

Educational Insights from Code: Mapping Learning Challenges in Object-Oriented Programming through Code-Based Evidence

André Menolli

Universidade Estadual do Norte do Paraná
Universidade Estadual de Londrina
Brazil
menolli@uenp.br

Bruno Strik

Instituto Federal do Paraná
Astorga, Brazil
bruno.strik@ifpr.edu.br

ABSTRACT

Object-Oriented programming is frequently challenging for undergraduate Computer Science students, particularly in understanding abstract concepts such as encapsulation, inheritance, and polymorphism. Although the literature outlines various methods to identify potential design and coding issues in object-oriented programming through source code analysis, such as code smells and SOLID principles, few studies explore how these code-level issues relate to learning difficulties in Object-Oriented Programming. In this study, we explore the relationship of the code issue indicators with common challenges encountered during the learning of object-oriented programming. Using qualitative analysis, we identified the main categories of learning difficulties and, through a literature review, established connections between these difficulties, code smells, and violations of the SOLID principles. As a result, we developed a conceptual map that links code-related issues to specific learning challenges in Object-Oriented Programming. The model was then evaluated by an expert who applied it in the analysis of the student code to assess its relevance and applicability in educational contexts.

KEYWORDS

object-oriented programming, code quality, learning challenges, code analysis, code smells, SOLID

1 Introduction

Learning computer programming is a well-known challenge in undergraduate Computer Science education [37, 59]. Although numerous educational technologies and simplified environments have been developed to support novice learners, the cognitive complexity of programming—particularly within the Object-Oriented Programming (OOP) paradigm—remains significant. Abstract concepts such as encapsulation, inheritance, and polymorphism often pose difficulties that go beyond syntactic understanding, affecting not only program correctness but also the quality of software design [39].

Several studies [7, 12, 23] have shown that many students reach advanced programming courses with an insufficient conceptual grasp of OOP fundamentals, often resulting in low-quality code and persistent misconceptions. While introductory courses tend to emphasize syntactic and functional correctness, structural and stylistic aspects are frequently overlooked. As a result, students may develop ineffective mental models and poor programming practices that endure throughout their academic journey [38].

A review of the literature reveals a limited number of studies that classify the specific difficulties in learning OOP. Gutiérrez, Guerrero, and López-Ospina [12] identified 14 categories of such

difficulties based on a review of 56 studies. Other works, such as those by Thomasson, Ratcliffe, and Thomas [56], Biju [5], Holland, Griffiths, and Woodman [16], and Or-Bach and Lavy [42], organize learning challenges around fundamental OOP concepts: abstraction, encapsulation, inheritance, and polymorphism [50].

Furthermore, in the context of OOP learning, it is common to evaluate students' source code based solely on whether it meets functional requirements, even if the implementation is poor and filled with *code smells*. Although such code may be accepted by the compiler, it often violates key OOP principles. In many introductory courses, these design-related issues are ignored to prevent overwhelming inexperienced learners with excessive complexity. However, in OOP, it is essential not only to check if the code runs correctly, but also to assess whether it adheres to the paradigm's design principles.

Therefore, it is important not only to identify deficiencies in OOP learning but also to understand the underlying factors that contribute to them. One effective approach is source code analysis, through which signs of OOP misuse can reveal the specific learning challenges students may be facing.

This study aims to map the relationship between indicators of problematic object-oriented code and common learning difficulties in OOP. To achieve this, we conduct a qualitative analysis grounded in theories of code smells and the SOLID principles, establishing connections between these design violations and documented challenges in learning OOP [12].

The main contributions of this work are twofold. First, we refine and extend the learning challenges previously identified by [12] by defining specific categories of learning problems associated with each challenge. Second, we present a conceptual map that illustrates how particular design flaws in students' code may reflect underlying learning difficulties related to core object-oriented principles.

2 Theoretical Background

This section provides the theoretical background that supports this study. It addresses three key aspects: the main challenges associated with learning programming—particularly object-oriented programming, the role of code quality in programming education, and the identification of code-level indicators that may reflect learning difficulties in OOP contexts.

2.1 Programming Learning and Challenges

Moser [39] describes programming as an intimidating process requiring multilayered skill development. Learning progresses bottom-up, starting with syntax, then structure, and finally style. Tan, Ting,

and Ling [53] note that early focus on syntax can lead to misunderstandings of deeper programming concepts, resulting in a reliance on specific languages rather than general programming skills. This often leads to poor code quality and difficulty transitioning to other languages. Several studies have analyzed learning difficulties in programming [23, 25, 37–39, 53, 59]. Lahtinen, Ala-Mutka, and Järvinen [28] collected feedback from over 500 students worldwide, confirming the widespread difficulty of programming education, particularly in abstraction and program construction. Cheah [8], through a literature review, identified key factors contributing to these challenges: lack of logical foundations, use of industry-focused tools unsuited for learning, and high levels of anxiety. A critical issue is the formation of incorrect mental models, which leads to design flaws and bugs. Focusing more narrowly on OOP, few studies have explored specific learning challenges. Ismail, Ngah, and Umar [20] argue that OOP instruction requires different strategies than procedural or structured programming, as conventional techniques like pseudocode and flowcharts are insufficient.

Kölling [24] criticizes the typical pedagogical sequencing, where procedural programming precedes OOP, reinforcing the misconception that OOP is merely an additional feature. He emphasizes that OOP is a distinct paradigm that fundamentally reshapes how problems are modeled and should be taught from the outset. Among the reviewed works, only Gutiérrez, Guerrero, and López-Ospina [12] provide a classification of OOP-specific learning difficulties. Their study defines 14 categories of student learning challenges, offering a structured framework that serves as the basis for the OOP learning challenges adopted in this research.

2.2 Code Quality in Programming Education

Software quality and code quality are closely related but distinct concepts. Code quality is understood as a more specific aspect of the broader software quality defined in ISO/IEC 25000 [19]. The notion of code quality [23] focuses on static characteristics of programs that are directly observable from the source code, excluding aspects related to dynamic behavior, such as runtime performance. In the educational context, Stegeman, Barendsen, and Smetsers [51] proposed a model with six criteria to assess student-written code. These criteria, developed from best practice guides and programming instructors' experience, include:

- **Comments:** content and summarization;
- **Formatting:** consistency and expressiveness;
- **Layout:** cohesion, organization, and presence of dead code;
- **Naming:** consistency and meaningfulness;
- **Structure:** abstraction, duplication, modularization, type use, method adequacy, and fragmentation;
- **Expressiveness:** phrasing, clarity, and flow control.

Other studies have also contributed to evaluation criteria for code quality in education. Hamm et al. [14] emphasize documentation, structure, and functionality. Howatt [17] proposes evaluating executability, adherence to requirements, effective comments, readability, and planning.

Based on these and other works [3, 49], it is evident that code quality is a well-established topic in programming education. However, no studies were found that specifically parameterize the evaluation of code quality in the context of OOP. Although general quality criteria may apply, they often fail to address specific aspects of OOP, which introduces distinct concepts that must be validated in educational settings. While summative evaluation through classification and conformance checking plays a role in education, such approaches offer limited support for formative learning. As this study aims to propose a formative rather than summative strategy, it focuses on identifying the weaknesses that lead to code quality problems.

2.3 Indicators of Code Quality Issues

Identifying issues in object-oriented code often requires the detection of recurring structural and behavioral patterns that undermine readability, maintainability, and extensibility. Key indicators of such issues include code smells [9], as well as violations of widely accepted design principles, such as the Law of Demeter [30], the Tell, Don't Ask principle [55], and the SOLID principles [36]. These violations not only reflect poor code quality but also suggest fundamental misunderstandings of core object-oriented programming concepts like encapsulation, abstraction, and responsibility. Identifying these issues in students' source code can reveal several indicators of difficulties in grasping fundamental OOP concepts.

Among these issues, code smells are perhaps the most extensively discussed in the literature, a term popularized by Fowler's work [9]. A "smell" refers to an underlying problem in the software, which can manifest at both the code level [9] and the design level [6]. These "smells" are symptoms in software components that can hinder the system's evolution. Depending on the level of abstraction, they are classified as code smells or design smells. Unlike bugs, which often result in immediate faults, smells do not directly cause application errors but can lead to long-term negative consequences, such as difficulties in maintenance and future development.

Several authors have contributed to the conceptualization of code smells. Brown et al. [6] introduced 40 anti-patterns, which describe common problems that typically result in negative consequences. Fowler [9] cataloged 22 distinct smells and proposed sequences of refactorings to mitigate each one. Wake [58] explored problematic patterns commonly identified by practitioners in the field. Kerievsky [22] expanded this discussion, focusing on the role of design patterns in addressing these issues.

An effective way to improve understanding of code smells is through their categorization based on potential relationships, which can support deeper comprehension and analysis. One of the most accepted classifications of code smells is presented by [34], who introduced a detailed taxonomy, grouping smells into the following categories:

- **Bloaters:** These are code elements that have grown excessively large and become difficult to manage or understand.
- **Object-Orientation Abusers:** These represent an improper or suboptimal use of object-oriented principles, often involving workarounds that ignore good OO design practices.
- **Change Preventers:** Structures that make software modifications difficult, increasing the cost and risk of changes.

- **Dispensables:** Code fragments that are unnecessary and should be removed to improve clarity and maintainability.
- **Couplers:** Smells that indicate excessive coupling between classes or components, which can reduce modularity and hinder reuse.

Furthermore, over the years, the concept of code smells has expanded beyond traditional object-oriented code, with research identifying smells in various domains. These include test code [11, 15] aspect-oriented systems [1, 33], software reuse [32], and web applications [41] among others.

Another important conceptual framework for identifying design issues in object-oriented code is **SOLID**, an acronym for five principles introduced by Martin [36] to promote robust and maintainable software design. These principles are: (1) the *Single Responsibility Principle* (SRP), which states that a class should have only one reason to change; (2) the *Open/Closed Principle* (OCP), which advocates for designing modules that are open to extension but closed to modification; (3) the *Liskov Substitution Principle* (LSP), which ensures that subclasses can be substituted for their base classes without compromising program correctness [31]; (4) the *Interface Segregation Principle* (ISP), which encourages the creation of small, role-specific interfaces rather than large, general-purpose ones; and (5) the *Dependency Inversion Principle* (DIP), which promotes depending on abstractions rather than concrete implementations.

In this work, we are specifically interested in indicators related to implementation and design. For this reason, we focus on *code smells* and the **SOLID** principles, as summarized in Table 1.

Table 1: Code quality indicators classified by focus and type

Focus	Type	References
Implementation	Code Smell	[6, 9, 44, 57]
Design	SOLID	[36]
	Code Smell	[35, 52]

3 Research Structure

The schematic representation of the research structure applied in the study is shown in Figure 1, which illustrates the stages and the output artifacts produced in each phase. The study unfolded in six key phases, starting with the identification of challenges in learning object-oriented programming and culminating in a mapping between code-level issues and object-oriented learning challenges.

3.1 Identification of learning object-oriented programming challenges

In the study by [12], a systematic review was conducted, in which 56 selected studies were analyzed, leading to the identification of 14 challenges related to the teaching and learning of object-oriented programming. The work of [12] consolidated the main difficulties encountered in learning object-oriented programming, providing a comprehensive overview of the topic based on the existing literature. This study serves as the starting point for our mapping.

Among the 14 difficulties identified, several are not directly observable in source code or are primarily related to the teaching process rather than the learning process. Examples include the Difficulty in teaching and understanding general programming topics (D09) and the Difficulty with project administration and management methodologies and techniques (D13). Considering this in our mapping, we focused on six learning challenges that can be identified through source code, as presented by [12] and supported by many works as presented below:

- (1) **Difficulties related to understanding classes (D02).** This difficulty is described as the complexity presented by the students when assimilating the static nature and depth of classes. It is challenging for them to understand the hierarchy and the identification of correct classes. The students even refer to the difficulty in distinguishing between class and object. They generally assimilate class as a collection of objects, rather than an abstraction [4, 10, 18, 21, 29, 40, 43, 45, 47, 48, 54, 61].
- (2) **Difficulty in understanding the concept of method (D03).** In this case it is referred as the complexity presented when assimilating the concept of method, there is no clarity on how to make the method calls. The students do not know how to determine the number of methods needed or what labels or names to assign to them [10, 18, 21, 40, 45, 47, 54].
- (3) **Difficulty in implementing object-orientation (D04).** This problem is specified as the challenge of performing object-oriented analysis, design, and programming. The students present difficulties when adopting the object-oriented paradigm, because their initial formative process is generally based on purely structural programming. The modular nature of the object-oriented paradigm is conceived as a challenge for educators, since in this process it is common for students to assimilate erroneous conceptions and to present problems in understanding and implementing object-oriented standards [4, 10, 18, 21, 29, 40, 43, 45, 47, 48, 54, 61].
- (4) **Difficulty in understanding object-oriented relationships (D05).** It refers to the difficulty that the students have when understanding and implementing object-oriented relationships, such as association, dependency, generalization / specialization-inheritance, composition and aggregation. These problems are common due to the learners' lack of experience in relation to the object-oriented programming paradigm. The students generally present difficulties in the process of modeling these relationships, and consequently in the implementation and application of concepts that are often conceived as complex [4, 10, 13, 21, 29, 40, 43, 47, 48, 54, 61].
- (5) **Difficulty in understanding polymorphism and overload (D06).** In this case it is indicated the high level of complexity the concepts of polymorphism and overload have at the moment of initiating a student into the programming area [4, 29, 40, 43, 47, 54, 61].
- (6) **Difficulty in understanding encapsulation (D07).** This problem is related to the assimilation of several misconceptions related to understanding encapsulation, modularity and information hiding [10, 18, 21, 29, 40, 43, 45, 47, 48, 54, 60, 61].

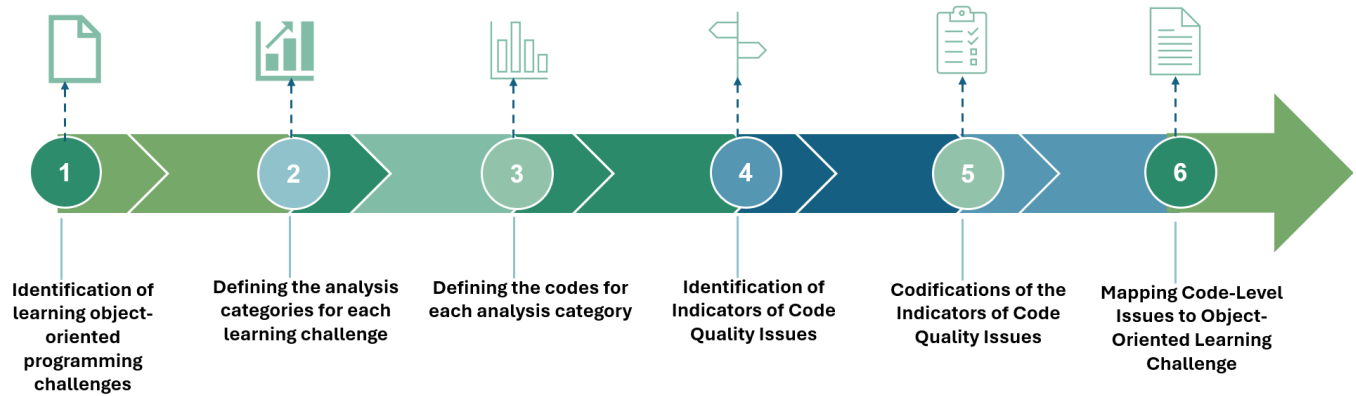


Figure 1: The Research Structure of the Study

Difficulty in understanding the concept of method (D03). In this case it is referred as the complexity presented when assimilating the concept of method, **there is no clarity on how to make the method calls**. The students **do not know how to reuse methods** or **what labels to name to assign to them**. They **do not understand how to reuse methods** or their proper placement.

Difficulty calling methods. Uncertainty about method quantity. Lack of understanding about... Issues with correct method call.

Difficulty in understanding the concept of method (D03). In this case it is referred as the complexity presented when assimilating the concept of method, **there is no clarity on how to make the method calls**. The students **do not know how to reuse methods** or **what labels to name to assign to them**. They **do not understand how to reuse methods** or their proper placement.

Method Call Syntax Errors. Cannot Reuse Method. Method Naming Confusion. Method Placement Issue. Wrong Class For Method. Methods Rewritten Unnecessarily.

Figure 2: Identification of analysis categories for the learning challenge *Difficulty in understanding the concept of method (D03)*.

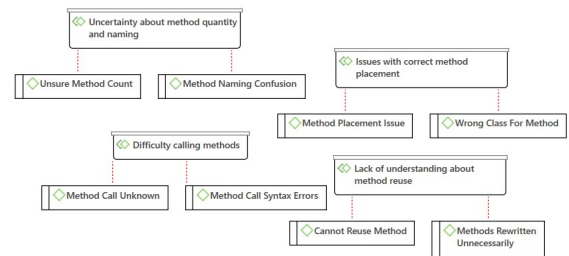
3.2 Defining the analysis categories for each learning challenge

To define the analysis categories, we employed content analysis, a qualitative research method. Content analysis systematically examines the content and structure of communication, aiming to identify patterns, themes, and relationships within the data [2, 26]. It also allows for inferences by interpreting evidence and indicators, supported by a structured framework for technical validation [2].

At this stage, we applied an inductive approach, which involves an open coding process in which categories are created during the analysis. We examined the textual descriptions of each learning challenge and defined distinct analysis categories accordingly. For example, as illustrated in Figure 2, we identified four analysis categories for the challenge *D03 (Difficulty in understanding the concept of a method)*. These categories reflect specific issues that may contribute to the learning difficulty. In this analysis process, we used the Atlas.ti¹ software to support and organize the analysis, and this process was repeated for the six learning challenges, where we have identified 22 categories.

3.3 Defining the codes for each analysis category

The next stage involved defining codes for each category. Once again, we adopted an inductive approach through an open coding process. In this process, codes were created for each category. As an example, Figure 3 presents the codes identified from the text describing the learning challenge *Difficulty in understanding the concept of method (D03)*, organized according to each category. As a

Figure 3: Identification of codes for the learning challenge *Difficulty in understanding the concept of method (D03)*.Figure 4: Analysis categories and its codes for the learning challenge *Difficulty in understanding the concept of method (D03)*.

result of the analysis of challenge D03, we identified eight codes associated with the four analysis categories defined for this challenge, as presented in Figure 4. We repeated this process systematically for all other learning problems.

3.4 Identification of Indicators of Code Quality Issues

The analysis of students' source code can reveal a range of indicators that point to difficulties in assimilating the fundamental concepts of OOP. Some of these indicators are reflected in the presence of code smells as well as in violations of recognized design principles such as the Law of Demeter [30], the Tell, Don't Ask principle [55], and the SOLID principles [36]. The occurrence of these violations, in addition to representing a code quality issue, indicates challenges in learning the core concepts of OOP. Such violations suggest an insufficient understanding of encapsulation, abstraction, responsibility, and other pillars of object orientation.

¹Atlas.ti <https://atlasti.com>

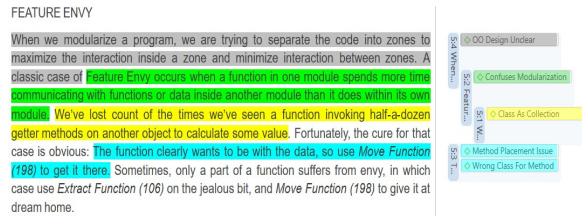


Figure 5: Codes applied to the code smell *Feature Envy* [9]

Considering the code problem indicators identified in the literature, many of them tend to overlap, as they address similar aspects. In this work, we selected a set of implementation and design code smells, as well as violations of SOLID principles, as indicators. Based on the literature, we analyzed the most relevant implementation and design smells and SOLID principles, and, drawing on the work of [27, 46] we compiled a list of code indicators that may point to difficulties in learning object-oriented programming concepts as presented in Table 2.

3.5 Codifications of the Indicators of Code Quality Issues

The next stage involved applying the codes to each indicator identified in Table 2. In this step, we employ a deductive analysis approach, in which a predefined set of categories is established, and the collected data is coded according to these categories [26].

We analyze each original description of the indicator indicators of the problem and apply the codes defined in Stage 3 (Section 3.3), in order to map the relationships between the indicator of the quality of the code and the categories of analysis of learning problems and, consequently, the associated learning difficulties.

As an example, Figure 5 presents the original description of the code smell *Feature Envy* along with the codification applied. It shows five codes applied, corresponding to four different analysis categories. We repeat the coding process for all indicators in the Table 2.

3.6 Mapping Code-Level Issues to Object-Oriented Learning Challenge

The final stage of our research involved mapping the indicators of code quality issues to the broader challenges of learning object-oriented programming, and compiling the results presented in the following section.

4 Results

After executing the entire coding process, Table 3 summarizes one of the main results of the study. It organizes the identified difficulties in learning Object-Oriented Programming into analysis categories, each associated with specific codes (registration units). Additionally, for each code, related code smells and design issues are listed, reflecting how these learning difficulties can manifest in students' source code.

Regarding all the processes carried out and the results presented in Tables 2 and 3, we developed a conceptual map to represent the relationships between learning challenges and code-related issues.

To improve data visualization and clarity, the map was divided into parts.

Figure 6 illustrates the conceptual mapping developed from the analysis process for learning challenges D02 and D03. Figure 7 presents the mapping for challenges D04 and D05, while Figure 8 focuses on challenges D06 and D07. The dark green elements represent the main learning challenges encountered in the context of OOP. Light green elements indicate underlying cognitive or conceptual difficulties that contribute to the emergence of these learning challenges. Observable issues in students' source code—code smells that act as indicators of such difficulties—are marked in light gray. Additionally, dark gray elements represent related code smells that, while not directly observed, are theoretically associated with the identified problems and may also signal learning difficulties. Finally, the yellow elements refer to violations of SOLID principles, which were also identified in the code and serve as indicators of design flaws linked to conceptual misunderstandings. The diagrams provide a visual representation of how abstract learning issues relate to concrete problems in students' code.

4.1 How to Navigate in the Conceptual Map

Through the analysis of object-oriented source code, it is possible to identify the presence of *code smells* or violations of the SOLID principles. Once this analysis is conducted—either manually or with the aid of automated tools such as *SonarQube*, *JDeodorant*, or *Checkstyle*—it becomes feasible to map students' learning difficulties.

For instance, whether the code analysis reveals the presence of the code smells *Long Method* and *Switch Statements*, in the Figure 6 is possible to observe that these smells are associated with the problems *Uncertainty about method quantity and naming* and *Lack of understanding about method reuse*, respectively. Both are indicative of the learning challenge labeled *Difficulty in understanding the concept of method*.

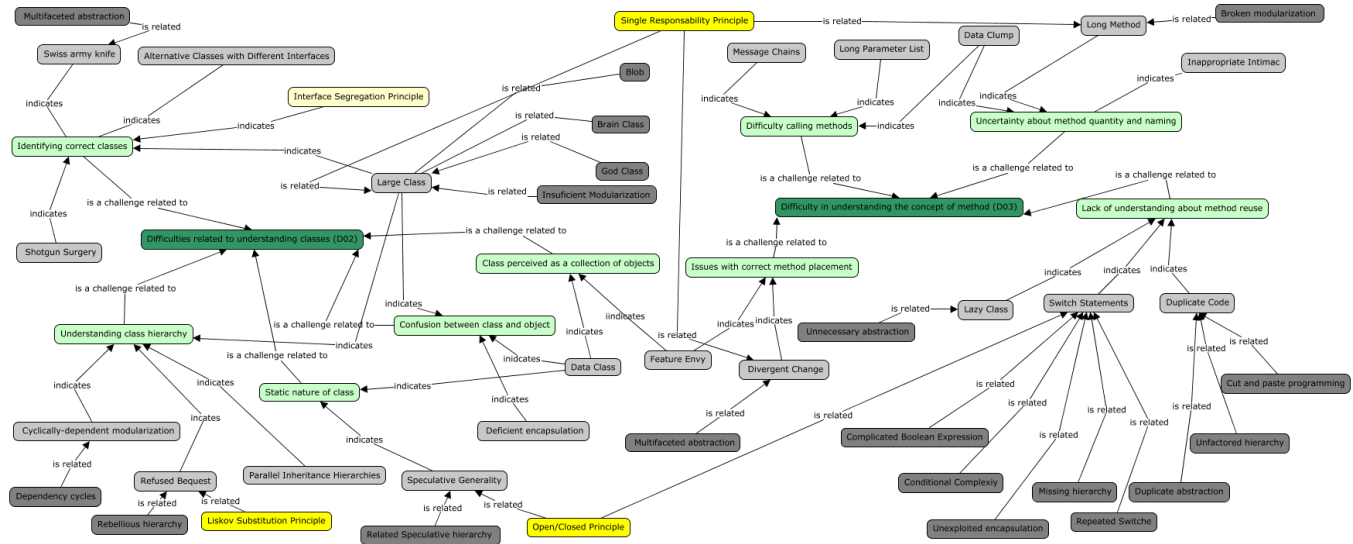
Additionally, *Long Method* is also linked to the issue *Difficulty shifting from structured to OO thinking*, which falls under the challenge *Difficulty in implementing object-orientation* as presented in Figure 7. Furthermore, as shown in Figure 8, the *Switch Statement* smell is also related to the problem *Polymorphism is too abstract*, which signals a *Difficulty in understanding polymorphism and overload*.

The combined analysis of these two code smells suggests that the student faces broader challenges in learning object-oriented programming. These include difficulties with basic concepts such as decomposing methods into smaller units, which could also violate the Single Responsibility Principle (SRP) of SOLID, and carrying over practices from structured programming into an object-oriented context.

It is also important to highlight that the dark gray *code smells* shown in Figure 6,7 and 8 were not directly examined in the qualitative analysis. However, based on their known relationships with other analyzed *code smells*, one can infer that their presence in a student's code may indicate associated learning difficulties. For example, if a *God Class* is detected, it is plausible to infer that the student is experiencing learning difficulties similar to those linked with the *Large Class* smell.

Table 2: Code-Based Indicators of Difficulties in Learning Object-Oriented Programming Used. Adapted from [46]

Indicators of Object-Oriented Learning Challenges	Description
Large class [9]	A class that centralizes too many responsibilities, violating the Single Responsibility Principle. Related Insufficient modularization [52], Blob [6], Brain class [57] - God Class [44], Single Responsibility Principle [36].
Feature envy [9]	A method that accesses data from another object more than from its own class.
Shotgun surgery [9]	A single change requires modifications in many different classes simultaneously.
Data class [9]	A class that contains only fields and accessors with little or no behavior.
Long method [9]	A method that is too long and complex, making it hard to understand or maintain. Related Broken modularization [52], Single Responsibility Principle [36].
Functional decomposition [6]	Procedural-style code in OO programming that lacks true object orientation.
Refused bequest [9]	A subclass inherits methods or data it doesn't need or use. Related Rebellious hierarchy [52], Liskov Substitution Principle [36].
Spaghetti code [6]	Code with tangled logic and flow, making it difficult to follow and modify.
Divergent change [9]	A class that is often changed in different ways for different reasons. Related Multifaceted abstraction [52], Single Responsibility Principle [36].
Long parameter list [9]	A method that takes too many parameters, making it hard to use and refactor.
Duplicate code [9]	Identical or very similar code exists in more than one place. Related Duplicate abstraction [52], Unfactored hierarchy [52], Cut and paste programming [6].
Cyclically-dependent modularization [52]	Modules that depend on each other in a circular way, harming modularity. Related Dependency cycles [35].
Deficient encapsulation [52]	Internal implementation details are exposed, reducing flexibility and safety.
Speculative generality [9]	Code designed for future needs that may never occur, adding unnecessary complexity. Related Speculative hierarchy [52], Open/Closed Principle [36].
Lazy class [9]	A class that does too little to justify its existence. Related Unnecessary abstraction [52].
Switch statement [9]	Complex conditional logic spread through code instead of using polymorphism. Related Complicated Boolean Expression [58], Conditional Complexity [22], Unexploited encapsulation [52], Missing hierarchy [52], Repeated Switches [9], Open/Closed Principle [36].
Primitive obsession [9]	Overuse of primitive types instead of creating small objects for concepts. Related Missing abstraction [52].
Swiss army knife [6]	A class with too many unrelated responsibilities or utilities. Related Multifaceted abstraction [52].
Data Clump [9]	Groups of variables that appear together repeatedly and should be encapsulated.
Inappropriate Intimacy [9]	Classes that know too much about each other's internals.
Temporary Field [9]	Instance variables that are only sometimes used, depending on the context.
Middle Man [9]	A class that delegates all work to another class, adding unnecessary indirection.
Message Chains [9]	Chained method calls that expose navigation through multiple objects.
Parallel Inheritance Hierarchies [9]	Adding a subclass in one hierarchy forces changes in another related hierarchy.
Alternative Classes with Different Interfaces [9]	Classes that perform similar work but have different interfaces, complicating usage.
Interface Segregation Principle [36]	Interfaces should be specific and focused. A module should not be forced to depend on methods it does not use. This avoids large, general-purpose interfaces and promotes low coupling and high cohesion.
Dependency Inversion Principle [36]	High-level modules should depend on abstractions, not on implementations. Details should depend on abstractions, not the other way around. This principle encourages the use of interfaces and dependency injection to reduce coupling between components.

**Figure 6: Conceptual map showing the relationships between learning challenges (D02 and D03) and observable code issues**

4.2 Expert Evaluation

To assess the educational value and practical applicability of the proposed approach, we conducted an expert evaluation involving

two experienced computer science educators from Brazilian universities, with significant backgrounds in programming and software engineering instruction. One expert holds a Ph.D. and has 6 years of teaching experience, primarily in software engineering, project

Table 3: Challenges in OOP learning associated with code issues

Learning Challenges	Analysis Category	Codes (Registration Units)	Code Issues
D02 Classes	Static nature of class	D02.01 Static Not Assimilated	<i>Speculative Generality, Data Class</i>
	Understanding class hierarchy	D02.02 No Hierarchy, D02.03 Wrong Abstraction Level	<i>Refused Bequest, Large Class, Cyclically-dependent modularization, Parallel Inheritance Hierarchies</i>
	Identifying correct classes	D02.04 Cannot Identify Classes, D02.05 RealWorld Obj As Class	<i>Large Class, Shotgun Surgery, Swiss army knife, Alternative Classes with Different Interfaces, Interface Segregation Principle</i>
D03 – Methods	Confusion between class and object	D02.06 Class Equals Object	<i>Data Class, Large Class, Deficient encapsulation</i>
	Class perceived as a collection of objects	D02.07 Class As Collection, D02.08 No Abstraction In Class	<i>Data Class, Feature Envy</i>
	Difficulty calling methods	D03.01 Method Call Unknown, D03.02 Method Call Syntax Errors	<i>Message Chains, Long Parameter List, Data Clump</i>
D04 – OO Paradigm	Uncertainty about method quantity and naming	D03.03 Unsure Method Count, D03.04 Method Naming Confusion	<i>Long Method, Inappropriate Intimacy, Data Clump</i>
	Lack of understanding about method reuse	D03.05 Cannot Reuse Method, D03.06 Methods Rewritten Unnecessarily	<i>Duplicated Code, Lazy Class, Switch Statements</i>
	Issues with correct method placement	D03.07 Method Placement Issue, D03.08 Wrong Class For Method	<i>Feature Envy, Divergent Change</i>
D05 – OO Relationships	Difficulty shifting from structured to OO thinking	D04.01 Structured Thinking Dominance	<i>Large Class, Long Method, Functional decomposition, Spaghetti code</i>
	Difficulty in OO analysis and design	D04.02 OO Design Unclear	<i>Feature Envy, Shotgun Surgery, Cyclically-dependent modularization</i>
	Difficulty implementing OO concepts	D04.03 OO Implementation Failures, D04.04 Confused OO Syntax	<i>Divergent Change, Large Class, Primitive Obsession, Functional decomposition, Spaghetti code, Swiss army knife, Dependency Inversion Principle, Interface Segregation Principle</i>
D06 – Polymorphism and Overload	Misconceptions about OO paradigm	D04.05 OO Misconceptions, D04.06 Confuses Modularization	<i>Shotgun Surgery, Feature Envy, Functional decomposition, Spaghetti code, Swiss army knife, Dependency Inversion Principle</i>
	Trouble with association and dependency	D05.01 Association Unclear, D05.02 Dependency Misuse	<i>Inappropriate Intimacy, Middle Man, Dependency Inversion Principle</i>
	Difficulty with generalization/specialization (inheritance)	D05.03 Inheritance Not Applied, D05.04 No Generalization Modeling	<i>Refused Bequest, Temporary Field, Parallel Inheritance Hierarchies</i>
D07 – Encapsulation	Confusion with composition and aggregation	D05.05 Aggregation Or Composition Confusion	<i>Inappropriate Intimacy, Large Class</i>
	Difficulty modeling and implementing relationships	D05.06 Modeling Relationships Error, D05.07 Confused Concept Map	<i>Divergent Change, Shotgun Surgery</i>
	Polymorphism is too abstract	D06.01 Cannot Apply Polymorphism	<i>Switch Statements, Refused Bequest</i>
D07 – Encapsulation	Confusion about method overloading	D06.02 Overload Not Clear, D06.03 Duplicated Methods Instead	<i>Long Parameter List, Duplicated Code</i>
	Misunderstandings about encapsulation	D07.01 Encapsulation Misunderstood, D07.02 Encapsulation As Hiding Only	<i>Data Class, Inappropriate Intimacy, Deficient encapsulation</i>
	Lack of understanding about modularity	D07.03 Modularity Not Applied, D07.04 Mixes Concerns In Class	<i>Large Class, Divergent Change, Alternative Classes with Different Interfaces, Interface Segregation Principle</i>
	Problems with information hiding	D07.05 Exposes Internal State, D07.06 Uses Public Attributes	<i>Data Class, Inappropriate Intimacy, Deficient encapsulation, Data Clump</i>

management, and IT governance. The other expert holds a master's degree and has 14 years of teaching experience, with expertise in software engineering, systems analysis, programming, and artificial intelligence.

Each expert was asked to review the proposed model, apply it in selected student code samples, and evaluate its ability to accurately represent real classroom challenges as well as its potential impact on teaching and learning in introductory object-oriented programming. The evaluation included a combination of structured questionnaires and open-ended feedback.

The analysis focused on the ability of the proposed mapping to faithfully reflect real learning challenges observed in the classroom. To this end, participating educators submitted authentic code excerpts from students, each containing identifiable problems commonly encountered during instruction.

Each expert then navigated the full diagnostic path proposed by our visual model: starting from the Code Issues present in the

snippet, through the Analysis Categories, and finally arriving at the proposed Learning Challenges. The goal was to evaluate whether the final output corresponded meaningfully to the actual learning difficulties perceived in the classroom context for that particular student.

Table 4 presents representative samples from this process, outlining the classification path and the expert's judgment on the accuracy and relevance of the result. The samples are available at <https://github.com/brunostrik/BadQualityCodeExamples>

The analysis carried out highlighted the strong potential of the proposed mapping to represent, in a structured way, the learning challenges faced by students in introductory object-oriented programming courses. In several cases, such as examples S1 and S2, a significant correspondence was observed between the quality issues in the students' code and the fundamental concepts that require further development, such as encapsulation, class structuring, and understanding hierarchies. The ability to follow a clear diagnostic

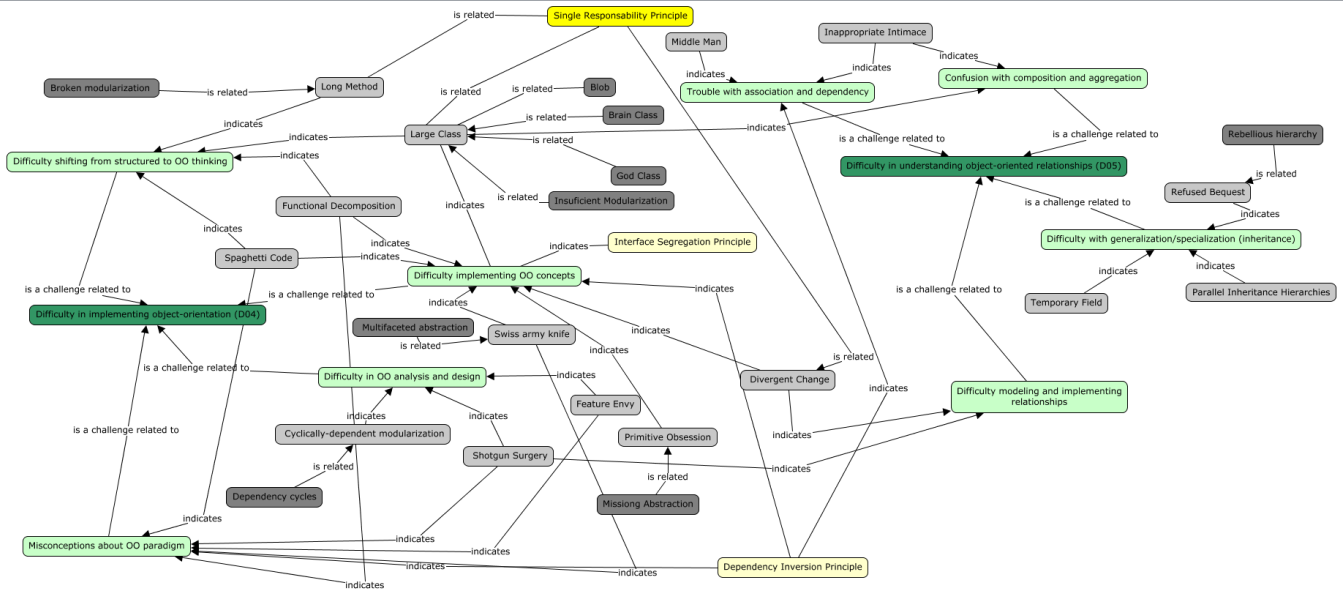


Figure 7: Conceptual map showing the relationships between learning challenges (D04 and D05) and observable code issues

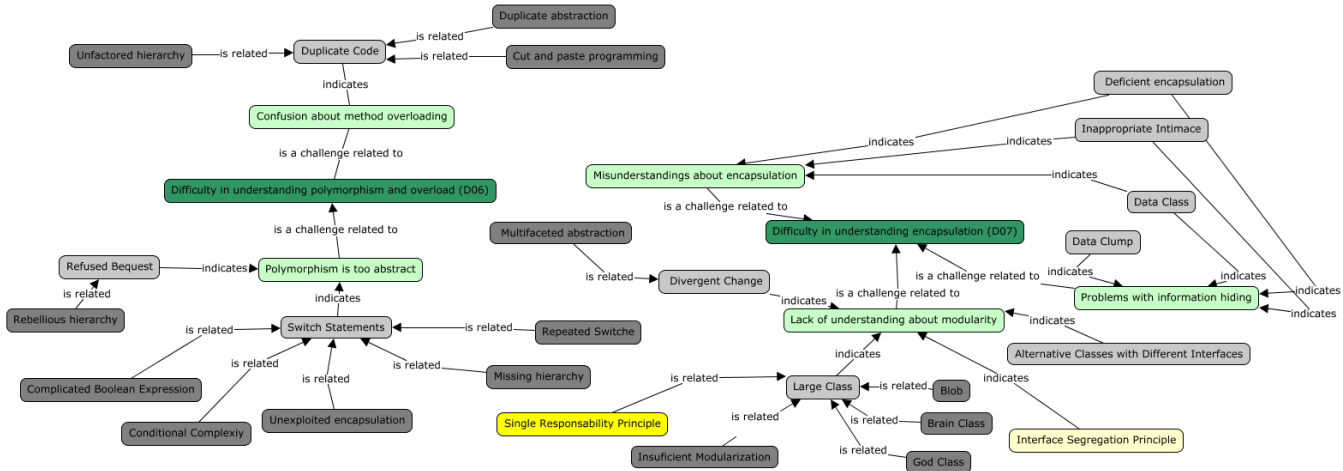


Figure 8: Conceptual map showing the relationships between learning challenges (D06 and D07) and observable code issues

path—from the identified problem to the underlying conceptual difficulties—represents a meaningful step forward for pedagogical practice.

In other cases, such as examples S3 and S4, the richness of the analyzed situations helped to reveal opportunities for refinement within the model, particularly in representing certain nuances that emerge in the teaching context. Even so, the model proved to be a promising tool to support instructors in identifying and analyzing students' difficulties. Overall, the evaluation suggests that the mapping is a useful complementary resource to the pedagogical perspective, with room to evolve as new situations and contexts are incorporated into its knowledge base.

5 Research Limitations

While the conceptual model presented in this study offers a novel approach to diagnosing OOP learning challenges through code analysis, it is important to acknowledge certain limitations. First, the mapping between code issues and learning difficulties was based on qualitative coding and expert interpretation.

Although we employed rigorous content analysis methods and triangulated findings with expert evaluations, subjective biases may still influence the categorization and associations identified.

Moreover, while the model incorporates widely recognized code smells and SOLID principles, it does not account for all possible design or implementation flaws that students might exhibit. Certain learning difficulties may manifest in ways not captured by the

Table 4: Summary of expert evaluation

Sample	Code Issues	Analysis Category	Learning Challenges	Expert Judgment
S1	Divergent change, Innapropriate intimacy	Lack of understanding about modularity, Trouble with association and dependency	D07 – Encapsulation, D05 – OO Relationships	Learning difficulties could be correctly identified through the mapping, although the problem indicator—understanding the context—was essential for identifying the correct relationships made possible by the mapping
S2	Large Class, Data Class	Understanding class hierarchy, Identifying correct classes	D02 Classes	The mapping was able to identify, based on the quality issues present in the source code, a poor implementation of the class structure and the underlying learning difficulties that motivated these problems
S3	Long Parameter List, Duplicated Code	Confusion about method overloading	D06 – Polymorphism and Overload	The problems caused by the incorrect implementation of method overloading were accurately mapped; however, the mapping did not capture the connection between the excessively long parameter list and the flawed class structure, which is the underlying cause of the identified issue
S4	Switch Statements	Confusion about method overloading	D06 – Polymorphism and Overload	The switch statement code smell was correctly mapped as a misunderstanding of polymorphism and overloading; however, the existing codes and analysis categories could be expanded with additional elements to more clearly capture the complete absence of an appropriate polymorphic structure, as observed in the analyzed code snippet

selected indicators or may result from non-code-related factors such as instructional design, student motivation, or prior knowledge.

Finally, the model's application in classroom settings depends on educators' familiarity with both software quality indicators and the conceptual framework. Without adequate training or supporting tools, instructors may face challenges in adopting the model effectively for formative assessment or instructional planning.

6 Conclusion and Future Work

Although OOP education, as well as issues such as code smells and violations of SOLID principles, are widely discussed in the literature, no studies explicitly explore the relationship between these topics. In this work, we investigated these connections through a qualitative analysis, drawing from prior studies that identify and consolidate the main learning difficulties in OOP, as well as from the literature on code smells and SOLID principles.

Our goal was to propose a conceptual map that supports the understanding of learning difficulties related to the object-oriented paradigm, based on evidence found in students' source code. This analysis, supported by the conceptual map, enables a deeper examination of students' difficulties in understanding core OOP concepts, rather than simply evaluating whether they are able to write functioning code. Considering this, the main contributions of the work are:

- (1) We identified the main object-oriented programming learning challenges that can be observed through source code analysis, based on the work of [12].
- (2) We categorized learning problems associated with each of these challenges.
- (3) We provided a visual representation of the relationships among different code smells identified by various authors.
- (4) We identified potential learning difficulties associated with the presence of each code smell or violation of SOLID principles.
- (5) We presented a visual representation that connects code smells and SOLID principle violations with object-oriented programming learning problems and their corresponding challenges.

The expert evaluation demonstrated the model's practical applicability, with educators confirming its ability to accurately reflect

real-world learning difficulties observed in student code. By linking observable code issues to underlying cognitive challenges, this work bridges the gap between software engineering practices and programming education.

To extend this research, we propose the following directions:

- **Tool Development:** Implement automated tools to analyze student code and map issues to learning challenges, integrating with popular educational platforms.
- **Expanded Language Support:** Validate the model with code written in languages beyond Java, such as Python or C++, to ensure broader applicability.
- **Longitudinal Studies:** Investigate how early identification and intervention based on code smells impact long-term OOP proficiency.
- **Instructor Dashboards:** Develop dashboards to help educators track common challenges across student cohorts and tailor instruction accordingly.

This work lays the foundation for a code-centric approach to OOP education, where quality issues serve as actionable insights into student learning. Future efforts will focus on scaling its adoption in classrooms and refining the model through empirical studies.

ARTIFACT AVAILABILITY

The samples used in the expert evaluation and the conceptual maps are available at:

<https://github.com/brunostrik/BadQualityCodeExamples>

ACKNOWLEDGMENTS

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) – Finance Code 001. Grammar and text structure were reviewed and refined with the assistance of AI tools, including Claude, Perplexity, and ChatGPT.

REFERENCES

- [1] Pérciles Alves, Eduardo Figueiredo, and Fabiano Ferrari. 2014. Avoiding code pitfalls in aspect-oriented programming. In *Programming Languages: 18th Brazilian Symposium, SBLP 2014, Maceio, Brazil, October 2-3, 2014. Proceedings* 18. Springer, 31–46.
- [2] Laurence Bardin. 2011. Content analysis. *São Paulo: Edições* 70, 279 (2011), 978–8562938047.

- [3] Katrin Becker. 2003. Grading programming assignments using rubrics. In *Proceedings of the 8th annual conference on Innovation and technology in computer science education*. 253–253.
- [4] Alan Benander, Barbara Benander, and Janche Sang. 2004. Factors related to the difficulty of learning to program in Java—an empirical study of non-novice programmers. *Information and Software Technology* 46, 2 (2004), 99–107.
- [5] Soly Mathew Biju. 2013. Difficulties in understanding object oriented programming concepts. In *Innovations and Advances in Computer, Information, Systems Sciences, and Engineering*. Springer, 319–326.
- [6] William H Brown, Raphael C Malveau, Hays W* Skip* McCormick, and Thomas J Mowbray. 1998. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc.
- [7] Omar Freddy Chamorro Atalaya. 2020. Analysis of Learning Difficulties in Object Oriented Programming in Systems Engineering Students at UNTELS. (2020).
- [8] Chin Soon Cheah. 2020. Factors contributing to the difficulties in teaching and learning of computer programming: A literature review. *Contemporary Educational Technology* 12, 2 (2020), ep272.
- [9] Martin Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.
- [10] Tony Gorschek, Ewan Tempero, and Lefteris Angelis. 2010. A large-scale empirical study of practitioners' use of object-oriented concepts. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering—Volume 1*. 115–124.
- [11] Michaela Greiler, Arie Van Deursen, and Margaret-Anne Storey. 2013. Automated detection of test fixture strategies and smells. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 322–331.
- [12] Luz E. Gutiérrez, Carlos A. Guerrero, and Héctor A. López-Ospina. 2022. Ranking of problems and solutions in the teaching and learning of object-oriented programming. *Education and Information Technologies* 27, 5 (June 2022), 7205–7239. doi:10.1007/s10639-022-10929-5
- [13] Irit Hadar. 2013. When intuition and logic clash: The case of the object-oriented paradigm. *Science of Computer Programming* 78, 9 (2013), 1407–1426.
- [14] R Wayne Hamm, Kenneth D Henderson Jr, Marilyn L Repsher, and Kathleen M Timmer. 1983. A tool for program grading: The Jacksonville University scale. In *Proceedings of the fourteenth SIGCSE technical symposium on Computer science education*. 248–252.
- [15] Benedikt Hauptmann, Maximilian Junker, Sebastian Eder, Lars Heinemann, Rudolf Vaas, and Peter Braun. 2013. Hunting for smells in natural language tests. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 1217–1220.
- [16] Simon Holland, Robert Griffiths, and Mark Woodman. 1997. Avoiding object misconceptions. *ACM Sigcse Bulletin* 29, 131–134. doi:10.1145/268084.268132
- [17] James W Howatt. 1994. On criteria for grading student programs. *ACM SIGCSE Bulletin* 26, 3 (1994), 3–7.
- [18] Peter Hubwieser and Andreas Mühling. 2011. What students (should) know about object oriented programming. In *Proceedings of the seventh international workshop on Computing education research*. 77–84.
- [19] International Organization for Standardization. 2014. *ISO/IEC 25000:2014 - Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE*. International Organization for Standardization, Geneva, Switzerland. Disponível em: <https://www.iso.org/standard/64764.html>. Acesso em: 12 ago. 2024.
- [20] Mohd Nasir Ismail, Nor Azilah Ngah, and Irfan Naufal Umar. 2010. Instructional strategy in the teaching of computer programming: a need assessment analyses. *The Turkish Online Journal of Educational Technology* 9, 2 (2010), 125–131.
- [21] Amela Karahasanović, Annette Kristin Levine, and Richard Thomas. 2007. Comprehension strategies and difficulties in maintaining object-oriented systems: An explorative study. *Journal of Systems and Software* 80, 9 (2007), 1541–1559.
- [22] Joshua Kerievsky. 2005. *Refactoring to patterns*. Pearson Deutschland GmbH.
- [23] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2023. A Systematic Mapping Study of Code Quality in Education. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1 (Turku, Finland) (ITICSE 2023)*. Association for Computing Machinery, New York, NY, USA, 5–11. doi:10.1145/3587102.3588777
- [24] Michael Kölling. 1999. The problem of teaching object-oriented programming, Part 1: Languages. *Journal of Object-oriented programming* 11, 8 (1999), 8–15.
- [25] Mario Konecki. 2014. Problems in programming education and means of their improvement. *DAAAM international scientific book 2014* (2014), 459–470.
- [26] Klaus Krippendorff. 2018. *Content analysis: An introduction to its methodology*. Sage publications, Thousand Oaks, California.
- [27] Guilherme Lacerda, Fabio Petrillo, Marcelo Pimenta, and Yann Gaël Guéhéneuc. 2020. Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software* 167 (2020), 110610.
- [28] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. 2005. A study of the difficulties of novice programmers. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (Caparica, Portugal) (ITICSE '05)*. Association for Computing Machinery, New York, NY, USA, 14–18. doi:10.1145/1067445.1067453
- [29] Tracy L Lewis, Mary Beth Rosson, and Manuel A Pérez-Quinones. 2004. What Do The Experts Say? teaching introductory design from an expert's perspective. *ACM SIGCSE Bulletin* 36, 1 (2004), 296–300.
- [30] Karl J. Lieberherr and Ian M. Holland. 1989. Assuring good style for object-oriented programs. *IEEE software* 6, 5 (1989), 38–48.
- [31] Barbara H Liskov and Jeannette M Wing. 1994. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 6 (1994), 1811–1841.
- [32] John Long. 2001. Software reuse antipatterns. *ACM SIGSOFT Software Engineering Notes* 26, 4 (2001), 68–76.
- [33] Isela Macia Bertran, Alessandro Garcia, and Arndt Von Staa. 2011. An exploratory study of code smells in evolving aspect-oriented systems. In *Proceedings of the tenth international conference on Aspect-oriented software development*. 203–214.
- [34] Mika Mantyla, Jari Vanhanen, and Casper Lassenius. 2003. A taxonomy and an initial empirical study of bad smells in code. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings*. IEEE, 381–384.
- [35] Klaus Marquardt. 2001. Dependency Structures N Architectural Diagnoses and Therapies.. In *EuroPLoP*. Citeseer, 11–52.
- [36] Micah Martin and Robert C Martin. 2006. *Agile principles, patterns, and practices in C*. Pearson Education.
- [37] Valéria F Martins, Ilana de Almeida Souza Concilio, and Marcelo de Paiva Guimarães. 2018. Problem based learning associated to the development of games for programming teaching. *Computer Applications in Engineering Education* 26, 5 (2018), 1577–1589.
- [38] D. Mazaitis. 1993. The object-oriented paradigm in the undergraduate curriculum: a survey of implementations and issues. *ACM SIGCSE Bulletin* 25, 3 (1993), 58–64. doi:10.1145/165408.165432
- [39] Robert Moser. 1997. A fantasy adventure game as a learning environment: why learning to program is so difficult and what can be done about it. In *Proceedings of the 2nd conference on Integrating technology into computer science education (ITICSE '97)*. Association for Computing Machinery, New York, NY, USA, 114–116. doi:10.1145/268819.268853
- [40] Wejdan Eissa Moussa, Raniyah Mutlaq Almalki, Maryam Abdulrahman Alamoudi, and Arwa Allinjaw. 2016. Proposing a 3d interactive visualization tool for learning oop concepts. In *2016 13th Learning and Technology Conference (L&T)*. IEEE, 1–7.
- [41] Hung Viet Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, Anh Tuan Nguyen, and Tien N Nguyen. 2012. Detection of embedded code smells in dynamic web applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 282–285.
- [42] Rachel Or-Bach and Ilana Lavy. 2004. Cognitive activities of abstraction in object orientation: an empirical study. *SIGCSE Bull.* 36, 2 (jun 2004), 82–86. doi:10.1145/1024338.1024378
- [43] KMM Rajashekharaiyah, Manjula Pawar, Mahesh S Patil, Nagarajna Kulenavar, and GH Joshi. 2016. Design thinking framework to enhance object oriented design and problem analysis skill in Java programming laboratory: An experience. In *2016 IEEE 4th International Conference on MOOCs, Innovation and Technology in Education (MITE)*. IEEE, 200–205.
- [44] Arthur J Riel. 1996. *Object-oriented design heuristics*. Addison-Wesley Longman Publishing Co., Inc.
- [45] Kate Sanders, Jonas Boustedt, Anna Eckerdal, Robert McCartney, Jan Erik Moström, Lynda Thomas, and Carol Zander. 2008. Student understanding of object-oriented programming as expressed in concept maps. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education*. 332–336.
- [46] Tushar Sharma and Diomidis Spinellis. 2018. A survey on software smells. *Journal of Systems and Software* 138 (2018), 158–173.
- [47] Steven D Sheetz, Gretchen Irwin, David P Tegarden, H James Nelson, and David E Monarchi. 1997. Exploring the difficulties of learning object-oriented techniques. *Journal of Management Information Systems* 14, 2 (1997), 103–131.
- [48] Ven Sien and David Chong. 2012. Threshold concepts in object-oriented modelling. *Electronic Communications of the EASST* 52 (2012).
- [49] Lon Smith and Jose Cordova. 2005. Weighted primary trait analysis for computer program evaluation. *Journal of Computing Sciences in Colleges* 20, 6 (2005), 14–19.
- [50] Ian Sommerville. 2015. *Software Engineering* (10th ed.). Pearson.
- [51] Martijn Stegeman, Erik Barendsen, and Sjaak Smetters. 2014. Towards an empirically validated model for assessment of code quality. In *Proceedings of the 14th Koli Calling international conference on computing education research*. 99–108.
- [52] Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. 2014. *Refactoring for software design smells: managing technical debt*. Morgan Kaufmann.
- [53] Phit-Huan Tan, Choo-Yee Ting, and Siew-Woei Ling. 2009. Learning difficulties in programming courses: undergraduates' perspective and perception. In *2009 International Conference on Computer Technology and Development*, Vol. 1. IEEE, 42–46.
- [54] David P Tegarden and Steven D Sheetz. 2001. Cognitive activities in OO development. *International Journal of Human-Computer Studies* 54, 6 (2001), 779–798.
- [55] David Thomas and Andrew Hunt. 2019. *The Pragmatic Programmer: your journey to mastery*. Addison-Wesley Professional.

- [56] Benjy Thomasson, Mark Ratchliffe, and Lynda Thomas. 2006. Identifying novice difficulties in object oriented design. *ACM SIGCSE Bulletin* 38, 3 (2006), 28–32.
- [57] Santiago A Vidal, Claudia Marcos, and J Andrés Díaz-Pace. 2016. An approach to prioritize code smells for refactoring. *Automated Software Engineering* 23 (2016), 501–532.
- [58] William C Wake. 2004. *Refactoring workbook*. Addison-Wesley Professional.
- [59] C. Watson and F. W. B. Li. 2014. Failure rates in introductory programming revisited. In *Proceedings of the 2014 conference on Innovation & technology in computer science education - ITiCSE '14*. ACM Press, New York, New York, USA, 39–44. doi:10.1145/2591708.2591749
- [60] Stelios Xinogalos. 2015. Object-oriented design and programming: an investigation of novices' conceptions on objects and classes. *ACM Transactions on Computing Education (TOCE)* 15, 3 (2015), 1–21.
- [61] Jeong Yang, Young Lee, and Kai H Chang. 2018. Evaluations of JaguarCode: A web-based object-oriented programming environment with static and dynamic visualization. *Journal of Systems and Software* 145 (2018), 147–163.