Автономная некоммерческая организация высшего образования
«Университет Иннополис»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
(МАГИСТЕРСКАЯ РАБОТА)
по направлению подготовки
09.04.01 - «Информатика и вычислительная техника»

GRADUATION THESIS
(MASTER'S GRADUATION THESIS)
Field of Study
09.04.01 – «Software Engineering»

Направленность (профиль) образовательной программы
«Информатика и вычислительная техника»
Area of Specialization / Academic Program Title:
«Software Engineering»

Тема / Topic	**Анализ объектных моделей в современных языках программирования/ Analysis of object models in modern programming languages**

Работу выполнил / Thesis is executed by	**Шалагин Егор Сергеевич / Shalagin Egor**	подпись / signature
Руководитель выпускной квалификационной работы / Supervisor of Graduation Thesis	**Зуев Евгений Александрович / Zouev Eugene**	подпись / signature
Консультанты / Consultants		подпись / signature

Иннополис, Innopolis, 2025

Contents

1 Introduction **7**

2 Literature Review **9**

2.1 The Concept and Components of the Object Model 9

2.1.1 Major Components of the Object Model 11

2.1.2 Minor Components of the Object Model 14

2.1.3 The Selected Definition of the Object Model 15

2.2 Critical Analysis of Object-Oriented Programming Approach . . 17

2.2.1 Theoretical Inconsistencies in Inheritance and Subtyping 17

2.2.2 The Fragile Base Class Problem 18

2.2.3 Excessive Coupling and Tight Interdependencies 20

2.2.4 Multiple Inheritance and Ambiguity Resolution 25

2.2.5 The Reusability Paradox 25

2.2.6 State Management Complexity and Object Identity . . . 26

2.2.7 Absence of Clear Separation Between Specification and Implementation 27

2.2.8 Empirical Evidence of Design Difficulties 27

2.2.9 Systematic Classification of OOP Limitations 28

2.3 Review of Related Work 30

2.3.1 Fundamental taxonomies and theoretical foundations . . 30
2.3.2 Empirical Performance Comparisons 31
2.3.3 Narrowly Focused and Engineering Comparisons 31
2.3.4 Conclusion 32
2.4 Methodology of Comparative Analysis 33

3 Methodology 36

4 Implementation 37

5 Evaluation and Discussion 38

6 Conclusion 39

Bibliography cited 40

List of Tables

List of Figures

2.1 Inheritance without proper subtyping 19
2.2 Inheritance without proper subtyping: manifestation 20
2.3 Fragile Base Class: first implementation base class 21
2.4 Fragile Base Class: first implementation deriving class 22
2.5 Fragile Base Class: misspelled implementation base class 23
2.6 Fragile Base Class: misspelled implementation deriving class . . 24
2.7 Fragile Base Class: demonstration 24

Abstract

This work conducts a systematic comparison of the object models in ten modern programming languages: C#, C++, Golang, Java, Python, Ruby, JavaScript, Scala, Smalltalk, and Zonnon. The research is motivated by the operational challenges faced by software engineers when choosing technologies given the significant differences in how object models are implemented in different language ecosystems. The main objective of the research is to identify the similarities, differences, strengths, and weaknesses of various object models, and based on this analysis, to develop proposals for an enhanced model. As a result of the study, the theoretical foundations of models are systematized, a detailed examination of their implementation in the selected languages is conducted, identifying application domains and common pitfalls, and an original model integrating best practices is proposed. The results obtained serve as a practical guide for developers when choosing a language and for software architects when designing object-oriented systems.

Chapter 1

Introduction

Object-oriented programming continues to be one of the fundamental paradigms in the software industry. However, despite decades of its existence, a unified standard for its implementation has never been established. Modern programming languages offer a variety of object models: from the strict and static ones in Java and C# to the prototype-based in JavaScript and the minimalist in Golang. This diversity, on one hand, provides developers with freedom of choice, but on the other, creates inconveniences when applying these approaches in various situations. Most of the research conducted focuses on individual languages or syntax, without offering a systematic understanding of the architectural principles underlying their object systems.

The goal of this master's thesis is to conduct a comprehensive comparative analysis of the object models of a number of modern programming languages and, based on it, to develop an original, enhanced model. To achieve this goal, the work addresses a series of tasks: it unveils the theoretical foundations of object models, analyzes the criticism directed at OOP, and conducts a detailed analysis of the implementation of models in languages such as C#, C++, Golang,

Java, Python, Ruby, JavaScript, Scala, Smalltalk, and Zonnon, identifying their advantages, disadvantages, and typical application areas. The practical section illustrates characteristic usage errors with specific examples. The final part presents the result of a comparative analysis with the identified common trends and fundamental differences, which serves as the basis for the proposal of the author's unified object model designed to mitigate the shortcomings of existing approaches identified during the study.

The object of the study is the object models themselves, and the subject is their architectural principles, similarities, differences, and practical aspects of implementation. The methodological foundation of the work consists of theoretical methods of analysis and systematization, as well as practical methods of comparative analysis based on the study of documentation and code writing. The theoretical significance of the work lies in the systematization of knowledge about object models, while the practical significance lies in the proposal of a new, more effective model for future developments. The structure of the work includes....

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:
$object - oriented = dataabstractions + objecttypes + typeinheritance$

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 four major components of the object model form the fundamental basis on which object-oriented programming is built. Without these four elements, a programming language cannot achieve true object orientation, since they collectively determine how systems are broken down into components, 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.

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.

Encapsulation hides the internal representation of an object from external access, enabling interaction with the object exclusively through its public interface—a set of methods specifically provided for this purpose. Besides providing data protection, encapsulation provides a stable interface that allows developers to refactor and optimize the internal implementation without affecting dependent code.

Modularity

The principle of modularity is designing software systems as sets of discrete, independent units or components that can be designed, implemented, and maintained separately. Parnas' [4] fundamental idea of modular decomposition: modules should be designed and implemented independently, be simple enough to be fully understandable, and allow changing the implementation of a module without affecting the behavior of other modules.

In object-oriented programming, modularity comes about by packaging classes and objects into coherent units with well-defined boundaries and minimal interdependencies. According to Liskov [3], the data abstractions offer

a mechanism to structure the programs into modules in which each module is responsible for implementing an abstraction.

Hierarchy

Hierarchy is the organization of abstractions into ordered structures in which more general or abstract concepts are connected to more specific or concrete concepts through "is-a" and "part of" relationships. In object-oriented programming, hierarchy is implemented using two main mechanisms: inheritance hierarchies and composition hierarchies.

In inheritance hierarchies, classes are organized into tree-like or lattice structures in which subclasses inherit attributes and behavior from their superclasses. The principle underlying such structures is Liskov's substitution principle, formulated by Liskov [3] and formalized in subsequent works [5]. This principle states that objects of a base type must be replaceable with objects of derived types without changing the correctness of the program.

In composition hierarchies, objects combine simpler objects through a "part-of" relationship, creating structures where complex objects are aggregates from simpler components. This organizational principle allows systems of any complexity to be built from well-understood, reusable building blocks. As noted in the seminal work on design patterns [6], composition provides a more flexible alternative to inheritance, allowing changes to be made to the relationships between objects at runtime 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 improve the object model's expressiveness and usability, but aren't strictly needed for a language to be considered as object-oriented. Presence of this components in modern programming languages shows the evolution of object-oriented principles for solving practical problems in modern software development.

Typing

Typing is defined a system of rules and mechanisms that determine how data types are associated with variables, expressions, and values, as well as how operations on values of different types are permitted. The type system assigns specific types to each element in the program, defining both the kind of data that can be stored and the operations that are allowed on that data.

Concurrency

Concurrency is defined as the ability of a system to manipulate several computations that are logically parallel, either through true simultaneous execution on multiprocessor systems or through alternating execution on single-processor systems. In the context of the object model, concurrency resolves the issue on how objects maintain consistency and correctness when multiple threads or processes simultaneously access and modify them.

Different object-oriented languages solve the problem of synchronizing values during parallel execution using different mechanisms. Some use 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, in which parallel operations are executed sequentially, ensuring consistency.

Persistence

Persistence refers to the ability to store and retrieve objects or their state between program runs. Without persistence, all objects created during program execution are lost when the program terminates, which limits the applicability of object-oriented programming to stateless or short-lived computations.

The object model includes persistence through various approaches. The orthogonal persistence approach automatically saves and restores objects using special mechanisms without the need for explicit serialization code. With explicit programming, objects must implement serialization interfaces [7], or developers must write code to convert objects to storage representations and back.

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.

2.2 Critical Analysis of Object-Oriented Programming Approach

The object-oriented programming paradigm, despite its significant influence on software development practices, demonstrates fundamental limitations and unresolved problems that hinder its applicability and effectiveness in various problem-solving areas. This section presents a systematic and critical examination of the main shortcomings of OOP. The critical perspective developed here provides a theoretical basis for evaluating the actual implementation of object models in modern programming languages.

2.2.1 Theoretical Inconsistencies in Inheritance and Subtyping

A fundamental issue in typed object-oriented languages concerns the conflation of inheritance with subtyping relations. Cook, Hill, and Canning demonstrate that inheritance, when used as a mechanism for incremental modification of recursive object definitions, does not necessarily produce subtypes [8]. The traditional assumption that inheritance hierarchies determine conformance relations proves inadequate when dealing with recursive types and contravariant method types.

The distinction between inheritance and subtyping stems from type-theoretic considerations. Cardelli and Wegner in their analysis of type systems establishes that inheritance as an implementation mechanism and subtyping represent orthogonal concerns [1]. Their concepts demonstrates that subtyping relations require adherence to the Liskov substitution principle, whereas inheritance mechanisms impose additional constraints that compromise expressiveness [9]. The consequence of this theoretical inconsistency is that inheritance-based type hierarchies

cannot enforce type safety without either compromising expressiveness or introducing type insecurities. Eiffel exemplifies this problem: by identifying classes with types and inheritance with subtyping, the language exhibits type insecurities as documented by Cardelli and Wegner [1].

The following C# example 2.1 demonstrates how inheritance does not automatically guarantee proper subtyping behavior 2.2, particularly with covariance and contravariance in method signatures:

2.2.2 The Fragile Base Class Problem

One of the most significant architectural deficiencies in inheritance-based OOP systems is the fragile base class problem, formally studied by Mikhajlov and Sekerinski [10]. This problem occurs in open object-oriented systems employing code inheritance as an implementation reuse mechanism: seemingly safe modifications to a base class can cause derived classes to malfunction, despite no explicit violation of method contracts or public interfaces. Mikhajlov and Sekerinski distinguish between two manifestations: the syntactic fragile base class problem, which necessitates recompilation of extension and client classes when base classes change, and the semantic fragile base class problem, wherein self-recursion vulnerabilities create situations where base class revisions break extension class functionality without changing the external interface [10].

The fundamental cause lies in the interaction between dynamic dispatch, self-reference (encoded as self or this), and encapsulation boundaries. When a subclass overrides methods that the base class uses internally through dynamic dispatch, changes to the base class implementation can alter method invocation sequences, causing subclass assumptions about execution order to be violated.

```
public abstract class Animal
{
    public virtual void Reproduce(Animal mate)
    {
        Console.WriteLine("Animals reproducing");
    }

    public abstract void Feed(Food food);
}

public class Dog : Animal
{
    // VIOLATION: Contravariant parameter - accepts broader type
    // This breaks Liskov Substitution Principle when used polymorphically
    public override void Reproduce(Animal mate)
    {
        if (!(mate is Dog))
            throw new ArgumentException("Dogs can only reproduce with dogs");
        Console.WriteLine("Dogs reproducing");
    }

    // VIOLATION: Covariant return would be OK, but Food parameter is problematic
    public override void Feed(Food food)
    {
        Console.WriteLine($"Dog eating: {food.Name}");
    }
}

public class Cat : Animal
{
    public override void Reproduce(Animal mate)
    {
        if (!(mate is Cat))
            throw new ArgumentException("Cats can only reproduce with cats");
        Console.WriteLine("Cats reproducing");
    }

    public override void Feed(Food food)
    {
        Console.WriteLine($"Cat eating: {food.Name}");
    }
}

public class Food { public string Name { get; set; } }
```

Fig. 2.1. Inheritance without proper subtyping

```
Animal dog = new Dog();
Animal cat = new Cat();

// This code violates the contract established by the base class
// The compiler allows it, but runtime fails - TYPE INSECURITY
try
{
    dog.Reproduce(cat); // Runtime exception: "Dogs can only reproduce with dogs"
}
catch (ArgumentException ex)
{
    Console.WriteLine($"Type safety violation: {ex.Message}");
}
```

Fig. 2.2. Inheritance without proper subtyping: manifestation

This coupling violation fundamentally undermines the promise of modular reuse through inheritance. In open systems where subclasses are developed independently of base class implementations, predicting the consequences of base class modifications becomes impractical [10].

2.2.3 Excessive Coupling and Tight Interdependencies

The inheritance mechanism introduces structural coupling between base and derived classes that compromises modularity and reusability. Booch's analysis of class quality metrics identifies coupling as a critical measure of design quality, noting that inheritance creates strong coupling between superclasses and subclasses [2]. This coupling creates dependencies where modifications in parent classes propagate through inheritance chains, a phenomenon that Hitz and Montazeri characterize as particularly problematic: excessive coupling between object classes is detrimental to modular design and prevents reuse of individual components in alternative applications [11].

Beyond inheritance-induced coupling, OOP systems exhibit architectural issues with circular dependencies among classes. Circular dependencies create

```
public class BankAccount
{
    protected decimal _balance;

    public virtual void Deposit(decimal amount)
    {
        _balance += amount;
        Console.WriteLine($"Deposited {amount}. Balance: {_balance}");
    }

    public virtual void Withdraw(decimal amount)
    {
        if (amount <= _balance)
        {
            _balance -= amount;
            Console.WriteLine($"Withdrawn {amount}. Balance: {_balance}");
        }
    }

    public virtual decimal GetBalance()
    {
        return _balance;
    }
}
```

Fig. 2.3. Fragile Base Class: first implementation base class

tight mutual coupling that reduces the possibility of separate reuse and creates domino effects where changes in one module spread unwanted side effects to dependent modules. This architectural pattern violates the foundational principle of modular decomposition, particularly problematic in large-scale systems where dependency management becomes increasingly complex.

The Gang of Four Design Patterns documentation explicitly acknowledges this limitation, stating that inheritance exposes a subclass to details of its parent's implementation, and therefore "inheritance breaks encapsulation" [6]. The authors warn that implementation of a subclass can become so bound to the implementation of its parent that any change in the parent's implementation forces the subclass to change. They further note that polymorphism combined with implementation inheritance can create systems where it is difficult to predict which

```
public class PremiumAccount : BankAccount
{
    private decimal _monthlyFee = 10m;
    private int _freeTransactions = 10;
    private int _transactionCount = 0;

    public override void Deposit(decimal amount)
    {
        _transactionCount++;
        if (_transactionCount > _freeTransactions)
        {
            base.Deposit(amount - _monthlyFee); // Account for fee
        }
        else
        {
            base.Deposit(amount);
        }
    }

    public override void Withdraw(decimal amount)
    {
        _transactionCount++;
        base.Withdraw(amount); // Assumes Withdraw will update _balance
    }

    public void ApplyMonthlyFee()
    {
        _balance -= _monthlyFee;
    }
}
```

Fig. 2.4. Fragile Base Class: first implementation deriving class

```
public class BankAccount_V2
{
    protected decimal _balance;
    protected decimal _totalFees = 0;

    // CHANGE: Added automatic fee application
    public virtual void Deposit(decimal amount)
    {
        _balance += amount;
        ApplyTransactionFee(); // NEW: automatic fee
        Console.WriteLine($"Deposited {amount}. Balance: {_balance}");
    }

    public virtual void Withdraw(decimal amount)
    {
        if (amount <= _balance)
        {
            _balance -= amount;
            ApplyTransactionFee(); // NEW: automatic fee
            Console.WriteLine($"Withdrawn {amount}. Balance: {_balance}");
        }
    }

    protected virtual void ApplyTransactionFee()
    {
        _balance -= 1m; // Transaction fee
        _totalFees += 1m;
    }

    public virtual decimal GetBalance()
    {
        return _balance;
    }
}
```

Fig. 2.5. Fragile Base Class: misspelled implementation base class

```
public class PremiumAccount_V2 : BankAccount_V2
{
    private decimal _monthlyFee = 10m;
    private int _freeTransactions = 10;
    private int _transactionCount = 0;

    public override void Deposit(decimal amount)
    {
        _transactionCount++;
        if (_transactionCount > _freeTransactions)
        {
            // PROBLEM: Base.Deposit now also applies transaction fee!
            // Double deduction: _monthlyFee + automatic ApplyTransactionFee()
            base.Deposit(amount - _monthlyFee);
        }
        else
        {
            base.Deposit(amount); // Unexpected fee added by base class
        }
    }

    public override void Withdraw(decimal amount)
    {
        _transactionCount++;
        base.Withdraw(amount); // Unexpected fee added by base class
    }
}
```

Fig. 2.6. Fragile Base Class: misspelled implementation deriving class

```
var account = new PremiumAccount_V2();
account.Deposit(100); // Expected: balance = 100; Actual: balance = 89 (100 - 10 - 1)
Console.WriteLine($"Balance: {account.GetBalance()}"); // Shows unexpected deduction

// Subclass assumptions about execution order are violated
// The fragile base class problem manifests as silent data corruption
```

Fig. 2.7. Fragile Base Class: demonstration

method will be invoked in response to a message, complicating both debugging and maintenance.

2.2.4 Multiple Inheritance and Ambiguity Resolution

Languages supporting multiple inheritance encounter the well-documented diamond problem, wherein a derived class inheriting from multiple parents creates ambiguity about which implementation of inherited methods should be invoked when those parents themselves share a common ancestor. This problem reflects fundamental challenges in the inheritance model: the attempt to linearize and resolve multiple method resolution paths introduces complexity that defeats the transparency and simplicity promised by object-oriented design.

Languages addressing this through single inheritance restrictions or interface-based approaches acknowledge that inheritance as a reuse mechanism creates structural problems that require linguistic constraints. The very fact that languages must prohibit or heavily restrict inheritance patterns suggests limitations in the fundamental mechanism itself.

2.2.5 The Reusability Paradox

Despite promises of enhanced reusability through inheritance and polymorphism, empirical software development experience reveals significant obstacles to achieving code reuse in OOP systems. Graham observes that object-oriented programming offers “a sustainable way to write spaghetti code,” suggesting that the anticipated reusability gains often fail to materialize [12]. The problem lies partly in the tightly coupled nature of inheritance-based designs: inheriting from an existing class binds the derived class not just to the inherited interface but to

the entire implementation dependency graph of the parent class.

Effective reuse typically requires either shallow inheritance hierarchies or careful extraction of orthogonal concerns into separately reusable components. This requirement contradicts the hierarchical organization that inheritance encourages, creating a tension between the inheritance mechanism and practical reusability goals.

2.2.6 State Management Complexity and Object Identity

Object-oriented systems organize code around mutable state encapsulated within objects. This approach fundamentally differs from functional programming's emphasis on immutability and pure functions. The combination of mutable state, object identity, and encapsulation creates complexity in reasoning about program behavior. When objects maintain mutable internal state, analyzing program correctness requires understanding not only the method calls but also the complete history of state modifications, increasing the difficulty of formal verification and testing.

Reynolds' distinction between values (immutable, state-based equality) and entities (distinct identity, mutable state) illustrates the conceptual tension in OOP [13]. In value-oriented systems, equality testing is unambiguous and immutability simplifies reasoning. In object-oriented systems, the notion of object identity introduces questions about whether two references point to the same object or equivalent objects, complications that functional approaches avoid entirely through immutable values.

Furthermore, the emphasis on object identity creates problems in concurrent systems where mutable shared state becomes a liability. Multiple threads

attempting to access and modify object state require extensive synchronization, complicating concurrent programming compared to functional approaches based on immutable data and pure transformations [13].

2.2.7 Absence of Clear Separation Between Specification and Implementation

OOP languages typically conflate class specifications with implementation details through a single construct. Unlike languages supporting explicit separation between interfaces and implementations, OOP classes bundle these concerns, making it difficult to understand what behavior clients should depend upon versus what represents implementation-specific detail. This conflation particularly affects evolution and maintenance: modifications to implementation details that preserve external contracts can nonetheless break derived classes through the fragile base class problem.

2.2.8 Empirical Evidence of Design Difficulties

The widespread adoption of design patterns and anti-pattern literature documents systematic difficulties with OOP system design. The Gang of Four patterns book identifies recurring design problems that OOP practitioners encounter; the very necessity of these patterns suggests that raw OOP mechanisms inadequately support common design requirements [14]. Anti-patterns such as the God Object problem reflect systematic tendency toward violation of single responsibility principles.

Chidamber and Kemerer's metrics suite for measuring OOP design qual-

ity reveals correlations between high metric values and undesirable properties: high coupling between object classes indicates fault-proneness and maintenance difficulty; excessive inheritance tree depth indicates reduced comprehensibility; high weighted methods per class indicates complexity [15]. These metrics-based findings empirically validate concerns about OOP's inherent tendency toward coupled, complex designs. Their research demonstrates that systems exhibiting high CBO values require more rigorous testing and experience higher defect rates, a pattern consistent across multiple industrial software projects [15].

2.2.9 Systematic Classification of OOP Limitations

The critical examination conducted in the preceding subsections reveals a coherent set of fundamental limitations inherent in the object-oriented programming paradigm. These limitations can be systematically classified into several interconnected categories, which collectively explain the challenges encountered in the design, implementation, and maintenance of large-scale OOP systems.

1. **Theoretical and Type-System Deficiencies:** At the most foundational level, OOP suffers from a conflation of distinct concepts, primarily inheritance (an implementation reuse mechanism) and subtyping (a type compatibility relation). This conflation, formalized by type theory, leads to inherent tensions between expressiveness and type safety, as exemplified by the need for complex workarounds.

2. **Architectural and Structural Deficiencies:** The inheritance mechanism itself introduces severe architectural flaws. It creates tight, often hidden, coupling between base and derived classes, fundamentally breaking encapsulation. This results in the well-documented *Fragile Base Class Problem*,

where the modularity of a system is compromised because changes in one module (a base class) can unpredictably break functionality in dependent modules (derived classes), despite adherence to public interfaces.

3. **Compositional and Reusability Deficiencies:** Contrary to its core promises, OOP often hinders effective code reuse. The *Reusability Paradox* highlights that deep inheritance hierarchies bind classes to extensive implementation dependency graphs, making isolated reuse difficult. Furthermore, the challenges of *Multiple Inheritance*, such as the diamond problem, demonstrate the paradigm's struggle to manage complexity when composing behaviors from multiple sources.

4. **State Management and Concurrency Deficiencies:** The paradigm's emphasis on mutable state and object identity complicates reasoning about program behavior, especially in concurrent and distributed environments. The need to manage and synchronize shared mutable state stands in stark contrast to the simplicity offered by immutable data models, making OOP a suboptimal choice for highly concurrent systems.

5. **Design and Evolvability Deficiencies:** The absence of a clear separation between specification and implementation within the class construct impedes system evolution and maintenance. This, combined with the inherent coupling, leads to well-documented design anti-patterns. Empirical evidence, including software metrics and the proliferation of design patterns, confirms a systematic tendency toward complex, tightly coupled designs that are fault-prone and difficult to comprehend.

2.3 Review of Related Work

The problem of comparative analysis of programming languages is not new to computer science. Over recent decades, numerous works have been published dedicated to the classification and comparison of object-oriented languages. Through a systematic literature review, we have identified three main groups of studies: fundamental taxonomies, empirical performance comparisons, and narrowly focused comparative analyses of specific language pairs.

2.3.1 Fundamental taxonomies and theoretical foundations

The foundational works in the classification of object models were conducted in the late 1980s. The seminal work by P. Wegner, “Dimensions of Object-Oriented Language Design“ [16], laid the groundwork by introducing strict criteria for “object-based“ languages. However, contemporary research indicates that these hierarchical taxonomies are losing their relevance in the era of multi-paradigm languages.

In a recent work, M. Vandeloise [17] argues that traditional classifications fail to account for hybrid languages (e.g., Scala, Rust) and proposes a “compositional reconstruction“ of paradigms based on type theory, where the key factor becomes safety guarantees rather than the presence of classes. This is supported by research from Crichton et al. [18], who propose treating the ownership semantics in Rust as a new fundamental dimension of the object model, which ensures memory safety without a garbage collector—an aspect absent from earlier taxonomies like Armstrong’s [19].

2.3.2 Empirical Performance Comparisons

A second extensive group of works is dedicated to the quantitative comparison of languages. The well-known study by L. Prechelt [20] compares seven languages (C, C++, Java, Python, among others) based on criteria such as execution time and memory consumption. Similar contemporary studies, for instance, the work by Farooq and Khan [21], propose frameworks for assessing the popularity and technical efficiency of languages.

A key limitation of this group of works, in relation to the goals of our dissertation, is their focus on *quantitative metrics* (speed, code volume) rather than on the *architectural distinctions* between object models. They answer the question “which language is faster,“ but do not explain how differences in the implementation of v-tables, prototypes (JavaScript), or structural typing (Go) influence the architecture of software systems.

2.3.3 Narrowly Focused and Engineering Comparisons

The third category encompasses works that compare specific language pairs or their individual mechanisms. Numerous publications exist that contrast Go’s interface model with the class-based model of Java/C++ [22], [23]. Hybrid functional-object-oriented models in Scala compared to Java have also been extensively documented [24].

Nevertheless, a significant gap remains in the academic literature regarding works that conduct a comprehensive, end-to-end analysis of the entire spectrum of modern models: ranging from “pure“ OOP (Smalltalk, Ruby) to prototype-based (JavaScript), hybrid (Scala, C++), and active object models (Zonnon). The Zonnon language and its unique model, based on active objects and dialogs as

described by Gutknecht [25], receives virtually no attention in broad comparative surveys, remaining confined to specialized niche literature.

2.3.4 Conclusion

The analysis of the literature demonstrates that, despite the abundance of comparative studies, a comprehensive investigation is lacking—one that would:

1. Unify both classical and modern languages within a single comparative framework.
2. Examine not only syntax or performance but also the internal implementation of object models (dispatch mechanisms, memory management, object layouts).
3. Identify the architectural consequences of choosing a particular model for designing complex systems.

This dissertation aims to fill this gap by proposing a systematization that includes both industry standards and academic languages, with the goal of developing recommendations for an enhanced object model.

2.4 Methodology of Comparative Analysis

When developing object models for programming language, a systematic and reproducible approach to their analysis must be employed. This section provides a list of criteria for comparing the implementation of object models.

Rationale for the Chosen Methodology

The selection of comparison criteria is justified by a number of fundamental works on programming language design. Cardelli and Wegner [1] established three key requirements for defining an object-oriented language: support for objects as data abstractions, the association of objects with a specific type, and type inheritance. These requirements form a basic classification of languages and allow one to distinguish object-oriented languages from those that merely support some object-based features.

Booch [2] defined four core elements of the object model — abstraction, encapsulation, modularity, and hierarchy — as well as three secondary elements — as well as three secondary elements—typing, concurrency, and persistence. According to Booch, without the core elements, a model ceases to be object-oriented; however, the secondary elements significantly impact the model's practical applicability. This hierarchical approach to identifying elements helps determine which language characteristics are critical.

The methodology for evaluating programming languages is based on a comprehensive approach discussed in works on language assessment criteria. Sebesta [26] proposed four primary criteria for evaluating languages: readability, writability, reliability, and cost. Furthermore, readability and writability depend on a wide range of language characteristics, including syntactic simplicity, support for

abstraction, and orthogonality. Modern works on code quality and software maintainability emphasize the importance of metrics such as modularity, reusability, analyzability, and modifiability [27], which are directly influenced by the characteristics of a specific language's object model.

Object Model Comparison Criteria

Based on the fundamental works mentioned above and an analysis of programming language standards, the following criteria have been identified for the systematic comparison of object models:

1. **Type System:** allows to evaluate how the language ensures code reliability and builds its infrastructure through data validation rules.

2. **Inheritance Mechanism:** is necessary to compare how languages support code reuse and hierarchy organization, which directly affects maintainability and extensibility.

3. **Abstraction and Encapsulation:** helps analyze how language manages complexity by hiding implementation details and protecting the internal state of objects.

4. **Polymorphism:** comparing the trade-off between performance (static dispatching) and code flexibility (dynamic dispatching) in different languages.

5. **Object Creation Mechanism:** Compare fundamental approaches to defining objects: based on classes (instances) or prototypes (delegation).

6. **Metaclasses and Metaprogramming:** assessments of the language's ability to introspect and modify its own structure during execution.

7. **Multiple Inheritance and Mixins:** Compare alternative mechanisms for composition and extending functionality beyond a strict class hierarchy.

8. **Memory Management:** analysis of trade-offs between performance, predictability of behavior, and development complexity in different languages.

9. **Exception Handling:** Compare the philosophies of languages regarding error handling guarantees: strict control at the compilation stage (checked exceptions) or a more flexible runtime model.

10. **Modularity and Packages:** assess how the language supports the organization and scaling of large projects through code structuring mechanisms.

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] M. Atkinson and R. Morrison, "Orthogonally persistent object systems," *The VLDB Journal*, vol. 4, no. 3, pp. 319–401, 1995.

[8] W. R. Cook, W. Hill, and P. S. Canning, "Inheritance is not subtyping," in *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1989, pp. 125–135.

[9] B. C. Pierce and D. N. Turner, "Simple type-theoretic foundations for object-oriented programming," *Journal of functional programming*, vol. 4, no. 2, pp. 207–247, 1994.

[10] L. Mikhajlov and E. Sekerinski, "A study of the fragile base class problem," in *European Conference on Object-Oriented Programming*, Springer, 1998, pp. 355–382.

[11] M. Hitz and B. Montazeri, *Measuring coupling and cohesion in object-oriented systems*. na, 1995.

[12] P. Graham, *The Hundred-Year Language*, https://www.paulgraham.com/hundred.html, [Online; accessed 27-11-2025], Apr. 2003.

[13] J. C. Reynolds, "Three approaches to type structure," in *Colloquium on Trees in Algebra and Programming*, Springer, 1985, pp. 97–138.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: Abstraction and reuse of object-oriented design," in *European conference on object-oriented programming*, Springer, 1993, pp. 406–431.

[15] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[16] P. Wegner, "Dimensions of object-based language design," *ACM Sigplan Notices*, vol. 22, no. 12, pp. 168–182, 1987.

[17] M. Vandeloise, "Towards a unified framework for programming paradigms: A systematic review of classification formalisms and methodological foundations," *arXiv preprint arXiv:2508.00534*, 2025.

[18] W. Crichton, G. Gray, and S. Krishnamurthi, "A grounded conceptual model for ownership types in rust," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA2, pp. 1224–1252, 2023.

[19] D. J. Armstrong, "The quarks of object-oriented development," *Communications of the ACM*, vol. 49, no. 2, pp. 123–128, 2006.

[20] L. Prechelt, "An empirical comparison of seven programming languages," *Computer*, vol. 33, no. 10, pp. 23–29, 2002.

[21] M. S. Farooq et al., "Comparative analysis of widely use object-oriented languages," *arXiv preprint arXiv:2306.01819*, 2023.

[22] R. Pike, "Go at google: Language design in the service of software engineering. online," *URL: https://talks. golang. org/2012/splash. article*, 2012.

[23] I. Balbaert, *The way to Go: A thorough introduction to the Go programming language*. IUniverse, 2012.

[24] M. Odersky and T. Rompf, "Unifying functional and object-oriented programming with scala," *Communications of the ACM*, vol. 57, no. 4, pp. 76–86, 2014.

[25] J. Gutknecht and E. Zueff, "Zonnon language report," *Zurich, Institute of Computer Systems ETH Zentrum*, 2004.

[26] R. W. Sebesta, *Concepts of programming languages*. Pearson Education India, 2016.

[27] *Iso/iec 25010: Systems and software engineering – system and software product quality models*, 2023.