Reengineering process applied to serious games: The T-TEA console case study

Alexandre Altair de Melo[1,2], **Luis Eduardo Bet**[3], **André Bonetto Trindade**[4], **Marcelo da Silva Hounsell**[2,3,4]

[1]Department of Teaching, Research and Extension – Instituto Federal de Santa Catarina (IFSC) – Jaraguá do Sul – SC – Brazil

[2]Postgraduate Program in Applied Computing – Universidade do Estado de Santa Catarina (Udesc) – Joinville – SC – Brazil

[3]Department of Computer Science – Universidade do Estado de Santa Catarina (Udesc) – Joinville – SC – Brazil

[4]Postgraduate Program in Electrical Engineering – Universidade do Estado de Santa Catarina (Udesc) – Joinville – SC – Brazil

alexandre.melo@ifsc.edu.br, {luis.bet,andre.trindade}@edu.udesc.br, marcelo.hounsell@udesc.br

Abstract. Introduction: *The T-TEA console is an interactive floor interface designed for exergames targeting children with Autism Spectrum Disorder (ASD). Initially developed in an academic context, the project prioritized functionality over software engineering practices, resulting in a procedural and hard-to-maintain codebase.* ***Objective:*** *This study analyzes the impact of a reengineering process on the T-TEA software, focusing on the transition to Object-Oriented (OO) programming, adoption of the MVC architecture, and use of design patterns to improve quality and maintainability.* ***Methodology:*** *The process included code analysis using Pylint and SonarQube to measure metrics such as Pylint score, Cyclomatic Complexity, and Technical Debt, followed by refactoring, migration to an OO paradigm, implementation of the MVC architecture, and the addition of new features.* ***Results:*** *Code reengineering significantly reduced Technical Debt (from 1,738 to 37 minutes) and Cyclomatic Complexity per file (from 42 to 13), while Pylint scores improved by up to 318%, demonstrating enhanced maintainability and software quality. The results prove that performing software reengineering can have significant impact on code metrics.*

Keywords *Game Development, Refactoring, Serious Games, Software Engineering, Software Quality*

1. Introduction

Serious games are designed around goals or challenges, whether mental or physical, structured through game actions and defined rules. Beyond entertainment, their primary purpose is to achieve at least one additional objective, often referred to as a 'characteristic goal' such as enhancing player's skills or fostering awareness of their values and perspectives [Conradt et al. 2021].

Games that require the player to make some type of movement or physical effort, are classified as active serious games or exergames [Miranda et al. 2025]. Exergames

can use various forms of interface to capture user reactions to the game, for example, specific remote controls, interactive boards and projections [Trindade et al. 2022b]. Due to their characteristics, exergames have emerged as a promising strategy for enhancing cognitive and physical performance, supporting both educational and therapeutic goals [Miranda et al. 2025]. These types of interventions are particularly relevant for addressing challenges faced by children on the autism spectrum, providing a more engaging and accessible means of promoting motor and executive skills.

Among the interface alternatives to exergames, Interactive Floor has particular affordances to Autism Spectrum Disorder (ASD) individuals thanks to its multisensorial nature (kinesthetic and visual). ASD is a condition characterized by persistent difficulties in communication and social interaction in various contexts [American Psychiatric Association 2013]. T-TEA [Trindade et al. 2022a], is an example of Interactive Floor interface, but its development has not undergone rigorous software engineering principles.

Software development in academic contexts sometimes does not include fundamental aspects of software engineering. This situation derives from several factors, such as reduced deadlines for delivery, the inexperience of those involved, the absence of perspectives regarding the continuity of the project, or the limitation of human resources. It is quite common to academic projects to focus on functionality instead of modularity, clarity, maintainability, and so on [Arvanitou et al. 2022]. This scenario also occurs in the field of digital games, particularly in exergames, as was the case with the T-TEA project.

This article presents a case study on the reengineering process applied to the T-TEA console software. The case study methodology enables an in-depth investigation of a contemporary phenomenon, namely software improvement, within its real-world context. The objective of this investigation is to analyze how transitioning from a structured model to an object-oriented approach, adopting an MVC architecture, applying design patterns, and introducing new features have impacted the overall quality of the software. The process was guided by objective metrics, using Cyclomatic Complexity, Technical Debt and Pylint Score to assess the improvements in code maintainability and clarity.

2. Background

Software is designed to solve a problem in a specific application domain. However, over time, the system demands evolution, whether through improvements, bug fixes, or other needs. Therefore, the software will evolve over time, with changes driving this process. All systems have a limited lifetime. Each implemented change erodes the structure which makes the following change more expensive. As time goes on, the cost to implement a change will be too high, and the system will then be unable to support its intended task [Jacobson and Lindström 1991]. In such cases, software reengineering may be a viable option to address software maintenance challenges. The reengineering process consists of understanding and transforming an existing software system, commonly a legacy system, in order to make it easier to maintain. The reengineering process may involve redocumenting the system, reorganizing the system architecture, and reimplementing the code using a more modern language or technique [Sommerville 2011].

[Singh et al. 2021], which uses the validation of reengineering with Agile Scrum, and [Sianturi et al. 2024], which demonstrates the iterative reengineering process in an

application, show that the reengineering process improves code quality, maintainability, and user adoption. Another study [Penteado et al. 1998] presents a reengineering process based on migrating from a procedural paradigm to an object-oriente (OO) approach, resulting in improvements in code reusability, maintainability, and readability.

2.1. Procedural to Object-Oriented Migration

Maintenance of software developed within the procedural paradigm represents many problems. In procedural programs it is rather difficult to define the artifacts and their relationships, because they are often not visible in procedural programs (e.g. side effects, etc.) [Gall and Klosch 1995]. To make this paradigm shift, it is necessary to consider several factors that can assist in the process, from the primary source code, system documentation, training materials, etc. These factors contribute to the understanding of system construction, thus generating an initial analysis model. The first step, i.e., preparation of an analysis model, requires that we assimilate the existing information about the system [Jacobson and Lindström 1991]. The existing information has many different forms, e.g. requirements specifications, user operating instructions, maintenance manuals, training manuals, design documentation, source code files, and database schema descriptions [Jacobson and Lindström 1991]. This survey contributes to the search for a general understanding of how the system works. But when it comes to source code, how do you identify code snippets that are candidates to be transformed into classes? An "object" in a conventional programming language can be identified as a collection of routines, types, and/or data items [Liu and Wilde 1990]. The routines will implement the methods associated with the object, the types will structure the data it conceals or processes and the data items will represent or point to actual instances of the object class [Liu and Wilde 1990].

Once you've identified the characteristics of procedural code for migration to the object-oriented model, separating responsibilities between components, from viewing data on screen to persisting it, is a key consideration. Every software program that requires at least a bit of interactivity with users requires a user interface. This makes the integration of the user interface with the application domain a recurring engineering problem [Syromiatnikov and Weyns 2014]. Design patterns provide generic solution schemes for recurring design problems, offering reference materials that give engineers access to the field's systematic knowledge [Clements and Shaw 2009]. Model-View-Controller (MVC) is a widely known design pattern to integrate a user interface with the application domain. Central to MVC is the separation of the representation of the application domain (the model) from the display of the application's state (the view) and the user interaction processing (the controller) [Syromiatnikov and Weyns 2014].

3. Metrics

To assess software quality we used the following metrics:

- Pylint Code Quality Score;
- Mccabe's Cyclomatic Complexity;
- Technical Debt.

3.1. Pylint - Code Quality Score

Pylint is an automated tool that can be used to assess the quality of a python code based on a weighted linear calculation that takes into account number of errors, warnings, refractors and conventions found during code analysis [Dasgupta and Hooshangi 2017]. In this context[1], warning and convention flags are raised when the code does not follow standard Python convention or style guides or there is an issue with variable assignments. Another scenario with a warning, as a variable name or having unused variables in the code. A typical Pylint output is a basic terminal output that includes all the warnings, convention issues and error messages, basic statistics about the code (such as number of modules, classes, and methods) and raw metrics related to the number of lines of actual code, docstring, comments and empty space within the code. A total rating score is provided at the end of the output as an overall measure of the code quality [Dasgupta and Hooshangi 2017]. A resulting score can be classified as follows: code that exemplifies excellence in syntax, compliance to coding standards, and absence of linting errors are classified as "High (Pylint score greater than 7). Code that manifests minor linting errors or warnings, albeit being functional, is deemed Moderate (Pylint score between 3 to 7). At the lowest tier, Low (Pylint score lesser than 3) is assigned to code fraught with significant issues, including but not limited to syntax errors, flagrant deviations from accepted coding practices, and critical linting errors" [Chavan and Mondal 2024]. Pylint uses the Equation (1) to calculate the rating score [Dasgupta and Hooshangi 2017]:

$$\text{Score} = 10.0 - \left(\frac{5 \times \text{error} + \text{warning} + \text{refactor} + \text{convention}}{\text{statement}} \right) \times 10 \qquad (1)$$

3.2. Mccabe's Cyclomatic Complexity Metric - CCM

Software complexity is one branch of software metrics that is focused on direct measurement of software attributes, as opposed to indirect software measures such as project milestone status and reported system failures [Watson et al. 1996]. Cyclomatic complexity [McCabe 1976] measures the amount of logic decision in a single software module. Another use for this measure is the use in software test. [Watson et al. 1996] explains about this metric that first, it gives the number of recommended tests for software; second, it is used during all phases of the software lifecycle, beginning with design, to keep software reliable, testable, and manageable. Cyclomatic complexity is based entirely on the structure of software's control flow graph. There is a strong connection between the bug density and the CCM [Yu and Zhou 2010]. Usually the CCM should be less than 10, and not be more than 20. When the CCM is close to 100, the software will be so complicated that, when a bug is fixed, a new bug will be introduced there is more them 60% of probability[Yu and Zhou 2010]. Thus, the code is nearly out of control.

3.3. Technical Debt

Technical Debt refers to postponed tasks or immature artifacts within a software project that may provide short-term benefits, such as higher productivity and

[1]Pylint supports the Python standard library out of the box. Third-party libraries are not always supported, so a plugin might be needed, source: https://pylint.readthedocs.io/en/stable/

lower costs, but eventually lead to increased effort and costs during the system's evolution and maintenance [Avgeriou et al. 2016]. The systematic literature review in [Kleinwaks et al. 2023] shows that the main causes of technical debt in software systems are external factors, deviations from standards, and poor code quality. The first cause is related to cost and schedule constraints, while the second and the last refer to failures in following established standards and the development of poor or incomplete work. Also, regarding the game development industry, [Borowa et al. 2021] presents a survey showing that professional game developers are aware of technical debt, but its management is still a rare practice in the video game industry.

4. Design the case study

When conducting a case study, there are five major process steps to go through [Runeson and Höst 2009]: Case study design, preparation for data collection, collecting evidence, analysis of collected data and reporting. This section describes the steps taken for the case study presented here.

4.1. Research questions

This article aims to examine how the use of software engineering through code reengineering, moving from a structured to an OO model combined with the use of software metrics, can contribute to improving the development of an exergame-type system like T-TEA console outgame on interface system and auxiliary records. This leads to the following research questions (RQs):

- RQ1: What is the impact of a reengineering method applied with the OO paradigm and the MVC pattern, on the Technical Debt, Pylint Quality Score and Average Cyclomatic Complexity of the T-TEA console?
- RQ2: How does the adoption of software engineering practices (coding standards) influence code maintainability during the process?

4.2. Data collection

To collect data regarding the quality of code maintenance, the SonarQube tool will be used. This tool will be used to extract metrics for Mccabe's complexity, lines of code, technical debt, aiming to understand the quality of the code in terms of its maintainability. SonarQube is an open source Static Application Security Testing (SAST) capable of detecting a wide range of code quality issues in more than 30 languages, including Python [SonarSource SA 2026]. It has been utilized in numerous software engineering empirical studies like [Avgeriou et al. 2021], [Lenarduzzi et al. 2020], [Yu et al. 2023], [Hassan et al. 2024] for example. In our study, SonarQube served as the tool for measuring Mccabe's CCM. The Pylint tool was used to identify the code's adherence to Python language best practices, measuring the code's quality in terms of documentation, creation and correct use of variables, among other points mentioned in section 3.1.

4.3. Data Analysis Procedures

For the quantitative analysis of the process, information regarding the metrics used before and after the intervention was tabulated, as provided by the automated tools used in the study. From a qualitative point of view, the process of reengineering and rewriting the

code to follow the OO paradigm, the adoption of the MVC design pattern, and their contribution to the observed improvements will be described. Other reference sources, such as system documentation, training manuals, etc., will also serve as support for analyzing the new architecture model.

5. Rethinking T-TEA architecture

The T-TEA console uses conventional equipment such as a video projector, camera, Windows 10+ computer and support structure to project the exergame onto the floor [Trindade et al. 2022a]. All assembled in a specific configuration (Figure 1).

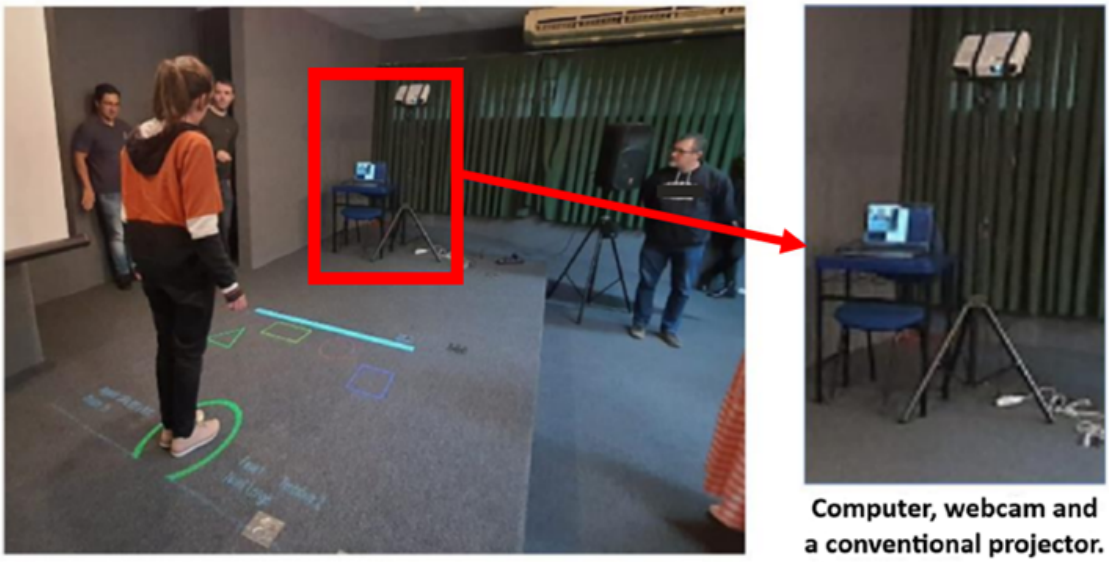

Figure 1. Configuration of the T-TEA console [Trindade et al. 2022a].

The console currently has 3 games. These games were built using the Python language, using the OpenCV2 computer vision module and the MediaPipe3 machine learning module. The RepeTEA game is for single player and focus on attention, short-term memory and motor training [Trindade et al. 2022a]. KarTEA assist in the development of multisensory integration, through the stimulation of concentration, attention, motor coordination and laterality of the player [Pereira et al. 2023]. VesTEA requires the player to analyze the challenge presented, and select the clothing that best suits the challenge while at the same time moving through a virtual maze [TTEA 2025].

The console interface base software provide three functionalities:

- The main menu is responsible for logging the player in and providing access to the games;
- An specific post-installation configuration step called calibration function;
- An analytics and database support to register users and save data regarding their use of the games.

To reduce errors in determining the player's position on the projection due to the positioning of the camera in relation to the floor, the user must perform a calibration after installing the T-TEA. Therefore, addressing the issues in the code began with an analysis of the original codebase and a refactoring process aimed at reducing CCM and improving maintainability and readability in the future.

5.1. Analyzing the Code

The T-TEA software restructuring process follows a hybrid approach that combines the use of tools such as Pylint and SonarQube to collect information on code quality, with system restructuring carried out by the current project team. When using Pylint to check code compliance to the best practices suggested by the Python language and PEP 8 coding convention [van Rossum and Warsaw 2010], we had the following scores before and after the reworking in Table 1. The scores are on a scale from 0 to 10, where higher values indicate better quality.

Using SonarQube on T-TEA's original codebase, which includes the pre-game and exergames software, was analysed (see Table 1). These values revealed several issues, such as duplicated files in the codebase, and a lack of a well-defined architectural pattern, programming paradigm, and coding conventions.

One part of the code that presented significant issues was the console interface, also known as the pre-game interface, which consists of three files: a menu file, an archive management file, and a calibration file. The refactoring data analysis, shown in Table 1, refers only to the first two files, as the calibration module has not yet been included in the refactoring process. Data highlighted in red in Table 1 refers to worsening, and highlighted in green refers to metrics improvements.

Table 1. SonarQube code metrics before and after the console interface refactoring

Metrics	Whole Codebase	Menu and Archive Files	
		Before	After
Lines of Code	7130	591	1903
Comments (%)	15	23	62
Number of Files	35	2	23
Number of Classes	25	0	23
Number of Functions	312	77	138
Technical Debt (min.)	9060	1738	37
Duplicated Code (%)	27	0	0
Total CCM	1447	85	299
Average CCM (per file)	41	42	13
Highest CCM File	485	66	58
Lowest CCM File	0	19	3
Pylint Quality Score	-	5.8/3.0	8.5/9.4

Refactoring was motivated by the need to reduce the TD of 1738 minutes (approximately 1 day and 5 hours) and CCM, with the goal of improving the system's maintainability and evolvability.

5.2. Reengineering and Refactoring the previous code

To improve code quality while addressing reported bugs and newly identified requirements, a code reengineering and refactoring process was initiated:

1. The PEP 8 coding convention was adopted to establish a consistent pattern that makes the written code more readable, maintainable, and easier to understand in the future. To apply this convention, the existing modules, files, and variables were refactored to follow the pattern [van Rossum and Warsaw 2010];
2. Object-Oriented Programming (OOP) was selected as the primary programming paradigm because of its advantages in terms of reusability, testability, and maintainability. Since a significant part of the system was originally implemented using a procedural approach, it was reengineered and rewritten following OOP paradigm. Also, during this process several refactoring techniques were applied, such as renaming, variable extraction, and dead code removal. This include software design patterns seek to codify expert knowledge to share experience about successful design structures [Zhang and Budgen 2012];
3. The Model-View-Controller (MVC) architectural pattern was defined as a standard over a previously undefined architecture. The MVC architecture, commonly used with the OOP paradigm, provides a structure that separates the graphical interface, event handling, and data storage and management;
4. The Kanban framework was used to visualize the workflow stages, and GitHub, was employed for version control and collaboration.

Following these definitions, the menu and archive management modules of the console interface were selected as the first components to be refactored, due to the high TD and CCM. Also, new features were added to the pre-game interface, like:

- User manuals for both the console and the exergames;
- Splash Screen when open the console interface;
- A System Logging Mechanism for error capture;
- Language support for both portuguese and english language;
- Windows and Linux support.

After the reengineering and refactoring process, the console interface code was analysed by Pylint and SonarQube again. Table 1 shows the results.

6. Discussions

The main goal of this case study is to improve the T-TEA code quality, initially focusing on the console interface, by applying software engineering techniques, primarily code refactoring methods, and using SonarQube code analysis to provide evaluation metrics. Comparing the code metrics before and after the refactoring process, almost all maintainability issues were resolved, but the total CCM increased by 352%.

The increase in CCM is due to the greater number of lines and files, arising from the addition of five new features. However, this complexity is distributed throughout the codebase, with an average CCM of 13 per file. The file with the highest CCM is the main application file, which is a candidate for improvement in the near future. Also, a critical bug related to user data persistence and consistency was fixed along the process and improvements in code readability were implemented due to the adoption of coding conventions and the refactoring methods applied. This type of benefit is especially important, given the plurality of people who currently work in the codebase.

On the other hand, the quality of the code in relation to the adoption of good practices measured by Pylint, both in the adoption of the PEP 8 style guide and in the

quality of the code in general, has increased 46.38% in Menu Interface (console main interface) and 318.30% on Archive File.

Furthermore, a qualitative analysis of the existing materials, including the platform's training manual and games, contributed to understanding how the console operates. However, there was little internal documentation within the code or related analysis documents, a limitation that was gradually overcome throughout the process.

6.1. Threats to Validity

Study validity denotes the reliability of the results. As with any empirical study, there are potential threats to the validity of this research. Following the guidelines for case studies in SE proposed by [Runeson and Höst 2009], we analyzed four categories of threats and the countermeasures adopted:

- Construct Validity - Construct validity reflects the extent to which the operational measures studied actually represent what the researcher has in mind:
  - Threat: The main construct threat is whether the metrics of TD and CCM from SonarQube, and the score reported by Pylint, indeed represent the concepts of "code quality" and "maintainability".
  - Countermeasure: To mitigate this threat, we use automated tools widely accepted in industry and software engineering research (SonarQube and Pylint). The use of standardized tools ensures that data collection is objective and repeatable, rather than depending on the subjective interpretation of the researchers. Furthermore, CCM and TD are well-established constructs in the literature.
- Internal Validity - Internal validity is a concern when examining causal relationships, i.e., whether the intervention (refactoring) actually caused the change in outcomes (metrics):
  - Threat: The main threat to internal validity is the presence of confounding factors. During the refactoring process, new features were added to the console, such as user manuals, splash screens, and system logs. This explains why the total number of lines of code and total CCM increased. Therefore, the improvement in metrics cannot be attributed solely to the refactoring.
  - Countermeasure: The authors recognized this threat and addressed it by analyzing not only total metrics but also average metrics. As shown in Table 1, the Average CCM (per file) dropped from 42 to 13, and TD dropped from 1738 minutes to 37. The drastic reduction in these metrics, despite the addition of new code, provides strong evidence that the refactoring intervention had the desired causal effect in improving code structure.
- External Validity - External validity refers to the possibility of generalizing the study findings beyond the specific case:
  - Threat: This is a single case study, focusing on a specific piece of software (T-TEA) developed in an academic context. The results may not be directly generalizable to large-scale commercial software projects, or even to other types of games.

  - Countermeasure: The goal of a case study is not statistical generalization, but rather "analytical generalization". The results are theoretically generalizable to other cases with similar characteristics. T-TEA represents a common class of projects (academic/scientific software) that evolved without rigorous engineering practices. Therefore, the lessons learned from applying the OO paradigm and the MVC pattern are highly relevant to other projects in similar contexts that face maintainability issues with legacy procedural code.
- Reliability - Reliability refers to the extent to which the data and analysis depend on the specific researchers:
  - Threat: If the analysis were purely qualitative, researcher bias could be a threat.
  - Countermeasure: The reliability of this study is high. We followed an explicit case study protocol in Section 4. More Importantly, data collection for key outcome metrics was performed using automated tools (SonarQube and Pylint). This ensures that if another researcher ran the same tools on the same codebase (before and after), they would obtain the same quantitative results.

7. Conclusion

The code analyses and modifications applied to the T-TEA software using software engineering techniques represent a significant improvement in the academic approach to developing scientific software, particularly in the field of exergames. The refactoring process applied helped to shrink TD from 1738 minutes to 37 minutes and average CCM from 42 to 13 at the same time that new functions were added and bugs fixed. Pylint, another tool used in the study, also indicated improvements in code quality and the adoption of best practices in the general context of the platform's new architecture. Based on the current data, it is evident that the code is already better than before, especially in terms of maintainability and the CCM per file. It can be concluded that software engineering practices should be integrated throughout the entire software lifecycle, especially reengineering and refactoring, in order to enhance code clarity and support both corrective and evolutionary maintenance over time.

For future work, it is expected that all parts of the codebase[2] will be refactored including the games themselves, tests and continuous integration will be applied throughout the process, and code documentation will be more incremented.

Acknowledgments

The authors thank Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) – Brazil for the financial support (Finance Code 001); the National Council for Scientific and Technological Development (CNPq-Brazil) for the undergraduate research and productivity grant (Process No. 306613/2022-0); to Grupo de Automação de Sistemas e Robótica (GASR) and to the Laboratory for Research on Visual Applications (LARVA).

[2]The code developed to the project is available at: https://github.com/larvattea

References

American Psychiatric Association (2013). *Diagnostic and Statistical Manual of Mental Disorders*. American Psychiatric Publishing, Arlington, VA, 5th edition.

Arvanitou, E., Nikolaidis, N., Ampatzoglou, A., and Chatzigeorgiou, A. (2022). Practitioners' perspective on practices for preventing technical debt accumulation in scientific software development. In *Proceedings of the 17th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE*, pages 282–291. INSTICC, SciTePress.

Avgeriou, P., Kruchten, P., Ozkaya, I., and Seaman, C. (2016). Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162). *Dagstuhl Reports*, 6(4):110–138.

Avgeriou, P. C., Taibi, D., Ampatzoglou, A., Arcelli Fontana, F., Besker, T., Chatzigeorgiou, A., Lenarduzzi, V., Martini, A., Moschou, A., Pigazzini, I., Saarimaki, N., Sas, D. D., de Toledo, S. S., and Tsintzira, A. A. (2021). An overview and comparison of technical debt measurement tools. *IEEE Software*, 38(3):61–71.

Borowa, K., Zalewski, A., and Saczko, A. (2021). Living with technical debt—a perspective from the video game industry. *IEEE Software*, 38(6):65–70.

Chavan, S. B. and Mondal, S. (2024). Do large language models recognize python identifier swaps in their generated code? In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE 2024, page 663–664, New York, NY, USA. Association for Computing Machinery.

Clements, P. and Shaw, M. (2009). "The Golden Age of Software Architecture" Revisited. *IEEE Software*, 26(4):70–72.

Conradt, J., Eckert, T., Caserman, P., Schaub, M., Bruder, R., and Göbel, S. (2021). Towards a quality label for educational games and serious games. In *European Conference on Games Based Learning*, pages 151–159, Reading, UK. Academic Conferences International Limited, Academic Conferences International Limited.

Dasgupta, S. and Hooshangi, S. (2017). Code quality: Examining the efficacy of automated tools. In *Proceedings of the 23rd Americas Conference on Information Systems (AMCIS)*, pages 2422–2426, Boston, MA, USA. Association for Information Systems (AIS).

Gall, H. and Klosch, R. (1995). Finding objects in procedural programs: an alternative approach. In *Proceedings of 2nd Working Conference on Reverse Engineering*, pages 208–216.

Hassan, H. B., Sarhan, Q. I., and Beszédes, Á. (2024). Evaluating python static code analysis tools using fair principles. *IEEE Access*, 12:173647–173659.

Jacobson, I. and Lindström, F. (1991). Reengineering of old systems to an object-oriented architecture. In *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '91, page 340–350, New York, NY, USA. Association for Computing Machinery.

Kleinwaks, H., Batchelor, A., and Bradley, T. H. (2023). Technical debt in systems engineering—a systematic literature review. *Systems Engineering*, 26(5):675–687.

Lenarduzzi, V., Saarimäki, N., and Taibi, D. (2020). Some sonarqube issues have a significant but small effect on faults and changes. a large-scale empirical study. *Journal of Systems and Software*, 170:110750.

Liu, S.-S. and Wilde, N. (1990). Identifying objects in a conventional procedural language: an example of data design recovery. In *Proceedings. Conference on Software Maintenance 1990*, pages 266–271.

McCabe, T. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320.

Miranda, J. M., Browne, R. A. V., da Silva, W. Q. A., Dos Santos, J. P. R., Campbell, C. S. G., and Ramos, I. A. (2025). Effects of a session of exergames and traditional games on inhibitory control in children with autism spectrum disorder: Randomized controlled crossover trial. *JMIR Serious Games*, 13(1):e65562.

Penteado, R., Masiero, P., Do Prado, A., and Braga, R. (1998). Reengineering of legacy systems based on transformation using the object oriented paradigm. In *Proceedings Fifth Working Conference on Reverse Engineering (Cat. No.98TB100261)*, pages 144–153.

Pereira, G. B., Trindade, A. B., Dickel, M. R. B., and Hounsell, M. d. S. (2023). Jogo sério para estímulo sensorial de crianças com transtorno do espectro autista. In *Simpósio Brasileiro de Informática na Educação (SBIE)*, pages 572–583, Porto Alegre, RS, Brazil. SBC, Sociedade Brasileira de Computação.

Runeson, P. and Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical Softw. Engg.*, 14(2):131–164.

Sianturi, R. A., Jawak, P. K. I., and Pangibulan, W. H. (2024). Reengineering of markopi mobile application. In *2024 International Conference on Information Technology Systems and Innovation (ICITSI)*, pages 426–432.

Singh, J., Dhindsa, K. S., and Singh, J. (2021). Software quality improvement and validation using reengineering. *Journal of Engineering Research*, 9(4, Part A):59–73.

Sommerville, I. (2011). *Software Engineering*. International Computer Science Series. Pearson, 9th edition.

SonarSource SA (2026). Sonarqube. Available at: `https://www.sonarsource.com/products/sonarqube`. Accessed: April 23, 2026.

Syromiatnikov, A. and Weyns, D. (2014). A journey through the land of model-view-design patterns. In *2014 IEEE/IFIP Conference on Software Architecture*, pages 21–30.

Trindade, A. B., Pereira, G. B., and Hounsell, M. d. S. (2022a). Chão interativo e jogos sérios ativos para autistas: A plataforma t-tea e o jogo repetea. In *Simpósio Brasileiro de Jogos e Entretenimento Digital (SBGames)*, pages 512–521, Porto Alegre, RS, Brasil. SBC, Sociedade Brasileira de Computação.

Trindade, A. B., Pereira, G. B., and Hounsell, M. d. S. (2022b). Requisitos para jogos sérios ativos em chão interativo para o público autista. In *Simpósio Brasileiro de Jogos e Entretenimento Digital (SBGames)*, pages 542–551, Porto Alegre, RS, Brasil. SBC, Sociedade Brasileira de Computação.

TTEA (2025). Jogo 3: Vestea. Available at: https://udescmove2learn.wordpress.com/2023/06/26/t-tea. Accessed: April 23, 2026.

van Rossum, G. and Warsaw, B. (2010). *Style Guide for Python*, pages 283–297. Apress, Berkeley, CA.

Watson, A. H., McCabe, T. J., and Wallace, D. R. (1996). Structured testing: A testing methodology using the cyclomatic complexity metric. *NIST Special Publication*, 500:235.

Yu, P., Wu, Y., Peng, J., Zhang, J., and Xie, P. (2023). Towards understanding fixes of sonarqube static analysis violations: A large-scale empirical study. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 569–580.

Yu, S. and Zhou, S. (2010). A survey on metric of software complexity. In *2010 2nd IEEE International Conference on Information Management and Engineering*, pages 352–356.

Zhang, C. and Budgen, D. (2012). What do we know about the effectiveness of software design patterns? *IEEE Trans. Softw. Eng.*, 38(5):1213–1231.