer comments compared to the mutable programs. The qualitative data showed lower perceived workload, difficulty, and complexity by the participants in the treatment group. Teams using immutability perceived the difficulty of the project to be lower than the other teams.

The differences between the two groups were not statistically significant for individual measures and, thus, we cannot draw strong conclusions. Yet, every trend indicated that immutability may have a positive impact on OOP and we found no evidence of disadvantages. **We did not observe any significant disadvantage of using immutability in the data collected for this study, both quantitative and qualitative and both internal (measures, survey) and external (grades). We conclude that the disadvantages of using immutability in the context of OOP are outweighed by the advantages.** This conclusion also indirectly and a-posteriori supports language designers who added immutability in OOP languages, based on their anecdotes and developers' lore (e.g., in C#).

Chapter 5

Studying New Features for Immutability Support in C#

5.1 Introduction

In the last chapter, we showed that immutability has a positive impact on OOP software development. In this chapter, we discuss language features which make use of or facilitate immutability in OOP. In particular, we study the impact of specific new features added to C# which enhances support for immutability.

We target a set of features that were recently added to C# and perform an experiment to measure the impact of these features on the development of immutable programs. The features targeted include Record Types, Record Updating, Pattern Matching, and Multiple Values Return. These features all have ties to functional programming and facilitate the development and usage of immutable programs.

We perform a multi-method empirical experiment using 12 participants: 10 graduate students and two professional developers. During the experiment, the participants assess a base program and extend it by adding new functionalities while enforcing transitive immutability.

The base programs are file system simulators that allow the creation and deletion of folders and files in a hierarchy. There are two base programs: one for the treatment group, and one for the control group. They implement the same functionalities, but the treatment program uses the new features, while the control program does not. We ask the participants to add new functionalities to their base programs. These new functionalities include adding a `Collect` method which aggregates components from a root folder according to specified criteria (e.g., files and folder, files only, recursively, etc.), adding a `Duplicate` method to copy part of the file system hierarchy, and adding an `Undo` method to rollback any operation made on the file system.

We collect qualitative data on the submitted programs by performing manual code analysis and quantitative data using a static analysis tool. We also collect subjective insight by asking the participants to fill out a survey and conducting an interview with the professional developer in the control group.

Results indicate that the new features have a positive impact on software maintainability. Quantitative data shows improvement in the Maintainability Index, Cyclomatic Complexity, and Number of Executable Lines of Code. Qualitative analysis reveals that both groups opted for a similar approach to implementing the functionalities. Even though a Visitor-based approach would have been possible for the control group, they opted for a Pattern Matching-based approach and compensated for the lack of language support by doing manual casting and type checking.

While these results could be interpreted as the student participants in the control group doing a poor job due to a lack of expertise, the interview conducted with the professional developer in the control group gives more insight. They also adopted the same approach as the students, and explain that they felt the Visitor approach, while having its advantages, adds too much indirection to be worth these advantages. They preferred the Pattern Matching approach, even without language support, and compensated with other features as a workaround.

The rest of the chapter is divided as follows. Section 5.2 situates this study in relationship with other studies. Section 5.3 presents our research questions and the structure of the experiment and methods used to collect and analyse data. Section 5.4 presents the data and results of the experiment and Section 5.5 discusses these results and answers each research question. Section 5.6 discusses potential threats to the validity of the results. Finally, Section 5.7 summarizes and concludes with future work.

This chapter is the basis of a paper sent for a special issue of Journal of Systems and Software on Software Languages Engineering. It is currently under review.

5.2 Background

While usability features for immutability have become more prevalent in programming languages, not much research has been done on their use by developers, as discussed in Chapters 2 and 4. We could not find any paper discussing these features and their empirical evaluations. Research in the domain focuses on language features for immutability enforcement.

In this study, we evaluate the impact of usability features for immutability on the development of immutable OOP programs. While languages do not explicitly label their features as supporting immutability, we chose features which were recently added to the C# language and were related to functional programming. Specifically, we consider the Records, Record Updating, Pattern Matching, and Multiple Values Return features. By adding these features, Microsoft implicitly assumes that they improve software quality or development efficiency. The main goal of this study is to confirm or infirm this assumption.

We execute a multi-method empirical study using graduate students and professional developers. The study requires the participant to read and understand a program developed using transitive immutability, then add new functionalities to that program. We ask half the participants to use the new C# features, while we prohibit the other half from using them. We then ask the participants to fill out a survey to evaluate the relative difficulty of the task. We use quantitative measures of the source code and qualitative measures for the survey, as well as an interview with one of the professional developers, to answer our research questions.

The main research question of this study is: *Do the recently added immutability-related features have a positive impact on writing immutable code in C#?*

We divide this research question into three secondary questions:

(1) *R5.1: What is the quantitative impact of using the immutability-related features on immutable code maintainability?*

(2) *R5.2: What is the qualitative impact of using the immutability-related features on workload, difficulty and complexity of writing immutable code?*

(3) *R5.3: What are the differences between immutable code written with and without the immutability-related features?*

5.3 Method

In this section, we describe our empirical study and the methods used to answer our research questions. We first discuss a pilot study. We then introduce the base software used for the experiment and the added functionalities that we ask our participants to add. Finally, we present the methods used to obtain quantitative and qualitative data from the source code and participants.

5.3.1 Language Features

This study aims at evaluating the impact of specific features added to C# for immutability support. We chose functional programming features added to the language in versions 7.0 to 9.0 (2017 to 2020). We present each feature and how they interact with immutability in what follows.

Record Types

Records are a type of data structure, similar to a class or a struct. Their main characteristic is structural equality. Two records are equivalent if their properties (declared next to the name of the Record Type) are equal. This structural equality allows using equality operators (i.e, `==` and `!=`) between records and using Record Updating. Reference equality can still be used using the `ReferenceEquals` method.

Records use a syntax similar to classes, with a more concise constructor and property definitions declared at the top of the record:

```
record MyRecord(string Field1, int Field2)
{
    void SomeMethod(string arg)
    {
        Field1 = arg;
        return Field2;
    }
}
```

While Records in C# are not required to be immutable, they mirror a similar feature of the same name in functional programming languages such as OCaml, F#, and Haskell. In an immutable context, using Records allows you to make use of Record Updating, a useful tool for updating immutable objects.

Record Updating

Record Updating (also called "non-destructive mutation" in C#) allows the duplication of a record while changing the values of the properties of the duplicate. This update does not affect the original record. Record Updating is used with the `with` keyword:

```
var entry1 = new MyRecord("Content", 0);
var entry2 = entry with { Field2 = 42 };
```

While Record Updating can be used outside of an immutable context, its main attraction is the ability to "simulate" mutation without breaking immutability. Using Record Updating, one can create methods which modify an object by returning a new version of itself with specific properties changed. This allows the usage of "setter" methods in an immutable context.

Pattern Matching

Pattern Matching is a syntax feature used to choose a branch based on the type of an object. Its syntax is similar to a `switch` statement, but it branches based on the type of the object.

There are two ways to use Pattern Matching: in an `if` expression or in a `switch` expression:

```
// If-expression example
if (someVariable is int i)
{
    return i + 2;
}
else
{
    return 0;
}

// Switch-expression example
return (someVariable switch
{
    int i => i + 2;
    string s => 1;
    _ => throw new
        InvalidOperationException();
});
```

Pattern Matching is a recurring feature of functional programming languages. It appears in Haskell, OCaml, F#, and many other functional languages, sometimes also under the form of function polymorphism. In these languages, it is used to destructure a record or list and branch depending on the success of this destructuring. It is the functional programming equivalent to object polymorphism (which is why it can be used to replace the Visitor pattern). The C# version of Pattern Matching is more limited in its abilities to destructure objects. It can branch on the type of an object, but can only destructure the contents of a Record:

```
// Definition of a Record family
// of addresses.
record Address(string Country);
record CanadaAddress(string Province)
    : Address("Canada");
record USAddress(string State)
    : Address("US");

...
// Switch expression destructuring
// an address.
return (address switch
{
    CanadaAddress(var country, var province)
        => ShipToCanada(country, province),
    USAddress(var country, var state)
        => ShipToUS(country, state),
    (var country)
        => UnsupportedCountry(country),
});
```

Out of the four features discussed in this study, Pattern Matching is the one that applies most to both mutable and immutable code. While it originates from immutable languages, the C# implementation can easily be used on mutable code to branch on object types, as well as to destructure mutable Records.

Multiple Values Return

In functional programming, it is frequent to create functions that return multiple values. A typical use case is creating a new element in a hierarchy: to return both the created element and the updated hierarchy as a result of a function.

You can return multiple values from a method using a special syntax for Tuples:

```
(int, string) MakeTuple()
{
    return (42, "Content");
}

// ...

(int, string) tuple = MakeTuple();
// Use tuple.Item1 and tuple.Item2
// to access the values.

// or alternatively

(int i, string s) = MakeTuple();
// Use i and s as normal variables.
```

Multiple Values Return is an important feature when working with immutability. Immutable (or “pure”) functions cannot have side effects, which means they must always return any updated object. As soon as a function updates more than one object type at a time, the ability to return multiple values at once because very useful.

5.3.2 Pilot Study

To assess the feasibility of the experiment, we opted to perform a pilot study. This pilot study was done with two recently graduated Ph.D. students. We gave them the instructions for the experiment and asked them to perform it and give their feedback.

The initial structure of the experiment involved the participants developing a program using given specifications and following a set of rules defined by Bloch [13] to ensure transitive immutability. The participants would then fill out a survey asking them to evaluate the resulting program's workload, difficulty, and complexity.

The feedback from the pilot study, however, indicated that this may be too big a task to ask of volunteer participants. The program must be complex enough to obtain significant results. However, asking the participants to develop a non-trivial program using a set of rules they were not used to, as well as using language features they may not have known existed, introduces significant risk. The participants may opt out of the experiment, or the quality of their work may be impacted.

To mitigate this risk, we redesigned the experiment so that the participants would instead be asked to understand and evaluate an existing program and add new features to it. We would still ask the participants to develop part of the program, but that part would be significantly smaller. They would need to give their subjective assessment of their base program, as they need to study and understand it to extend it. This turned out to be necessary, as the results of the survey show that the student participants found the experiment very difficult and frustrating. We discuss this further in Section 5.5.2.

5.3.3 Participants

We perform the experiment with 10 M.Sc. and Ph.D. students, as well as two professional developers. The students and developers are split into treatment and control groups evenly and given specific instructions for their group. They are given half a day to read and understand a base program and extend it with the same new functionalities.

Before the experiment, the treatment group is given a short training on the four C# features presented in Section 5.3.1. We did not give any information concerning the new features to the students in the control group, as none of them knew about the features and there was no risk of them accidentally using them.

In the case of the professional developer in the control group, we showed them the features and specifically instructed them not to use them because the professional developer knew about the features and could have used them otherwise.

Each of the student participants received, in the weeks prior to the experiment, training on a subset of the 23 "Gang of Four" design patterns [33], including the Observer and Visitor patterns. We also assume that they had a basic knowledge of the design patterns as part of their prior academic curricula.

At the end of the experiment, we ask the participants to answer a survey to obtain their subjective assessment of the experience. We also conduct an interview with one of the professional developers to obtain further insight into the results.

5.3.4 The Base Programs

In this subsection, we describe the specifications of the base programs and the improvements asked of the participants. For the experiment, we need a program simple enough to be understood and extended in a short amount of time, but also non-trivial to obtain significant data. The program took the form of a file system simulator.

The file system simulator supports creating a hierarchy of folders and files. A folder may include files and other folders (which we refer to as components). The base program supports a few simple operations: obtaining a component from a given path string, renaming a component, deleting a component, and observing a component, which involves getting notifications whenever the component is renamed or deleted using the Observer design pattern [33].

The file system simulator is developed while ensuring transitive immutability, done by following the same set of five rules suggested by Bloch [13] that we used in Chapter 4.

We made two versions of the program: one for the treatment group, and one for the control group. The only difference between versions is the usage of the four language features described in Section 5.3.1 in the treatment program:

(1) We changed classes containing data to Records. The more succinct syntax of Records contributed to reducing the number of lines of code in the treatment program.

(2) We used Record Updating to update the file system and its components. This allowed us to remove copy constructors and some code duplication when creating new objects from old ones.

(3) We used Pattern Matching instead of object polymorphism. For example, instead of having a `Rename` method in both `File` and `Folder` classes, which the file system called whenever a component needed to be renamed, we could consolidate all renaming code inside the file system method using Pattern Matching and Record Updating:

```
IComponent newComp = component switch
{
    File file
        => file with
            { Name = newName },
    Folder folder
        => folder with
            { Name = newName },
    _ => throw new ArgumentException()
};
```

We ask the participants to extend their base program with the same new functionalities. They must add a *Collect* operation, which walks through the hierarchy of files and folders from a given point and collects files and folders under this point. They must add an *Undo* operation, which is used to walk back any rename or delete operation. Finally, they must add a *Duplicate* operation, which, given a root component, creates a duplicated hierarchy in the file system.

Both programs, with detailed specifications, are accessible online via GitHub[1][2]. We made these repositories accessible to their respective group as part of the experiment.

[1]Treatment program: https://github.com/wflageol-uqtr/CSharpCaseStudy
[2]Control program: https://github.com/wflageol-uqtr/CSharpCaseStudyControl

5.3.5 Survey

After the experiment, we asked the participants to fill out a survey to obtain qualitative data on their experience. The survey was structured into five sections. For each question in Sections 2 to 5, the survey offered the respondents to elaborate on their answers in free text fields.

The first section related to general questions about participants:

- How many years of professional experience do you have in software development?
- What is the highest education degree you have obtained?
- How many years of experience (professional or otherwise) do you have with object-oriented programming?
- How many years of experience (professional or otherwise) do you have with the C# language?
- Have you used Record Types or Pattern Matching before?

The second section measured their subjective appraisal of their base program. Each question was answered using the Likert scale (1-5):

- What was your impression of the workload for inspecting and understanding the existing code?
- What was your impression of the difficulty felt in inspecting and understanding the existing code?
- What is your impression of the complexity of the existing code?

The third section measured their subjective appraisal of the code they developed to extend the program. Each question was answered using the Likert scale (1-5), and asked the respondent to elaborate in a free text field:

- What was your impression of the workload for implementing the new functionalities?

- What was your impression of the difficulty felt in implementing the new functionalities?
- What is your impression of the complexity of your own code?

The fourth section used the NASA Task Load Index questionnaire [46]. Questions were answered on a scale of 1 to 7:

- Mental Demand: How mentally demanding was implementing the specifications?
- Physical Demand: How physically demanding was implementing the specifications?
- Temporal Demand: How hurried or rushed was the pace of implementing the specifications?
- Performance: How successful were you in accomplishing what you were asked to do?
- Effort: How hard did you have to work to accomplish your level of performance?
- Frustration: How insecure, discouraged, irritated, stressed, and annoyed were you?

Finally, the treatment group had a fifth section asking their opinion on the new features they used:

- Do you feel like the features presented in this study helped with the implementation of the specifications?
- Would you consider using the features presented in this study in a future project?

5.4 Results

In this section, we present the results of the experiment in the form of quantitative and qualitative data obtained from the programs and survey answers.

5.4.1 Program Analysis

To obtain qualitative data from the experiment, we manually analyse each completed program submitted by the participants. We look for trends and patterns in the code and differences between the treatment and control groups using a grounded theory approach which involved categorizing the approaches used to implement the *Collect*, *Duplicate*, and *Undo* features. Then, we obtain quantitative data by extracting a set of measures using automated code analysis tools and perform statistical analysis tests to see if the measures are significantly different from one group to the other.

Qualitative Analysis

We manually examine the code of the programs submitted by the participants to determine which approach was used for implementing each functionality. We intended to categorise the approaches, but the results are much more homogenous than expected across groups.

Each participant in the treatment group made use of Pattern Matching to implement the *Collect* and *Duplicate* functionalities, which we expected considering that the participants were shown the feature just prior to the experiment and it was appropriate for implementing these functionalities. Pattern Matching was specifically used to discriminate components between the `IFile` and `IFolder` interfaces.

+Interestingly, all participants in the control group also made use of ad-hoc Pattern Matching by using manual casting and type-checking, despite not having received any training or prompt to use such a technique. None of them used the Pattern Matching syntax presented in Section 5.3.1, because either they did not know it, or we specifically forbade them from using that feature.

For the *Undo* functionality, the implementation was trivial considering the program was already implemented using transitive immutability. Each participant, irrespective of their group, used a stack or list to keep previous versions of the file system and implemented *Undo* by simply "popping" the last version.

Quantitative Analysis

We obtain quantitative data by analysing the programs submitted by the participants with a static analysis tool. We are interested in the impact of using the new features on code maintainability. We collected data for five measures: Maintainability Index, Cyclomatic Complexity, Coupling Between Object, Number of Lines of Code, and Number of Executable Lines of Code. We chose these measures because they represent the usual dimensions against which we measure software quality and maintainability.

Maintainability Index, proposed by Oman and Hagemeister [70], assesses program maintainability by using a combination of the Halstead Volume [43], McCabe's Cyclomatic Complexity [65], and Number of Lines of Code. According to the Software Assurance Technology Center (SATC) at NASA [75], a combination of Cyclomatic Complexity and code size is an effective evaluation of software reliability. We use the version available in Microsoft Visual Studio, which normalizes its values between 0 and 100[3]. Microsoft suggests that a Maintainability Index between 20 and 100 is considered "good".

Cyclomatic Complexity was introduced by McCabe [65] to measure the structural complexity of programs. The measure represents the number of branching paths in a method (e.g., `if` statements, `switch`, etc.). A program with more branching paths leads to more test effort to achieve good coverage and negatively affects maintainability.

Coupling Between Objects is a measure introduced by Chidamber and Kemerer [21] that counts a program's dependencies between objects (or classes). An object has a dependency when one of its attributes, base classes, or method arguments references another object. Class Coupling can be used as a predictor of software failure and high coupling is known to have a negative effect on maintainability [83].

Number of Lines of Code is the exact number of source code lines present in the files of the program, including comments and blank lines. Number of Executable Lines of Code is an approximation of the number of operations or lines of code that are executed at run-time. We use these measures as indicators of code size, which can have an effect on

[3]https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-maintainability-index-range-and-meaning?view=vs-2022

maintainability [70].

Table 16 shows the measures collected for each submitted program. The table shows only 10 entries, despite there being a total of 12 participants (10 students and 2 professionals), because two of the participants (one in the control group and one in the treatment group) did not implement all the required functionalities. To avoid biasing the quantitative measures, we removed their programs from the analysis.

We observe differences between the two groups for most measures. Maintainability Index averages 91.8 in the control group and 93.2 in the treatment group, indicating that the final programs submitted by the treatment group were slightly more maintainable. We observe the same differences in most other measures: Cyclomatic Complexity (80.8 control vs 61.2 treatment), Number of Lines of Code (401 vs 347.8), and Number of Executable Lines of Code (117 vs 89.4).

The programs developed by the treatment group seem more maintainable than those developed by the control group, which would be clear if each participant had fully developed their program independently. However, they started from base programs that they extended by adding new functionalities. These base programs would have an influence on the measures collected.

Table 17 shows the measures collected for the base programs for both groups. We see that the treatment version of the program already has better maintainability than the control version of the program. While the only difference between the base programs is the usage of the new features, it is important to disclose that the measures obtained in Table 16 are clearly influenced by the base programs.

We cannot, however, apply simple arithmetic to obtain the difference between base measures and final measures. For example, if we subtracted the number of lines of code of the base program from that of the submitted programs, we would be ignoring any line that was modified by the participant. We wrote a script to calculate the number of changed lines of code between each submitted program and the base program. Our script only considered added or modified lines and did not count deleted lines. The reason for this choice is that it is impossible to determine if a line has been modified or deleted and replaced by a new one. We obtained an average of 120.8 changed lines of code for the control group and 95 for the treatment group. This difference is consistent with the total number of lines of code of the final programs.

As shown by the relative complexity of calculating the seemingly simple number of lines of code measure, it is not clear that we can perform the same on other measures. For example, measures such as Maintainability Index and Cyclomatic Complexity require the context of the full program and cannot be calculated easily on partial programs. Furthermore, as they may not be linear in their distribution, we cannot perform arithmetic operations on them (e.g., subtracting the base measures from the final measures).

The alternative would have been to have the participant develop the full program by themselves. However, as discussed in Sections 5.3.2 and 5.5.2, this would not have been feasible with student participants as the workload would have been too high. We believe the best compromise in this situation is to use the final program measures while disclosing their limitations.

We perform further statistical analysis to gain additional insight into the effect of the treatment group on the data. Our data is non-normal, as evidenced by performing a Shapiro-Wilk test [82], thus we perform the non-parametric Mann-Whitney U test to determine if the sets of data from each group come from the same population. We use Cliff's Delta to obtain the effect size of the independent variable on the measures.

Table 18 shows the results of the statistical analysis. It shows that the data from the treatment group is from a different population than the data from the control group, which means that the group influences the measures. We adjust the standard p-value threshold of 0.05 using the Bonferroni adjustment by dividing by four (we consider NLoC and NLoCExec to be highly correlated and be essentially the same measure) to obtain the threshold value of 0.0125, which three measures meet: Maintainability Index, Cyclomatic Complexity, and NLoCExec. For each of the three measures, the Cliff's Delta value is 1.00 (or -1.00) indicating that the populations are completely disjoint.

These results mean that the Maintainability Index is systematically higher in the treatment group and that the Cyclomatic Complexity and Number of Executable Lines of code are systematically lower.

ID	Group	MI	CC	CBO	NLoC	NLoCE
1	Control	92	74	25	375	106
2	Control	91	99	27	513	170
3	Control	92	81	26	381	103
4	Control	92	76	26	359	102
5	Control	92	74	25	375	104
6	Treatment	93	60	25	339	89
7	Treatment	93	59	25	346	90
8	Treatment	93	63	26	368	86
9	Treatment	93	66	26	355	98
10	Treatment	94	58	25	331	84

Table 16: Program Measures

Group	MI	CC	CBO	NLoC	NLoCE
Control	93	61	25	318	78
Treatment	95	48	25	276	59

Table 17: Base Program Measures

Measure	P-Value	Effect Size
Maintainability Index	0.0074	-1.00
Cyclomatic Complexity	0.0119	1.00
Coupling Between Objects	0.4884	0.28
NLoC	0.0211	0.92
NLoCExec	0.0122	1.00

Table 18: Mann Whitney U test results

5.4.2 Survey Analysis

To obtain qualitative data from the participants, we asked them to fill out a survey following the completion of the experiment. The content of the survey was described in Section 5.3.5. We detail the results of that survey in what follows. There was a large discrepancy between the answers of the students and of the professional developers, we separated them into different tables.

General Questions

When asked about their professional experience, most of the student participants (7 out of 10) answered having less than three years of experience in software development. The professional developers answered having 6 to 10 years and 10 to 15 years.

When asked about their education level, most of the students answered having a master's degree (i.e., being Ph.D. students). The professional developers answered having a bachelor's degree and a professional degree.

Most of the student participants had little experience with the C# language and had never used the new features presented. The professional developers had more extensive experience with the language and were familiar with the features.

Appraisal of the Base Program

The second section of the survey asked the participants to appraise their base program as to the workload for analysing it, the difficulty they had understanding it, and their perceived complexity of the program.

Table 19 shows the averages and standard deviations for the answers of students in both treatment and control groups. Table 20 shows the individual results for each professional developer.

Results are similar for both groups, with the treatment group having slightly lower values indicating they may have had less difficulty understanding the base program. Additional analysis (using the Mann-Whitney U test as in Section 5.4.1) indicates this difference to be non-significant (p-values > 0.4). The professional developer in the control group seems to have found the exercise very easy, whereas their counterpart in the treatment group found it somewhat harder.

Measure	Control Avg (SD)	Treatment Avg (SD)
Workload	3.8 (0.74)	3.6 (0.8)
Difficulty	4.0 (0.63)	3.4 (0.49)
Complexity	3.6 (0.8)	3.4 (1.02)

Table 19: Student Answers for Appraisal of the Base Program

Measure	Control	Treatment
Workload	1	2
Difficulty	1	2
Complexity	1	3

Table 20: Professional Developers Answers for Appraisal of the Base Program

Appraisal of the Extended Program

The third section of the survey asked the participants to appraise the workload, difficulty, and complexity of their own program extension to add new functionalities to their base program.

Table 21 shows the averages and standard deviations for the answers of students in both treatment and control groups. Table 22 shows the individual results for each professional developer.

The same trend as in the previous section can be seen, with the treatment group having slightly lower values than the control group. Complexity in particular has a large gap between groups (3.2 vs. 2.2, each outside of the other's standard deviation range) indicating that the treatment group found their resulting code less complex than those in the control group. Further analysis using the Mann-Whitney U test produces a p-value of 0.069 which, while not significant to draw conclusive results, shows that the populations of the two groups may be different.

The answers of professional developers match each other at much lower values than those of the students.

Measure	Control Avg (SD)	Treatment Avg (SD)
Workload	3.8 (0.75)	3.4 (0.49)
Difficulty	3.2 (0.98)	3.2 (0.75)
Complexity	3.2 (0.75)	2.2 (0.4)

Table 21: Student Answers for the Appraisal of the Implementation of the New Functionalities

Measure	Control	Treatment
Workload	1	1
Difficulty	2	1
Complexity	2	2

Table 22: Professional Developers Answers for Appraisal of the Implementation of the New Functionalities

NASA Task Load Index

The fourth section asked the participants to answer questions from the NASA Task Load Index questionnaire [46] regarding their experience.

Table 23 shows the averages and standard deviations for the answers of students in both treatment and control groups. Table 24 shows the individual answers for each professional developer.

While the answers are similar for both groups, the treatment group has slightly higher values, suggesting that they found the experience generally more demanding and frustrating than their control group counterparts. Further analysis using Mann-Whitney U testing does not reveal any statistically significant differences.

The professional developers generally found the experience less demanding and frustrating than the students, and both marked that they felt they were very successful in implementing the requirements by rating 7 on the Performance question.

Measure	Control	Treatment
Mental Demand	5.4 (1.2)	6.0 (1.10)
Physical Demand	3.6 (2.33)	4.4 (1.85)
Temporal Demand	5.4 (1.36)	4.6 (2.33)
Performance	5.2 (2.4)	5.6 (1.36)
Effort	4.8 (1.72)	5.8 (1.17)
Frustration	5.0 (1.79)	5.6 (2.33)

Table 23: Student Answers for NASA Task Load Index

Measure	Control	Treatment
Mental Demand	3	2
Physical Demand	1	1
Temporal Demand	2	2
Performance	7	7
Effort	4	3
Frustration	1	3

Table 24: Developer Answers for NASA Task Load Index

Questions on New Features

Only the survey given to the treatment group participants included this final section. The section was related to their opinion on the new features that they used in the experiment.

Every participant answered that the new features helped with the implementation of the specifications and that they would consider using the new features in the future. In particular, comments mentioned Pattern Matching as an especially useful feature. The professional developer mentioned they frequently used Pattern Matching and Multiple Values Return. They were not as familiar with records but said they already saw situations where they would be helpful.

5.4.3 Interview

We interviewed the professional developer of the control group after the experiment to gain further qualitative insight into the new features. For reasons of availability, we could not conduct the same interview with the professional developer of the treatment group. The interview took place after the developer had filled out the survey and lasted 30 minutes. What follows is a summary of each question and answer given during the interview.

Would you have used the new features if allowed?

The developer was part of the control group and did not have access to use the new features. However, they were aware of the existence of the features and knew how to use them. The developer answered that they would have used the new features to help implement the functionalities. They had to simulate Pattern Matching by using a combination of the `is` and `as` keywords for discriminating between `IFile` and `IFolder` objects. They would have rather used Pattern Matching.

Do you think being unable to use the new features impacted your implementation?

The developer answered they felt it was a minor annoyance, but overall the structure of their code was not majorly impacted. They still used ad-hoc Pattern Matching using available features, so their overall approach did not change.

Did you consider using the Visitor design pattern to implement the *Collect* and *Duplicate* functions?

As the developer was not familiar with that specific design pattern, we showed them a full implementation of the features using the Visitor pattern. After examination, they saw the advantages of such an approach on reusability and maintainability but thought the added layers of indirection complexified the code compared to the Pattern Matching approach. Knowing the Visitor pattern, they would still instead use the Pattern Matching approach.

5.5 Discussion

In this section, we review and discuss the results and answer each research question.

5.5.1 Research Question 5.1

Our first research question is What is the quantitative impact of using the immutability-related features on immutable code maintainability?

We collected data from the programs developed by each participant during the experiment. To assess immutability, we collected measures for the Maintainability Index, Cyclomatic Complexity, Coupling Between Objects, Number of Lines of Code, and Number of Executable Lines of Code. In this discussion, we will consider the full programs, including the base program and the participants' extensions, as these are the only viable measurements we can obtain.

There was a clear difference between the treatment and control groups for every measure, except Coupling Between Objects.

The Cyclomatic Complexity varies widely in the control group programs, while it is much more consistent in the treatment group (smaller standard deviations). Each program submitted by the treatment group has a lower Cyclomatic Complexity than those submitted by the treatment group. This difference may be due to the control group participants using ad-hoc Pattern Matching with manual casting and type-checking operators. This approach would contribute to increasing Cyclomatic Complexity by adding multiple unnecessary conditions and branches to ensure proper type safety. Further statistical analysis reveals that the difference in Cyclomatic Complexity between groups is significant (p-value of 0.0119 against the threshold of 0.0125).

Between the two measures of the number of lines of code, the most significant is the Number of Executable Lines of Code, both because the difference between groups is greater and because the measure itself is more significant when discussing maintainability, as it ignores lines without statements such as blank lines. Again, the treatment group has fewer lines of executable code than the control group. The same reasoning as for Cyclomatic Complexity may apply here: the extra verification steps needed for manual casting for the Pattern Matching approach contributed to increasing the number of lines of code. Further statistical analysis reveals the difference between groups is significant (p-value of 0.0122 against the threshold of 0.0125).

Finally, the Maintainability Index for each program was above 90, which means although there was a difference between groups, the overall maintainability of the programs is very good. As Maintainability Index is calculated in part using Cyclomatic Complexity and Number of Lines of Code, it is not surprising to see the same relationship between groups. The difference between group averages is about 1.4 and further statistical analysis reveals this difference to be significant (p-value of 0.0074, against a threshold of 0.0125).

These results reveal that the treatment group, who were using the new features for immutability support, produced programs with greater maintainability than the control group who did not use these features. We conclude that the new features added to the C# language, in particular Pattern Matching, contribute to improving the quality and maintainability of programs.

5.5.2 Research Question 5.2

Our second research question is What is the qualitative impact of using the immutability-related features on workload, difficulty and complexity of writing immutable code?

To answer this question, we asked the participants to fill out a survey after the experiment and give their subjective opinion on the workload, difficulty and complexity of the code during the experiment.

Survey results were similar between the treatment and control groups. For most answers, both group averages are within each other's standard deviation and further statistical analysis shows no significant difference between the datasets.

The only exception is regarding the complexity of the code written by the participants. The control group rated the complexity of their code one point higher on average than the treatment group. However, the statistical analysis does not show the difference to be significant by itself (p-value of 0.069), so further study is needed before drawing any conclusion.

The answers to the NASA Task Load Index section of the survey reveal that the experiment seems to have been difficult, particularly for the students. Both groups rated mental demand and frustration above 5 (out of 7), with temporal demand and effort also rated high. Some participants even rated physical demand very high, although this may have been a misunderstanding of the meaning of that specific question.

These results confirm that our decision to lighten the load by simplifying the experiment following the pilot study was the right one. In fact, even the resulting experiment was still difficult for the students.

We notice a distinction between the survey answers of the professional developers and the students. In general, the professional developers found the experiment easier in terms of workload and had less difficulty understanding the base programs. Their NASA Task Load Index answers were much lower in general and they both rated their own performance at 7. Their ratings on the complexity of the code were close to the students. This difference can be due to the gap in experience between the students and the professional developers. Most students stated they had no experience with C#.

Overall, we did not notice any significant difference between the qualitative data of the treatment and control groups. While we could not observe any advantage of the new features on the workload, difficulty, and complexity of the tasks, we could not observe any disadvantage either. Perhaps the overall difficulty of the experiment hid any possible difference. We conclude that the immutability-related features do not impact the workload, difficulty, and complexity of writing immutable code.

5.5.3 Research Question 5.3

Our third question is What are the differences between immutable code written with and without the immutability-related features?

Before we performed the experiment, we assumed that Visitor vs. Pattern Matching would define the main difference between the treatment and control implementations. The *Collect* and *Duplicate* features were good fits for a Visitor implementation. A `IComponent-Visitor` interface could be declared, containing methods to visit files and folders. Then, for the *Collect* functionality, a `CollectVisitor` could be implemented defining how to walk through the file system hierarchy and whether or not individual files and folders should be collected. A similar implementation could have been done for the *Duplicate* functionality.

Pattern Matching offers an alternative to the Visitor design pattern because it effectively allows the equivalent of dynamic dispatching: to associate behaviour discriminated on the type of an object. Reusing the examples above, a Pattern Matching implementation of the *Collect* functionality would be a function that branches on the type of an object (whether it's a file or folder), which removes the need for a Visitor because the "visiting" functionality is covered by the Pattern Matching branching.

We expected that some of the participants from the control group would use the Visitor design pattern, and the treatment group would instead opt for the Pattern Matching implementation. However, the experiment results show that both groups went for the Pattern Matching approach. The control group did not have access to the proper language feature for Pattern Matching, and so implemented it ad-hoc, using manual type casting operators such as `as`, `is`, and `typeOf`.

During the interview with the control group's professional developer, we presented them with a Visitor implementation of the experiment. While they understood the advantages of such an approach, they felt the added layers of indirection made the program more complex and were not worth the advantages. When asked what they would have done differently if they had access to the immutability-related features, they answered that they would have used the features, but that the general structure of their code would not have changed, because they used the same approach and simply compensated for the lack of features.

There were not large differences in the code implemented by both groups, because both groups adopted the approach that the new features, specifically Pattern Matching, allow. The control group simply compensated for the lack of features by using manual casting and type-checking conditions. Their code would have been improved by using the new features. We conclude that the immutability-related features enhance the approach which developers naturally use to solve problems and that the lack of these features results in them being implemented ad-hoc.

5.5.4 Main Research Question

With the secondary research questions answered, we can now answer our main research question: Do the recently added immutability-related features have a positive impact on writing immutable code in C#?

We found that the immutability-related features contribute to improving the maintainability of programs. We could not identify any advantage or disadvantage of using the immutability-related feature when considering the subjective workload, difficulty, and complexity of program understanding and implementing specifications. We also observed that the control group, lacking the immutability-related features, still used the same approach as the treatment group to implement their program, leading to ad-hoc implementations of the new features.

We conclude that the immutability-related features have a positive impact on writing immutable programs in C#. Other OOP languages should consider implementing these features, in particular Pattern Matching, into their language to better support immutability and improve the maintainability of immutable programs and the developers' quality of life.

5.6 Threats to Validity

In this section, we discuss potential threats to the validity of this study. While we tried to keep the study and its results as objective as possible, some threats inevitably exist. We divided the threats into Internal, External, Construct, and Conclusion Validity threats.

5.6.1 Interval Validity

The study was done using a convenience sampling of graduate students and professional developers. The students' supervisor asked them to participate and the professional developers volunteered after a general announcement via social media.

The majority of the participants were students, which limits the generalization of results in any study. We included professional developers to lessen any bias brought by students. Students also have advantages, as they have limited experience in software development and thus, have fewer biases towards particular technologies or coding styles. They also can dedicate more time and focus to the given tasks, as professional developers would have to complete the tasks during their personal spare time (which is why it is difficult to find professional developers for these experiments in the first place). Nonetheless, we must accept threats to internal validity and suggest a future replication of the study with mainly professional developers.

We avoided biasing the participants toward the immutability-related features. We trained only the treatment group to use the new features and did not mention them to the control group. An exception was made for the professional developer in the control group. They were already aware of the features and used some of them regularly in their work, we instructed them specifically to not use them. Fortunately, this instruction did not seem to affect the results, because the code submitted by the professional developer used a similar approach as the other participants in the control group.

5.6.2 External Validity

The Hawthorne effect may have occurred: participants perform better knowing they are participating in this study. Participants, particularly in the treatment group, knew about the subject of the study, which could have yielded a positive attitude toward the immutability-related features. However, we did not observe such an effect in the results, as evidenced by the results of the NASA Task Load Index section of the survey. For the students, survey revealed that frustration level was high and mental demand was slightly higher for the treatment group.

5.6.3 Construct Validity

The design of the study is a significant threat that must be addressed. Following the pilot study, which revealed that having the participants develop the whole program themselves was much too big of a task, we redesigned the study. Instead of developing the full program, participants would analyse a base program and extend it by adding new functionalities. The base programs were different for both groups, as the treatment base program made use of the immutability-related features, while the control base program did not.

This design implies that part of the code submitted by the participants was not developed by them, but was still counted when collecting quantitative data on the programs.

It was not feasible to accurately subtract the base programs measures from the submitted programs because of the non-linear nature of most of the measures collected (e.g., Maintainability Index). We did calculate the difference in lines of code between the base program and submitted programs and the results showed the same tendency as before, with the treatment group having a lower average difference of lines of code than the control group.

This trade-off was necessary, as evidenced by the results of the survey. The student participants found even the current simplified experiment very difficult and frustrating to complete. To mitigate the effect of this threat, this study uses multiple methods, in the form of quantitative analysis, qualitative analysis, a survey, and an interview, to gain more contrasting insight.

5.6.4 Conclusion Validity

The statistical analysis done in the survey in Section 5.4.2 did not reveal any significant difference between groups. We thus do not have any strong conclusion when discussing this analysis, even though there was a gap between the answers of the groups concerning the complexity of their resulting code. The treatment group rated their complexity generally lower than the control group by one point out of five. However, we believe we lacked enough data for the statistical analysis to show a significant difference here. We believe further studies may show a significant relationship between the usage of the immutability-related features and the complexity of the resulting code. The Maintainability Index results shown in Section 5.4.1, also support this conclusion.

5.7 Conclusion

We presented a multi-method experiment on the impact of immutability-related features added to C# from 2017 to 2022. We divided a set of 12 participants, composed of 10 graduate students, mostly Ph.D. level, and 2 professional developers, into a treatment and a control group. Both groups added the same functionalities to two different base programs: one using the new features and one without them. We followed a multi-method approach to assess the results.

We collected quantitative measures on the programs using a static analysis tool and qualitative data by performing a manual analysis of the programs. We also asked the participants to fill out a survey to obtain their subjective opinion on the workload of the study, the difficulty of completing the tasks, and the complexity of the code. Finally, we performed an interview with one of the professional developers and asked questions about their decisions and their thoughts on the immutability-related features.

We observed a significant difference in the quantitative data collected on the programs of the two groups. The treatment group programs showed a higher Maintainability Index, lower Cyclomatic Complexity, and lower Number of Executable Lines of Code than the control group programs. These results seem to show that in general, using the new features improves the maintainability of the code.

With the manual analysis, we noticed that both groups used the same approach to implement the new features. We expected that some participants in the control group would choose a Visitor-based approach, as that was an appropriate design pattern for solving this particular situation, but every participant instead used a pattern matching approach. The control group worked around the lack of features for pattern matching by manual casting and type-checking. Having access to the new features would have improved their code. When asked whether or not they would consider using the immutability-related feature in the future, every participant in the treatment group said they would.

We did not observe any significant difference between groups in the survey results. We did notice a difference between the professional developers and the students. The students reported a very high mental demand and frustration, whereas the professional developers reported a very low one. In contrast, both professional developers considered they had accomplished their tasks perfectly, whereas the students had varying degrees of success. Two students could not finish implementing the required functionalities. Even though we had lowered the workload of the experiment significantly following a pilot study, it seems it was still difficult for the students to perform.

During the interview with the professional developer, which was part of the control group, we asked if they would have used the new features if allowed, how they thought this would have impacted their code, and whether or not they considered using the Visitor pattern. They answered that they would have used the new features in a normal setting, but that the structure of their code would not have changed significantly because they used the same approach but with workarounds to compensate for the lack of pattern matching. When shown the Visitor approach to implementing the program, they said they understood the advantages, but were not sure they were worth the added indirection layers.

We conclude that the immutability-related language features added to C# have a positive impact when used to implement immutable programs. We suggest that other OOP languages add these features to better support immutability-style code in the future. In particular, a pattern matching approach seems more intuitive to developers than the Visitor-based one, so language designers should consider features allowing safe pattern matching (without requiring manual casting).

Chapter 6

Solving Immutable Method Reusability Problems with a New Design Pattern

6.1 Introduction

As discussed in Chapter 4, introducing immutability in OOP presents some challenges. One of these challenges is method overriding and the covariance of the return type. Inheritance and subtyping are the main mechanisms for code reuse in OOP. As objects inherit methods and attributes from their parent, mutators (methods that modify the receiver) continue to function because their context (attributes and related methods) is preserved during inheritance. Immutable objects cannot have mutators by definition and instead must use functional updating (methods which return a new object instead of modifying the receiver). Such methods can be problematic when using inheritance as illustrated in Section 6.2.

In this chapter, we analyse an example of a situation where the naive subtyping approach for immutable objects creates code duplication and scalability problems and we present a solution to mitigate these problems in a new design pattern. We also discuss the advantages and limitations of this design pattern as well as its implementation in multiple languages (i.e., Clojure, Java, and Kotlin) and which language features can improve these implementations. We identify functional updating and dynamic typing as features that directly affect the implementation of the design pattern. We then extend a language where the problems exist, Common Lisp, with a new functional updating feature and solve the problems.

Thus, the contributions of this chapter are as follows:

(1) A new design pattern to solve problems of code duplication and scalability when combining subtyping and immutability.

(2) Variants of the design pattern presented in Clojure, Java, and Kotlin.

(3) Discussion of specific features (functional updating and dynamic typing) which have an impact on the implementation of the design pattern.

We opted to use a more narrative-focused approach instead of the more traditional Coplien Form as used by Gamma et al. [33], because the pattern applies to a specific situation which needs to be introduced first to understand the problems and why there is a need of a design pattern solution in the first place.

The rest of the chapter is divided as follows. Section 6.2 presents a running example of a problem which we will discuss throughout the study. Section 6.3 presents our solution to that problem in a new design pattern based on the Factory Method pattern [33] and a Java implementation of this design pattern. Section 6.4 shows the implementation of that solution in the Clojure and Kotlin languages and discusses differences. Section 6.5 generalises the solution and discusses limitations and the language features that mostly affect the implementation. Finally, Section 6.7 summarises the chapter.

This chapter is the basis of a paper sent for the EuroPLoP 2023 conference. It was accepted and will be presented at the conference in July 2023.

6.2 Problem outline

We use a simple running example to illustrate the problems encountered while developing immutable objects in OOP languages. While this example will be shown using the Java

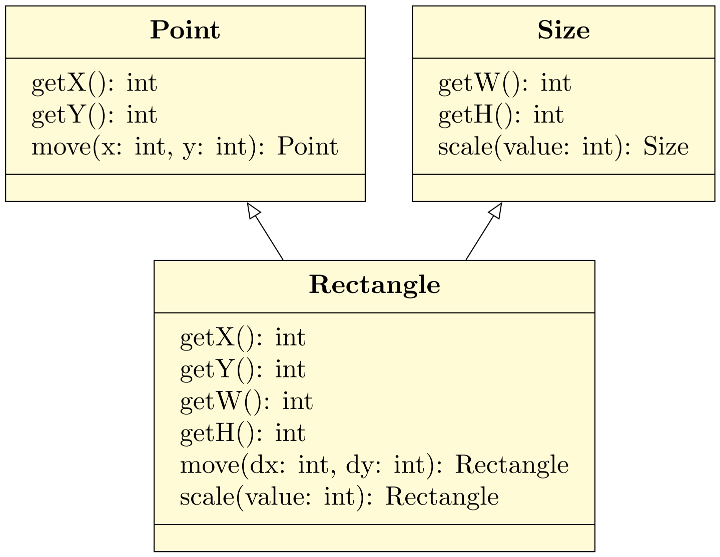

Figure 13: Simple class diagram of the geometry library specifications

language, the problems shown here are present in many statically typed OOP languages, including C# and C++. For the sake of brevity, we keep the specifications simple with few methods, without loss of generality.

6.2.1 Specifications

We want a library for managing geometry objects, such as points, sizes, and rectangles. A rectangle has both the attributes of a point (its position) and a size (its extent). As such, we expect it to use all the methods available on both points and sizes. Furthermore, we want every object to be immutable. Every method on an immutable object should always return a newly created object and not modify the receiver. Figure 13 shows a class diagram of these specifications.

The `Point` class defines X and Y attributes. It also defines a `move` method, which creates a new `Point` with the same position, offset by a `dx` value and a `dy` value.

The `Size` class defines W and H attributes. It also defines a `scale` method, which creates a new `Size` with its width and height scaled by the specified value.

We want to avoid code duplication, in particular of the definitions of the attributes and of the `move` and `scale` methods. Adding a new method to any of its parents should make it automatically available and usable to `Rectangle` without having to define delegating methods for every single method. Although reasonable from the point of view of OOP development, these specifications bring two problems.

6.2.2 Problem 1: Inheritance and immutability

Bloch [13] established a set of rules which, when followed, ensures transitive immutability in any OOP language. The second rule is that classes in an immutable program should not be extendable. This rule exists to avoid future users breaking immutability by using inheritance to add mutating methods. Furthermore, Java (and many other OOP languages) only supports single-class inheritance and our design would require `Rectangle` inheriting both `Point` and `Size` for it to work.

There is also the issue of the "is-a" relationship. Is a rectangle a point and a size? One could argue the relationship here is "has-a": a rectangle *has a* position and an extent. This would indicate that composition should be used here instead of inheritance. Instead of having X, Y, W, and H attributes, Rectangle could instead have Position (Point) and Extent (Size) attributes.

However, this composition approach causes another problem, illustrated in the following lines of code:

```
Rectangle r = new Rectangle(1, 3, 2, 2);
Point p = r.getPosition().move(1, 2);
```

`getPosition()` returns a `Point` object, on which calling `move` always returns a new `Point`. To obtain a new `Rectangle`, we must create a new Rectangle object using the new `Point`. This instantiation of a `Rectangle` would create a burden every time we use a method on Rectangle.

Alternatively, we could reimplement `move` and `scale` inside `Rectangle`, delegating to the respective objects and creating the required new `Rectangle`. This composition does not scale well, however, as any new method added to `Rectangle` parents requires its own forwarding method, creating significant code duplication (e.g., if there were 100 methods associated with these data types, there would be 100 forwarding methods in `Rectangle`).

6.2.3 Problem 2: Return-value Polymorphism

Even if we were to use multiple inheritance, we still face a problem where the methods in `Point` and `Size`, which return new `Point` and `Size` objects respectively, must return `Rectangle` objects when called on `Rectangle` objects.

In Java, it is possible to parameterize arguments and return-values by using generic programming. For example, if `move` was a static method, we could parameterize it:

```
static T move<T>(T movable, int x, int y) { ... }
```

which would make the return-value the same type as the first argument. However, there is no syntax available to do the same with instance methods, i.e., there is no syntax to signify that a method returns the type of `this`.

Furthermore, even if this was possible, the method would not know how to create the new object. If we call `move` on a `Point`, we create a new `Point` with a constructor requiring two arguments. If we call `move` on a `Rectangle`, we create a new `Rectangle` with a constructor requiring four arguments.

We could copy the objects and return the copy after modifying their attributes accordingly, but such copying is not practical in Java because there is no generic way to copy objects, and, even if we added a `clone` method, we could not update the attributes of the clones because they are immutable as per the five rules presented in Section 4.2.1 of Chapter 4.

6.3 Solution: The Immutable Factory Method

We now propose a solution to implement the specifications while addressing the problems described above.

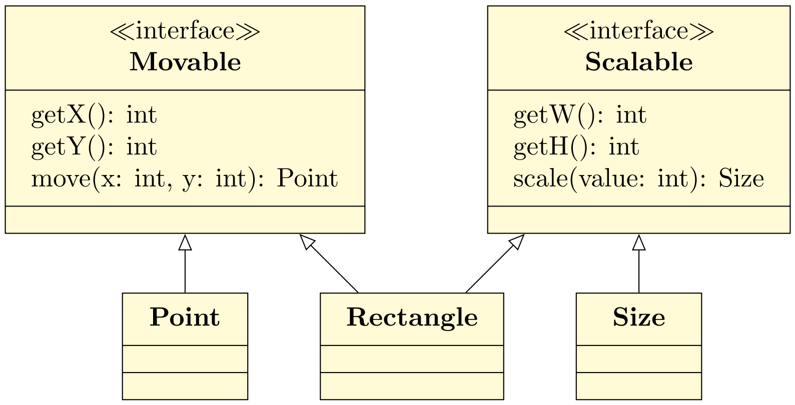

Figure 14: Specifications

6.3.1 Addressing Problem 1: Inheritance

The preferred mechanism in Java to address multiple inheritance problems is subtyping using interfaces. We could have `Movable` and `Scalable` interfaces and have `Rectangle` implement both, as illustrated in Figure 14.

The new problem with using interfaces is that every class that implements an interface must define every method declared in the interface. `Point` would define `move` and `Rectangle` would also have to define it, possibly through delegation. This need creates code duplication, as discussed earlier. Java 8 added the concept of default methods for interfaces, which solves this problem: we can implement `move` as a default method for `Movable`:

```
interface Movable {
    int getX();
    int getY();

    default Point move(int dx, int dy) {
        return new Point(this.getX() + x, this.getY() + y);
    }
}
```

Using the names `Movable` and `Scalable` also alleviates some of the problems with the *is-a* relationship between `Rectangle` and its parents. Now, instead of `Rectangle` *being a*

`Point` and a `Size`, it *behaves* like a `Point` and a `Size`.

6.3.2 Addressing Problem 2: Return-type Polymorphism

The `Movable` defined above has a significant problem: the method `move` necessarily returns a new `Point`. Thus, the `move` method in `Rectangle` would also return a new `Point` instead of a new `Rectangle`.

We could attempt to solve this problem with generic programming, but would face the constructor problem: how do we abstract the constructors for `Point`, `Size` and `Rectangle`? We leverage the Factory Method design pattern [33] with a common method that could be supplied by both `Point`, `Size`, and `Rectangle` classes to create new objects of the expected type:

```
interface Movable<T> {
    // ...
    T updateMovable(int x, int y);
}
```

which is a Factory Method that creates generic types from x and y values. The resulting type could be a `Point` or a `Rectangle`. Using this concept, we modify our `Movable` interface as such:

```
interface Movable<T> {
    int getX();
    int getY();

    T updateMovable(int x, int y);

    default T move(int dx, int dy) {
        return updateMovable(this.getX() + dx, getY() + dy);
    }
}
```

Now `Movable` is a generic interface that can create new instances of any type T when calling its `move` method. The implementor must define a `updateMovable` method that creates the expected instance using the given with x and y values. For `Point`, the implementation is trivial:

```
final class Point implements Movable<Point> {
    // ...

    @Override
    public Point updateMovable(int x, int y) {
        return new Point(x, y);
    }
}
```

`Rectangle` must implement both `Movable` and `Scalable`. Both of them require new `Rectangle` objects instantiated using only two arguments (either x and y or w and h). The following shows how we implement these Factory Methods in `Rectangle`:

```
final class Rectangle
    implements Movable<Rectangle>, Scalable<Rectangle> {
    // ...

    @Override
    public Rectangle updateMovable(int x, int y) {
        return new Rectangle(x, y, this.getW(), this.getH());
    }

    @Override
    public Rectangle updateScalable(int w, int h) {
        return new Rectangle(this.getX(), this.getY(), w, h);
    }
}
```

We use the original values of the `Rectangle` object on which the method is called, only replacing the appropriate old values with the supplied values. Thus, creating a new `Rectangle` using `updateMovable` keeps the original extent and creating a new Rectangle using `updateScalable` keeps the original position. Thus, we fulfill our specifications with few code repetition. The full listing of this implementation is available at https://github.com/wflageol-uqtr/ImmutableGeometry

6.4 Variants in other languages

The example described in the previous section is not arbitrary. We argue it is reasonable to require a geometry library to supply immutable objects. It is in fact the idiomatic style in some programming languages, such as Clojure or Haskell. Implementing our specifications is simpler in these languages. In this section, we look at specific language features that help with the above-stated problems in a more elegant way.

6.4.1 Clojure

Typing is a significant problem for our specifications, especially returning the expected type on inherited methods, such as `move`. Using a language without static typing solves this part of the problem. It does not, however, solve the constructor problem.

Some dynamic languages, such as JavaScript or Python, provide a generic way of copying objects. Yet, copying is not enough, because we must modify the immutable object to update some of its attributes.

In Clojure, this problem is solved with the `assoc` function. Clojure objects are actually immutable associative lists of keywords and values. The `assoc` function allows copying such a list while adding or replacing some of its values.

We define our `Point` and `Rectangle` structures as associative lists:

```
(defn make-point [x y]
        {:x x :y y})


    (defn make-rectangle [x y w h]
        {:x x :y y :w w :h h})
```

Both `make-point` and `make-rectangle` define the `x` and `y` values (`make-rectangle` also defining `w` and `h`). We can now implement the move function:

```
    (defn move [point dx dy]
        (assoc point
               :x (+ (point :x) dx)
               :y (+ (point :y) dy)))
```

This function creates a new list identical to the one provided as the first argument while offsetting the `x` and `y` values, which means, for a `point`, changing every value, effectively creating a new associative list and for `rectangle`, the values of `w` and `h` are untouched.

The Clojure implementation of our specifications is trivial and idiomatic.

6.4.2 Kotlin

We now implement our specifications with a statically typed language, Kotlin, which has many similarities to Java (for example, it is also implemented on top of the JVM), but includes some notable features that will simplify our solution. As with the other solutions, we address the constructor problem. As in Java, in Kotlin we cannot specify constructors in an interface and use the Factory Method design pattern again:

```
interface Movable<T> {
    val x: Int
    val y: Int


    fun updateMovable(newX: Int, newY: Int) : T
    fun move(moveX: Int, moveY: Int) : T {
        return updateMovable(moveX + x, moveY + y)
    }
}

class Point(override val x: Int, override val y: Int)
        : Movable<Point> {

    override fun updateMovable(newX: Int, newY: Int): Point {
        return Point(x, y)
    }
}

data class Rectangle(override val x: Int, override val y: Int,
                     override val w: Int, override val h: Int)
                 : Movable<Rectangle> {
    override fun updateMovable(newX: Int, newY: Int): Rectangle {
        return copy(x = newX, y = newY)
    }
}
```

This implementation is similar to the Java one, with one difference: in the `Rectangle` class, instead of creating a new `Rectangle` in `updateMovable` and provide every attribute of the object, we use the `copy` function of data classes in Kotlin. We still need the `updateMovable` Factory Method in this implementation because the `copy` function cannot be used on interfaces.

Data classes are classes designed to hold data and to be easily compared and copied[1]. We declare `Rectangle` as a data class, which assigns it a `copy` function. The `copy` function creates a duplicate of the object on which it is called and can modify the attributes of the

[1]https://kotlinlang.org/docs/data-classes.html

copy as needed, similar to how `assoc` works with associative lists in Clojure. By copying a rectangle and modifying its x and y values only, the rest of the attributes (w and h) remain unchanged. This feature reduces code redundancy and makes the design easier to modify.

Kotlin interfaces are actually traits and can contain method implementations, thus we added the `move` method to the `Movable` interface. The `move` method is available to both `Point` and `Rectangle`, because they inherit it from `Movable`, and, using generic programming, returns the expected type specified as the generic type of `Movable`. This implementation fulfills our specifications with minimal code duplication.

6.5 Discussion

We now discuss the general form of the proposed design pattern, its advantages and limitations, and how it is affected by some specific language features.

6.5.1 General Form

The pattern that we proposed is an adaptation of the Factory Method pattern [33] but allows changing the return type of methods when subtyping.

Figure 15 shows the general form of the design pattern. In this figure, `TypeABehaviour` and `TypeBBehaviour` are generic interfaces with a generic type T defined when implementing them. For example, `TypeA` implements `TypeABehaviour<TypeA>`, defining the generic type to be itself. Similarly, `AggregateType` implements the same interface but defines the generic type also to be itself (`TypeABehaviour<AggregateType>`) so that the `typeACreator` method returns `TypeA` objects when called on a `TypeA` object, and returns `AggregateType` objects when called on an `AggregateType` object.

When programming with transitive immutability, methods that return new objects of the type of the receiver are common. These methods allow incremental changes to objects without mutating them. Our proposed pattern allows inheriting and reusing such methods in a type hierarchy.

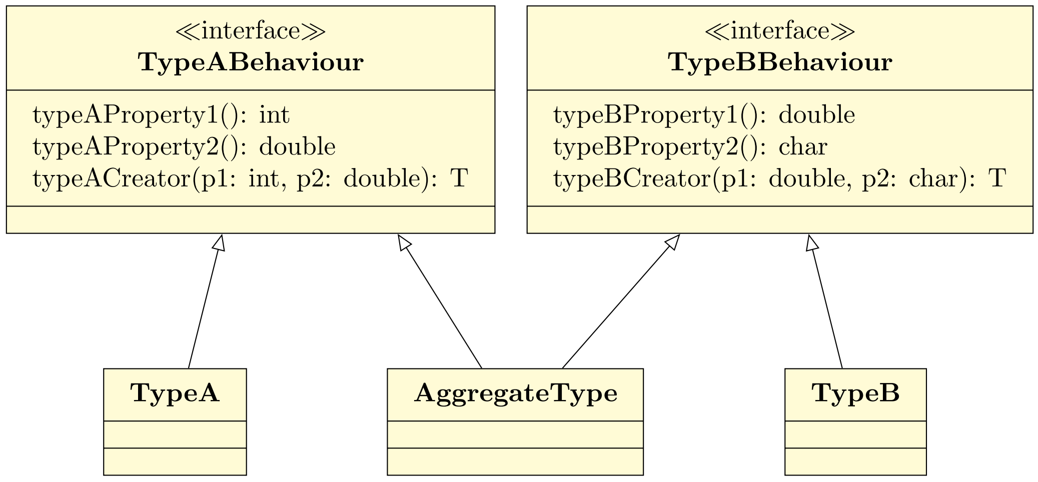

Figure 15: General Form

6.5.2 Disadvantages and Limitations

Using this design pattern in practice comes with some disadvantages and limitations. We discuss one main disadvantage and one main limitation of this pattern. The disadvantage concerns the method declarations. Behaviour must be encapsulated in interfaces or traits and defined by default methods, which in some languages involves a different syntax than regular method definition. In Java, default methods require the `default` keyword. Generic programming must also be used to ensure that the methods can be inherited by any type. The style can differ significantly from the usual way of defining methods.

If we take the `move` method from our running example, it could be defined using the usual style in Java like this:

```
Rectangle move(int dx, int dy) {
    return new Rectangle(this.getX() + dx, this.getY() + dy,
                         this.getW(), this.getH());
}
```

This method would be found in the `Rectangle` object. When applying our design pattern, however, the method must be declared in a `Movable` interface and look like this:

```
default T move(int dx, int dy) {
        return updateMovable(this.getX() + x, this.getY() + y);
    }
```

Furthermore, when using our design pattern, object creation no longer uses the default language mechanism (i.e., `new` in Java and Kotlin), but instead must use the factory methods included in the behaviour objects (e.g., `updateMovable` in our example).

The difference between the two approaches could impact readability and understandability and make it more difficult for a new developer to understand the program and create a steeper learning curve.

The limitation of the design pattern concerns scalability. While the design pattern exists to improve the scalability of a naive approach by reusing methods, a problem remains with object creation. Consider the following code:

```
    @Override
    public Rectangle updateMovable(int x, int y) {
        return new Rectangle(x, y, this.getW(), this.getH());
    }
```

We must supply the constructor with the W and H values of the current `Rectangle`, which implies that every attribute that must be conserved as-is must be assigned in this method (and every other factory method). If we add new attributes to `Rectangle`, we must modify every factory method that creates `Rectangle` objects to conserve these attributes. This recurring modification can be tedious and possibly error-prone, as a developer could forget to update one or all of them at some point, resulting in some perplexing bugs where some, but not all, the attributes of an object would be conserved between changes. The result could also become unwieldy at some point if dozens of attributes were involved.

6.5.3 Language Feature: Functional Updating

The disadvantages and limitations are affected by the implementation language of our design pattern. In the above examples, we considered Java. Other languages have features that mitigate or remove parts of these limitations. Functional updating is one such feature adopted by multiple OOP languages, such as C# and Kotlin.

The Kotlin example in Section 6.4.2 makes use of functional updating to mitigate the scalability limitation. Here is how the same factory method example discussed above could be done in Kotlin:

```
override fun updateMovable(newX: Int, newY: Int): Rectangle {
    return copy(x = newX, y = newY)
}
```

We do not specify any other attributes than the ones that change, similar to how normal mutation would be done. There is no scalability issue, as adding new attributes to `Rectangle` would not affect the implementation of this method. In the Clojure example (Section 6.4.1), the functional updating operation is the `assoc` function. Other functional languages such as Haskell, OCaml, and F# all have this feature. Functional updating is the main replacement for mutation in immutable programming.

6.5.4 Dynamic vs Static Typing

The disadvantage concerning the syntax required to define reusable immutable methods relates to static typing. As discussed in Section 6.2.3, we must express that `Movable` applies to any implementing type and should return its own type, which requires the use of generic programming in statically-typed languages.

In dynamically-typed languages, we can consider the Clojure implementation of the `move` function:

```
(defn move [point dx dy]
    (assoc point
           :x (+ (point :x) dx)
           :y (+ (point :y) dy)))
```

Because no type declaration is required on functions, there is no need for a special syntax when declaring reusable immutable methods. This style is idiomatic in Clojure.

Statically-typed languages have the additional burden of expressing a sound type structure for the compiler, which requires specifying a sound return type for the method, which in turn leads to the use of generic programming when that return type can vary depending on the receiver.

6.6 Common Lisp Implementations

In this section, we will discuss a Common Lisp implementation of this design pattern. We will then extend the language by adding the functional updating feature and observe how it impacts our implementation.

Common Lisp is a dynamically-typed general programming languages with meta-programming capabilities. Those meta-programming capabilities will allow us to add functional updating to the language without writing a third-party tool.

Unlike Clojure, Common Lisp supports full OOP through its Common Lisp Object System (CLOS) library. The combination of dynamic typing, OOP support, and meta-programming makes Common Lisp a prime language for this experiment.

6.6.1 Implementation in base Common Lisp

Our first implementation will only use Common Lisp features included in the base language, without any added macros.

Implementing the data structures is straightforward, thanks to multiple inheritance support:

```
(defclass point ()
  ((x :reader x :initarg :x)
   (y :reader y :initarg :y)))

(defclass size ()
  ((w :reader w :initarg :w)
   (h :reader h :initarg :h)))
```

```
(defclass rectangle (point size) ())
```

`Rectangle` inherits both the attributes of `Point` and `Size`. Note that this is not necessary and only saves a few lines of code. We could have declared the `x`, `y`, `w`, and `h` attributes manually in `Rectangle` without having to use inheritance at all.

Common Lisp does not support functional updating for objects, so we need to implement the Factory Methods to update objects:

```
(defgeneric update-movable (movable x y))


(defmethod update-movable ((movable point) x y)
  (make-instance 'point :x x :y y))


(defmethod update-movable ((movable rectangle) x y)
  (make-instance 'rectangle
                 :x x :y y
                 :w (w movable) :h (h movable)))


(defgeneric update-scalable (scalable w h))


(defmethod update-scalable ((scalable size) w h)
  (make-instance 'size :w w :h h))


(defmethod update-scalable ((scalable rectangle) w h)
  (make-instance 'rectangle
                 :w w :h h
                 :x (x scalable) :y (y scalable)))
```

This implementation is similar to the Java or Kotlin implementations presented in Sections 6.3 and 6.4.2. One notable difference is the lack of `movable` and `scalable` interfaces.

CLOS does not associate methods to classes or interfaces, but instead to generic functions. Methods can then target any type of objects, without the need of inheritance or subtyping.

We can then implement `move` and `scale` as functions:

```
(defun move (movable dx dy)
  (update-movable movable
   (+ dx (x movable))
   (+ dy (y movable))))


(defun scale (scalable scale)
  (update-scalable scalable
    (* scale (w scalable))
    (* scale (h scalable))))
```

The lack of static typing makes this implementation shorter and simpler than the Java or Kotlin ones. It also does not share the method declaration disadvantage: the `move` and `scale` functions look like idiomatic Common Lisp functions. However, it still retains the scalability limitation. Adding attributes to any of the data structures would require modifying the various `update` Factory Methods to include these new attributes.

6.6.2 Extending Common Lisp

To solve the scalability issue, we must remove the need for these `update` Factory Methods. In Section 6.4.1, we presented a Clojure implementation which made use of functional updating instead of Factory Methods. However, Clojure uses property lists instead of objects and could use the `assoc` function to operate on those lists. Our Common Lisp implementation uses objects and no language feature currently exist to update an object in a functional way. Thanks to Common Lisp's meta-programming capabilities, we can add such a feature to the language.

To add functional updating, we first need a way to clone an object, similar to the `copy` method of data classes in Kotlin. We can create a `clone-object` function by using the closer-mop[2] compatibility layer in Common Lisp. This layer is required to standardize the Meta-Object Protocol (MOP) across the many Common Lisp implementations. Specifically, we need it to obtain the list of attributes in an object. Our `clone-object` function looks like this:

```
(defun clone-object (instance)
  (let* ((class (class-of instance))
         (clone (allocate-instance class)))
    (dolist (slot-name
              (mapcar #'closer-mop:slot-definition-name
                      (closer-mop:class-slots class)))
      (when (slot-boundp instance slot-name)
        (setf (slot-value clone slot-name)
              (slot-value instance slot-name))))
    clone))
```

This function creates a shallow copy of a given object. A shallow copy is sufficient when working with immutable code, as embedded data structures are also immutable. We could use our `clone-object` function to implement the `move` and `scale` functions directly. Instead, we will go one step further and create a macro with a structure idiomatic to Common Lisp. We create the `with-new` macro:

```
(defmacro with-new (slots instance &body body)
  (let ((instance-sym (gensym)))
    `(let ((,instance-sym (clone-object ,instance)))
       (with-slots ,slots ,instance-sym
         ,@body
         ,instance-sym))))
```

This macro is structurally similar to the `with-slots` macro, which is used to update the attributes of an object. Our macro does the same, but instead of modifying the object

[2] https://github.com/pcostanza/closer-mop

in place, creates a copy and modifies the copy instead. We can use `with-new` to implement the `move` and `scale` functions:

```
(defun move (movable dx dy)
  (with-new (x y) movable
    (incf x dx)
    (incf y dy)))


(defun scale (scalable scale)
  (with-new (w h) scalable
    (setf w (* scale w))
    (setf h (* scale h))))
```

We no longer need the `update` Factory Methods and our code looks like idiomatic Common Lisp code. This implementation solves the scalability issue, as adding new attributes to the data structures will not impact the rest of the code. Adding new functions is also very simple, as there is almost no difference between the implementation of an immutable function and a mutable one (the only difference is replacing `with-new` with `with-slots`).

6.7 Conclusion

In this chapter, we discussed problems with the combination of OOP and immutability regarding the reuse of immutable methods through subtyping.

We analysed a naive implementation of a running example and found that it had code duplication and scalability problems, caused by a need for multiple inheritance and return-value polymorphism. We proposed a design pattern as a solution to mitigate those problems, based on the Factory Method pattern.

We first presented our design pattern in Java to show its general implementation and discussed the limitations of this implementation. While it does mitigate the problems with a naive approach, it introduces some complexity, decreased understandability and readability, as well as an increased learning curve.

We then presented its implementations in other languages, namely Clojure and Kotlin. We reported that the Clojure implementation was idiomatic to the language thanks to the presence of functional updating. The Kotlin implementation was closer to the Java one, as they both are statically-typed languages. Kotlin could also make use of functional updating to solve the scalability problem present in the Java implementation.

When finally presented an implementation of the design pattern in Common Lisp, showing that it had the same scalability problem present in other implementations, except Clojure. We then extended the language by adding a functional updating feature. Using this new feature, create a new implementation with idiomatic Common Lisp code and no scalability issue.

We conclude that functional updating is an essential feature when working with immutability. Languages that support functional programming, such as F#, Haskell, Kotlin, and OCaml, all provide this feature. In recent years, it has been added to Microsoft C# to increase immutability support. Java has also recently increased its immutability support but has yet to add a form of functional updating.

Static typing could limit reusability and understandability but has other advantages that may well outweigh this limitation (most functional languages are statically typed).

Chapter 7

Conclusion and Future Work

In this thesis, we have presented various language features which could improve OOP. We first performed a **mapping study** of the literature to catalogue the language features suggested by researchers to improve OOP design patterns. We then **explored** the impact of immutability on OOP software development. We also **studied** a set of language features added to C# that improved immutability support. We finally **solved** a problem in software that combines OOP and immutability by introducing a design pattern, and improving that solution with functional updating. What follows is a summary of our findings.

7.1 Contributions and Findings

In Chapter 3, we presented a mapping study of the primary studies on language features to improve Object-Oriented Programming design pattern implementations. Our objective in finding these language features was to improve the OOP paradigm itself. Design patterns are solutions to recurring problems, and thus, including features in an OOP language that makes it easier to solve these problems makes the paradigm and languages themselves easier to use for practical purposes. The main contribution of this chapter was a catalogue of 18 language features claiming to improve design pattern implementations. This catalogue can be used as a road map of existing literature that we intend to use to research improvements to the Object-Oriented Programming paradigm. For the rest of the thesis, we chose to focus on one of the least explored language features in our catalogue: immutability.

Chapter 4 reported a multi-method exploratory empirical study on the impact of immutability on object-oriented software development. We performed an experiment on 67 undergraduate students to assess the impact of using immutability in OOP software development. The treatment group had to develop a program using OOP and immutability, while the control group would develop the same software, but using classical mutable OOP. We found no significant negative impact related to using immutability on programs or participants. By extracting measures from the code, we observed an increase in number of methods and a decrease in method size as well as number of comments. The survey of the participants showed lower perceived workload, difficulty, and complexity by the participants in the treatment group. Teams using immutability perceived the difficulty of the project to be lower than the other teams. **We concluded that the disadvantages of using immutability in the context of OOP are outweighed by the advantages.**

Chapter 5 was a multi-method empirical study on the impact of immutability-related features added to C# from 2017 to 2022. We performed an experiment on 12 participants, 10 of whom were graduate students, and 2 professional developers. The treatment group would analyse and extend a program made using the new features, while the control group would do the same on a similar program made without using the new features. The treatment group programs showed a higher Maintainability Index, lower Cyclomatic Complexity, and lower Number of Executable Lines of Code than the control group programs. We expected that some participants in the control group would choose a Visitor-based approach to extend the program, but every participant instead used a pattern matching approach. **We concluded that the immutability-related language features added to C# have a positive impact when used to implement immutable programs.**

Finally, Chapter 6 presented a problem that emerges with the combination of OOP and immutability. We analysed a naive implementation of a running example and found that it had code duplication and scalability problems. We presented a design pattern to solve this problem in existing OOP languages. Our first implementation was in Java, which mitigated the problem, but introduced increased complexity, decreased understandability and readability, as well as an increased learning curve. We reported that the Clojure implementation was idiomatic to the language thanks to the presence of functional updating. The Kotlin implementation was closer to the Java one, as they both are statically-typed languages. Kotlin could also make use of functional updating to solve the scalability problem present in the Java implementation.

We identified dynamic typing and functional updating as two key features which can solve the problem. To test these findings, we implemented the new design pattern in Common Lisp, a dynamically-typed language with OOP support. We extended Common Lisp using its meta-programming capabilities (i.e., macros) and showed that by adding functional updating to the language, we could solve the problem without the need for a design pattern and without any increased complexity or scalability issues.

The main contributions of this thesis are as follows:

(1) A catalogue of 18 language features suggested to improve OOP design implementations, which formed the basis of a paper submitted to the Information and Software Technology journal.

(2) The first exploratory study on the impact of immutability on OOP, which formed the basis of a paper submitted to the Empirical Software Engineering journal.

(3) An empirical study on the impact of adding immutability-related features to C#, which formed the basis of a paper submitted to a special issue of Journal of Systems and Software.

(4) A new design pattern to solve a problem that emerges with the combination of OOP and immutability, which formed the basis of a paper submitted to the EuroPLoP 2023 conference.

(5) An extension to Common Lisp which adds functional updating to the language.

7.2 Future Work

There are many open research opportunities that arise from the results of this thesis. We discuss some of them in what follows.

Out of the 18 language features catalogued in Chapter 3, we only explored immutability in this thesis. The other 17 features could each make for interesting research topics in the future, in particular studying further their interaction with design patterns and with the OOP paradigm. We find Reactive Programming particularly promising in that respect and have already started studying how its concepts can be integrated into OOP to solve the Observer pattern's underlying problem.

An alternative to studying the language features directly would be to study individual design patterns. The features mapped in Chapter 3 each impact specific design patterns, thus future research could focus on specific patterns and discuss the features that impact them and perhaps how to combine them to improve the pattern or solve its underlying problem.

We want to perform a quasi-replication of Chapter 4 using other programming languages, including object-oriented programming languages with immutable properties. In particular, we want to focus on the impact of immutability on code granularity and understandability, as well as compare the workload between mutable and immutable software development.

Considering the relative difficulty faced by the students in participating in the experiment of Chapter 5, it would be interesting to replicate that study using only professional developers instead. While there is a risk that professionals by the features that they use regularly, they would bring additional insight into the impact of the new features. In particular, a replication study could confirm whether or not the complexity of the code is affected by the presence of the new features. Future studies could also consider other language features, such as the functional-style LINQ in C# or Streams in Java.

We intend to continue researching interactions between OOP and functional programming. When combining the two approaches, problems and solutions in the form of other design patterns may emerge, giving us more insight into how to improve OOP languages and which features are needed. Perhaps a new paradigm could also emerge from the combination of OOP and functional programming in the future.

Pattern matching is a good example of a functional programming feature which interacts with OOP in an interesting way. As discussed in Chapter 5, the Visitor design pattern can be replaced by pattern matching in certain situations. In Chapter 2, we also discussed multimethods, or multiple dispatch, which can also be used to replace the Visitor pattern. Interestingly, multimethods were not featured in our feature catalogue in Chapter 3. Future work could study the interaction between pattern matching, multimethods, and the Visitor pattern to find the best way of solving its underlying problem.

At various points in this thesis, we discussed Rust, a relatively new programming language with a unique take on mutability. Rust uses an ownership system to determine which part of a program can mutate an object, and only allows one “owner” of any object. Other parts of the program can only “read” the object, unless they are given ownership. We are interested in studying the impact of this ownership system on OOP and how it interacts with structural design patterns.

Bibliography

[1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1984.

[2] G. Kiczales et al. "Aspect-oriented programming". In: *Proceedings of the European Conference on Object-Oriented Programming* (1997), pp. 220–242.

[3] Khalid Aljasser. "Implementing design patterns as parametric aspects using ParaAJ". In: *Computer Languages, Systems & Structures* 45 (2016), pp. 1–15.

[4] 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 (Apr. 2016), pp. 1–15.

[5] Apostolos Ampatzoglou, Sofia Charalampidou, and Ioannis Stamelos. "Design pattern alternatives: what to do when a GoF pattern fails". en. In: *Proceedings of the 17th Panhellenic Conference on Informatics - PCI '13*. Thessaloniki, Greece: ACM Press, 2013, p. 122.

[6] 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.

[7] Eyvind W. Axelsen, Fredrik Sørensen, and Stein Krogdahl. "A reusable observer pattern implementation using package templates". In: *Proceedings of the 8th workshop on Aspects, components, and patterns for infrastructure software - ACP4IS '09*. the 8th workshop. Charlottesville, Virginia, USA: ACM Press, 2009, p. 37.

[8] Pavol Baca and Valentino Vranic. "Replacing Object-Oriented Design Patterns with Intrinsic Aspect-Oriented Design Patterns". In: *2011 Second Eastern European Regional Conference on the Engineering of Computer Based Systems.* 2011 2nd Eastern European Regional Conference on the Engineering of Computer Based Systems (ECBS-EERC 2011). Bratislava, Slovakia: IEEE, Sept. 2011, pp. 19–26.

[9] B. Bafandeh Mayvan, A. 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 (Mar. 2017), pp. 93–118.

[10] Marc Bartsch and Rachel Harrison. "Design Patterns with Aspects: A case study". In: *Proceedings of the 12th European Conference on Pattern Languages of Programs.* EuroPLoP '2007. ACM, 2007, pp. 797–810.

[11] Ruslan Batdalov and Oksana Nikiforova. "Towards Easier Implementation of Design Patterns". In: *The Eleventh International Conference on Software Engineering Advances.* ICSEA 2016. Rome, Italy: International Academy, Research, and Industry Association, 2016.

[12] M.L. Bernardi and G.A. Di Lucca. "Improving Design Pattern Quality Using Aspect Orientation". In: *13th IEEE International Workshop on Software Technology and Engineering Practice (STEP'05).* 13th IEEE International Workshop on Software Technology and Engineering Practice (STEP'05). Budapest, Hungary: IEEE, 2005, pp. 206–218.

[13] J. Bloch. *Effective Java, Second Edition.* Addison-Wesley, 2008.

[14] J. Borella. "The observer pattern using aspect oriented programming". In: *Proceedings of the Viking Pattern Languages of Programs.* Viking PLOP. 2003.

[15] John Boyland, James Noble, and William Retert. "Capabilities for Sharing". In: *ECOOP 2001 — Object-Oriented Programming.* Ed. by Jørgen Lindskov Knudsen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 2–27.

[16] Yegor Bugayenko and Sergey Zykov. "The Impact of Object Immutability on the Java Class". In: *24th International Conference on Knowledge-Based and Intelligent Information & Engineering Systems*. Elsevier, 2020.

[17] Eden Burton and Emil Sekerinski. "Using dynamic mixins to implement design patterns". In: *Proceedings of the 19th European Conference on Pattern Languages of Programs - EuroPLoP '14*. the 19th European Conference. Irsee, Germany: ACM Press, 2014, pp. 1–19.

[18] Nelio Cacho, Claudio Sant'anna, Eduardo Figueiredo, Francisco Dantas, Alessandro Garcia, and Thais Batista. "Blending design patterns with aspects: A quantitative study". In: *Journal of Systems and Software* 98 (Dec. 2014), pp. 117–139.

[19] M. Ceccato and P. Tonella. "Measuring the effects of software aspectization". In: *1st Workshop on Aspect Reverse Engineering* (2004).

[20] C. Chambers, B. Harrison, and J. Vlissides. "A Debate on Language and Tool Support for Design Patterns". In: *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2000).

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

[22] M. Coblenz, W. Nelson, J. Aldrich, B. Myers, and J. Sunshine. "Glacier: Transitive Class Immutability for Java". In: *39th International Conference on Software Engineering*. IEEE/ACM, 2017.

[23] Michael Coblenz, Joshua Sunshine, Jonathan Aldrich, Brad Myers, Sam Weber, and Forrest Shull. "Exploring Language Support for Immutability". In: *38th International Conference on Software Engineering*. Austin, Texas: Association for Computing Machinery, 2016, pp. 736–747.

[24] Arie van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. "Refactoring test code". In: *2nd International Conference on Extreme Programming and Flexible Processes (XP2001)*. 2001.

[25] Daniel von Dincklage. “Iterators and encapsulation”. In: *Proceedings 33rd International Conference on Technology of Object-Oriented Languages and Systems* (2000).

[26] J. Dolado, M. Harman, M. Otero, and L. Hu. “An empirical investigation of the influence of a type of side effects on program comprehension”. In: *IEEE Transactions on Softare Engineering.* Vol. 29. 7. 2003, pp. 665–670.

[27] Tobias Dürschmid. “Design pattern builder: a concept for refinable reusable design pattern libraries”. In: *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity.* SPLASH ’16: Conference on Systems, Programming, Languages, and Applications: Software for Humanity. Amsterdam Netherlands: ACM, Oct. 20, 2016, pp. 45–46.

[28] N. El Maghawry and A. R. Dawood. “Aspect oriented GoF design patterns”. In: *The 7th International Conference on Informatics and Systems.* INFOS. 2010, pp. 1–7.

[29] Jonathan Eyolfson and Patrick Lam. “How C++ Developers Use Immutability Declarations: An Empirical Study”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering.* 2019, pp. 362–372.

[30] Jose M. Felix and Francisco Ortin. “Aspect-Oriented Programming to Improve Modularity of Object-Oriented Applications”. In: *Journal of Software* 9.9 (Sept. 1, 2014), pp. 2454–2460.

[31] L. Ferreira and C. M. F. Rubira. “The Reflective State Pattern”. In: *Proceedings of the 5th Conference on Pattern Languages of Programs (PLOT ’98)* (1998).

[32] Harold Fortuin. “A Modern, Compact Implementation of the Parameterized Factory Design Pattern.” In: *The Journal of Object Technology* 9.1 (2010), p. 57.

[33] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1994.

[34] Alessandro Garcia, Cláudio Sant'Anna, Eduardo Figueiredo, Uirá Kulesza, Carlos Lucena, and Arndt von Staa. "Modularizing Design Patterns with Aspects: A Quantitative Study". In: *Transactions on Aspect-Oriented Software Development I*. Ed. by Awais Rashid and Mehmet Aksit. Vol. 3880. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 36–74.

[35] Taher Ahmed Ghaleb, Khalid Aljasser, and Musab A. Alturki. "An Extensible Compiler for Implementing Software Design Patterns as Concise Language Constructs". In: *International Journal of Software Engineering and Knowledge Engineering* 31.7 (July 2021), pp. 1043–1067.

[36] Jeremy Gibbons. "Design patterns as higher-order datatype-generic programs". In: *Proceedings of the 2006 ACM SIGPLAN workshop on Generic programming - WGP '06*. the 2006 ACM SIGPLAN workshop. Portland, Oregon, USA: ACM Press, 2006, p. 1.

[37] Rosario Giunta, Giuseppe Pappalardo, and Emiliano Tramontana. "AODP: refactoring code to provide advanced aspect-oriented modularization of design patterns". In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing - SAC '12*. the 27th Annual ACM Symposium. Trento, Italy: ACM Press, 2012, p. 1243.

[38] Rosario Giunta, Giuseppe Pappalardo, and Emiliano Tramontana. "Using aspects and annotations to separate application code from design patterns". In: *Proceedings of the 2010 ACM Symposium on Applied Computing - SAC '10*. the 2010 ACM Symposium. Sierre, Switzerland: ACM Press, 2010, p. 2183.

[39] João L. Gomes and Miguel P. Monteiro. "Design pattern implementation in object teams". In: *Proceedings of the 2010 ACM Symposium on Applied Computing - SAC '10*. the 2010 ACM Symposium. Sierre, Switzerland: ACM Press, 2010, p. 2119.

[40] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. "Uniqueness and Reference Immutability for Safe Parallelism". In: *SIGPLAN Not.* 47.10 (Oct. 2012), pp. 21–40.

[41] Q. Hachani and D. Bardou. “Using aspect-oriented programming for design patterns implementation”. In: *Workshop on Reuse in Object-Oriented Information Systems Design.* 2002.

[42] Harry Hakonen, Ville Leppänen, Timo Raita, Salakoski Tapio, and Jukka Teuhola. “Improving object integrity and preventing side effects via deeply immutable references”. In: *6th Fenno-Urgic Symposium on Software Technology (FUSST’ 99).* 1999.

[43] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series).* USA: Elsevier Science Inc., 1977.

[44] Jan Hannemann and Gregor Kiczales. “Design pattern implementation in Java and AspectJ”. In: *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming* (2002), pp. 161–173.

[45] Jan Hannemann and Gregor Kiczales. “Design pattern implementation in Java and aspectJ”. In: *ACM SIGPLAN Notices* 37.11 (Nov. 17, 2002), pp. 161–173.

[46] Sandra G. Hart and Lowell E. Staveland. “Development of NASA-TLX (Task Load Index): Results of Empirical and Theoretical Research”. In: *Human Mental Workload.* Ed. by Peter A. Hancock and Najmedin Meshkati. Vol. 52. Advances in Psychology. North-Holland, 1988, pp. 139–183.

[47] P. Helland. “Immutability changes everything”. In: 59.1 (2016), pp. 64–70.

[48] Bilal Hussein, Aref Mehanna, and Yahia Rabih. “Visitor Design Pattern Using Reflection Mechanism:” in: *International Journal of Software Innovation* 8.1 (Jan. 1, 2020), pp. 92–107.

[49] Joshua Kerievsky. *Refactoring to Patterns.* Addison-Wesley Professional, 2004.

[50] B. Kitchenham and S. Charters. “Guidelines for Performing Systematic Literature Review in Software Engineering”. In: *Technical Report EBSE 2007-001, Keele Univ. and Durham Univ. Joint Report* (2007).

[51] B.A. Kitchenham. “Procedures for Undertaking Systematic Reviews”. In: *Joint Technical Report, Computer Science Department, Keele University (TR/SE-0401) and National ICT Australia Ltd. (0400011T.1)* (2004).

[52] Günter Kniesel and Dirk Theisen. “JAC—Access right based encapsulation for Java”. In: *Software: Practice and Experience* 31.6 (2001), pp. 555–576.

[53] Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. “Synthesizing Object-Oriented and Functional Design to Promote Re-Use”. In: *Proceedings of the 12th European Conference on Object-Oriented Programming.* ECCOP ’98. ACM, 1998, pp. 91–113.

[54] Justin Kruger and David Dunning. “Unskilled and unaware of it: how difficulties in recognizing one’s own incompetence lead to inflated self-assessments.” In: *Journal of personality and social psychology* 77.6 (1999), p. 1121.

[55] Martin Kuhlemann, Sven Apel, Marko Rosenmüller, and Roberto Lopez-Herrejon. “A Multiparadigm Study of Crosscutting Modularity in Design Patterns”. In: *Objects, Components, Models and Patterns.* Ed. by Richard F. Paige and Bertrand Meyer. Red. by Will van der Aalst, John Mylopoulos, Norman M. Sadeh, Michael J. Shaw, and Clemens Szyperski. Vol. 11. Series Title: Lecture Notes in Business Information Processing. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 121–140.

[56] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development.* Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997.

[57] Douglas Lea. *Concurrent programming in Java: design principles and patterns.* Addison-Wesley Professional, 2000.

[58] Barbara H. Liskov and Jeannette M. Wing. “A behavioral notion of subtyping”. In: *ACM Transactions on Programming Languages and Systems* (1994), pp. 1811–1841.

[59] Liu Jicheng, Yin Hui, and Wang Yabo. “A novel implementation of observer pattern by aspect based on Java annotation”. In: *2010 3rd International Conference on Computer Science and Information Technology.* 2010 3rd IEEE International Conference on Computer Science and Information Technology (ICCSIT 2010). Chengdu, China: IEEE, July 2010, pp. 284–288.

[60] F. S. Løkke. “Scala & design patterns”. In: *Master’s thesis, University of Aarhus* (2009).

[61] Yun Mai and Michel Champlain. “Reflective Visitor Pattern”. In: *European Conference on Pattern Languages of Programs.* EuroPLoP. ACM, 2001, pp. 299–316.

[62] Ingo Maier and Martin Odersky. “Higher-Order Reactive Programming with Incremental Lists”. In: *ECOOP 2013 – Object-Oriented Programming.* Ed. by Giuseppe Castagna. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 707–731.

[63] S. Majukmder, P. Mody, and T Menzies. “Revisiting process versus product metrics: a large scale analysis”. In: *Empirical Software Engineering* 27.60 (2022).

[64] Robert C. Martin. *Design Principles and Design Patterns.* 2000. URL: https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf.

[65] T.J. McCabe, McCabe Associates, and IEEE Computer Society. *Structured Testing.* Ieee computer society. IEEE Computer Society Press, 1983.

[66] James D. McCaffrey and Adrian Bonar. “A Case Study of the Factors Associated with Using the F# Programming Language for Software Test Automation”. In: *2010 Seventh International Conference on Information Technology: New Generations.* 2010, pp. 1009–1013.

[67] Peter Norvig. *Design Patterns in Dynamic Programming.* 1996. URL: https://norvig.com/design-patterns/design-patterns.pdf.

[68] Peter Norvig. “Techniques for Automatic Memoization with Applications to Context-Free Parsing”. In: *Comput. Linguist.* 17.1 (Mar. 1991), pp. 91–98.

[69] Bruno C.d.S. Oliveira, Meng Wang, and Jeremy Gibbons. “The visitor pattern as a reusable, generic, type-safe component”. In: *ACM SIGPLAN Notices* 43.10 (Oct. 27, 2008), pp. 439–456.

[70] P. Oman and J. Hagemeister. “Metrics for assessing a software system’s maintainability”. In: *Proceedings Conference on Software Maintenance 1992*. 1992, pp. 337–344.

[71] K. Peterson, S. Vakkalanka, and L. Kuzniarz. “Guidelines for conducting systematic mapping studies in software engineering: An update”. In: *Information and Software Technology* 64 (2015), pp. 1–18.

[72] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[73] Sara Porat, Marina Biberstein, Larry Koved, and Mendelson Bilha. “Automatic detection of immutable fields in Java”. In: *2000 Conference of the Center for Advanced Studies on Collaborative Research*. Vol. 10. IBM Press, 2000.

[74] Rolando P. Reyes, Oscar Dieste, Efraìn R. Fonseca, and Natalia Juristo. “Statistical Errors in Software Engineering Experiments: A Preliminary Literature Review”. In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE ’18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 1195–1206.

[75] L. Rosenberg, T. Hammer, and J. Shaw. “Software Metrics and Reliability”. In: *Proceedings of IEEE International Symposium on Software Reliability Engineering*. IEEE, 1998.

[76] Kazunori Sakamoto, Hironori Washizaki, and Yoshiaki Fukukaza. “Extended design patterns in new object-oriented programming languages”. In: *Proceedings of the International Conference on Software Engineering and Knowledge Engineering*. SEKE. Eslevier, 2013, pp. 600–605.

[77] Guido Salvaneschi, Sven Amann, Sebastian Proksch, and Mira Mezini. “An empirical study on program comprehension with reactive programming”. In: *Proceedings of*

the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. SIGSOFT/FSE'14: 22nd ACM SIGSOFT Symposium on the Foundations of Software Engineering. Hong Kong China: ACM, Nov. 11, 2014, pp. 564–575.

[78] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. "REScala: Bridging between Object-Oriented and Functional Style in Reactive Applications". In: *Proceedings of the 13th International Conference on Modularity*. MODULARITY '14. Lugano, Switzerland: Association for Computing Machinery, 2014, pp. 25–36.

[79] Guido Salvaneschi, Sebastian Proksch, Sven Amann, Sarah Nadi, and Mira Mezini. "On the Positive Effect of Reactive Programming on Software Comprehension: An Empirical Study". In: *IEEE Transactions on Software Engineering* 43.12 (2017), pp. 1125–1143.

[80] Cláudio Sant'Anna, Alessandro Garcia, Uirá Kulesza, Carlos Lucena, and Arndt von Staa. "Design patterns as aspects: a quantitative assessment". In: *Journal of the Brazilian Computer Society* 10.2 (Nov. 2004), pp. 42–55.

[81] Mahnoosh Shahidi, Mehrdad Ashtiani, and Morteza Zakeri-Nasrabadi. "An automated extract method refactoring approach to correct the long method code smell". In: *Journal of Systems and Software* 187 (2022), p. 111221.

[82] S. S. SHAPIRO and M. B. WILK. "An analysis of variance test for normality (complete samples)†". In: *Biometrika* 52.3-4 (Dec. 1965), pp. 591–611.

[83] Raed Shatnawi. "A Quantitative Investigation of the Acceptable Risk Levels of Object-Oriented Metrics in Open-Source Systems". In: *IEEE Transactions on Software Engineering* 36.2 (2010), pp. 216–225.

[84] Matthias Springer, Hidehiko Masuhara, and Robert Hirschfeld. "Classes as Layers: Rewriting Design Patterns with COP: Alternative Implementations of Decorator, Observer, and Visitor". In: *Proceedings of the 8th International Workshop on Context-Oriented Programming*. ECOOP '16: European Conference on Object-Oriented Programming. Rome Italy: ACM, July 17, 2016, pp. 21–26.

[85] J. Stylos and S. Clarke. “Usability Implications of Requiring Paramters in Objects’ Constructors”. In: *International Conference on Software Engineering.* 2007.

[86] Antero Taivalsaari. “On the notion of object”. In: *Journal of Systems and Software* 21 (1993), pp. 3–16.

[87] R Teebiga and S Senthil Velan. “Comparison of applying design patterns for functional and non-functional design elements in Java and AspectJ programs”. In: *2016 International Conference on Advanced Communication Control and Computing Technologies (ICACCCT).* 2016 International Conference on Advanced Communication Control and Computing Technologies (ICACCCT). Ramanathapuram, India: IEEE, May 2016, pp. 751–757.

[88] Matthew F. Tennyson. “A study of the data synchronization concern in the Observer design pattern”. In: *2010 2nd International Conference on Software Technology and Engineering.* 2010 2nd International Conference on Software Technology and Engineering (ICSTE 2010). San Juan, PR, USA: IEEE, Oct. 2010, p. 5608911.

[89] “The Pascal Programming Languages”. In: *ANSI/IEEE 770X3.97-1983.* 1983.

[90] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. “Ten years of JDeodorant: Lessons learned from the hunt for smells”. In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER).* 2018, pp. 4–14.

[91] Matthew S. Tschantz and Michael D. Ernst. “Javari: Adding Reference Immutability to Java”. In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications.* OOPSLA ’05. San Diego, CA, USA: Association for Computing Machinery, 2005, pp. 211–230.

[92] Christopher Unkel and Monica S. Lam. “Automatic Inference of Stationary Fields: A Generalization of Java’s Final Fields”. In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL ’08. San Francisco, California, USA: Association for Computing Machinery, 2008, pp. 183–195.

[93] Cheng Zhang and David Budgen. “What Do We Know About the Effectiveness of Software Design Patterns”. In: *IEEE Transactions on Software Engineering* 38 (2012), pp. 1213–1231.

[94] Weixin Zhang and Bruno C. D. S. Oliveira. “EVF: An Extensible and Expressive Visitor Framework for Programming Language Reuse”. In: (2017). In collab. with Marc Herbstritt. Artwork Size: 32 pages Medium: application/pdf Publisher: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany, 32 pages.

[95] Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kielun, and M. D. Ernst. “Object and reference immutability using Java generics”. In: *6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering.* ACM, 2007, pp. 75–84.