Code Smells Detective - An Educational Digital Game for Supporting the Learning-Teaching Process in Code Smells

Jefferson Wgner Francisco
Universidade Federal de Lavras
Lavras/MG, Brazil
jefferson.francisco@estudante.ufla.br

Heitor Costa
Universidade Federal de Lavras
Lavras/MG, Brazil
heitor@ufla.br

Abstract

We present the preliminary evaluation of Code Smells Detective, an educational digital game designed to improve students' understanding of code smells and code quality. The game was developed as an interactive, dynamic learning resource and was evaluated by 14 undergraduate and graduate students who had previously completed a Software Engineering course. The assessment used a structured questionnaire based on the MEEGA+ model, covering usability, learning, engagement, and player experience. The results showed a highly positive reception, with participants reporting strong usability, high engagement, and favorable perceptions of learning, suggesting that participants perceived the game as supportive of the assimilation of code-quality concepts. Overall, the study indicates that Code Smells Detective is a promising pedagogical tool for teaching code quality and recommends broader future evaluations with larger, more diverse samples, including longitudinal studies.

Keywords

Educational Games, Software Engineering Education, Code Smells

1 Introduction

The continuous pursuit of quality is a pillar of contemporary software engineering, both a technical ideal and an economic necessity for managing what is known as "technical debt". This concept, coined by Ward Cunningham [8], employs a financial metaphor to describe how design decisions that prioritize the short term accumulate "interest" over time, manifesting itself as reduced productivity and high maintenance costs. In this scenario, "code smells", a term popularized by Kent Beck and Martin Fowler [12], emerge as one of the indicators of this debt. While they do not prevent compilation, code smells are symptoms of deeper design issues, signaling weaknesses that compromise the future maintainability and reliability of software. Consequently, the ability to identify and mitigate these smells is a relevant skill for software engineers [12]. Despite its importance, teaching code smell identification faces pedagogical barriers. Novice students perceive concepts such as code quality as abstract and uninteresting [15, 25], partly due to a temporal disconnect: the negative impacts of a code smell are typically felt during maintenance, a phase often far removed from the reality of a professional focused on immediate delivery. Traditional pedagogical approaches, such as lectures, often fail to bridge this experiential gap, hindering the internalization of concepts [14] [28]. To overcome this engagement barrier, this paper focuses on `Code Smell Detective`, an educational digital game designed as a targeted pedagogical tool. While other playful approaches tend to focus on complex tasks, such as refactoring, the `Code Smells Detective` game fills a gap by targeting the first cognitive step. Pedagogically, the game targets the most basic level of Revised Bloom's Taxonomy ("Remembering"), which involves recognizing and recalling basic facts and concepts [3]. This focused approach uses the universally recognized mechanics of the memory game to create a low-cost, highly accessible activity that builds the vocabulary and knowledge base needed for higher-order thinking.

The choice of memory game mechanics is not arbitrary but based on cognitive science principles of learning. Our working memory capacity is limited, and practical instructional design should minimize extrinsic cognitive load (the mental effort to understand the learning task itself) so that learners can devote their resources to initial cognitive load (the process of constructing knowledge) [31]. The simplicity of the memory game rules serves this purpose precisely, allowing students to focus on the central task of forming mental associations. This process of direct pairing between a representation of a smell (be it a piece of code or an icon) and its name/concept is a classic example of Paired Associate Learning, a fundamental mechanism for memory consolidation [26, 33, 35]. In this study, Cognitive Load Theory (CLT) serves only as a design lens; we do not test the theory itself, but rather evaluate learners' perceptions of the CLT-informed game.

We aim is to present the design, pedagogical foundation, and the Code Smells Detective game, and analyze its potential as a complementary tool for introducing software quality concepts in an engaging and cognitively efficient way. Our main contribution is empirical evidence on the perceived quality and effectiveness of the Code Smells Detective game as a pedagogical tool. To guide this investigation, we formulated the following research question:

> How is a digital educational game, designed based on Cognitive Load Theory principles for the "Remembering" level of Bloom's Taxonomy, perceived by Computer Science students in terms of usability and player experience in supporting the learning of code smells?

TTo answer this research question, a case study was conducted with 14 undergraduate and graduate Computer Science students, using the MEEGA+ model [27] to evaluate the game. The quantitative results demonstrated a positive perception of the game, attesting to its quality in the Usability and Player's Experience factors, with a particular emphasis on participants' perceptions of learning.

The remainder of this paper is structured as follows. Section 2 presents the theoretical framework that guided the game design and the analysis of the results. Section 3 provides a brief description of the `Code Smells Detective` game. Section 4 details the empirical evaluation. Section 5 discusses the implications of the findings.

Section 6 presents threats to the validity of this work. Section 7 concludes the work and points to directions for future research.

2 Background

2.1 Code Smells and Source Code Quality

In Software Engineering, the relentless pursuit of quality transcends mere functional correctness, encompassing software maintainability, reliability, and efficiency [20]. Within this context, the concept of code smells, popularized by Kent Beck and Martin Fowler in their work "Refactoring: Improving the Design of Existing Code" [12], has become a pillar. A code smell is not a bug or an error that prevents compilation; it is a symptom, a feature in the source code structure that, while functional, serves as a strong indicator of a deeper design problem [34]. Identifying these smells is relevant, as they signal weaknesses that hinder software evolution, slow future development, and increase the risk of defects [30]. Identifying code smells can be subjective, depending on the project context and the developer's experience [2]. However, their importance is such that Robert C. Martin considers them part of an essential value system for software development, a marker of professionalism and quality [22]. Teaching code smell identification, therefore, is more than just passing along a list of rules; it is about cultivating a design sensibility in future software engineers and fostering a shift in mindset from an exclusive focus on "does the code work?" to a more mature focus on "is the code well-designed, readable, and maintainable?". We can find several taxonomies to structure the study and identification of these problems. Mika Mäntylä's classification, for example, groups code smells into intuitive categories such as Bloaters (elements that have grown excessively), Object-Orientation Abusers (misuse of object-oriented principles), Change Preventers (structures that hinder modifications), Dispensables (unnecessary elements), and Couplers (excessive coupling problems) [21]. More recently, Jerzyk and Madeyski's work presents a comprehensive online catalog and taxonomy of 56 code smells, reflecting the ongoing evolution of the field [18]. Typical examples, particularly relevant to beginning students, include Long Method (an overly lengthy method), Large Class or God Object (a class with too many responsibilities), Duplicated Code (identical or similar pieces of code in multiple places), Feature Envy (a method more interested in data of another class than its own), and Primitive Obsession (excessive use of primitive types to represent complex concepts). The ability to recognize these patterns is a relevant skill in Software Engineering education [17]. It is a prerequisite for learning refactoring techniques, controlled procedures for eliminating smells, and improving the internal quality of code. Therefore, providing students with concrete vocabulary and the ability to identify these problems early is an essential step in training professionals who can build robust, sustainable software over the long term [15].

2.2 The Impact of Code Smells: Technical Debt

The presence of code smells is not just a matter of aesthetics or abstract "best practices"; it has tangible economic and productivity consequences encapsulated by the metaphor of "Technical Debt". Coined by Ward Cunningham in 1992, the term draws an analogy between shortcuts in software development and incurring financial debt. In his experience report for the OOPSLA (Object-Oriented Programming, Systems, Languages & Applications) conference, Cunningham explained that delivering code non fully aligned with the best understanding of the problem ("not-quite-right code") is like taking out a loan: it can speed up initial delivery, but it will require "interest" payments in the future [8]. In this metaphor, "interest" represents the extra effort and lost productivity the development team faces each time they interact with compromised code. A code smell, like a Long Method or a Large Class, is a visible manifestation of this debt. Each time a developer needs to modify or understand this piece of code, they spend more time and cognitive effort than they would if the code were clean and well-structured. The developer must pay the additional time ("interest"). The continuous, unmanaged accumulation of this debt can lead an entire organization into stagnation, where most of the development effort is consumed solely by paying "interest" rather than adding new value to the product [8]. The discussion of technical debt introduces a relevant concept: intentionality. Not all debt is inherently flawed. Prudent and deliberate debt can be a strategic decision, such as releasing an initial version of a product more quickly to gather market feedback, with a clear plan to "pay off the debt" (by refactoring the code) later [8]. The real danger lies in inadvertent and reckless debt arising from negligence, lack of knowledge, or pressure to meet unrealistic deadlines, without awareness of its long-term consequences [12]. Code smells are often the primary manifestation of this type of dangerous debt [30]. Introducing this concept into Software Engineering teaching is transformative. It shifts the argument about code quality from a purely technical or aesthetic perspective ("write clean code because it is the right thing to do") to a pragmatic and economic perspective ("manage your technical debt so as not to compromise future productivity"). For a student focused on short-term goals, such as delivering functional work, the notion that shortcuts today can lead to a productivity "collapse" tomorrow provides a much more powerful and concrete motivation to focus on design quality from the outset.

2.3 Educational Games in Software Engineering

Faced with the challenge of teaching abstract concepts sometimes perceived as uninteresting, such as code quality [29], innovative pedagogical approaches are essential. Educational games emerge as a particularly promising strategy in this scenario. They aim to promote active learning and skill development in an engaging way. David Michael and Sande Chen's work ("Serious Games: Games That Educate, Train, and Inform") helped to consolidate the field, demonstrating its applicability across diverse sectors [23]. In the context of Software Engineering education, games and gamification have been successfully applied to teach a variety of topics, including agile methodologies such as Scrum (PlayScrum [10]) and project management (SimSE [24]), as well as software testing and programming [6]. The literature suggests that these approaches can enhance student motivation and engagement, facilitate a deeper understanding of complex concepts, and improve knowledge retention [19]. The *Code Smells Detective* game follows this proposal, but with a strategic focus. While many existing games in the code quality domain, such as the REFACTORY card game [16] or the 3D environment CodeArena [4], focus on refactoring (a complex skill that involves applying techniques to correct smells), our game addresses a fundamental gap in the learning process. It focuses

on the preliminary and indispensable cognitive stage: recognizing and naming code smells. To build a comprehensive understanding of impact and refactoring, developers must identify problematic code patterns and associate them with their canonical names. The choice of a simple game mechanic, such as a memory game, is not accidental; it is an instructional design decision intended to isolate and strengthen this fundamental skill before introducing additional layers of complexity. Several games have been proposed specifically to address code smells or code quality. Refactor4green [1] and CleanGame [9], for example, ask learners to identify and refactor code smells in concrete code snippets, combining recognition with higher-order skills such as applying refactoring techniques. REFACTORY [16] and CodeArena [4] similarly focus on refactoring and improving code metrics. These games target upper levels of Bloom's taxonomy (e.g., Applying, Analyzing), assuming that learners can already recognize and name common smells. In contrast, Code Smells Detective does not attempt to simulate refactoring or complex maintenance scenarios. Its contribution is to provide a tool focused on the Remembering level: helping learners build and reinforce the basic vocabulary and pattern recognition required to benefit from these more advanced games and activities.

2.4 Memory Games and Associative Learning

The effectiveness of the memory game as a teaching tool stems from its connection with fundamental cognitive mechanisms. Its central mechanic, turning cards to find matching pairs, is a direct implementation of the Associative Learning paradigm, specifically Paired Associate Learning (PAL) [35]. This paradigm describes the process by which the brain forms and strengthens a connection between two distinct stimuli. This theory claims that human cognition processes information through two distinct yet interconnected channels: a verbal system, which deals with language, and a non-verbal (or imagistic) system, which handles images and sensory representations. When information is presented in a way that allows both channels to process it simultaneously, learning and memorization are significantly enhanced [7].

2.5 Cognitive Load Theory

Learning a complex topic such as code smells requires significant mental effort for students. Cognitive Load Theory (CLT), developed by John Sweller, offers a robust framework for understanding and managing this effort. In this work, CLT is used solely as a design guideline to simplify game mechanics, not as a theory subject to direct empirical testing [31]. The CLT distinguishes three types of cognitive loads: i) **Intrinsic Load**: Refers to the inherent complexity of the material to learn, determined by the number of information elements and their interactivity. The topic of code smells has a high intrinsic load because it involves understanding multiple abstract concepts and their relationships; ii) **Extraneous Load**: This is the load imposed by the way of presenting information. Confusing instructions, poorly designed interfaces, or irrelevant information create extraneous, unproductive load on learning, minimizing its effectiveness; and iii) **Germination Load (Germane Load)**: This is the productive mental effort that the learner devotes to constructing mental schemas and integrating new knowledge with pre-existing knowledge in long-term memory. The goal of instructional design is to optimize this load. The approach used in the game aligns with the "worked-example effect", a key finding in CLT research [32]. For novice learners, studying worked examples is a more cognitively efficient approach than solving problems in an open-ended, unguided manner [31]. A traditional approach, such as giving a student a large block of code and asking them to "find the smells", represents unguided problem-solving with a very high external load. In contrast, each pair found in the memory game functions as a "worked micro-example": it explicitly and isolatedly presents the "solution" (the name and definition of the smell) to a "problem" (its representation).

2.6 Revised Bloom's Taxonomy

To ensure the effectiveness of a pedagogical intervention, a precise definition of its learning objectives is relevant. Revised Bloom's Taxonomy provides a widely accepted framework for categorizing these objectives by cognitive complexity [11]. Originally published in 1956 by a committee led by Benjamin Bloom [5], the taxonomy underwent significant revision in 2001 by Anderson and Krathwohl, who restructured it around action verbs, making it more practical for contemporary instructional design [3]. The revised taxonomy organizes cognitive processes into a six-level hierarchy, where mastery of one level is generally a prerequisite for the following: i) **Remembering**: Retrieving, recognizing, and recalling relevant knowledge from long-term memory; ii) **Understanding**: Constructing meaning from oral, written, and graphic messages; iii) **Applying**: Executing or using a procedure in a specific situation; iv) **Analyzing**: Breaking down material into constituent parts and determining how the parts relate to each other and an overall structure; v) **Evaluating**: Making judgments based on criteria and standards; and vi) **Creating**: Bringing elements together to form a coherent or functional whole; rearranging aspects into a new pattern or structure. This framework is relevant to the `Code Smells Detective` game because it enables us to position it precisely within the learning spectrum. The game focuses on the most fundamental level of the taxonomy: Remembering. The pair-finding mechanic directly trains the ability to "Remember", aligning with verbs such as recognize, identify, and recall.

3 Code Smells Detective

We conceived, designed, and developed the *Code Smells Detective* game using a systematic, theory-informed methodology that extends beyond software development, applying principles from instructional design, cognitive psychology, and game design frameworks. The central goal was to create a pedagogical tool that would achieve specific learning objectives and be engaging and accessible to beginner students in Software Engineering, since they often perceive code quality as abstract and uninteresting.

Building on the theoretical foundation presented in Section 2, three principles directly shaped the design of the game. First, following CLT (Section 2.5), we minimized extraneous load by adopting the memory-game mechanic, freeing working memory capacity for the intrinsic load of the code-smell concepts themselves. In practice, this translated into a small, fixed set of rules, a single interaction type (flipping cards to find pairs), and immediate visual and textual feedback on each match attempt. It helped to avoid interface- or rule-related confusion competing with the learning task. Second,

following Dual Coding Theory and the PAL paradigm (Section 2.4), each card pair combines a non-verbal representation of a code smell (a code snippet or icon) with its verbal representation (the name and a concise description of the code smell). Third, following the Remembering level of Bloom's Taxonomy (Section 2.6), the learning objective was scoped to recognition and recall, i.e., matching a description to its name.

These principles guided two design decisions. The code smells included in the game (Table 1) were selected based on their frequency, impact, and relevance to developers, prioritizing concepts that beginning students are likely to encounter early. The representation and description of each card were written to be concise and unambiguous, to facilitate the pairing process rather than introduce additional interpretive load. To move the game beyond a plain memorization exercise, we added a scoring system, attempt and pair counters, and visual/textual feedback (elements intended to increase motivation and provide a sense of progress, consistent with the engagement factors discussed in Section 2.3).

Table 1: The set of code smells currently implemented as card pairs in Code Smells Detective.

Code Smells	Description
Long Method	A method with many lines and/or responsibilities.
Large Class	A class with many responsibilities, methods and attributes.
Duplicate Code	Identical or very similar code snippets in multiple places.
Bad Names	Variables, methods, or classes with names that do not reveal their purpose.
Feature Envy	A method that seems more interested in another class's data than its own.
Data Class	A class that has only fields and methods to access them (get/set), with no behavior of its own.
Shotgun Surgery	A small change in one part of the code requires many small changes in many classes.
Message Chains	A Long sequence of nested method calls (e.g. obj.getA().getB().doSomething()).
Primitive Obsession	Excessive use of primitive types (int, string) instead of creating small classes for concepts.
Comments	A method is filled with explanatory comments.
Switch Statements	Excessive use of "switch-case" or "if-else if" which could be polymorphism.
Middle Man	A class that delegates most of its work to another class, serving only as a conduit.
Lazy Class	A class that doesn't do enough to justify its existence or maintenance.
Temporary Field	An instance attribute that is used only in specific situations or for a short period of time.
Speculative Generality	Code created "just in case" for future features that are never implemented.
Data Clumps	Groups of variables that frequently appear together in various parts of the code (e.g., host, port, user, pass).
Long Parameter List	More than three or four parameters for a method.
Refused Bequest	If a subclass uses only some of the inherited methods and properties, the hierarchy becomes unbalanced.
Alternative Classes with Different Interfaces	Two classes perform identical functions but have different method names.
Divergent Change	Need to change many unrelated methods when making changes to a class.
Parallel Inheritance Hierarchies	Whenever you create a subclass for a class, you need to create a subclass for another class.
Dead Code	A variable, parameter, field, method, or class is no longer used (usually because it is obsolete).
Inappropriate Intimacy	A class uses the internal fields and methods of another class.
Incomplete Library Class	Libraries that no longer meet users' needs.

To evaluate the feasibility of this design and turn it into a functional tool, we developed a web application using HTML5, CSS3, and JavaScript, prioritizing accessibility: the game runs in any modern browser, on desktop, tablet, or smartphone, without installation. Each session randomly selects a subset of code smells from a larger repository and shuffles them onto the board, supporting replayability and varied practice across sessions. Figure 2 shows the game's board interface, displaying the score counters (pairs found and attempts) and the grid of face-down cards used in the matching mechanic. The resulting artifact should be understood not only as a final product but as a research instrument, built to investigate whether memory-game mechanics can effectively support the teaching of code smells.

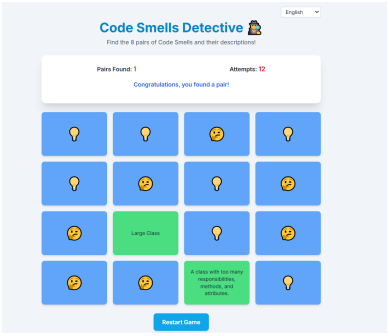

Figure 1: The Code Smells Detective game board, showing the score panel and card grid.

By constraining interaction to "match description to smell name", the game isolates this foundational skill rather than covering the full complexity of smell detection and refactoring. The design choice that motivates the evaluation reported in Section 4.

4 Evaluation

This section presents the evaluation of the quality and pedagogical potential of the `Code Smells Detective` game. We structured the evaluation using the MEEGA+ model (Model for the Evaluation of Educational Games), a validated instrument for evaluating educational games in computer science education.

We designed the study as a non-experimental case study with a single intervention group and post-test data collection, an approach justified by MEEGA+ for classroom assessments where control groups can be impractical. The objective was to analyze the perceived quality of the game from the perspective of computer science students. The academic profile of these students revealed a high level of qualifications: 57% (8 of 14) were currently enrolled in or had completed a postgraduate program, with the remaining 43% in advanced undergraduate studies in fields such as Computer Science and Software Engineering. Regarding Experience, 71% (10 of 14) classified their programming experience as "Intermediate" or "Advanced", and the majority had prior familiarity with the topic (71% were familiar with the term "Code Smell"). This participant's profile, more experienced than the primary target audience of beginning students, is a central point in the discussion of the results.

The measurement instrument used was the MEEGA+ questionnaire, consisting of 34 standardized items (Table **??**) evaluated on a 5-point Likert scale (1 = "Strongly Disagree" and 5 = "Strongly Agree"). The items measure the two quality factors of the model: Usability and Player's Experience. We conducted the study remotely; participants received a link to the game and, after completing the game session, were directed to the online evaluation form. The study followed a simple procedure. Participants received an invitation with a brief explanation that the goal was to evaluate an

educational game about code smells. After providing informed consent, they accessed the game through a web link. Each participant played at least one full session, defined as matching all pairs on the board. The game maintained a retry counter, and players were free to restart as many times as they wished; we did not impose time limits or a minimum number of sessions, mirroring a typical classroom scenario in which the game is used as a brief reinforcement activity. Immediately after completing their last session, participants were redirected to the online MEEGA+ questionnaire, whose responses were automatically stored in a spreadsheet.

To provide a more detailed analysis of the 14 participants' perceptions, we analyzed the frequency distributions of responses for each item in the MEEGA+ questionnaire. We organized them according to Usability and Player's Experience. The collected data were analyzed using descriptive statistics, calculating the mean (M), median (MD), and standard deviation (SD) for the item responses. Table **??** presents a comprehensive summary of these results, organized by the factors and dimensions of the MEEGA+ model.

The results indicate a positive perception of the Usability of the game. All dimensions within this factor received consistently high average scores, with medians of 4.00 or 5.00 across almost all items. Player's Experience was generally very positive, with highlights for the Relevance, Satisfaction, and Fun dimensions. The Challenge dimension showed more moderate results, and Social Interaction, as expected for a single-player game, received the lowest scores. The assessment can be found at: https://doi.org/10.5281/zenodo.17088683.

5 Discussion

The analysis of the data enables an in-depth discussion that connects the quantitative results to the theoretical foundation guiding the design of the `Code Smells Detective` game. Rather than testing Cognitive Load Theory itself, our goal was to design a game informed by CLT and examine learners' perceptions of its usability, relevance, and contribution to remembering code-smell concepts. The results of the empirical evaluation provide strong support for this goal. **Usability.** It shows that participants received the game very well. Most responses are concentrated at the highest scores (4 and 5), indicating that participants found the game easy to learn and play, with clear rules and an intuitive interface. This result supports the design decision to use simple mechanics, minimizing unnecessary cognitive load and allowing for a focus on learning. **Player´s Experience.** It also reveals a positive perception. The most significant result is that the Relevance and Perceived Learning dimensions received the highest ratings, indicating that participants perceived the game as an effective means of teaching and as directly related to the subject. While the game was considered fun and satisfying, scores for deep immersion (e.g., "losing track of time") were lower, as expected for a short, focused learning activity. Besides, social interaction items received the lowest scores, as expected, since the game is single-player.

In Table **??**, the high scores across all Usability dimensions, especially Learnability and Operability, directly indicate low extrinsic cognitive load. Participants reported that the game was straightforward to learn and play, with clear rules and an intuitive interface. This ease of use is not merely a convenience; it embodies the design principle of Cognitive Load Theory (CLT). By choosing the universally recognized mechanics of the memory game, the design eliminated the need for users to expend cognitive resources figuring out how to interact with the tool. This minimization of extrinsic load allowed participants to allocate their working memory capacity to the primary task: forming and strengthening associations between code smell representations and their formal names and concepts; the high scores on the Perceived Learning dimension are consistent with this design intention. Participants not only considered the game an adequate teaching method (M = 4.29), but also felt that it effectively contributed to their learning (M = 4.00). This connection between usability and perceived learning supports the design strategy: the simplicity of the tool served as a gateway to the content complexity.

This positive reception came from a group of mostly experienced participants. Although we designed the game with a focus on the "Remember" level of Bloom's Taxonomy for novices, the results suggest that it is useful beyond this audience. For professionals or advanced students, the game serves as a mechanism for reinforcing and practicing fluency. The high scores on Relevance (M = 4.57) and Satisfaction (M = 4.29) from this group indicate that the cognitive task of memorization and rapid association was sufficiently engaging. If experienced individuals find the tool relevant and valuable, the impact on students encountering these concepts for the first time is likely to be even more pronounced.

Concerning contribution to the teaching and learning process, the `Code Smells Detective` game benefits students in learning about code smells, offering several benefits, including: i) **Facilitates the understanding of abstract concepts**. Code smells are typically complex and abstract concepts for beginners. The game translates these ideas into conceptual and playful situations; ii) **Makes learning more engaging**. By incorporating game mechanics, learning becomes more dynamic, interactive, and engaging, increasing student engagement compared to traditional methods; iii) **Helps with memorization and recognition**. Repetition and the association of code smell names and definitions reinforce memorization and the ability to identify these problems in code, a relevant skill for developers; iv) **Promotes the development of technical vocabulary**. Students learn the correct nomenclature and become better at communicating about code smells in professional and academic settings; and v) **Builds a foundation for more advanced skills**. By focusing on recognizing and understanding code smells, the game prepares students for advanced topics such as refactoring and code reviews. In this sense, the simplicity of the mechanic is not a limitation of the tool, but a design choice to isolate and train a foundational micro-skill that other games typically embed as a secondary element. By explicitly targeting this micro-skill, Code Smells Detective complements more complex games and activities that focus on refactoring and higher-order problem solving.

6 Threats to Validity

The **external validity** is limited by its small convenience sample of 14 participants, who were mostly graduate and advanced undergraduate students with intermediate to advanced experience rather than novice learners. As a result, the positive findings cannot be confidently generalized to the game's primary target audience, and further validation with beginners is needed. The **internal validity** is limited because the study used a non-experimental case study

Quality Factor	Dimension	Questionnaire Item	M	MD	SD
Usability	Aesthetics	1. The game design is attractive (interface, graphics, etc.).	3.79	4.00	1.05
		2. The texts, colors and fonts match and are consistent.	4.21	4.00	0.70
	Learnability	3. I needed to learn a few "things" to be able to start playing.	4.21	5.00	1.25
		4. Learning to play this game was easy for me.	4.29	5.00	1.16
		5. I think most people would learn to play quickly.	4.21	5.00	1.05
	Operability	6. I consider the game to be easy to play.	4.50	5.00	0.85
		7. The rules of the game are clear and understandable.	4.14	4.00	0.95
	Accessibility	8. The fonts (size and style) of the letters used in the game are legible.	4.14	4.50	1.17
		9. The colors used in the game are understandable.	4.00	4.00	1.24
	Error protection	10. The game protects me from making mistakes.	4.14	4.50	1.23
		11. When I make a mistake, it's easy to recover quickly.	4.57	5.00	0.76
Player's Experience	Trust	12. When I first looked at the game, I had the impression that it would be easy.	4.71	5.00	0.61
		13. The organization of the content helped me feel confident that I would learn.	3.71	4.00	1.14
	Challenge	14. This game is adequately challenging for me.	3.29	3.50	1.27
		15. The game offers new challenges at a suitable pace.	2.86	3.00	1.35
		16. The game does not become monotonous in its tasks.	3.57	4.00	1.28
	Satisfaction	17. Completing the tasks of the game gave me a sense of accomplishment.	4.29	4.50	0.99
		18. It is due to my personal effort that I am able to advance in the game.	4.21	4.00	0.97
		19. I feel satisfied with what I learned in the game.	4.29	4.50	0.83
		20. I would recommend this game to my colleagues.	4.14	4.00	1.23
	Social Interaction	21. I was able to interact with other people during the game.	1.59	1.00	1.10
		22. The game promotes moments of cooperation and/or competition.	1.71	1.00	1.33
		23. I felt good interacting with other people during the game.	1.71	1.00	1.20
	Fun	24. I had fun with the game.	4.29	5.00	1.14
		25. Something happened during the game that made me smile.	2.93	3.00	1.44
	Focused Attention	26. There was something interesting early in the game that caught my attention.	4.07	4.50	1.21
		27. I was so involved in the game that I lost track of time.	2.50	2.00	1.45
		28. I forgot about the environment around me.	2.50	2.00	1.45
	Relevance	29. The game content is relevant to my interests.	4.57	5.00	0.65
		30. It is clear to me how the content relates to the topic of code smell.	4.71	5.00	0.47
	Perceived Learning	31. The game is an appropriate teaching method.	4.29	4.50	0.83
		32. The game contributed to my learning.	4.00	4.00	1.11
		33. The game was efficient for my learning.	3.71	4.00	1.07
		34. I'd rather learn from this game than any other way.	3.21	3.00	1.25

M = Mean, MD = Median, and SD = Standard Deviation

Figure 2: The Code Smells Detective game board, showing the score panel and card grid.

with only one intervention group and no control group, which prevents strong causal conclusions about the game's effect. Although participants reported positive perceptions, these results can also reflect other factors, such as the novelty effect, so the game cannot be identified as the sole cause of the observed outcomes. The **conclusion validity** is limited mainly by the small sample size (N = 14), which reduces statistical power and makes it harder to detect subtle effects or draw stronger statistical inferences. This limitation was partially mitigated by using MEEGA+, a validated questionnaire with high internal consistency in the literature, which increases confidence in the reliability of the measurements. The **construct validity** is limited because learning was assessed only through students' self-reported perceptions using the MEEGA+ scale. Although this captures perceived learning, it does not provide objective evidence of actual knowledge gain. Therefore, the results can reflect satisfaction or confidence rather than true learning. To strengthen validity, future studies should combine self-assessment with objective measures such as pre- and post-tests.

7 Final Remarks

This paper presented the conception, design, implementation, and evaluation of the `Code Smells Detective`, an educational game that teaches code smells. The theoretical basis demonstrated the relevance of the approach, and the developed game evaluated its technical feasibility. The empirical evaluation, conducted using the MEEGA+ model, provided evidence that the game is perceived as highly usable, with a positive player experience, and is perceived as relevant and effective for learning by a qualified audience. Our preliminary results suggest that simple game mechanics, when informed by cognitive principles, can be perceived as an effective strategy for demystifying abstract concepts and supporting the acquisition of foundational knowledge about code smells. The game fills a gap identified in the literature by focusing on the initial stage of learning (Revised Bloom's Taxonomy - "Remember"), which is relevant for developing more advanced software quality skills.

For the evolution of this work, follow some suggestions for future work: i) **Learning Validation with a Control Group**: Conduct an experimental study with pre- and post-tests of knowledge, comparing the group using the game with a control group using a traditional method. It will enable the objective measurement of learning gains and knowledge transfer, thereby addressing the main limitations of this study; ii) **Functionality Expansion**: Implement the functionalities foreseen in the conceptual design, such as difficulty levels that gradually increase the number of pairs and a "Smell Encyclopedia" that bridges to higher cognitive levels; and iii) **Integration with Learning Environments**: Explore the possibility of integrating the game into platforms such as Moodle or Canvas, using interoperability standards such as SCORM (Sharable Content Object Reference Model) or LTI (Learning Tools Interoperability), allowing instructors to monitor student progress centrally.

Artifact Availability

To support the principles of open science and ensure the reproducibility and replicability of our study, we have made a comprehensive replication package available to the program committee. This package includes all information used to generate the results presented in this paper, as well as links to the game and its source code. The information can be accessed at Zenodo repository [13].

Acknowledgments

We thank the volunteers who tested the game and completed the survey and the Google Gemini LLM, which assisted in reviewing the text.

References

[1] Vartika Agrahari and Sridhar Chimalakonda. 2020. Refactor4green: A game for novice programmers to learn code smells. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. 324–325.

[2] Farah Khiled AL-Jibory. 2023. Code smells in software. *International Journal of Nonlinear Analysis and Applications* 14, 1 (2023), 1339–1345.

[3] Lorin W Anderson and David R Krathwohl. 2001. *A taxonomy for learning, teaching, and assessing: A revision of Bloom's taxonomy of educational objectives: complete edition*. Addison Wesley Longman, Inc.

[4] Simon Baars and Sander Meester. 2019. CodeArena: Inspecting and improving code quality metrics using minecraft. In *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*. IEEE, 68–70.

[5] BS Bloom. 2001. Taxonomy of educational objectives and writing intended learning outcomes statements. *International Assembly for Collegiate Business Education (IACBE), USA* (2001).

[6] Lea C. Brandl and Andreas Schrader. 2024. Serious Games in Higher Education in the Transforming Process to Education 4.0—Systematized Review. *Education Sciences* 14, 3 (2024). doi:10.3390/educsci14030281

[7] James M Clark and Allan Paivio. 1991. Dual coding theory and education. *Educational psychology review* 3 (1991), 149–210.

[8] Ward Cunningham. 1992. The WyCash portfolio management system. *ACM Sigplan Oops Messenger* 4, 2 (1992), 29–30.

[9] Hoyama Maria dos Santos, Vinicius HS Durelli, Maurício Souza, Eduardo Figueiredo, Lucas Timoteo da Silva, and Rafael S Durelli. 2019. Cleangame: Gamifying the identification of code smells. In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*. 437–446.

[10] João M Fernandes and Sónia M Sousa. 2010. Playscrum-a card game to learn the scrum agile method. In *2010 Second International Conference on Games and Virtual Worlds for Serious Applications*. IEEE, 52–59.

[11] Ana Paula do Carmo Marcheti Ferraz and Renato Vairo Belhot. 2010. Taxonomia de Bloom: revisão teórica e apresentação das adequações do instrumento para definição de objetivos instrucionais. *Gestão & produção* 17 (2010), 421–431.

[12] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 2003. Refactoring: Improving the Design of Existing Code. 2000. *DOI= http://www. martinfowler. com/books. html/refactoring* (2003).

[13] Jefferson Francisco and Heitor Costa. 2025. *Code Smells Detective Educational Game: Quality and Pedagogical Perception Evaluation Dataset*. doi:10.5281/zenodo.17088684

[14] Scott Freeman, Sarah L. Eddy, Miles McDonough, Michelle K. Smith, Nnadozie Okoroafor, Heidi Jordt, and Mary Pat Wenderoth. 2014. Active learning increases student performance in science, engineering, and mathematics. *Proceedings of the National Academy of Sciences* 111, 23 (2014), 8410–8415. doi:10.1073/pnas.1319030111

[15] Paul C Grabow. 2015. Software Engineering I: Teaching Challenges. In *ACMS Conference Proceedings 2015*.

[16] Thorsten Haendler. 2019. A Card Game for Learning Software-Refactoring Principles.. In *GamiLearn*.

[17] Zijie Huang, Huiqun Yu, Guisheng Fan, Zhiqing Shao, Ziyi Zhou, and Mingchen Li. 2024. On the effectiveness of developer features in code smell prioritization: A replication study. *Journal of Systems and Software* 210 (2024), 111968.

[18] Marcel Jerzyk and Lech Madeyski. 2023. Code smells: A comprehensive online catalog and taxonomy. In *Developments in Information and Knowledge Management Systems for Business Applications: Volume 7*. Springer, 543–576.

[19] Serhii S Korniienko, Pavlo V Zahorodko, Andrii M Striuk, and Andrey I Kupin. 2024. A systematic review of gamification in software engineering education. In *CEUR Workshop Proceedings 83–95*. CEUR Workshop Proceedings.

[20] Letaw Lara. 2024. *Handbook of Software Engineering Methods*. Disponível em: https://open.oregonstate.edu/.

[21] Mika V Mäntylä and Casper Lassenius. 2006. Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering* 11 (2006), 395–431.

[22] Robert C Martin. 2013. *Clean Code-Refactoring, Patterns, Testen und Techniken für sauberen Code: Deutsche Ausgabe*. MITP-Verlags GmbH & Co. KG.

[23] D.R. Michael and S. Chen. 2006. *Serious Games: Games that Educate, Train and Inform*. Thomson Course Technology. https://books.google.com.br/books?id=49kTAQAAIAAJ

[24] Emily Oh Navarro and André van der Hoek. 2004. SIMSE: An Interactive Simulation Game for Software Engineering Education.. In *CATE*, Vol. 1. 12–17.

[25] Sofia Ouhbi and Nuno Pombo. 2020. Software Engineering Education: Challenges and Perspectives. In *2020 IEEE Global Engineering Education Conference (EDUCON)*. 202–209. doi:10.1109/EDUCON45650.2020.9125353

[26] Allan Paivio. 1991. Dual coding theory: Retrospect and current status. *Canadian Journal of Psychology/Revue canadienne de psychologie* 45, 3 (1991), 255.

[27] Giani Petri, Christiane Gresse Von Wangenheim, and Adriano Ferreti Borgatto. 2019. MEEGA+: Um Modelo para a Avaliação de Jogos Educacionais para o ensino de Computação. *Revista Brasileira de Informática na Educação* 27, 03 (2019), 52–81.

[28] Michael Prince. 2004. Does active learning work? A review of the research. *Journal of Engineering Education* 93, 3 (2004), 223–231. doi:10.1002/j.2168-9830.2004.tb00809.x

[29] Philipp Straubinger, Tim Greller, and Gordon Fraser. 2025. Teaching Software Testing and Debugging with the Serious Game Sojourner under Sabotage. In *Proceedings of the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 16–20.

[30] Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. 2014. *Refactoring for software design smells: managing technical debt*. Morgan Kaufmann.

[31] John Sweller. 1988. Cognitive load during problem solving: Effects on learning. *Cognitive science* 12, 2 (1988), 257–285.

[32] John Sweller. 2006. The worked example effect and human cognition. *Learning and instruction* (2006).

[33] Paul Thibodeau, Isaac Levy, and Mikaela de Lemos. 2021. Exploring mental representation with a memory matching game. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, Vol. 43.

[34] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2015. When and Why Your Code Starts to Smell Bad. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 403–414. doi:10.1109/ICSE.2015.59

[35] Jin-Hui Wang and Shan Cui. 2018. Associative memory cells and their working principle in the brain. *F1000Research* 7 (2018), 108.