

SHOULD BE REPLACED ON REQUIRED TITLE PAGE

Instruction

1. Open needed docx template (folder "title"/<your department or batch 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
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	18
2.2	Critical Analysis of Object-Oriented Programming Approach . .	19
2.2.1	Theoretical Inconsistencies in Inheritance and Subtyping	19
2.2.2	The Fragile Base Class Problem	20
2.2.3	Excessive Coupling and Tight Interdependencies	22
2.2.4	Multiple Inheritance and Ambiguity Resolution	27
2.2.5	The Reusability Paradox	28
2.2.6	State Management Complexity and Object Identity	28
2.2.7	Absence of Clear Separation Between Specification and Implementation	29
2.2.8	Empirical Evidence of Design Difficulties	30
2.2.9	Systematic Classification of OOP Limitations	30
2.3	Review of Related Work	33

2.3.1	Fundamental taxonomies and theoretical foundations	33
2.3.2	Empirical Performance Comparisons	34
2.3.3	Narrowly Focused and Engineering Comparisons	34
2.3.4	Conclusion	35
2.4	Methodology of Comparative Analysis	36
3	Methodology	41
4	Implementation	42
5	Evaluation and Discussion	43
6	Conclusion	44
	Bibliography cited	45
A	Extra Stuff	49
B	Even More Extra Stuff	50

List of Tables

List of Figures

2.1	Inheritance without proper subtyping	21
2.2	Inheritance without proper subtyping: manifestation	22
2.3	Fragile Base Class: first implementation base class	23
2.4	Fragile Base Class: first implementation deriving class	24
2.5	Fragile Base Class: misspelled implementation base class	25
2.6	Fragile Base Class: misspelled implementation deriving class . .	26
2.7	Fragile Base Class: demonstration	26

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:

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

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.

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.

2.2 Critical Analysis of Object-Oriented Programming Approach

The object-oriented programming paradigm, despite its widespread adoption and significant influence on software development practices, exhibits fundamental limitations and unresolved challenges that constrain its applicability and effectiveness across diverse problem domains. This section presents a systematic and critical examination of the core deficiencies of OOP. The critical perspective developed here establishes a theoretical foundation for evaluating the real-world implementation of object models in contemporary 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 [12]. The traditional assumption that inheritance hierarchies determine conformance relations proves inadequate when dealing with recursive types and contravariant method types. Cook et al. show through formal analysis that in strongly-typed OOP languages like Simula, C++, and Eiffel, the restriction of inheritance to satisfy subtyping requirements substantially limits the flexibility of inheritance mechanisms, making them less expressive than untyped inheritance found in languages such as Smalltalk [12].

The distinction between inheritance and subtyping stems from fundamental type-theoretic considerations. Cardelli and Wegner's comprehensive analysis of

type systems establishes that inheritance as an implementation mechanism and subtyping as a type-theoretic relation represent orthogonal concerns [1]. Their framework demonstrates that subtyping relations require adherence to the Liskov substitution principle, whereas inheritance mechanisms, particularly when dealing with recursive object definitions through F-bounded polymorphism, impose additional constraints that compromise expressiveness [13]. The practical consequence of this theoretical inconsistency is that inheritance-based type hierarchies cannot soundly 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 [14]. 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

```

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

self-recursion vulnerabilities create situations where base class revisions break extension class functionality without changing the external interface [14].

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. 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 [14].

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

```

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

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 [15].

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

```
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

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 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 [16]. 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 [17]. 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 [17].

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 [18]. 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 quality 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 [19]. 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 [19].

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” [20], 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 [21] 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. [22], 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 [23].

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 [24] 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 [25], 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++ [26], [27]. Hybrid functional-object-oriented models in Scala compared to Java have also been extensively documented [28].

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 [29], 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 [30] 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 [31], 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:

Type System A language's type system defines the rules for assigning types to various program constructs — variables, expressions, and functions. This is a fundamental aspect of any programming language, essential both for enhancing reliability and for defining its infrastructure.

A distinction is made between static typing (type checking at compile time) and dynamic typing (type checking at runtime). Additionally, languages may use nominal typing (where type compatibility is determined by their explicit declaration) or structural typing (where compatibility is determined by the structure of types). The choice of type system impacts code reliability, the potential for detecting errors in early development stages, and the overall architecture of the object model.

Inheritance Mechanism Inheritance is a central mechanism in object-oriented languages for defining new classes based on existing ones. It enables code reuse and establishes hierarchical relationships between classes. Object-oriented design principles, such as the Liskov Substitution Principle and the Open/Closed

Principle, rely on inheritance and subtyping. A correct implementation of the inheritance mechanism is critical for code maintainability and extensibility.

Abstraction and Encapsulation Abstraction is the process of hiding implementation details and exposing only the essential features of an object. Encapsulation is the mechanism of bundling data and methods into a single unit and restricting direct access to an object's internal state. Programming languages support various access control levels (public, private, protected), which influence the degree of encapsulation. Some languages allow for modifying class member visibility, while others enforce stricter control mechanisms.

Polymorphism Polymorphism enables a single interface to represent different data types. It is a fundamental mechanism for achieving flexibility and extensibility in object-oriented systems. The choice of dispatch mechanism directly impacts performance: static dispatch allows the compiler to optimize function calls, while dynamic dispatch provides flexibility at the cost of some performance overhead.

Object Creation Mechanism Programming languages differ in their approach to object creation. Most modern languages employ a class-based model where objects are instances of classes. However, some languages utilize prototype-based inheritance, where objects inherit properties directly from other objects through delegation. The class-based model provides clear structure and predictable behavior, making code more understandable and maintainable. The prototype-based model offers greater flexibility by enabling dynamic alteration of object behavior during runtime. The distinction between these models impacts how developers

organize code, system performance, and the ease of debugging and maintenance.

Metaclasses and Metaprogramming Metaclasses are classes whose instances are themselves classes. They enable control over class creation and behavior. Metaprogramming refers to techniques that allow programs to create or modify other programs or themselves. Not all languages support metaclasses; those that do provide varying degrees of control over the metaclass protocol.

Multiple Inheritance and Mixins To facilitate code reuse and avoid the complexities of multiple inheritance, some languages support the mixin mechanism. Mixins allow functionality to be added to classes without using a strict inheritance hierarchy, providing an alternative composition approach.

Memory Management The memory management strategy is a critical aspect of any programming language, directly impacting performance, predictability, and development convenience. Languages may employ manual memory management, automatic garbage collection, or other approaches such as reference counting or ownership-based management. The choice of memory management strategy affects performance, predictability of program behavior, and required developer skills.

Exception Handling Most modern object-oriented languages support exception handling mechanisms for managing runtime errors. A distinction is made between languages with checked exceptions, which are verified by the compiler, and those with unchecked exceptions. Some languages provide special constructs, such as finally blocks for the guaranteed execution of cleanup code.

Method Dispatch The mechanism by which a language determines which method to invoke is a crucial aspect of implementing polymorphism. Dispatch can be static (resolved at compile time) or dynamic (resolved at runtime). Dynamic dispatch is typically implemented using virtual method tables (vtables), though alternative approaches such as inline caching may also be employed.

Modularity and Packages A language must provide means for organizing code into modules to facilitate the development of large-scale systems. This includes support for packages, modules, namespaces, and visibility control of symbols.

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.
- [12] 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.
- [13] 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.
- [14] 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.
- [15] M. Hitz and B. Montazeri, *Measuring coupling and cohesion in object-oriented systems*. na, 1995.

- [16] P. Graham, *The Hundred-Year Language*, <https://www.paulgraham.com/hundred.html>, [Online; accessed 27-11-2025], Apr. 2003.
- [17] J. C. Reynolds, “Three approaches to type structure,” in *Colloquium on Trees in Algebra and Programming*, Springer, 1985, pp. 97–138.
- [18] 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.
- [19] 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.
- [20] P. Wegner, “Dimensions of object-based language design,” *ACM Sigplan Notices*, vol. 22, no. 12, pp. 168–182, 1987.
- [21] M. Vandeloise, “Towards a unified framework for programming paradigms: A systematic review of classification formalisms and methodological foundations,” *arXiv preprint arXiv:2508.00534*, 2025.
- [22] 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.
- [23] D. J. Armstrong, “The quarks of object-oriented development,” *Communications of the ACM*, vol. 49, no. 2, pp. 123–128, 2006.
- [24] L. Prechelt, “An empirical comparison of seven programming languages,” *Computer*, vol. 33, no. 10, pp. 23–29, 2002.
- [25] M. S. Farooq et al., “Comparative analysis of widely use object-oriented languages,” *arXiv preprint arXiv:2306.01819*, 2023.

- [26] R. Pike, “Go at google: Language design in the service of software engineering. online,” *URL: https://talks. golang. org/2012/splash. article*, 2012.
- [27] I. Balbaert, *The way to Go: A thorough introduction to the Go programming language*. IUniverse, 2012.
- [28] 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.
- [29] J. Gutknecht and E. Zueff, “Zonnon language report,” *Zurich, Institute of Computer Systems ETH Zentrum*, 2004.
- [30] R. W. Sebesta, *Concepts of programming languages*. Pearson Education India, 2016.
- [31] *Iso/iec 25010: Systems and software engineering – system and software product quality models*, 2023.

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.