Design Patterns and Code Maintainability in Games: A Case Study

Vitor Braga Estevam[1], **Alysson Diniz dos Santos**[1], **Matheus Paixao**[2]

[1]Instituto Universidade Virtual – Universidade Federal do Ceará (UFC)

[2]Universidade Estadual do Ceará (UECE)

Fortaleza – CE – Brasil

vitorestevam02@gmail.com,alysson@virtual.ufc.br,matheus.paixao@uece.br

Abstract. *Design patterns are a common strategy employed in software development to promote several aspects of software quality, including code maintainability. Game development is a multidisciplinary field that includes not only software development but also animations, interactivity, sound design and others. Given the short deadlines in which game developers commonly operate, on top of the intrinsic complexity of the field itself, code maintainability is often overlooked by game developers. Hence, this work investigates how design patterns can impact the code maintainability of games. First, we performed a review to understand how (and which) design patterns can be applied to different game mechanics. Next, we selected two small games and refactored the code by applying design patterns to some of the games' mechanics. By leveraging static analysis to assess the games' code maintainability before and after the application of the design patterns, we observed an overall code maintainability improvement of 5.5%. The preliminary results indicate the potential to use design patterns as a strategy to improve code maintainability in the context of game development.*

1. Introduction

Object-oriented design patterns (DPs) have been originally proposed by the Gang-of-Four (GoF) [Gamma et al. 1995], and they provide reusable structures for solving common software development problems. DPs are extensively adopted in the development of software systems, and their usage has been shown to positively affect software quality [Wedyan 2020]. One of the software quality attributes that has been associated with the use of DPs is code maintainability [Ampatzoglou et al. 2013], which refers to the degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers [IEEE/ISO 2011].

The development of a game embraces particularities that highlight the need for code maintainability. Indeed, game project management is a complicated task that diverges from traditional software project management [Ampatzoglou and Stamelos 2010], due to the fact that games deal with multiple interweaving areas, such as: interactivity, animation, audio, among others [Kanode and Haddad 2009]. In addition, the creation of a profitable and competitive game may require many hours and thousands of lines of code,

introducing significant complexity. Consequently, game developers must adopt specific software engineering practices to improve the structural quality of the game's assets, including its code [Paschali et al. 2021].

DPs have already been extensively studied in various domains of software development (for a literature review, we refer to [Wedyan 2020]). However, in the context of game development, the subject has been underexplored. Recent work has investigated some of benefits that DPs may achieve in the development of digital games [Nystrom 2014, Figueiredo 2015, Barakat 2019, Paschali et al. 2021]. In spite of such efforts, to the best of our knowledge, there has not been a study that investigated how the implementation of game mechanics with DPs impact code maintainability.

Therefore, the objective of this work is to investigate how the use of DPs to implement game mechanics can impact code maintainability. To achieve this goal, we tackled the following specific objectives: (i) map how (and which) DPs can be used to implement game mechanics; (ii) refactor two open source games considering the application of the chosen DPs to some of the games' mechanics; and (iii) measure the refactorings' impact on maintainability through automated static analysis.

2. Background and Related Work

Game mechanics are the various actions, behaviors and control mechanisms afforded to a player within a game context [Hunicke et al. 2004]. As game mechanics are often reoccurring among different games, recent studies [Nystrom 2014, Figueiredo 2015, Paschali et al. 2021] attempt to relate game mechanics to DPs, in order to introduce template instantiations of game mechanics (for a summary, see Table 1).

The book Game Programming Patterns [Nystrom 2014] reviews the GoF design patterns in the context of games. The author highlights a subset of the Gof DPs as having the most potential in game development: Observer, Command, State, Prototype and Flyweight. In a different work, [Figueiredo 2015] relates DPs to different game mechanics. After an experiment with six teams, the results indicate the teams instructed about DPs produced code in less time and with better quality. In a recent study, [Paschali et al. 2021] presents "GAME-DP Repository", an online repository of mappings between DPs and game mechanics. However, the repository is no longer available for consultation.

3. Case Study

To investigate the impact in code maintainability caused by the usage of design patterns, we conducted a case study composed of three steps: (i) selecting and analysing games to serve as the study's subjects; (ii) refactoring some of the games' mechanics to use DPs; and (iii) employing static code analysis in the games' versions before and after refactoring to compare code maintainability.

3.1. Games Selection and Analysis

We selected two small and open-source Unity games. Game 1, *2d Roguelike*[1], is a survivor turn-based game in which the player must avoid enemies and collect resources while advancing the levels. Game 2, *Tanks*[2], is a two player competitive game in which the

[1]https://assetstore.unity.com/packages/templates/tutorials/2d-roguelike-29825

[2]https://assetstore.unity.com/packages/essentials/tutorial-projects/tanks-tutorial-46209

Table 1. Mapping between Design Patterns (DP) and Game Mechanics

Mechanic	Pattern	Game Mechanic Definition and DP relationship
Achievements and Quests	Observer	Setting and tracking player goals [Nystrom 2014]
UI Elements	Observer	Synchronizing game instances and interface components [Barakat 2019, Paschali et al. 2021]
Entities Control	Command	Encapsulating, storing and manipulating entities' executed actions [Nystrom 2014]
Undo Actions	Command	Storing and Reversing the executed actions [Nystrom 2014]
Character Animations	State	Managing and executing the animation of characters and other game instances [Barakat 2019]
Behavior Variation	State	Varying the behavior of entities according to their state [Nystrom 2014, Barakat 2019, Figueiredo 2015]
Entities Replication	Prototype	Spawning new game instances [Nystrom 2014, Barakat 2019, Figueiredo 2015, Paschali et al. 2021]
Resources Repetition	Flyweight	Sharing computing resources between game instances [Nystrom 2014, Figueiredo 2015, Paschali et al. 2021]

players control tanks that can move and fire, looking to defeat the opponent. The code of both games was analyzed to identify the presence of the game mechanics defined in Table 1. The analyses indicated:

- **Game 1**: UI Elements, Entities Control, Entities Replication and Resources Repetition;
- **Game 2**: UI Elements, Entities Control and Entities Replication

3.2. Mechanics Refactoring

In both games, the UI Elements and Entities Control mechanics were implemented without a proper separation of concerns, where the code for both mechanics was entangled. Hence, we refactored both mechanics to use the Observer and Command patterns, respectively. As for the Entities Replication mechanic, we noticed that the Prototype pattern was already employed as part of Unity's instantiation function. Thus, nothing had to be done regarding this mechanic. Finally, for Game 1 only, we refactored the Resource Repetition mechanic to use static attributes as a way to implement the Flyweight pattern.

All the refactorings consisted in structural changes to the games' classes, where no behavior to the games was modified. This was validated by functional tests before and after the refactorings. A summary of the changes to the games' code is presented in Table 2. The refactored codes are available on Github[3,4] for consultation and replication.

[3]https://github.com/VitorEstevam/2d-roguelike
[4]https://github.com/VitorEstevam/tanks

Table 2. Summary of changes to games' code caused by the refactorings[5]

	Game 1	Game 2
Code lines before refactoring	934	922
Changed Files	22	17
Line Additions	358	241
Line Deletions	278	58

3.3. Measuring Maintainability

To measure the code maintainability before and after the refactorings, we employed the Microsoft Code Analysis tool [Jones 2022]. The maintainability metrics generated by the tool are: maintainability index, number of source code lines, inheritance depth, and class coupling.

The *maintainability index* is value between 0 and 100 that represents the relative ease of maintaining the code [Jones 2022] in which higher values mean better maintainability. This metric is calculated in terms of *number of source code lines* and *cyclomatic complexity*. The cyclomatic complexity measures the amount of decision logic in a source code function. Higher cyclomatic complexity means lower maintainability index. In turn, the *inheritance depth* represents the number of classes that inherit attributes from other classes. Finally, *class coupling* refers to the interdependence between classes in the code.

Table 3 depicts the values for each of the maintainability metrics for the original and refactored code.

Table 3. Maintainability metrics for the original and refactored subject games

Metric	**Game 1**		**Game 2**	
	Original	**Refactored**	**Original**	**Refactored**
Maintainability Index	76	81	75	81
Number of Source Code Lines	934	1039	922	1010
Inheritance Depth	6	6	5	5
Class Coupling	44	50	38	44

4. Discussion

Table 3 indicates the maintainability index improved in 5% (76 to 81) and 6% (75 to 81), for Games 1 and 2, respectively. In addition, the inheritance depth remained the same in spite of the new classes being added to the refactored code. On the other hand, class

[5]The Changed Files, Line Additions and Line Deletions were taken from GitHub's comparison tool

coupling increased for both games. We argue that this is due to the Command pattern, which introduces shared responsibilities between classes, as detailed in Figure 1.

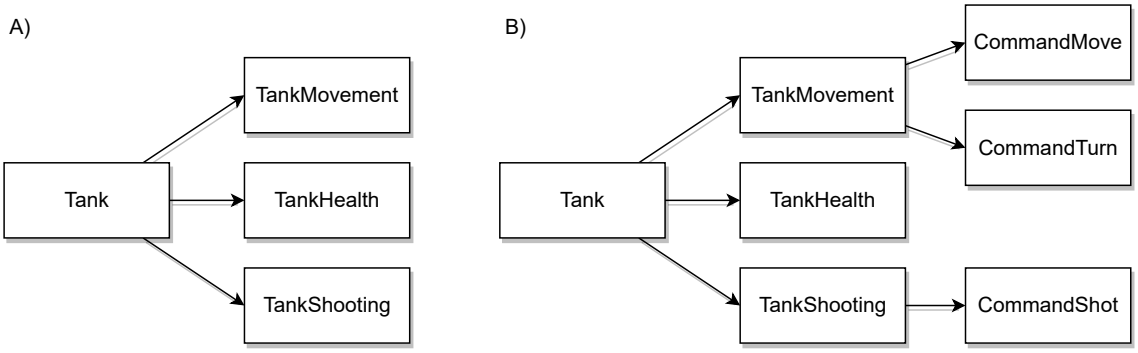

Figure 1. Simplified diagram for the class coupling of game 2's Tank class. A) Original and B) Refactored

The impact on the maintainability metrics caused by the Command pattern is an interesting case. On the one hand, it improved the maintainability index, but on the other hand, it also increased coupling, which may be considered detrimental to code maintainability. However, the application of the Command pattern allows for a better separation of concerns, which is depicted in the diagram in Figure 1. As pointed out in the literature [Borrelli et al. 2020], a clear separation of concerns contributes for a better overall maintainability in spite of the coupling increase.

5. Conclusion

This work investigated the impact of design patterns in the code maintainability of games. Even though this was a preliminary study, we believe the work showcased two important contributions. First, based on previous literature, we mapped which design patterns are more adequate to be used in the implementation of certain game mechanics. Second, through a case study with two small open-source games, we showed how the application of design patterns have the potential to improve the games' maintainability. We believe these initial results contribute to the game maintainability literature.

It is important to highlight that the scope of the experiment was limited, including only two small-sized games in which similar mechanics were refactored to use the same design patterns. An important limitation of the experiment is the fact that all the refactorings were performed by a single developer, which may bias the implementation. Nevertheless, the games' features were validated through functional tests.

Because of the limited scope of the experiment, we believe that for future work, the mapping between design patterns and game mechanics could be expanded with developer interviews. Furthermore, by reusing our methodology, a larger-scale empirical study could be conducted to enhance the observations we reported, by using more games and leveraging developers with different levels of experience for the refactorings.

References

[Ampatzoglou et al. 2013] Ampatzoglou, A., Charalampidou, S., and Stamelos, I. (2013). Research state of the art on gof design patterns: A mapping study. *Journal of Systems and Software*, 86:1945–1964.

[Ampatzoglou and Stamelos 2010] Ampatzoglou, A. and Stamelos, I. (2010). Software engineering research for computer games: A systematic review. *Information and Software Technology*, 52(9):888–901.

[Barakat 2019] Barakat, N. (2019). A framework for integrating software design patterns with game design framework. *Faculty of Informatics and Computer science,*.

[Borrelli et al. 2020] Borrelli, A., Nardone, V., Di Lucca, G. A., Canfora, G., and Di Penta, M. (2020). *Detecting Video Game-Specific Bad Smells in Unity Projects*, page 198–208. Association for Computing Machinery, New York NY USA.

[Figueiredo 2015] Figueiredo, R. (2015). Gof design patterns applied to the development of digital games. In *Proceedings of SBGames 2015*, Teresina PI Brazil. XIV SBGames.

[Gamma et al. 1995] Gamma, E., Helm, R., Johnson, R., Johnson, R. E., Vlissides, J., et al. (1995). *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH.

[Hunicke et al. 2004] Hunicke, R., LeBlanc, M., Zubek, R., et al. (2004). Mda: A formal approach to game design and game research. In *Proceedings of the AAAI Workshop on Challenges in Game AI*, volume 4, page 1722. San Jose, CA.

[IEEE/ISO 2011] IEEE/ISO (2011). *Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*. IEEE Computer Society.

[Jones 2022] Jones, M. (2022). Code metrics values. https://shorturl.at/dkowE.

[Kanode and Haddad 2009] Kanode, C. M. and Haddad, H. M. (2009). Software engineering challenges in game development. In *2009 Sixth International Conference on Information Technology: New Generations*, pages 260–265. IEEE.

[Nystrom 2014] Nystrom, R. (2014). *Game Programming Patterns*. Genever Benning.

[Paschali et al. 2021] Paschali, M.-E., Volioti, C., Ampatzoglou, A., Gkagkas, A., Stamelos, I., and Chatzigeorgiou, A. (2021). Implementing game requirements using design patterns. *Journal of Software: Evolution and Process*, 33(12):e2399.

[Wedyan 2020] Wedyan, F. (2020). Impact of design patterns on software quality: a systematic literature review. *IET Software*, 14(1):1–17.