

SHOULD BE REPLACED ON REQUIRED TITLE PAGE

Instruction

1. Open needed docx template (folder "title"/<your department or bach if bachelor student>.docx).
2. Put Thesis topic, supervisor's and your name in appropriate places on both English and Russian languages.
3. Put current year (last row).
4. Convert it to "title.pdf," replace the existing one in the root folder.

Contents

1	Introduction	7
1.1	Spacing & Type	7
1.1.1	Creating a Subsection	7
1.2	Theorems, Corollaries, Lemmas, Proofs, Remarks, Definitions and Examples	8
1.3	Optional table of contents heading	9
2	Literature Review	11
2.1	The Concept and Components of the Object Model	11
2.1.1	Major Components of the Object Model	13
2.1.2	Minor Components of the Object Model	17
2.1.3	The Selected Definition of the Object Model	20
3	Methodology	21
4	Implementation	22
5	Evaluation and Discussion	23
6	Conclusion	24

CONTENTS	3
-----------------	----------

Bibliography cited	25
---------------------------	-----------

A Extra Stuff	27
----------------------	-----------

B Even More Extra Stuff	28
--------------------------------	-----------

List of Tables

List of Figures

- 1.1 One kernel at x_s (*dotted kernel*) or two kernels at x_i and x_j (*left and right*) lead to the same summed estimate at x_s . This shows a figure consisting of different types of lines. Elements of the figure described in the caption should be set in italics, in parentheses, as shown in this sample caption. 8

Abstract

Данная работа посвящена сравнительному анализу объектных моделей современных языков программирования, включая C#, C++, Golang, Java, Python, Ruby, JavaScript, Scala, Smalltalk и Zonnon. Актуальность работы обусловлена широким распространением объектно-ориентированной парадигмы и значительным разнообразием в ее реализации, что создает сложности для осознанного выбора технологий и архитектурных решений. Целью исследования является выявление сходств, различий, сильных и слабых сторон различных объектных моделей, а также разработка на этой основе предложений по созданию усовершенствованной модели. В результате исследования систематизированы теоретические основы объектных моделей, проведен детальный разбор их реализации в выбранных языках с выявлением областей применения и типичных ошибок, а также предложена авторская модель, интегрирующая лучшие практики. Практическая значимость работы заключается в том, что ее результаты могут быть использованы разработчиками для выбора языка и парадигм, архитекторами — для проектирования систем.

Chapter 1

Introduction

1.1 Spacing & Type

1.1.1 Creating a Subsection

Creating a Subsubsection

Creating a Subsubsection

Creating a Subsubsection

This is a heading level below subsubsection And this is a quote:

This is a table:

The package `**upgreek**` allows us to use non-italicized lower-case greek letters. See for yourself: β , β , β , β . Next is a numbered equation:

$$\|\mathbf{X}\|_{2,1} = \underbrace{\sum_{j=1}^n f_j(\mathbf{X})}_{\text{convex}} = \sum_{j=1}^n \|\mathbf{X}_{:,j}\|_2 \quad (1.1)$$



Fig. 1.1. One kernel at x_s (*dotted kernel*) or two kernels at x_i and x_j (*left and right*) lead to the same summed estimate at x_s . This shows a figure consisting of different types of lines. Elements of the figure described in the caption should be set in italics, in parentheses, as shown in this sample caption.

The reference to equation (1.1) is clickable.

1.2 Theorems, Corollaries, Lemmas, Proofs, Remarks, Definitions, and Examples

Theorem 1. *Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.*

Proof. I'm a (very short) proof.



Lemma 1. *I'm a lemma.*

Corollary 1. *I include a reference to Thm. 1.*

Proposition 1. *I'm a proposition.*

Remark. I'm a remark.

Definition 1. I'm a definition. I'm a definition. I'm a definition. I'm a definition.
I'm a definition. I'm a definition. I'm a definition. I'm a definition. I'm a
definition. I'm a definition. I'm a definition.

Example. I'm an example.

1.3 Section with linebreaks in the name

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

This is the second paragraph. Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Chapter 2

Literature Review

2.1 The Concept and Components of the Object Model

The object model represents a fundamental framework that defines how a programming language represents and supports objects, classes, inheritance, encapsulation, polymorphism, and other related abstractions. According to commonly accepted terminology, the term object model has two interrelated meanings: (1) the properties of objects within a particular programming language, technology, notation, or methodology that employs these objects; and (2) a set of interfaces or classes through which a program can explore and manipulate specific aspects of its environment. Examples of object models include the Java Object Model, the Component Object Model (COM), and the Object Modeling Technique (OMT). Such object models are typically defined using concepts such as class, generic function, message, inheritance, polymorphism, and encapsulation.

Cardelli and Wegner in their work “On Understanding Types, Data Abstrac-

tion, and Polymorphism,” defined object-oriented languages through three key requirements [1]. A language is considered object-oriented if it satisfies the following conditions:

- Support objects that represent data abstractions with an interface composed of named operations and an encapsulated internal state
- Objects in the language are associated with a specific object type
- Types may inherit attributes from their supertypes

These requirements were formulated in the form of a formula:

$$\textit{object - oriented} = \textit{dataabstractions} + \textit{objecttypes} + \textit{typeinheritance}$$

Booch, in his seminal work *Object-Oriented Analysis and Design with Applications*, describes the object model as a conceptual representation of the organized complexity of software systems [2].

Booch (p. 40-41) concludes that the conceptual framework for anything object-oriented is the object model—a conceptual representation of the organized complexity of software. It consists of four major elements, i.e. abstraction, encapsulation, modularity, and hierarchy. In addition, the object model contains three minor element, i.e. typing, concurrency, and persistence.

This distinction between major and minor elements is fundamental: without the major elements, the model ceases to be object-oriented, whereas the minor elements are useful but not essential for supporting object orientation.

2.1.1 Major Components of the Object Model

The major components of the object model form the essential foundation upon which object-oriented programming is built. Without these four elements, a programming language cannot achieve true object orientation, as they collectively define how systems are decomposed, organized, and managed at the conceptual level.

Abstraction

Abstraction is the process of extracting essential characteristics of an object or concept while suppressing unnecessary implementation details. As Liskov [3] emphasizes, data abstraction is a valuable method for organizing programs to make them easier to modify and maintain.

According to Liskov's foundational work [3], a data abstraction is characterized by the separation of what an abstraction is from how it is implemented, such that implementations of the same abstraction can be substituted freely. The implementation of a data object is concerned with how that object is represented in memory, and this information is hidden from programs that use the abstraction by restricting those programs so that they cannot manipulate the representation directly, but instead must call the operations defined by the abstraction.

In the context of the object model, abstraction manifests through the definition of interfaces that specify what operations an object can perform, while concealing how these operations are actually implemented. This is achieved through the use of abstract classes, interfaces, and public method signatures that represent the contract between the object and its users.

Encapsulation

Encapsulation is the bundling of data (attributes) and the methods that operate on that data into a single cohesive unit, typically implemented as a class. More importantly, encapsulation involves the enforcement of access controls that restrict direct manipulation of an object's internal state. This concept originated from Parnas's pioneering work on information hiding [4], which established the principle that modules should be designed such that they hide information from other modules—information that is likely to change.

Parnas [4] articulated this principle by stating:

Every module in the second decomposition is characterized by its knowledge of a design decision which it hides from all others. Its interface or definition was chosen to reveal as little as possible about its inner workings.

This foundational idea has become the basis for encapsulation in object-oriented programming.

Through encapsulation, the internal representation of an object is hidden from external access, and interaction with the object is mediated exclusively through its public interface—a set of methods deliberately exposed for this purpose. The value of encapsulation extends beyond mere data protection. It provides a stable interface that enables developers to refactor and optimize the internal implementation without affecting dependent code. Furthermore, it facilitates the implementation of validation logic within setter methods, ensuring that objects never enter invalid states. This is particularly important in complex systems where maintaining object consistency is critical to correctness.

Modularity

Modularity is the principle of designing software systems as collections of discrete, largely independent modules or components that can be designed, implemented, and maintained separately. Parnas [4] established the foundational principles for modular decomposition, emphasizing that modules should be designed and implemented independently, be simple enough to be fully understandable, and possess the ability to change a module's implementation without affecting other modules' behavior.

In the context of object-oriented programming, modularity is achieved primarily through the organization of classes and objects into cohesive units with clear, well-defined boundaries and minimal interdependencies. As Liskov [3] notes, data abstractions provide a mechanism for organizing programs into modules where each module is responsible for implementing a specific abstraction.

Modularity in the object model manifests in several ways. At the most granular level, individual classes encapsulate a single, well-defined responsibility. At a higher level, packages or namespaces group related classes together. The module structure creates a hierarchy of abstractions, allowing developers to understand a system at different levels of detail.

Hierarchy

Hierarchy refers to the organization of abstractions into ordered structures where more general or abstract concepts are related to more specific or concrete ones through "is-a" and "part-of" relationships. In object-oriented programming, hierarchy is realized through two primary mechanisms: inheritance hierarchies (class structure) and compositional hierarchies (object structure).

In inheritance hierarchies, classes are organized into tree or lattice structures where subclasses inherit attributes and behaviors from their superclasses. When a subclass B is a subclass of a superclass A , every instance of B is also considered an instance of A , and B is said to inherit all features and behaviors from A . This relationship enables the creation of generalization-specialization hierarchies, where abstract base classes define common behavior applicable to a family of related classes, and derived classes specialize this behavior for specific scenarios.

The principle underlying such structures is the Liskov Substitution Principle, articulated by Liskov [3] and formalized in subsequent work [5]. This principle stipulates that objects of a base type should be replaceable by objects of derived types without altering the correctness of the program. In formal terms, if S is a subtype of T , then objects of type S may be substituted for objects of type T without breaking the invariants that hold for T .

In compositional hierarchies, objects combine simpler objects through the "part-of" relationship, creating structures where complex objects are aggregates of simpler components. This organizational principle allows for the construction of systems of arbitrary complexity from well-understood, reusable building blocks. As noted in the seminal work on design patterns [6], composition provides a more flexible alternative to inheritance, enabling runtime changes to object relationships that would otherwise be fixed at compile time.

Hierarchy provides several critical benefits: it enables the management of complexity through layered abstraction; it promotes code reuse through inheritance of common behavior; it establishes natural relationships between concepts that mirror real-world domain structures; and it facilitates polymorphism and dynamic dispatch, allowing for flexible, extensible program architectures.

2.1.2 Minor Components of the Object Model

The minor components are features that enhance the expressiveness and applicability of the object model but are not strictly necessary for a language to be considered object-oriented. Their presence in modern programming languages reflects the evolution of object-oriented principles to address practical challenges in contemporary software development.

Typing

Typing refers to the system of rules and mechanisms that govern how data types are associated with variables, expressions, and values in a programming language, and how operations are permitted on values of different types. A type system assigns specific types to each element in a program, defining both the kind of data that can be stored and the operations permissible on that data.

Milner's work on type inference in the ML programming language [7] established foundational principles for polymorphic type systems. The Hindley-Milner type system [8] allows type inference where the types of variables can be deduced from programs written in an entirely untyped style, while still supporting parametric polymorphism—the ability for a single definition to be applied to values of different types.

In object-oriented languages, typing serves multiple purposes. Static type checking, performed at compile time, allows detection of many categories of errors before program execution, increasing reliability and enabling certain optimizations. Type hierarchies, based on inheritance relationships, enable polymorphism—the ability for different types to respond to the same method call in type-specific ways. The relationship between types and subtypes is governed

by formal principles: in particular, subtype polymorphism allows objects of a supertype to be replaced by objects of a subtype, provided the subtype respects the contract established by the supertype.

Concurrency

Concurrency refers to the ability of a system to manage multiple computations that execute logically in parallel, either through true simultaneous execution on multi-processor systems or through interleaved execution on single-processor systems. In the context of the object model, concurrency addresses how objects maintain consistency and correctness when multiple threads or processes access and modify them simultaneously.

The integration of concurrency into object-oriented systems introduces significant complexity, as the mutable state encapsulated within objects may be accessed by concurrent executions. Johnsen and Owe [9] propose programming constructs for concurrent objects based on processor release points and asynchronous method calls, which allow objects to dynamically change between active and reactive behavior in a distributed environment. Active objects, as described by Lavender and Schmidt [10], decouple method execution from method invocation to simplify synchronized access to shared resources.

Different object-oriented languages address this challenge through various mechanisms: some employ mutual exclusion locks to ensure that only one thread can execute a particular method on an object at a time; others support active objects that manage their own internal threads; still others provide transactional semantics where concurrent operations are executed in a serializable manner, guaranteeing consistency.

Effective concurrency support is increasingly important in modern object-oriented systems, particularly those designed for distributed computing or multi-core processor environments. However, as concurrency mechanisms can significantly complicate both language design and program development, it remains a non-essential component of the object model. Many successful object-oriented languages provide limited concurrency support or delegate concurrency management to external libraries or frameworks.

Persistence

Persistence refers to the ability to store and retrieve objects or their state across program executions, typically through mechanisms such as serialization, object-relational mapping, or object-oriented databases. Without persistence, all objects created during program execution are lost when the program terminates, limiting the applicability of object-oriented programming to stateless or short-lived computations.

Atkinson and Morrison [11] introduced the concept of orthogonal persistence, which adheres to three fundamental principles: the principle of persistence independence (programs look the same whether they manipulate long-term or short-term data), the principle of data type orthogonality (all data objects are allowed to be persistent irrespective of their type), and the principle of persistence identification (the mechanism for identifying persistent objects is not related to the type system).

The object model incorporates persistence through various approaches. In some languages, persistence is achieved implicitly through orthogonal persistence, where objects are automatically saved and restored through special mechanisms

without requiring explicit serialization code [11]. In other approaches, persistence is achieved through explicit programming, where objects must implement serialization interfaces or developers must write code to translate objects to and from storage representations.

Several strategies for making objects persistent have been proposed: persistence by class, where all objects of certain classes are automatically persistent; persistence by creation, where special syntax creates persistent objects; persistence by marking, where objects are explicitly marked as persistent after creation; and persistence by reference, where designated root objects are declared persistent and all objects reachable from them are transitively made persistent.

2.1.3 The Selected Definition of the Object Model

Based on the conducted literature review, the following definition of the object model is adopted for the purposes of this work:

The object model is a conceptual framework that integrates a set of interrelated components — four major ones (abstraction, encapsulation, modularity, and hierarchy) and three minor ones (typing, concurrency, and persistence). It defines the means by which a programming language represents, organizes, and supports objects, classes, their interactions, inheritance relationships, mechanisms of polymorphism, and other related abstractions for the effective management of software system complexity.

Chapter 3

Methodology

Referencing other chapters 2, 3, 4, 5 and 6

...

Chapter 4

Implementation

...

Chapter 5

Evaluation and Discussion

...

Chapter 6

Conclusion

...

Bibliography cited

- [1] L. Cardelli and P. Wegner, “On understanding types, data abstraction, and polymorphism,” *ACM Computing Surveys (CSUR)*, vol. 17, no. 4, pp. 471–523, 1985.
- [2] G. Booch, R. A. Maksimchuk, M. W. Engle, B. J. Young, J. Connallen, and K. A. Houston, “Object-oriented analysis and design with applications,” *ACM SIGSOFT software engineering notes*, vol. 33, no. 5, pp. 29–29, 2008.
- [3] B. Liskov, “Keynote address-data abstraction and hierarchy,” in *Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum)*, 1987, pp. 17–34.
- [4] D. L. Parnas, “On the criteria to be used in decomposing systems into modules,” *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.
- [5] B. H. Liskov and J. M. Wing, “Behavioral subtyping using invariants and constraints,” Tech. Rep., 1999.
- [6] E. Gamma, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995, vol. 431.

- [7] R. Milner, “A theory of type polymorphism in programming,” *Journal of computer and system sciences*, vol. 17, no. 3, pp. 348–375, 1978.
- [8] R. Hindley, “The principal type-scheme of an object in combinatory logic,” *Transactions of the american mathematical society*, vol. 146, pp. 29–60, 1969.
- [9] E. B. Johnsen, O. Owe, and M. Arnestad, “Combining active and reactive behavior in concurrent objects,” in *Proc. of the Norwegian Informatics Conference (NIK’03)*, 2003, pp. 193–204.
- [10] R. G. Lavender and D. C. Schmidt, “Active object: An object behavioral pattern for concurrent programming,” in *Pattern languages of program design 2*, 1996, pp. 483–499.
- [11] M. Atkinson and R. Morrison, “Orthogonally persistent object systems,” *The VLDB Journal*, vol. 4, no. 3, pp. 319–401, 1995.

Appendix A

Extra Stuff

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Appendix B

Even More Extra Stuff

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.