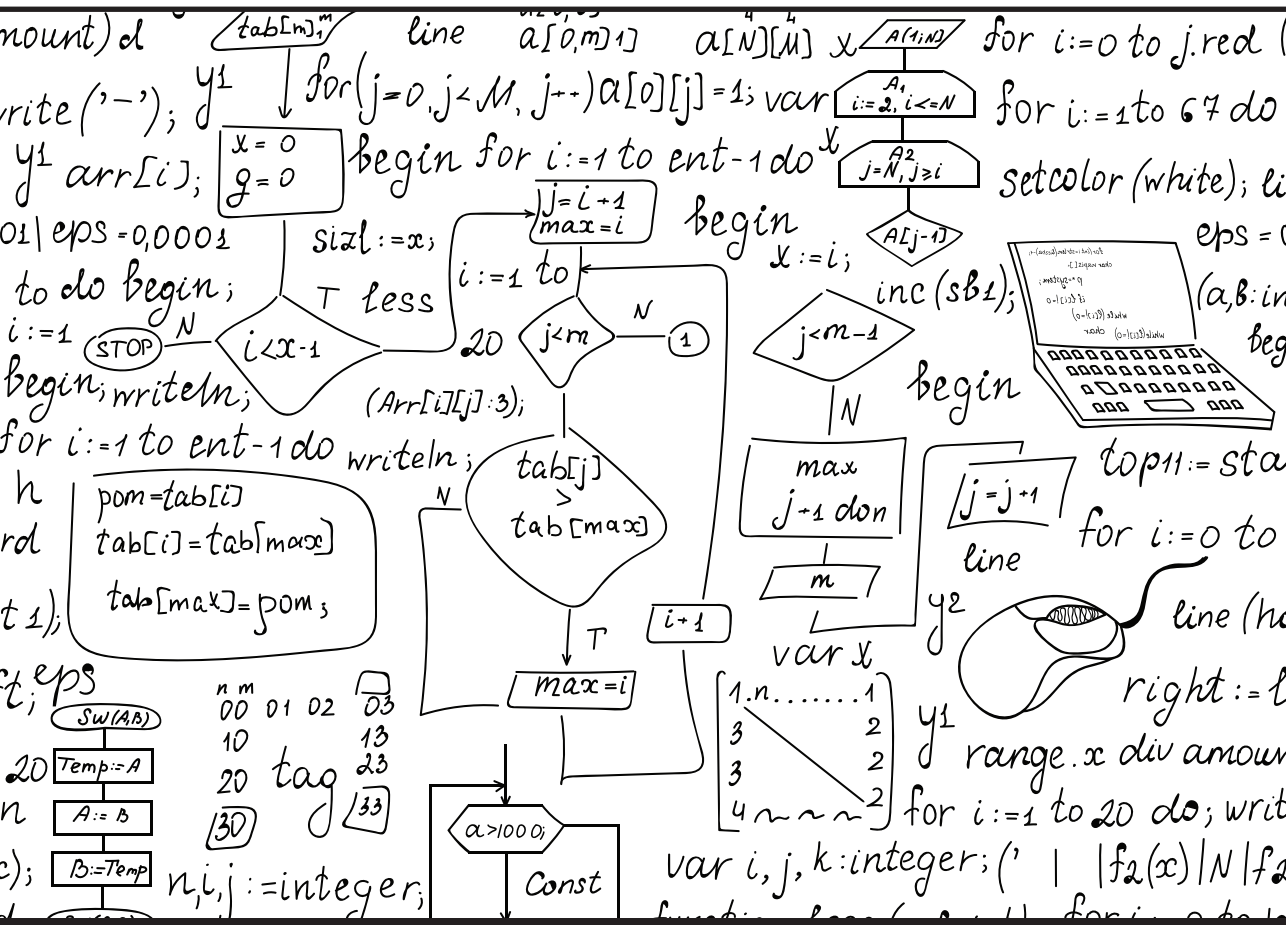
RIGA TECHNICAL UNIVERSITY

Ruslan Batdalov

DEVELOPMENT OF AN OBJECT-ORIENTED TYPE SYSTEM USING DESIGN PATTERN METHODOLOGY

Summary of the Doctoral Thesis



RTU Press
Riga 2024

RIGA TECHNICAL UNIVERSITY
Faculty of Computer Science, Information Technology and Energy

Ruslan Batdalov
Doctoral Student of the Study Programme "Computer Science and Information Technology"

DEVELOPMENT OF AN OBJECT-ORIENTED TYPE SYSTEM USING DESIGN PATTERN METHODOLOGY

Summary of the Doctoral Thesis

Scientific supervisor
Professor Dr. sc. ing.
OKSANA ŅIKIFOROVA

RTU Press
Riga 2024

Batdalov, R. Development of an Object-Oriented Type System Using Design Pattern Methodology. Summary of the Doctoral Thesis. – Riga: RTU Press, 2024. – 44 p.

Published in accordance with the decision of the Promotion Council "RTU P-07" of 24 May 2024, Minutes No. 1.

Cover picture from www.shutterstock.com

https://doi.org/10.7250/9789934371097
ISBN 978-9934-37-109-7 (pdf)

DOCTORAL THESIS PROPOSED TO RIGA TECHNICAL UNIVERSITY FOR PROMOTION TO THE SCIENTIFIC DEGREE OF DOCTOR OF SCIENCE

To be granted the scientific degree of Doctor of Science (Ph. D.), the present Doctoral Thesis has been submitted for defence at the open meeting of RTU Promotion Council on October 7, 2024, at the Faculty of Computer Science, Information Technology and Energy of Riga Technical University, 10 Zunda krastmala, Room 104.

OFFICIAL REVIEWERS

Professor Dr. sc. ing. Jānis Grundspeņķis
Riga Technical University

Professor Dr. sc. ing. Irina Arhipova
Latvia University of Life Sciences and Technologies, Latvia

Professor Dr. Jorge Luis Ortega Arjona
National Autonomous University of Mexico, Mexico

DECLARATION OF ACADEMIC INTEGRITY

I hereby declare that the Doctoral Thesis submitted for review to Riga Technical University for promotion to the scientific degree of Doctor of Science (Ph. D) is my own. I confirm that this Doctoral Thesis has not been submitted to any other university for promotion to a scientific degree.

Ruslan Batdalov ……. (signature)
Date: ……

The Doctoral Thesis has been written in English. It consists of an Introduction, 6 chapters, Conclusions, 23 figures, three tables, and two appendices; the total number of pages is 148, not including appendices. The Bibliography contains 128 titles.

TABLE OF CONTENTS

INTRODUCTION

Design patterns are extensively used in software engineering and development. Patterns provide standard solutions to recurring problems, document proven software design and architecture, identify higher-level abstractions, and facilitate communicating design by providing common vocabulary (Buschmann et al., 2013). Frank Buschmann et al. pointed out that any non-trivial design inevitably involves many patterns, consciously or otherwise (Buschmann et al., 2007b).

A widely recognised drawback of patterns-based design solutions is their complexity. Even though solving a problem inherently creates some complexity, indiscriminate application of design patterns often leads to solutions more complicated than necessary. It creates a non-trivial trade-off between the ease of applying a ready-to-use solution and the complexity of the resulting design.

Erich Gamma et al. warned about this danger at the beginning of the design patterns era (Gamma et al., 1995), and the course of events showed that these concerns were reasonable. Frank Buschmann et al. went from stating that patterns help developers manage software complexity in the first book of the 'Pattern-oriented Software Architecture' series (Buschmann et al., 2013) to the admission that many design failures had been caused by unnecessary and accidental architectural complexity, despite (or even due to) intentional and explicit applying design patterns (Buschmann et al., 2007b). Peter Sommerlad, a co-author of the same series, went even further and bluntly argued that design patterns are bad for software design because of this undue complexity (Sommerlad, 2007).

The author of the Doctoral Thesis proposed a hypothesis that the excessive complexity of pattern-based solutions is partially caused by insufficient expressiveness of the existing programming languages (Batdalov, 2016). There is a gap between patterns, representing mental constructs in which programmers reason about their programs, and programming language facilities, which have historically evolved from underlying technical means. As a result, the insufficient expressiveness of programming languages (which measures the breadth of ideas that programmers can express using the language (Leitão and Proença, 2014)) makes the implementation of pattern-based solutions more complicated than necessary.

The implementation difficulty is only one of many sources of complexity. Other sources include, for example, redundant flexibility, using inappropriate abstractions, or the complexity inherent to the problem. However, the language expressiveness problem, unlike other mentioned ones, is not specific to a particular problem and its solution. Therefore, addressing this problem can potentially help in more situations.

The Doctoral Thesis studies how the expressiveness of programming languages can be improved to make it closer to the mental constructs represented by design patterns. Raising the programming language's expressiveness is a never-ending process because humans can always invent new higher and higher-level abstractions. However, the proposed approach uses design patterns as a direction in which programming languages could evolve. It would make the implementation of pattern-based design solutions more straightforward.

An essential aspect of design complexity is how an existing system evolves. Introducing a pattern into an existing system is complicated, and eliminating it may be even more challenging (Sommerlad, 2007). In order to address this problem, the Doctoral Thesis strives to generalise the studied constructs as much as possible. It should facilitate the system evolution because substituting a subtype or an implementation of the general case with another one is usually simpler than switching to an unrelated type.

In order to achieve the necessary generalisation, the considered constructs themselves are described as patterns. Comparison of constructs used in different programming languages and identifying all pattern components allows finding their general case. Thus, patterns serve as the primary methodological means of the Doctoral Thesis.

An actual programming language based on the identified generalised constructs is outside the scope of the Doctoral Thesis. Its desired result is a theoretical ground for such a language, a system of formal types. The type system uses formalisms of the type-theoretical F_ω calculus (Pierce, 2002), which provides support for universal and existential types sufficient for the goals of the Doctoral Thesis. These types aim to be formal representations of generalised constructs used in reasoning about programs.

Topicality of the Subject

New programming languages emerge at a high pace. Some relatively recent languages that apply novel approaches and have already gained high popularity include Scala, Kotlin, Go, TypeScript, and Rust. Even creators of Java admitted the widespread desire to have 'the next great language' (Gosling et al., 2015). Besides, many research languages are created to test new approaches, concepts, and features before introducing them in mainstream languages. This situation demonstrates that the search for conceptual improvements in the programming languages field is going on permanently.

Older programming languages do not stay fixed, either. Such languages as C++ and Java, as well as others, undergo significant changes between versions, borrowing concepts from other languages and introducing new ones. The strive for higher expressiveness is a significant driver of changes for both new and existing programming languages (Batdalov, 2017).

The topic of design complexity has not been discussed recently in the patterns community as much as it was before. However, the problem is rather accepted than solved. The patterns community mainly concentrates on discovering new patterns, whereas conceptual revisions of the field have not appeared for a long time. The problem of excessive complexity of pattern-based solutions, admitted before, still exists.

The Doctoral Thesis supports the programming languages' movement towards higher expressiveness and partially addresses the problem of pattern-based solutions' complexity. It also partially systematises the ongoing processes in programming languages' evolution as the types described in the Thesis often generalise recent novelties. Thus, the Doctoral Thesis aligns with the programming languages evolution and design patterns' current needs.

The Goal of the Doctoral Thesis

The Doctoral Thesis aims to develop a system of types generalising existing programming languages' constructs to the abstraction level of mental constructs typical in reasoning about programs, thus increasing their expressiveness compared to existing languages.

The Tasks of the Doctoral Thesis

In order to achieve the goal of the Thesis, the following tasks were defined:

1. Analyse programming languages' evolution trends and previously described difficulties in design patterns' implementation.
2. Formulate requirements for the type system based on the mentioned trends and difficulties.
3. Describe common patterns of data composition and basic computation primitives.
4. Formalise type-theoretical constructs representing the described patterns.
5. Develop an example project illustrating the need for more expressive constructs.
6. Validate that the higher expressiveness of the proposed types would simplify the implementation of the example project and its evolution.

Research Object

The object of the Doctoral Thesis research is abstract concepts representing programming language-level constructs.

Research Subject

The subject of the Doctoral Thesis research is common patterns generalising basic programming-language level constructs and their type-theoretical formalisation.

Research Methods

The following methods were applied in the Doctoral Thesis:

1. Comparative analysis of programming languages to identify similar language-level constructs and describe their general forms.
2. Type-theoretical formalisation of the identified constructs using formalisms of the F_ω calculus.
3. Thought experiment on the applicability of the developed theoretical constructs in a practical project.

Scientific Novelty

1. Concepts representing typical programming language-level constructs are described as design patterns.
2. Using the language and structure of design patterns allowed for generalising the mentioned constructs.
3. The identified patterns are formalised using the type-theoretical apparatus.

Practical Significance of the Research

The validation results demonstrate that the language-level features proposed in the Thesis would simplify practical software development and evolution in certain situations. The methodology developed in the course of the present study can be used for other language and library features in the future.

Research Results Approbation

The results of the Doctoral Thesis have been reflected in eight publications in international and recognised by the Latvian Council of Science journals and proceedings.

1. Ruslan Batdalov. Inheritance and class structure. In: Pavel P. Oleynik, editor, *Proceedings of the First International Scientific-Practical Conference Object Systems — 2010*, pages 92–95, 2010. URL https://cyberleninka.ru/article/n/inheritance-and-class-structure/pdf
2. Ruslan Batdalov. Is there a need for a programming language adapted for implementation of design patterns? In: *Proceedings of the 21st European Conference on Pattern Languages of Programs (EuroPLoP '16)*, pages 34:1–34:3. Association for Computing Machinery, 2016. ISBN 978-1-4503-4074-8. doi: 10.1145/3011784.3011822
    - Indexed in Scopus.
3. Ruslan Batdalov and Oksana Nikiforova. Towards easier implementation of design patterns. In: *Proceedings of the Eleventh International Conference on Software Engineering Advances (ICSEA 2016)*, pages 123–128. IARIA, 2016
    - Personal contribution: related work analysis, development of the approach, and description of the proposed features.
4. Ruslan Batdalov, Oksana Nikiforova, and Adrian Giurca. Extensible model for comparison of expressiveness of object-oriented programming languages. *Applied Computer Systems*, 20(1): 27–35, 2016. doi: 10.1515/acss-2016-0012
    - Personal contribution: related work analysis, development of the comparison model, and performing the comparison.
    - Indexed in Web of Science.
5. Ruslan Batdalov and Oksana Ņikiforova. Implementation of a MIX emulator: A case study of the Scala programming language facilities. *Applied Computer Systems*, 22(1): 47–53, 2017. doi: 10.1515/acss-2017-0017
    - Personal contribution: related work analysis, emulator design and implementation, and Scala features analysis.
    - Indexed in Web of Science.
6. Ruslan Batdalov and Oksana Nikiforova. Three patterns of data type composition in programming languages. In: *Proceedings of the 23rd European Conference on Pattern Languages of Programs (EuroPLoP '18)*, pages 32:1–32:8. Association for Computing Machinery, 2018. ISBN 978-1-4503-6387-7. doi: 10.1145/3282308.3282341
    - Personal contribution: related work analysis and description of the proposed patterns.

- Indexed in Scopus and Web of Science.

7. Ruslan Batdalov and Oksana Nikiforova. Elementary structural data composition patterns. In: *Proceedings of the 24th European Conference on Pattern Languages of Programs (EuroPLoP '19)*, pages 26:1–26:13. Association for Computing Machinery, 2019. ISBN 978-1-4503-6206-1. doi: 10.1145/3361149.3361175
   - Personal contribution: related work analysis and description of the proposed patterns.
   - Indexed in Scopus and Web of Science.
8. Ruslan Batdalov and Oksana Nikiforova. Patterns for assignment and passing objects between contexts in programming languages. In: *Proceedings of the 26th European Conference on Pattern Languages of Programs (EuroPLoP '21)*, pages 4:1–4:9. Association for Computing Machinery, 2021. ISBN 978-1-4503-8997-6. Doi: 10.1145/3489449.3489975
   - Personal contribution: related work analysis and description of the proposed patterns.
   - Indexed in Scopus and Web of Science.

The main results of the Doctoral Thesis were presented at eight international scientific conferences.

1. The First International Scientific-Practical Conference on Object Systems – 2010, May 10–12, 2010, – Rostov-on-Don, Russia, "Inheritance and class structure."
2. EuroPLoP '16 – The 21st European Conference on Pattern Languages of Programs, July 6–10, 2016, Irsee, Germany, "Is there a need for a programming language adapted for implementation of design patterns?"
3. ICSEA 2016 – The Eleventh International Conference on Software Engineering Advances, August 21–25, 2016, Rome, Italy, "Towards easier implementation of design patterns."
4. Riga Technical University 57th International Scientific Conference, October 13–16, 2016, Riga, Latvia, "Extensible model for comparison of expressiveness of object-oriented programming languages."
5. Riga Technical University 58th International Scientific Conference, October 12–15, 2017, Riga, Latvia, "Implementation of a MIX emulator: A case study of the Scala programming language facilities."
6. EuroPLoP '18 – The 23rd European Conference on Pattern Languages of Programs, July 4–8, 2018, Irsee, Germany, "Three patterns of data type composition in programming languages."
7. EuroPLoP '19 – The 24th European Conference on Pattern Languages of Programs, July 3–7, 2019, Irsee, Germany, "Elementary structural data composition patterns."
8. EuroPLoP '21 – The 26th European Conference on Pattern Languages of Programs, July 7–11, 2021, Graz, Austria, "Patterns for assignment and passing objects between contexts in programming languages."

Theses Submitted for Defence

1. Design patterns methodology is applicable to generalising language-level constructs in the form of patterns.
2. The described patterns are formalisable as types (which potentially allows making them programming language constructs) and can facilitate the development of a software system.

Structure of the Thesis

Chapter 1 describes the patterns and type-theoretical context of the work, as well as presents the research methodology. Chapter 2 describes the evolution trends of programming languages and identifies related requirements to the developed type system. Chapter 3 describes patterns of data composition and their type-theoretical formalisation. Chapter 4 describes computation patterns, except for assignment, and their type-theoretical formalisation. Chapter 5 describes assignment patterns and their type-theoretical formalisation. Chapter 6 demonstrates the applicability of the developed type system to a software development project. The Conclusion summarises the Thesis and describes future research directions. Appendix 1 contains definitions of the terms used in the Thesis. Appendix 2 contains the list of described patterns with examples of their usage in various programming languages.

1. BACKGROUND AND MOTIVATION

The Doctoral Thesis studies the relationship between design patterns and elementary constructs of programming languages. Design patterns are important in this context to unveil the general case of a particular problem and a solution for it. The author, in his master's thesis, argued that existing programming languages often support only special cases of a particular pattern, which causes difficulties when a slightly different solution is needed (Batdalov, 2017). In this situation, describing the general case as a pattern can help understand the desired facilities of a programming language.

However, design patterns lack the formality required for programming languages' features. Type theory provides a toolset to define language facilities formally (Pierce, 2002). The generalisations described in the Doctoral Thesis as patterns are then formalised in the language of type theory to achieve the necessary strictness.

This chapter describes the design patterns approach, the general concepts of type theory, related studies, and how these approaches are applied together to achieve the goals of the Doctoral Thesis.

1.1. Design Patterns

The primary role of design patterns is to provide standard solutions for frequently arising problems in software design. Many common patterns are described in the literature in the nearly ready-to-use form, for example, Façade, Factory Method, Iterator, Visitor (Gamma et al., 1995), Broker, Publisher–Subscriber (Buschmann et al., 2013), and others. In addition to the role of software design components, they provide a common language to communicate design decisions between developers. In addition to classic popular books, thousands of other patterns are described in the literature (Booch, 2007). They are less widely known but still useful as design building blocks.

The design patterns approach originates from civil architecture. In 1977, Christopher Alexander et al. proposed using patterns as a common language for architects, expressing widely used architectural solutions (Alexander et al., 1977). According to them, '[e]ach pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice' (Alexander et al., 1977). The same approach applies in the software design patterns domain. Software design patterns were popularised by the seminal book "Design patterns" by Erich Gamma et al. (Gamma et al., 1995). Other prominent works concerning design patterns include the book series 'Pattern-oriented software architecture' (Buschmann et al., 2007a, 2007b, 2013; Kircher and Jain, 2013; Schmidt et al., 2013), 'Software patterns' by James O. Coplien (Coplien, 1996), 'Patterns of enterprise application architecture' by Martin Fowler (Fowler, 2012), and others. Proceedings of yearly conferences on the pattern languages of programs (PLoP, EuroPLoP, AsianPLoP, and others) contain descriptions of many other design patterns and pattern languages (sets of interrelated

patterns used together in the same domain). All these sources describe a multitude of patterns discovered in various software systems and ready to reuse.

The definitions of a design pattern vary but generally convey the same ideas. Erich Gamma et al. stated that patterns are 'descriptions of communicating objects and classes that are customised to solve a general design problem in a particular context' (Gamma et al., 1995). Frank Buschmann et al. proposed a similar definition: 'A pattern for software architecture describes a particular recurring design problem that arises in specific design contexts and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate' (Buschmann et al., 2013). Both definitions emphasise such principal attributes of a design pattern as a context, a problem and a solution for the problem.

The primary goal of pattern application, which Erich Gamma et al. placed in the subtitle of their book, is reusability (Gamma et al., 1995). Frank Buschmann et al. identified the following roles of patterns in software architecture: documenting existing best practices; identifying and specifying abstractions; a common vocabulary and shared understanding of design concepts; documenting software architectures; supporting the construction of software with well-defined properties; and capturing experience in a form that can be independent of specific project details and constraints, implementation paradigm, and often even programming language (Buschmann et al., 2007b).

An important aspect is that patterns exist in systems even when it is not a conscious decision. Grady Booch stated that every development culture tends to converge on a set of architectural patterns over time (Booch, 2007). According to Frank Buschmann et al., any non-trivial design uses many patterns, *consciously or otherwise* (Buschmann et al., 2007b). Similarly, researchers of patterns usually speak about pattern discovery, not inventing (Buschmann et al., 2007b, 2013). A pattern should be used in actual systems and proven before being described. It makes a described solution a pattern and not an ad-hoc solution.

Using patterns also has its costs. Frank Buschmann et al. mentioned the following common traps and pitfalls: the temptation to turn everything into patterns; describing design solutions that are not reusable or proven as patterns, considering a pattern as a fixed and unchangeable solution; describing guidelines that do not contain a solution to a problem; application of the wrong pattern; the belief that mechanical application of patterns ensures good architecture in all cases; lack of creativity; too high expectations; the impossibility of complete automation of pattern usage; the inability to be componentised; and using patterns instead of refactoring or vice versa (Buschmann et al., 2007b). These problems made Frank Buschmann et al. admit in the fifth volume of the 'Pattern-oriented Software Architecture' series that many systems in which patterns were used intentionally and explicitly ended up with unnecessarily complex architecture (Buschmann et al., 2007b). However, despite these problems, pattern application provides significant benefits and greatly facilitates software design.

1.2. Type Theory

Type theory as a field of computer science studies type systems existing in programming languages. Programming languages use types (such as integers, strings, arrays, maps, library- and user-defined classes) to define allowed operations with certain values. It protects programs against errors related to inappropriate data usage at the compile or run time.

Type theory provides a formal ground for reasoning about these protective measures taken by programming languages. It forms a basis for the calculi used to prove formal statements about programming languages. This way, one can formally prove that the rules of a programming language are sufficient or insufficient in certain situations.

Benjamin Pierce defined a type system as a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute (Pierce, 2002). This definition captures a few crucial properties of a type system: application to programs, being based on classifying language phrases, conservativeness (the ability to prove the absence of some bad behaviour but not its presence), limitedness to certain types of errors, and tractability (the opportunity to automatically check the type rules) (Pierce, 2002). Using type systems provides the following benefits: detecting errors early, helping in maintenance and refactoring, abstraction, documentation, language safety, efficiency, and security (Pierce, 2002).

A type may be assigned to a symbol in the program (a compile-time type) or to a value that this symbol holds (a run-time type). Languages vary in the usage of these types. Thus, statically typed languages significantly rely on compile-time types, perform as many checks as possible during compilation and often do not store the type information at the run time (though such features as subtype polymorphism prohibit this in some cases). By contrast, dynamically typed languages usually do not support compile-time types and perform all checks at the runtime based on the run-time type information. Somewhat paradoxically, all dynamically typed languages are safe in protecting their abstractions, whereas this is not always true for statically typed languages (Pierce, 2002). At the same time, statically typed languages provide efficiency of compile-time checks. Thus, both compile-time and run-time checks have benefits and are valuable in different situations.

Researchers usually differentiate between base (or uninterpreted) types (such as Booleans, numbers, and characters) and compound types (e.g., arrays, maps, and records) (Pierce, 2002). The base types are defined in the language specification in advance, whereas, for the compound types, only ways to construct them are specified. However, the language may be aware of some standard library-defined compound types (like iterators) and support syntactic constructs using these types. The boundary between base types and compound types is not universal, and a base type in one language may be a compound type in another (e.g., a string).

Concerning compound data types, type theory differentiates nominal and structural type systems. Traditionally, type theory primarily considers structural type systems, in which the type name is only an abbreviation, and structurally equivalent types are the same (AbdelGawad, 2017). However, most programming languages use nominal type systems, in which structurally equivalent types with different names are different (AbdelGawad, 2017). The advantages of

nominal type systems include the availability of run-time type information, natural support of recursive types, easy check of subtyping, and the differentiation between structurally compatible but semantically different types (Pierce, 2002). However, some advanced features (e.g., generics) are hard to implement in nominal type systems, so programming languages use hybrids of nominal and structural ones (Pierce, 2002).

Specific types and their semantics define calculus, empowering reasoning about programs and type systems. There are various calculi for different programming paradigms and other language properties. A general calculus can also be tailored to a particular language. Calculus allows proving properties of a type system, but it is not its only role. In Scala, at least two features (an explicit self type of a class (Odersky et al., 2006) and literal types (Leontiev et al., 2019)) first appeared in the formal description of the language semantics and only then in the language itself. Thus, the formal description of a language may inspire the development of its expressive power.

1.3. Related Work

A wide range of patterns literature is devoted to the patterns implementation problem. It is generally accepted that one cannot represent patterns in code but only provide a particular implementation (Alexandrescu, 2001). Implementation of common design patterns in different languages deserves separate books (for example, in C# (Bishop, 2008) or Scala (Løkke, 2009)). This fact suggests that implementing design patterns may be highly non-trivial, which has been recognised in the patterns community for a long time (Buschmann et al., 2007b; Sommerlad, 2007).

Several approaches to dealing with the implementation complexity have been proposed. For example, Frank Buschmann et al. tried to create configurable generic implementations but quickly showed that it is impossible even in relatively simple cases (Buschmann et al., 2007b). Some approaches employ advanced language features, for example, C++ template metaprogramming (Alexandrescu, 2001) or aspect-oriented languages (Kiczales et al., 1997; Hannemann and Kiczales, 2002; Monteiro and Gomes, 2013). Pavol Bača and Valentino Vranić proposed a similar idea to replace the commonly known object-oriented design patterns with aspect-oriented ones (Bača and Vranić, 2011). It seems that the separation of unrelated responsibilities is crucial in implementing patterns, but requiring using an aspect-oriented language for that looks redundant.

Another direction of research is pattern decomposition. The enormous number of discovered patterns suggests that they may consist of basic building blocks. Thus, Uwe Zdun and Paris Avgeriou tried to identify architectural primitives of patterns (Zdun and Avgeriou, 2008). Francesca Arcelli Fontana et al. described common design pattern micro-structures in order to facilitate pattern detection in existing systems (Fontana et al., 2011, 2013). Jan Bosch described design patterns as a composition of layers and delegations and proposed an implementation using the so-called Layered Object Model (Bosch, 1998). Despite the mentioned attempts, a limited set of universal building blocks is unknown and does not seem feasible.

The implementation complexity problem can also be solved from the other end: by raising the programming languages' level of abstraction and including patterns into languages directly. Joseph Gil and David H. Lorenz described the gradual percolation of design patterns into programming languages (Gil and Lorenz, 1998). However, there is no doubt that many patterns are too complicated and generative to be directly supported in programming languages, and the boundary between patterns that can and cannot be part of a language is unclear. Therefore, the patterns implementation problem still exists and deserves new solution proposals.

Concerning how the proposed type system could be built, one should consider structural and behavioural aspects. On the structural side, simple ways of building compound data types are known from type theory, e.g., pairs, tuples, records (including classes), lists, variants (including options and enumerations), and references (Pierce, 2002). However, these primitives do not necessarily reflect the ideas behind the programs (for example, an associative array is not considered a data composition primitive, though it is a part of some languages). The structural patterns described by Erich Gamma et al. (Gamma et al., 1995), on the contrary, represent a higher level of abstraction. Such patterns as Adapter, Proxy, or Façade are inappropriate language features because they describe specific cases of a complex interaction between multiple participants. Thus, an intermediate level of abstraction is required.

The Doctoral Thesis tries to find a balance in programming language-agnostic ways to describe data structures. Examples of such ways include Unified Modeling Language (UML) (OMG, 2017) and data transfer representations (Zimmermann et al., 2017).

The primary behavioural abstraction in type theory is a lambda expression. It represents a function that can be defined once and then applied to arguments calculated in other program fragments. The concept of a lambda expression originates from Alonzo Church's attempt to formalise the concept of an algorithm (Church, 1936). Church's lambda-calculus is not the only possible option but one of the most popular ways to formalise programs' behaviour (Pierce, 2002).

Another powerful abstraction, especially popular in functional programming studies, is a monad. Monads represent such impure features as a state, exceptions, and continuations in a purely functional environment (Wadler, 1992b). A monad is defined by a type operator and three functions that have to obey certain laws (Wadler, 1992a).

In procedural languages, behaviour mainly involves changing the name-value association (assignment). For this purpose, type theory describes such types as `Ref` (a reference, an abstraction of a location that contains a modifiable value), `Source` (a reference that can only be used for retrieving a value), and `Sink` (a reference that can only be used for assigning a value) (Pierce, 2002). Programming languages use these types explicitly or implicitly to implement state changes.

However, it is well known that the assignment operator in programming languages can have different meanings. For example, the Structural Operational Semantics (SOS) framework distinguishes between assignment (value copying) and binding (referential assignment) (Mosses, 2006). The authors of the Anzen programming language fought assignment ambiguity by introducing three different assignment operators to make the assignment semantics explicit (Racordon and Buchs, 2019). The assignment may be constrained by type modifiers, such as

the `const` keyword in C++. Type modifiers are a specific kind of subtyping relationship (Foster et al., 1999). Type modifiers often prohibit certain operations but can also bear different semantics (Carlson and Wyk, 2019). The ultimate form of assignment constraints is the single static assignment (SSA) form: each symbol is assigned only once in the program (Cytron et al., 1991). Although it sounds purely functional, compilers of imperative languages (for example, LLVM-based (Lattner and Adve, 2004)) may use the SSA form as an intermediate representation.

1.4. Research Methodology

The Doctoral Thesis is methodologically based on design patterns and type theory. The role of design patterns is twofold. On the one hand, the decomposition of well-known design patterns is used to identify features that would make their implementation easier. Thus, known design patterns are a source of ideas for possible directions of programming languages' development. On the other hand, the proposed features are themselves described in the form of patterns. In this sense, the patterns approach is the core methodology of the Doctoral Thesis: the type system features are not chosen arbitrarily but systematically described, generalised, and analysed in the patterns form.

Type theory is used for formalising the proposed patterns as language-level constructs. The identified patterns are formalised as types with corresponding semantics and operations. These types comprise the main result of the Doctoral Thesis.

2. TRENDS AND OBJECTIVES

This chapter considers the objectives of the developed type system that follow from the trends in the evolution of object-oriented languages and the potential features useful for design patterns implementation, identified by the author in his master's thesis. These objectives are primarily discussed in the context of statically typed languages. In a sense, the proposed features could increase the flexibility of static languages towards what dynamic ones provide, but without violating the typing rules.

2.1. Static and Dynamic Programming Languages Convergence

The traditional distinction between statically and dynamically typed languages is getting blurred. First, annotating every variable with a type in statically typed languages is tedious. Nowadays, many statically typed programming languages allow omitting type annotations if the compiler can infer them automatically. Type inference does not change the nature of statically typed languages but removes the burden of annotating variables from the programmer.

Second, statically typed programming languages historically tended to erase run-time type information, but it is not always possible. Polymorphic functions are resolved at the run time and thus require keeping the type information until that moment (Pierce, 2002). C# took a logical next step and introduced dynamic classes, which behave like classes in dynamically typed languages (ECMA-334, 2022).

On the other hand, dynamically typed languages are getting opportunities to specify symbol types in the code. PHP allows type specification for the parameters and return values of functions since version 7.0 (PHP, 2023). The opportunity to specify function parameter types was also introduced in Python 3.12 (Pyt, 2023). Similarly, TypeScript is a whole language that supports compile-time type declarations for JavaScript variables and functions (Mic, 2023).

Thus, even though statically and dynamically typed languages remain different in their fundamental principles, they gradually receive features that allow using the approaches usually associated with the other class. To support the convergence trend, Section 2.1 of the Doctoral Thesis formulates a generalisation that covers both the traditional statically typed and dynamically typed approaches as special cases.

2.2. Object-Oriented and Functional Languages Convergence

Functional programming is a programming paradigm based on strict assumptions, such as the absence of assignment, mutable data and iteration. Instead of assignment, functional programs create new values; instead of mutation of a data structure, they create another structure that may partially share data; and instead of iteration, they use recursion (Michaelson, 2011). Despite its restrictive environment, functional programming solves the same class of tasks as imperative programming and other programming paradigms.

The popularity of functional programming has been increasing recently. As a result, object-oriented programming languages have acquired certain features that used to be strongly

associated with functional programming only, e.g., immutable data types and higher-order functions. The mainstream languages only borrow some functional features, but such languages as Scala and Swift fully incorporate the functional style (though they stay impure as support assignment and mutable data, too).

Traditionally, the benefits of functional programming are associated with mathematical soundness. Usual mathematical systems work with generally valid truths rather than the mutable state and the order of operations. Theoretically, the validity of a functional program is a theorem that can be proved or invalidated. In practice, proving correctness is challenging, even for a functional program (Michaelson, 2011). However, for historical reasons, functional language creators pay much more attention to the type systems' guarantees and their formal proof.

Another benefit of functional languages is related to their independence of the order of execution. Thanks to this independence, functional programs are inherently safe in a concurrent environment. No locks or other synchronisation mechanisms are needed because there is no shared mutable state.

Restrictions of functional programming also cause some disadvantages. First, recursion is often less efficient than iteration. It is often possible to rewrite a recursive algorithm in the tail-recursive form, which allows the compiler to optimise the binary code (Pratt and Zelkowitz, 2001), but this form is more complicated and loses the recursive definition's main benefit, the natural mapping between the function definition in a program and the mathematical definition. Another source of inefficiency is that linked data structures, such as lists, suffer from inefficient memory access to non-contiguous data.

Another disadvantage is that not all widely used data structures allow recursive definitions. For example, sequences that support efficient random access to their elements (like arrays in imperative languages) are challenging to define recursively. Functional algorithms on them may be complicated because the recursive structure of an algorithm typically reflects the recursive structure of the data structure itself (Michaelson, 2011).

Despite the described problems, the features of functional programming languages are strongly associated with mathematical soundness and thread safety. Under certain conditions, programming in the functional style is also possible in languages that are not purely functional.

The main features to consider at the language level are functions as values (see Section 4.1) and algebraic data types (see Sections 3.6 and 3.8). Other important language-level constructs are guaranteed immutability and parametric polymorphism, but they are not considered in the Doctoral Thesis to keep the discussion scoped.

2.3. Lessening Traditional Object-Oriented Restrictions

Usage of some concepts and constructs of object-oriented programming languages is traditionally subject to restrictions for safety or good object-oriented design. However, some of these restrictions tend to be relaxed over time (Batdalov, 2017). It shows that the original constructs were too restrictive and required extension for better expressiveness.

An example of such a process is the extension of what an interface (mixin or trait) can define. In the classic Java form, interfaces might contain only declarations of public methods. However, since Java 8, interfaces may also define the so-called 'default' implementation of methods (Gosling et al., 2023). In TypeScript, interfaces can also define data members (Mic, 2023). In Scala, traits can define private members. Only constructor parameters are disallowed in Scala traits (Odersky et al., 2023). Thus, there is a tendency to extend the capabilities of interfaces.

Another example is related to the evolution of data access restrictions. Making all data members private and accessing them only through methods is considered a good style, except for classes having an intentionally open internal structure (Martin, 2008). However, having done this, programmers often create getters and setters to access the private fields. Even though creating getters and setters for every private data member should be avoided (Martin, 2008), this operation is so typical that it brought to life such constructs as properties in C# (ECMA-334, 2022) and get and set methods in TypeScript (Mic, 2023). Again, the original restrictions were too strict and required greater flexibility.

Another example is the variety of reference-like types in C++ and Java. C++ initially supported pointers, inherited from C, and references, their restricted version for safety. References addressed some pointers' shortcomings but were inappropriate in other situations, which led to the emergence of smart pointers, which ensure other aspects of safety (ISO/IEC 14882:2020, 2020). Similarly, in addition to the built-in Java reference types (arrays and objects), the standard library contains such types as `SoftReference`, `WeakReference`, and `PhantomReference`, which have special garbage collection rules (Ora, 2023). Memory management is impossible to reduce to one or two basic types.

The most important lesson from the examples above is that it does not make sense to define two or more similar concepts that differ in more than one aspect. Andrei Alexandrescu wrote that each independent design decision or constraint should become a separate policy, and every combination of orthogonal policies should be supported (Alexandrescu, 2001). The author tried to follow this principle in the considered type system as much as possible.

2.4. Operators as Class-Defined Methods

This trend is loosely related to the Doctoral Thesis because type systems have little to do with operators. Operators are primarily syntactic constructs, affecting how the compiler builds the abstract syntax tree. Therefore, operators are only briefly discussed in the Doctoral Thesis.

Usually, a programming language has a fixed set of operators, and the language provides an implementation of operators for the built-in types. When it comes to the user-defined types, some languages allow programmers to define operators for these types (e.g., C++ (ISO/IEC 14882:2020, 2020)), and some do not (e.g., Java (Gosling et al., 2023)). Scala has two novel features: defining an arbitrary operator and the call-by-name strategy in operators (just like in other methods) (Odersky et al., 2023). The latter strategy is not unique, but other languages usually use it only in built-in operators.

The discussed trend can be generalised in the following way: every feature or trait that can be applied to a built-in type can be applied to user-defined types as well. It is one of the main reasons to introduce generalised functions (see Section 4.1).

2.5. Constructs Corresponding to Common Design Patterns

The relationship between design patterns and language constructs is not trivial. Erich Gamma et al. said that a design pattern in one language is just a language construct in another (Gamma et al., 1995). It is also true for language development over time: many programming languages have acquired features corresponding to common design patterns. Examples of such features include for-each loop, implementation of the Iterator design pattern (Gamma et al., 1995), objects in Scala (Odersky et al., 2023), implementation of the Singleton design pattern (Gamma et al., 1995), channels in Go (Goo, 2023), implementation of the Pipes and Filters architectural pattern (Buschmann et al., 2013), and events in C# (ECMA-334, 2022) and observables in Kotlin (Jet, 2023), implementation of the Publisher-Subscriber design pattern (Buschmann et al., 2013).

The Doctoral Thesis research follows the same trend by describing patterns potentially able to be programming language features.

2.6. Generalisation and Interface Extraction

Inheritance in object-oriented programming languages is a special case of subtyping because any object of a subclass is considered an object of its superclass (Pierce, 2002). Some authors criticised the subtyping interpretation, arguing for the general incremental modification instead (Cook et al., 1989; Taivalsaari, 1996). However, most languages still assume the subtyping relationship between classes and subclasses.

The author of the Thesis argued that the opportunity to define a superclass later than a subclass would help cover a wider range of incremental modifications, as well as facilitate the implementation of design patterns (Batdalov, 2010; Batdalov and Nikiforova, 2016). The traditional object-oriented way of building type hierarchies is deductive: from the general to the special or from a supertype to a subtype. However, the cognitive process often goes in the opposite direction. For example, the more general complex numbers were historically introduced later than the special case of real numbers.

The order of definition is not directly related to the type system, as the latter deals with already defined types. However, Section 2.6 of the Doctoral Thesis defines the rules that have to be fulfilled to enable generalisation instead of specialisation in class hierarchies.

2.7. Predicate and Depth Subclassing

Most mainstream programming languages support only width subclassing, i.e., adding new members to a class. Type theory also describes depth subclassing, i.e., subtyping the types of existing members (Pierce, 2002). Depth subclassing can be generalised to predicate subtyping,

in which a subtype may be narrowed using an arbitrary predicate (e.g., a string matching a specific format) (Batdalov and Nikiforova, 2016).

The author previously showed that depth and predicate subclassing would be beneficial in implementing commonly known design patterns (Batdalov and Nikiforova, 2016). However, these types of subclassing create difficulties with using subclass references as mutable superclass references. Width subclassing guarantees that such usage will not violate type safety rules, but depth subclassing and predicate subclassing do not.

Section 2.7 of the Doctoral Thesis formulates the rules that would allow type-safe depth and predicate subclassing. They are based on the usage-site variance typing rules for generics (parameterised types).

2.8. Default Implementation and Execution Contexts

Usually, creating an object requires specifying a concrete class to instantiate. It is not enough to specify only an abstract class. The author of the Thesis argued that having rebindable default implementations of abstract classes (i.e., the concrete class to use when an abstract class is instantiated) would be beneficial in implementing commonly known design patterns (Batdalov and Nikiforova, 2016). A similar approach is applied in Scala, where the companion object of an abstract class may instantiate a concrete implementation (Odersky et al., 2023). However, the default implementation in Scala is set in the library, and a programmer cannot change it.

Assigning a default implementation to a class is a kind of binding, so its scope should be clearly defined (Batdalov and Nikiforova, 2016). The Doctoral Thesis proposes to store this binding in the execution contexts. An execution context is a space of globally accessible names inherited when spawning a new thread. Execution contexts are associated with executors (see Section 4.2).

2.9. Chameleon Objects

The State pattern allows an object to change its behaviour at runtime so that it appears to change its class (Gamma et al., 1995). The State pattern is reflected in the UML state machine diagram, although the UML assumes the behaviour of all states to be implemented in one class (OMG, 2017). However, placing unrelated behaviour in different classes is a better coding style (Martin, 2008). Therefore, the Doctoral Thesis considers the multi-class implementation of a state machine.

The original implementation of the State pattern employs an extra level of indirection, which redirects calls to actual objects (Gamma et al., 1995). The author of the Thesis proposed to support such behaviour directly at the language level by allowing objects to change their classes at the runtime (Batdalov and Nikiforova, 2016).

Section 2.9 of the Doctoral Thesis formulates the rules required for the type safety of this approach. They are based on the assumption that mutating methods of a class may change the

run-time type, but the type checks prohibit calling a method if the new run-time type is not a subtype of the compile-time type of the variable.

2.10. Extended Initialisation

A language may have different rules for different object lifecycle phases, e.g., assigning read-only fields is possible only during construction. As a result, all data necessary for object initialisation must be parameters of a single constructor. It is possible to partially overcome this restriction using creational patterns, such as the Builder and the Factory Method (Gamma et al., 1995). However, they do not have a special status and do not allow modification of constant fields outside the constructor.

The author proposed extending the initialisation phase and allowing it to consist of several operations (Batdalov and Nikiforova, 2016). It essentially means having different states for different lifecycle phases. Thus, the proposal may be implemented by combining the approaches described in Sections 2.7 and 2.9.

2.11. Object Interaction Styles

The most typical way of interaction between objects is synchronous function calls. However, programming languages tend to introduce alternative ways of interaction, such as asynchronous calls in many languages and channels in Go (Batdalov, 2017). These ways are reducible to synchronous calls but allow expressing object interaction more naturally.

However, other interaction styles are possible. A class may represent an external system or a component, e.g., when the Façade, Proxy (Gamma et al., 1995) or Broker (Buschmann et al., 2013) pattern is used, so interacting with the class is interacting with this external system. Component interaction styles are more diverse: synchronous request-response, asynchronous request-response, pipe&filter, broadcast, blackboard, and publish-subscribe (Crnković et al., 2011). Many design patterns representing these styles are described: Observer (Gamma et al., 1995), Blackboard, Forwarder-Receiver, Master-Slave, Pipes and Filters, Proxy, Publisher-Subscriber (Buschmann et al., 2013).

It is impossible to support every interaction style directly at the language level since the list of interaction styles is potentially open. Instead, the particular interaction styles should be defined as library classes and operators. The supporting features in the considered type system are generalised functions (see Section 4.1), executors (see Section 4.2), functions that return multiple values one by one (see Section 4.4), and the unboxing assignment (see Section 5.5).

2.12. Processing State Management

Conventional functions in programming languages and type theory are solely behavioural; they contain only code and do not hold any state. The Command pattern generalises this concept as a first-class object that allows inheritance, may store execution state, and supports undo, redo and logging (Gamma et al., 1995). Frank Buschmann et al. noticed that implementing all this

functionality in a Command may be unreasonable and described the Command Processor pattern (Buschmann et al., 2013). A Command Processor is an abstraction of an executor, such as a thread, a thread pool, or a debugging environment.

The considered type system uses these abstractions as basic building blocks. The corresponding type system features are generalised functions (see Section 4.1) and executors (see Section 4.2).

3. DATA COMPOSITION

Data composition is used in programming languages to build compound data types (such as arrays and classes) from simpler ones (Pierce, 2002). Similar mechanisms are applied in data transfer representation languages (such as JSON, XML and network protocols). However, the particular mechanisms used in these languages are different, which may create difficulties when converting between data transfer representation and in-memory objects.

Data transfer representation is a representation that is agnostic to the programming languages and paradigms used to interpret the data (Zimmermann et al., 2017). Therefore, data transfer representation languages are a good source of common patterns independent of particular programming languages. Conversion of a data object in a programming language to its data transfer representation is called marshalling, and the opposite operation is unmarshalling (Zimmermann et al., 2017). Unmarshalling usually requires additional information (a schema) to understand how to interpret the data.

This section describes the composition primitives that are used both in programming languages and in data transfer representation. These primitives are based on the list of basic compound types proposed by the author for comparison of programming language features (Batdalov, 2017; Batdalov et al., 2016) and later described in the form of patterns (Batdalov and Nikiforova, 2018, 2019).

3.1. Sequence

A sequence is a collection that contains possibly repeating values arranged in a certain order if the order generally does not depend on the values themselves (e.g., the values do not have to have ascending or descending order). The values can be accessed sequentially in the given order or by the index in the sequence. A sequence may support resizing to hold more or fewer values. The values in the sequence may be of the same type or not.

Examples of sequences in programming languages include such common data types as fixed- and variable-size arrays, singly and doubly-linked lists, and tuples. These basic types can serve as implementations of higher-level sequences, such as stacks and queues. In data transfer representation languages, sequences are usually represented by sequences of values (array literals in JSON or repeated fields in XML).

The used formalisation defines sequence as a universal (generic) existential (abstract) type that allows the creation of a sequence from a Traversable Once (see Section 4.4), iteration over a sequence, and iteration over assignable slots allowing modification of the sequence elements. Other operations, including access by index, are not common for all sequences, so they should belong to more specific types.

3.2. Set

A set is a collection of unique values whose order is either undefined or depends on the values themselves (i.e., an ordered set may iterate values in ascending order). Unlike sequences,

a set does not have an order that would be independent of the values. A set may support checking if a specific value is in the set. The values in the set may be of the same type or not.

Programming languages often support ordered and unordered sets, usually based on red-black trees and hash tables correspondingly. Other implementations are possible. Some languages do not have separate types for sets but allow using maps with trivial values to represent sets. In data transfer representation languages, the representation of sets is the same as the one of sequences, and the interpretation of data as a sequence or a set depends on the data schema.

The used formalisation defines a set as a universal (generic) existential (abstract) type that allows the creation of a set from a Traversable Once (see Section 4.4), iteration over a set, and adding and removing an element. Creation and iteration have the same signature as for sequences, but their semantics are different because a set does not keep the order of values insertion. Checking whether a value is in the set is not included in the formalisation because, for some implementations, this operation cannot be performed more efficiently than by iteration (though such implementations are rare).

3.3. Multiset

A multiset is a collection of potentially repeating values whose order is either undefined or depends on the values themselves (i.e., an ordered multiset may iterate values in ascending order). Like sets and unlike sequences, a multiset does not have an order that would be independent of the values. A multiset may support checking if a specific value is in the set. The values in the multiset may be of the same type or not.

Programming languages rarely support multisets, but they are easy to emulate with maps (by keeping the number of occurrences as the value). C++ is an exception that supports ordered and unordered multisets explicitly. In data transfer representation languages, the representation of multisets is the same as the one of sequences and sets, and the interpretation of data depends on the data schema.

The used formalisation defines a multiset as a universal (generic) existential (abstract) type that allows the creation of a set from a Traversable Once (see Section 4.4), iteration over a set, and adding and removing an element. This formalisation is the same as for sets; the only difference is whether the mutating operations ensure value uniqueness.

3.4. Map

A map is a correspondence between keys and values where each key may occur only once. It can be treated as a set of key-value pairs where keys do not repeat and do not have an order independent of the keys. The keys may be predefined or not. A value typically can be accessed by a key. There are no requirements for the values: they do not have to be unique, and efficient access to keys by values is not required.

A typical example of a fixed-keys map is a record. Examples of variable-keys maps include associative arrays, which may be ordered (e.g., based on red-black trees) or not (e.g., based on

hash tables). Depending on the language, a record or an unordered associative array is the underlying type for classes. Concerning data transfer representation languages, JSON supports object literals and arrays of key-value pairs, but XML does not have a direct equivalent of a map (though specific cases may be emulated with repeated tags).

The used formalisation defines a map as a universal (generic) existential (abstract) type that allows the creation of a map from a Traversable Once (see Section 4.4.) of key-value pairs, iteration over key-value pairs, iteration over pairs of keys and assignable slots to values, and adding and removing entries. Similarly to sets, the operation of access by key is omitted because some implementations (though rare) do not support this operation in a more efficient manner than iteration.

3.5. Multimap

A multimap is a correspondence between keys and values where keys may repeat. It can be treated as a set of key-value pairs where keys may repeat and do not have an order independent of the keys. Values typically can be accessed by a key. Multiple values associated with the same key may be organised in a sequence or a set. There are no requirements for the values: they do not have to be unique, and efficient access to keys by values is not required.

Programming languages rarely support multimaps, but they are easy to emulate with maps whose values are sets or sequences. C++ is an exception that supports ordered and unordered multimaps directly. In data transfer representation languages, JSON supports arrays of key-value pairs, but XML does not have a direct equivalent of a multimap (specific cases may be represented, but there is no uniform way for that).

The used formalisation defines a map as a universal (generic) existential (abstract) type that allows the creation of a map from a Traversable Once (see Section 4.4.) of key-value pairs, iteration over key-value pairs, iteration over pairs of keys and collections of values, iteration over pairs of keys and collections of assignable slots to values, and adding and removing entries.

3.6. Variant Type

A variant is a choice between two or more alternatives (e.g., a variable that can hold an integer or a string value, a union of types). Generally, variants make sense at the declaration site only (the compile-time type or data representation schema) since the usage site (the run-time type or the actual data representation) contains a particular alternative. A variant may be labelled (having a label indicating the chosen alternative) or unlabelled (a simple union of types). Important special cases of variants are optionals and enumerations.

The support for variants significantly varies between languages. Examples of labelled variants in programming languages include `std::variant` in C++ and `Either` in Scala. TypeScript supports the `|` type operator, which represents an unlabelled variant. Special cases are supported more often. Thus, in languages with exceptions, a function implicitly returns a variant of the return value and an exception. In Scala, this variant type exists also explicitly (`Try`). Many languages support enumerations, variants of trivial types, each having only one

value. The optional value, another special case, is considered in Section 3.7. In data transfer representation languages, XML Schema allows the declaration of alternatives that may be placed in a specific place of a document.

The used formalisation defines a labelled variant as a universal (generic) existential (abstract) type that allows the creation of a variant from a value of any alternative, applying a set of functions accepting different alternatives (so that only the function accepting the held alternative is executed), and applying a set of functions accepting assignable slots to different alternatives.

3.7. Optional Value

The optional value is a variable that may or may not hold a value. It is a special case of a variant: a variant of a type and a devoted null value. Similarly to general variants, an optional value may be labelled or unlabelled, though the difference emerges only in certain edge cases (labelled optional values allow nesting, like `Optional[Optional[Int]]`, but unlabelled ones do not).

In many languages, reference types (pointers in C++, objects and arrays in Java, etc.) may contain a special null value, which makes these types optional. These types allow dereferencing without checking the value for nullness, and an attempt to dereference null throws an exception. The creator of this approach later called it a billion-dollar mistake (Hoare, 2009). Languages with strong functional influence support the type-safe optional type, which cannot be dereferenced without checking that the value is present.

The used formalisation defines an optional value as a universal (generic) existential (abstract) type that allows creating an empty value, creating a non-empty value from a value of the underlying type, and two methods choosing one of two given functions depending on whether the value is empty, non-mutating and mutating.

3.8. Type Intersection

Type intersection naturally corresponds to multiple inheritance in object-oriented languages since a subclass of several classes is also a subtype of all of them. However, multiple inheritance from classes is rarely supported in object-oriented languages. Multiple inheritance from interfaces is typically supported and provides the opportunity to create type intersection. Since type intersection is a natural consequence of multiple inheritance, there is no need to describe a separate pattern.

4. ELEMENTS OF COMPUTATION

This chapter describes basic computational patterns, serving to model program behaviour. They do not include the assignment patterns described in Chapter 5. The Traversable Once and the Assignable Once patterns were previously described as patterns for linking program behaviour and data composition (Batdalov and Nikiforova, 2018); the others are based on the list of basic behavioural types proposed by the author for comparison of programming language features (Batdalov, 2017; Batdalov et al., 2016).

4.1. Generalised Functions

A function is the primary unit of behaviour in most programming languages. Functions allow developers to split the whole computation into logically separate reusable pieces with clearly defined inputs and outputs. In addition to basic stateless functions, there are extensions, which vary between languages. In particular, a function or a function-like object may be stored in variables, passed as an argument to other functions (higher-order functions), and hold an internal state.

The primary example is the conventional function, existing in virtually any general-purpose programming language. In the languages that treat functions as 'first-class citizens', functions (or function pointers) may be stored in variables, passed between contexts, and do whatever a piece of data can do. Some languages support anonymous functions (lambda expressions). Functors, i.e. objects that can hold internal state and be 'called' as if they were functions, cover the general case of the pattern. Functors may be defined in the program code or generated automatically by the compiler (closures and generators). Unfortunately, in some languages, various functions and function-like objects are not fully interchangeable, which hinders transitioning between them.

The used formalisation defines a generalised function as a universal (generic) existential (abstract) type that allows applying the object to an argument of the parameter type and receiving a value of the return value type. As usual in type theory, multi-parameter functions are not considered because they are easily reducible to single-parameter functions.

4.2. Executors

An executor is the abstraction of an environment that can execute code, such as a thread, a thread pool, a testing environment, or a remote system. An executor provides a unified interface for inter-environment calls. This interface is not limited to conventional synchronous function calls; other interaction styles (see Section 2.11) are possible. The interface of a remote system may be restricted (e.g., only certain functions can be called remotely). Another function of the executors is that different executors may have different values of global objects (singletons).

Standard and dedicated libraries often contain interfaces to support threads, thread pools, and remote procedure calls (RPCs) but rarely have a common interface for all these executors.

Interface definition languages define restricted interfaces. Substitution of global objects (such as fake clocks or mock remote systems) is common in testing environments.

The used formalisation defines an executor as a universal (generic) existential (abstract) type that allows executing a function and retrieving a (potentially modifiable) execution context.

4.3. Data Access Delegation

Many well-known design patterns are based on delegation. Behaviour delegation is usually trivial (it is simply a function that calls another function) and does not require a separate discussion. However, there is a specific case of delegation: delegation of data access. Some properties of objects may logically look like data attributes but not be stored in the object itself. In this case, whether the property is stored is an implementation detail, which may change without changing the public interface. In order to hide these details, a programming language may provide uniform access to stored and computable properties.

Such computable properties are supported, for example, in C#, JavaScript, TypeScript, Scala, and Python. Scala additionally documents the convention about side effects: a method without parameters may be declared without empty parentheses (`foo` instead of `foo()`), which means a promise to avoid side effects.

The used formalisation assumes that every object property has a getter and a setter. For stored properties, they are trivial, but a programmer can define them in a more complicated manner. The compiler should substitute data access in the code with calls to these functions.

4.4. Traversable Once

The Traversable Once pattern is a generalisation of the Iterator pattern, described by Erich Gamma et al. (Gamma et al., 1995). Unlike the Iterator pattern, it does not assume that the values to iterate over are held in an aggregate (collection). Instead, the values can be retrieved or computed as they are requested. In this case, the opportunity to traverse the same set of values the second time is not guaranteed. That is the reason to call the pattern Traversable Once. If the underlying structure allows multiple traversals, one can retrieve multiple Traversable Once objects, each of which is traversed once only. Retrieving values from a Traversable Once may be synchronous or asynchronous. A Traversable Once may apply a functional transformation (mapping) to the values returned by another Traversable Once.

Iteration over a container (e.g., an array, a list, or a map) is one of the basic operations in many programming languages. The appearance of the Iterator pattern popularised a special form of the for-loop, for-each loop, adapted for iteration. Some languages (e.g., Python, C#, and JavaScript) support generators, where the values do not have to come from a collection. Scala generalises both approaches in the `TraversableOnce` type (after which the pattern name is chosen). The asynchronous case is supported more rarely, but in Python and JavaScript, generators may be asynchronous. Another asynchronous example is the `Observable` interface in the ReactiveX library.

The used formalisation defines a Traversable Once as a universal (generic) existential (abstract) type that allows creating a Traversable Once from a value and retrieving one value from an existing Traversable Once.

4.5. Assignable Once

The Assignable Once pattern is used to request a value by giving the value producer a slot where the value may be stored. It is a general abstraction of an assignable slot decoupled from the nature of the particular slot (e.g., a variable, a constant, or a future). Since the slot may not be available after use, the pattern is called Assignable Once. As with the Traversable Once pattern, when the application logic allows multiple assignments, Assignable Once objects may be requested multiple times. In the general case, an Assignable Once is transferable between parts of code and in time.

In programming languages, variables are durable assignable slots. In terms of this pattern, they generate an Assignable Once for every assignment operation. Constants provide only one Assignable Once, which must be used during initialisation. Pointer and reference types support transferring assignable slots or storing them for later (though they do not usually support guarantees of a one-time assignment). Promises and futures support the asynchronous case (usually with guarantees of a one-time assignment).

The used formalisation defines an assignable once as a universal (generic) existential (abstract) type that supports only one operation: assigning a value. There is no generic way to create an assignable slot; it depends on the nature of the slot and thus is postponed to specific implementations.

5. ASSIGNMENT

This section describes patterns for various types of assignment, one of the most fundamental operations in programming languages. Assignment is understood in a broad sense: it includes initialising a variable or a constant, reassigning the value of a variable, and passing objects between contexts (passing an argument to a function or returning a function's return value). All these operations change the association between names and values in a program. There are different ways to change this association (for example, copying a value vs making two variables refer to the same memory fragment), which correspond to different patterns.

The identified patterns are based on the list of assignment types proposed by the author for comparison of programming language features (Batdalov, 2017; Batdalov et al., 2016) and later described in the form of patterns (Batdalov and Nikiforova, 2021).

5.1. Value Assignment

Value assignment is the most straightforward kind of assignment: the source value is directly stored in the target. In the prototypical case, a program copies the value from the source to the target. If the value is a linked data structure, it means recreating the data structure in new memory chunks (deep copying). This operation avoids the shared state but may be expensive and requires the compiler to understand the memory allocation of the data structure. Some languages copy only the structure base, e.g., the root node of a tree (shallow copying), which makes this operation a hybrid of the value and referential assignment, vulnerable to the drawbacks of the shared state. In some situations, shallow copying is safe, namely for immutable types or when the source is not used after assignment (the source is moved into the target).

Programming languages usually use value assignment for primitive types, but the situation with compound types may be more complicated. C++ uses value assignment with deep copying even for complex types and applies move and return value optimisation to avoid excessive copying. Such languages as Java, JavaScript, and C# consider all non-primitive types reference types and apply the referential assignment. If there is a need for the value assignment of a reference type in these languages, the programmer has to implement it manually. Perl supports the value assignment for compound types but performs only shallow copying.

The used formalisation reduces the value assignment to two operations: getting an assignable slot for the value type at the target and using the slot to assign the source value.

5.2. Referential Assignment

Referential assignment means that the destination symbol (name) starts referencing the memory location of the source. As a result, both symbols reference the same memory location. It is often more efficient than the value assignment and supports reusing non-copyable data (such as resource handles) in different parts of programs. However, referential assignment may create a shared mutable state, leading to such adverse effects as race conditions in a

multithreaded environment, accidental modification of an argument passed to a function, and use of a memory chunk after freeing it. Without a sophisticated memory management system, such as garbage collection or smart pointers, the referential assignment may lead to memory leaks or dangling references.

In C++, the assignment operator means the value assignment, but the assignment of pointers and references implements the referential assignment. Such languages as Java, JavaScript, and C# always treat non-primitive types as references, so their assignment is the referential assignment.

A separate formalisation of the referential assignment is redundant as it is technically simply the value assignment of references. It is enough for the type system to support reference types. This approach precisely reflects how referential assignment is typically *implemented* in programming languages. The described pattern provides a different *view* of this mechanism, but the underlying implementation and formalisation do not change.

5.3. Partial Assignment

Partial assignment arises when a part of a compound data structure (such as an array, a record, or a map) should be changed. A mutable data structure can be modified in place, and immutable structures apply the copy-change operation (creating a new object that has the new value of the assigned part and the old values in other parts). In the latter case, copying the whole unchanged part may be expensive, so the created structure typically at least partially shares data with the old structure (since the shared state is safe for immutable types).

In programming languages, the partial assignment is typical for container and container-like objects that maintain some form of key-value or index-value association (e.g., arrays, lists, records, and maps). Immutable counterparts of these types usually support the copy-change operation. Balancing partial modifications with avoiding excessive copying may be complicated. For example, the Vector class, an immutable counterpart of an array in the Scala standard library, is internally implemented as a tree with the branching factor 32 (Odersky and Spoon, 2023).

The used formalisation assumes that a compound type generates an assignable slot for the element to be modified (either during iteration or by index or key), and the assignable slot is used for the assignment.

5.4. Destructuring

Destructuring supports decomposing a compound data structure and assigning its elements to multiple targets in one statement. It is syntactic sugar and does not add new functionality. However, it may simplify code, especially when the source data structure comes from a call to another function. Assignment to individual targets may be the value or the referential assignment. Some parts of the source may not need to be stored, so destructuring may skip them. The destructuring assignment from user-defined types may require additional implementation work from the creators of these types (if the language supports it).

The destructuring assignment from standard types (such as arrays) is supported, for example, in C++, JavaScript, TypeScript, and Python. Scala and Kotlin support destructuring user-defined types, provided that they contain corresponding destructuring methods.

The used formalisation assumes that each operation that creates a compound object from multiple values has a counterpart for the opposite operation. The compiler may then call this counterpart and assign the returned values to the target variables.

5.5. Unboxing

Unboxing simplifies the syntax of using a value kept inside another object, for example, the result of an API call inside a promise or a future. Similarly to destructuring, the unboxing assignment is only syntactic sugar but makes a program syntax more concise. Unlike other assignment operators, the unboxing assignment is not a conventional function. Instead, when encountering the unboxing assignment, the compiler modifies the execution flow. For example, when unboxing a value from a promise, the compiler turns the subsequent code of the function into a lambda expression to be executed after the promise receives a value.

The `await` operator in C#, JavaScript, TypeScript, and Python unboxes the future value type (`Task`, `Promise`, or `Future`). It also supports user-defined types that define corresponding methods. Scala applies a more general case of unboxing with for-comprehensions usable with arbitrary types defining methods `map`, `flatMap`, and `filter` (for example, all standard collections).

The used formalisation describes the transformation the compiler should perform, defined as in the Scala case. The exact semantics of this transformation depends on the boxed type.

6. VALIDATION

The Doctoral Thesis illustrates the capabilities of the described type system on the example of an emulator of MIX, the mythical computer invented by Donald Knuth for his "The Art of Computer Programming" book series (Knuth, 1997). Despite being a 'toy' project, it shows potential improvements even in such an expressive language as Scala (Batdalov and Ņikiforova, 2017). The expected practical consequence of higher expressiveness of the proposed improvements is reducing the implementation complexity. The discussion is, to a certain degree, speculative because the Doctoral Thesis describes a type system but not an actual programming language, and the considered code changes are different from the actual history of the system. Despite that, this discussion demonstrates the potential aid the described type system could bring.

6.1. Project Description

Donald Knuth invented the MIX imaginary computer for his book series "The Art of Computer Programming" (Knuth, 1997). Most code fragments and exercise solutions in the series are written in the MIX assembly language (MIXAL).

A crucial feature of MIX is its incomplete determinism. MIX can work as a binary or a decimal computer, and a correct program should not depend on the byte size (Knuth, 1997). Similarly, programs should work correctly independently on the speed of asynchronous input/output operations. This indeterminism creates challenges in programming in the low-level assembly language.

The author developed a web-based emulator of MIX with additional correctness verification features (Batdalov and Ņikiforova, 2017). The emulator can execute programs in the binary or the decimal mode and verify the correctness of input/output synchronisation. It can also record every state during program execution and switch between them forward and backward (the feature inspired by Online Python Tutor (Guo, 2013)). These features provide tools to verify that a program follows the rules imposed by MIX architecture.

In order to support byte size variability, the emulator defines different families of byte size-dependent classes (`MixByte`, `MixWord`, and others) for the binary and the decimal mode and uses the family polymorphism in Scala (Odersky et al., 2006). The latter is necessary to ensure that classes in the same family are used together (e.g., the binary register state is incompatible with the decimal words). The user may run the emulator in the binary or the decimal mode and check that the result is the same.

To verify input/output synchronisation correctness, the emulator applies memory locks, inspired by SQL data locks (ISO/IEC 9075-2:2023, 2023). When a program initiates an input/output operation, the emulator locks the corresponding memory area and disallows operations whose results depend on whether the input/output operation is complete.

Tracking the emulator state and returning to previous states are achieved by storing each state in immutable data structures. The copy-change operation is applied to generate a new state

during a program execution. Since immutable data structures use shared storage, holding all states is not very expensive.

The source code of the author's implementation (in Scala and TypeScript) is available at https://github.com/linnando/MIXEmulator. A working copy of the emulator is available at https://www.mix-emulator.org.

6.2. Code Simplification

A possible simplification the proposed type system could provide is related to the feature parity for mutable and immutable implementations of structural types (described in Chapter 3 and their descendants). Among other things, the feature parity means that both mutable and immutable types support the partial assignment (assigning to a part of a compound object; see Section 5.3).

The partial assignment is extensively used in the emulator code because the state after every operation is a modification of the previous state. However, the partial update of immutable types requires complicated statements like `copy(forwardReferences = forwardReferences.updated(symbol, forwardReferences(symbol) :+ counter))`. The reason is that the copy-change operation on an immutable object is not reducible to an update of a single element, as in the mutable case. Instead, it involves rebuilding the whole data structure starting from the root.

The approach described in Section 5.3 proposes direct support for the partial assignment of immutable data types. It involves generating an assignable slot that would hold a pointer to the whole data structure instead of the assignable node (together with the information on finding the node). Assigning a value to this slot would rebuild the data structure and store a reference in the target. Then, the code updating the emulator state could look as simple as in the mutable case: `forwardReferences(symbol) :+= counter`.

6.3. Evolving from a Simpler Prototype

Since one of the goals of the proposed type system is to support code evolving (and not only simplify the code written once), it is important to understand potential simplifications in the emulator development from a simplified design to the current state. This mental experiment is not pure because it does not match the actual emulator history, but it makes sense because this way of development is typical for applications.

If the emulator had initially been binary only and the decimal mode had been added later, it would have required adding new implementations of the byte size-dependent classes. With the traditional object-oriented polymorphism, adding new implementations would have been easy, but the emulator uses Scala's family polymorphism. The reason is that classes from different families are incompatible and cannot be implementations of the same abstract type. Making them implementations of the same abstract type would require covariant method arguments, which are type-unsafe.

However, the family polymorphism in Scala requires the whole family of interrelated classes (e.g., the whole binary implementation) to be contained inside one singleton object. For a non-trivial logic, it creates enormous objects, violating the single responsibility principle (Batdalov and Ņikiforova, 2017). Thus, introducing the family polymorphism in an existing system would be difficult.

The Doctoral Thesis proposes an alternative related to the interpretation of inheritance in terms of usage-site variance, discussed in Section 2.7. This approach effectively allowed subtyping method parameters in the case of depth subtyping, provided that the relationship between parameter and return types guarantees type safety. The same approach can be applied to the family polymorphism. Then, the interface and multiple implementations can be defined in related but separate classes, making adding a new implementation as convenient as in the typical object-oriented case.

Another potential simplification is the treatment of singletons and executors. The current implementation passes the processing model (binary or decimal) as a parameter to functions that use it. In a binary-only emulator, the processing model elements would probably have been globally accessible as singletons. Moving from such a design to one with multiple processing models would require significant refactoring in different parts of the code.

As an alternative, Section 4.2 describes executors, which, among other things, hold singletons. Singletons are reachable as global objects, but the program can replace them during execution. Then, the code that chooses the emulator mode could set the processing model in the executor, and the rest of the code could access it as before.

6.4. Potential Future Development

A possible future improvement of the emulator is related to its interpretation of input/output operations determinacy. The conditions that the emulator imposes are very restrictive. For example, it would treat a program that periodically types `WAITING...` on the terminal until an input/output operation finishes as non-deterministic.

The emulator could be more permissive if it could track the range of possible virtual machine states instead of the exact state. As long as the difference in possible states stays reasonable, the execution continues (what exactly is reasonable is a matter of choice). The same approach would be helpful with the indeterminacy of the byte size. It would relieve the requirement of separate program runs in the binary and decimal modes.

However, tracking the range of states is complicated and unreasonable in most cases because most programs are deterministic. The emulator could track the exact state as long as possible and switch to the state boundaries mode when the behaviour is not deterministic anymore. The two modes would be significantly different, so it makes sense to apply the State pattern (Gamma et al., 1995). Section 2.9 describes chameleon objects, which would facilitate its implementation. The deterministic and non-deterministic implementations of the virtual machine would be different states that can convert to each other when needed.

CONCLUSIONS

The Doctoral Thesis is devoted to establishing a link between design patterns and expressive opportunities of programming languages. As known from the literature and experience, pattern-based design solutions are often associated with higher complexity, even though their original goal was the opposite. The proposed hypothesis is that the excessive complexity is partially caused by the insufficient expressiveness of programming languages, which struggle to convey mental constructs represented by patterns. In order to address this problem, the Doctoral Thesis proposes a set of generalisations describing general cases of various programming languages' constructs and formalises them using type-theoretical formalisms.

The tasks defined for the present work are fully fulfilled:

1. Analysis of programming languages' evolution trends and previously described difficulties in design patterns' implementation demonstrated problematic points even in modern programming languages.
2. Based on this analysis, the objectives and requirements for the desired type system are formulated.
3. The patterns of data composition and basic computation primitives, which provide generalisations of commonly used programming languages' constructs, are described.
4. The described patterns are formalised using a type-theoretical apparatus.
5. An emulator of the MIX computer is implemented in Scala to be used as an example practical project.
6. It is demonstrated how the proposed types could be beneficial in implementing the example project and its evolution.

The main result of the Doctoral Thesis is the developed system of types representing basic structural and computational patterns. The patterns describe general cases of typical constructs in programming languages (though actual languages may currently support a pattern in whole or only its special cases) and thus provide flexibility in modelling programmers' thoughts. Possession of such primitives in real programming languages could facilitate the implementation of design patterns and, more importantly, further development of an existing system.

Additional results of the work are:

1. The pattern-form descriptions of the described constructs explain how and why these constructs are used.
2. The list of known uses compares and contrasts implementations of the described patterns in various languages. It facilitates finding equivalent or similar constructs in different languages and understanding their limitations.
3. The methodology of describing language or library primitives as patterns to cover general cases and then formalising them as types can also be used for other constructs not covered by the Doctoral Thesis.
4. The developed emulator of MIX can be used in learning. It has certain benefits compared with other emulators, such as support for the decimal mode (in addition to the binary one) and verification of input/output synchronisation correctness.

The following conclusions can be made based on the conducted study:

1. Programming languages' constructs often represent only special cases of general patterns they implement. However, more expressive languages can support these patterns in their complete form.
2. More general cases of language constructs can be identified by describing them as patterns.
3. The described patterns are suitable for type-theoretical formalisation, which potentially allows them to serve as language constructs.
4. Even most expressive programming languages, such as Scala, have room for improvement concerning the described patterns.

The study can be continued in the following directions:

1. The same methodology can be applied to other constructs. For example, the Doctoral Thesis does not cover such essential aspects of a programming language as object lifecycle and memory management.
2. The general patterns described in the Doctoral Thesis can be implemented in new and existing programming languages.

BIBLIOGRAPHY

Moez A. AbdelGawad. A comparison of NOOP to structural domain-theoretic models of object-oriented programming. Preprint available at https://arxiv.org/abs/1603.08648, 2017.

Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Angel Shlomo. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press USA, 1977. ISBN 978-0-19-501919-3.

Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. C++ in-depth series. Addison-Wesley, 2001. ISBN 978-0-201-70431-0.

Pavol Bača and Valentino Vranić. Replacing object-oriented design patterns with intrinsic aspect-oriented design patterns. In: *Proceedings of the 2nd Eastern European Regional Conference on the Engineering of Computer Based Systems (ECBS-EERC)*, pages 19–26. IEEE, 2011. doi: 10.1109/ecbs-eerc.2011.13.

Ruslan Batdalov. Inheritance and class structure. In: Pavel P. Oleynik, editor, *Proceedings of the First International Scientific-Practical Conference Object Systems – 2010*, pages 92–95, 2010. URL https://cyberleninka.ru/article/n/inheritance-and-class-structure/pdf.

Ruslan Batdalov. Is there a need for a programming language adapted for implementation of design patterns? In: *Proceedings of the 21st European Conference on Pattern Languages of Programs (EuroPLoP '16)*, pages 34:1–34:3. Association for Computing Machinery, 2016. ISBN 978-1-4503-4074-8. doi: 10.1145/3011784.3011822.

Ruslan Batdalov. Comparative analysis of object-oriented programming languages in the context of language expressiveness. Master's thesis, Riga Technical University, 2017.

Ruslan Batdalov and Oksana Nikiforova. Towards easier implementation of design patterns. In *Proceedings of the Eleventh International Conference on Software Engineering Advances (ICSEA 2016)*, pages 123–128. IARIA, 2016.

Ruslan Batdalov and Oksana Ņikiforova. Implementation of a MIX emulator: A case study of the Scala programming language facilities. *Applied Computer Systems*, 22(1):47–53, 2017. doi: 10.1515/acss-2017-0017.

Ruslan Batdalov and Oksana Nikiforova. Three patterns of data type composition in programming languages. In: *Proceedings of the 23rd European Conference on Pattern Languages of Programs (EuroPLoP '18)*, pages 32:1–32:8. Association for Computing Machinery, 2018. ISBN 978-1-4503-6387-7. doi: 10.1145/3282308.3282341.

Ruslan Batdalov and Oksana Nikiforova. Elementary structural data composition patterns. In *Proceedings of the 24th European Conference on Pattern Languages of Programs (EuroPLoP '19)*, pages 26:1–26:13. Association for Computing Machinery, 2019. ISBN 978-1-4503-6206-1. doi: 10.1145/3361149.3361175.

Ruslan Batdalov and Oksana Nikiforova. Patterns for assignment and passing objects between contexts in programming languages. In: *Proceedings of the 26th European Conference on Pattern Languages of Programs (EuroPLoP '21)*, pages 4:1–4:9. Association for Computing Machinery, 2021. ISBN 978-1-4503-8997-6. doi: 10.1145/3489449.3489975.

Ruslan Batdalov, Oksana Nikiforova, and Adrian Giurca. Extensible model for comparison of expressiveness of object-oriented programming languages. *Applied Computer Systems*, 20(1):27–35, 2016. doi: 10.1515/acss-2016-0012.

Judith Bishop. *C# 3.0 Design Patterns*. O'Reilly Media, Sebastopol, CA, USA, 2008. ISBN 978-0-596-52773-0.

Grady Booch. The well-tempered architecture. *IEEE Software*, 24(4):24–25, 2007. ISSN 0740-7459. doi: 10.1109/ms.2007.122.

Jan Bosch. Design patterns as language constructs. *Journal of Object-Oriented Programming*, 11(2):18–32, 1998. ISSN 0896-8438.

Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture, A Pattern Language for Distributed Computing*, volume 4 of *Pattern-Oriented Software Architecture*. Wiley, 2007a. ISBN 978-0-470-06530-3.

Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture, On Patterns and Pattern Languages*, volume 5 of *Pattern-Oriented Software Architecture*. Wiley, 2007b. ISBN 978-0-470-51257-9.

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, A System of Patterns*, volume 1 of *Pattern-Oriented Software Architecture*. Wiley, 2013. ISBN 978-1-118-72526-9.

Travis Carlson and Eric Van Wyk. Type qualifiers as composable language extensions for code analysis and generation. *Journal of Computer Languages*, 50:49–69, 2019. ISSN 2590-1184. doi: 10.1016/j.jvlc.2018.10.008.

Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

William R. Cook, Walter Hill, and Peter S. Canning. Inheritance is not subtyping. In: *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 125–135, 1989

James O. Coplien. *Software Patterns*. SIGS management briefings. SIGS, 1996. ISBN 978-1-884842-50-4.

Ivica Crnković, Severine Séntilles, Aneta Vulgarakis, and Michel R. V. Chaudron. A classification framework for software component models. *IEEE Transactions on Software Engineering*, 37(5):593–615, 2011. ISSN 0098-5589. doi: 10.1109/tse.2010.83.

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991. ISSN 0164-0925. doi: 10.1145/115372.115320.

ECMA-334. *C# Language Specification*. Ecma International, sixth edition, 2022. Standard.

Francesca Arcelli Fontana, Stefano Maggioni, and Claudia Raibulet. Understanding the relevance of micro-structures for design patterns detection. *Journal of Systems and Software*, 84(12):2334–2347, 2011. ISSN 0164-1212. doi: 10.1016/j.jss.2011.07.006.

Francesca Arcelli Fontana, Stefano Maggioni, and Claudia Raibulet. Design patterns: a survey on their micro-structures. *Journal of Software-Evolution and Process*, 25(1):27–52, 2013. ISSN 2047-7481. doi: 10.1002/smr.547.

Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2012. ISBN 978-0-13-306521-3.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995. ISBN 978-0-201-63361-0.

Joseph Gil and David H. Lorenz. Design patterns and language design. *Computer*, 31(3):118–120, 1998. ISSN 0018-9162. doi: 10.1109/2.660196.

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification. Java SE 8 Edition*, 2015.

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. *The Java Language Specification. Java SE 21 Edition*, 2023.

Philip J. Guo. Online Python Tutor: Embeddable web-based program visualization for CS education. In: *Proceedings of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 579–584, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1868-6. doi: 10.1145/2445196.2445368.

Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. *ACM Sigplan Notices*, 37(11):161–173, 2002. ISSN 0362-1340. doi: 10.1145/583854.582436.

Tony Hoare. Null references: The billion dollar mistake. Available at https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare, August 2009.

ISO/IEC 14882:2020. *Information technology – Programming languages – C++*. ISO/IEC, 2020. Standard.

ISO/IEC 9075-2:2023. *Information technology – Database languages – SQL – Part 2: Foundation (SQL/Foundation)*. ISO/IEC, 2023. Standard.

Java® Platform, Standard Edition Core Libraries Release 21. Oracle, 2023. URL https://docs.oracle.com/en/java/javase/21/core/java-core-libraries1.html.

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In: Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings of the European conference on object-oriented languages (ECOOP '97)*, pages 220–242, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg. ISBN 978-3-540-69127-3.

Michael Kircher and Prashant Jain. *Pattern-Oriented Software Architecture, Patterns for Resource Management*, volume 3 of *Pattern-Oriented Software Architecture*. Wiley, 2013. ISBN 978-1-118-72523-8.

Donald E. Knuth. *The Art of Computer Programming: Fundamental algorithms*. Addison-Wesley series in computer science and information processing. Addison-Wesley, 1997. ISBN 978-0-201-89683-1.

Kotlin Language Documentation 1.9.0. JetBrains, 2023. URL https://kotlinlang.org/docs/kotlin-reference.pdf.

Chris Lattner and Vikram S. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In: *International Symposium on Code Generation and*

Optimization (CGO 2004), pages 75–86, New York, NY, USA, 2004. IEEE. doi:10.1109/CGO.2004.1281665.

António Leitão and Sara Proença. On the expressive power of programming languages for generative design: the case of higher-order functions. In: *Proceedings of the 32nd International Conference on Education and Research in Computer Aided Architectural Design in Europe (eCAADe)*, pages 257–266, 2014.

George Leontiev, Eugene Burmako, Jason Zaugg, Adriaan Moors, Paul Phillips, Oron Port, and Miles Sabin. SIP-23 - literal-based singletone types. Scala Improvement Proposal, 2019.

Fredrik Skeel Løkke. Scala & design patterns. Master's thesis, University of Aarhus, March 2009.

Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Robert C. Martin Series. Pearson Education, 2008. ISBN 978-0-13-608325-2.

Greg Michaelson. *An introduction to functional programming through lambda calculus*. Dover Publications, Inc., 2011. ISBN 978-0-486-47883-8.

Miguel P. Monteiro and João Gomes. Implementing design patterns in Object Teams. *Software: Practice and Experience*, 43(12):1519–1551, 2013. ISSN 1097-024X. doi:10.1002/spe.2154.

Peter D. Mosses. Formal semantics of programming languages: An overview. *Electronic Notes in Theoretical Computer Science*, 148(1):41–73, 2006. ISSN 1571-0661. doi:10.1016/j.entcs.2005.12.012.

Martin Odersky, Philippe Altherr, Vincent Cremet, Sébastien Doeraene, Gilles Dubochet, Burak Emir, Philipp Haller, Stéphane Micheloud, Nikolay Mihaylov, Adriaan Moors, Lukas Rytz, Michel Schinz, Erik Stenman, and Matthias Zenger. Scala Language Specification: Version 3.4, 2023. URL https://scala-lang.org/files/archive/spec/3.4/.

Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Lex Spoon, and Matthias Zenger. An overview of the Scala programming language. Technical Report LAMP-REPORT-2006-001, École Polytechnique Fédérale de Lausanne (EPFL), 2006.

Martin Odersky and Lex Spoon. *Scala Collections*, 2023. URL http://docs.scala-lang.org/overviews/collections/introduction.html.

PHP 8.2 Language Reference. PHP Group, 2023.

Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. ISBN 978-0-262-16209-8.

Terrence W. Pratt and Marvin V. Zelkowitz. *Programming Languages: Design and Implementation*. Prentice Hall PTR, fourth edition, 2001. ISBN 978-0-13-027678-0.

Dimitri Racordon and Didier Buchs. Implementing a language with explicit assignment semantics. In: *Proceedings of the 11th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL 2019)*, pages 12–21, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 978-1-4503-6987-9. doi:10.1145/3358504.3361227.

Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*, volume 2 of *Pattern-Oriented Software Architecture*. Wiley, 2013. ISBN 978-1-118-72517-7.

Peter Sommerlad. Design patterns are bad for software design. *IEEE Software*, 24(4):68–71, 2007. ISSN 0740-7459. doi: 10.1109/ms.2007.116.

Antero Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):438–479, Sep 1996. ISSN 0360-0300. doi: 10.1145/243439.243441.

The Go Programming Language Specification. Google, 2023.

The Python Language Reference 3.12.0. Python Software Foundation, 2023.

TypeScript Documentation. Microsoft Corporation, 2023. URL https://www.typescriptlang.org/docs/.

Unified Modeling Language Version 2.5.1. OMG, 2017. URL https://www.omg.org/spec/UML/2.5.1. Standard.

Philip Wadler. Comprehending monads. In: *Mathematical Structures in Computer Science*, pages 61–78, 1992a.

Philip Wadler. The essence of functional programming. In: *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14. Association for Computing Machinery, 1992b.

Uwe Zdun and Paris Avgeriou. A catalog of architectural primitives for modeling architectural patterns. *Information and Software Technology*, 50(9–10):1003–1034, 2008. ISSN 0950-5849. doi: 10.1016/j.infsof.2007.09.003.

Olaf Zimmermann, Mirko Stocker, Daniel Lübke, and Uwe Zdun. Interface representation patterns: Crafting and consuming message-based remote APIs. In: *Proceedings of the 22nd European Conference on Pattern Languages of Programs (EuroPLoP '17)*, pages 27:1–27:36. Association for Computing Machinery, 2017. ISBN 978-1-4503-4848-5. doi:10.1145/3147704.3147734.

Ruslan Batdalov was born in 1983 in Izhevsk, Russia. He obtained a mathematician's and system programmer's qualification in Applied Mathematics and Informatics from Kazan State University (2003, with distinction) and a Master's degree in Computer Systems from Riga Technical University (2017, with distinction). He has worked at "Smart Home" Ltd., CBOSS, Moscow City telephone network, Riga Technical University, and Rietumu Banka. Currently, he is a software engineer at Google. Since 2010, he has been a member of ACM. His research interests are related to design patterns and the expressiveness of programming languages.