Implementing Game Requirements using Design Patterns

Maria-Eleni Paschali[1], Christina Volioti[2], Apostolos Ampatzoglou[2], Anastasios Gkagkas[2], Ioannis Stamelos[1], and Alexander Chatzigeorgiou[2]

[1] Department of Informatics, Aristotle University of Thessaloniki, Greece

[2] Department of Applied Informatics, University of Macedonia, Greece

mpaschali@csd.auth.gr, chvolioti@gmail.com, a.ampatzoglou@uom.edu.gr, stamelos@csd.auth.gr, achat@uom.edu.gr

Game mechanics are high-level descriptions of common game requirements; however, they do not provide any guidance on their code implementation. Nevertheless, their implementation involves high volumes of essential complexity, which in turn leads to the introduction of accidental complexity (long methods, code repetition, etc.). A possible solution to this problem is to map game mechanics to design patterns, in order to provide template instantiations that handle structural complexity. The aim of this study is two-fold: (a) introduce template instantiations of game mechanics with GoF patterns; and (b) evaluate such mappings in terms of extendibility and reusability. To achieve these objectives, we developed an online repository of mappings between GoF patterns and game mechanics; and conducted an experiment to explore the benefits of the mapping. The results of the study suggest that the implementation of game mechanics with GoF patterns is beneficial, since the time to extent the current implementation or reuse code chunks is not prolonged, whereas the implementations are less faulty. Finally, the suggested mapping will equip game mechanics with sample implementations (that adhere to good design principles—ensuring extendibility) that can be reused, acting as a starting point for source code development.

1. Introduction

According to game industry statistics[1], during the last decades, the game industry has reached revenues of more than 90 billion dollars; and at the same time has attracted a significant attention from academia. One of the most striking characteristics of the game development industry is the very tight deadlines in terms of release of new versions (for series of titles), or patches / updates / extensions (for games provided as a service) due to market competition [1][2]. Such a time pressure usually leads to poor game design and reduced maintainability [3]. Nevertheless, ease of maintenance is an important factor for game success: the creation of a game that could be profitable and competitive requires many hours and thousands of code lines, introducing significant amounts of required complexity. Consequently, **game developers must adopt specific software engineering practices to improve** the structural quality of the game, placing special emphasis on *extendibility and reusability* [3][4] (*Need-A*).

By considering that maintainability cannot be uniformly improved in the complete game (it would be too expensive [5]), software engineers must identify the most fitting spots of the design to apply best software engineering practices (e.g., refactorings, patterns etc.). Such spots can be identified by exploiting game mechanics. Game mechanics are common requirements that exist in multiple games, and provide a common ground for communication among development stakeholders [6]. **Game mechanics abstractly** present high-level requirements that are re-occurring interactions in game scenarios and gameplay: for instance, the *Agents* mechanic suggests that *game entities which take the role of players must be controlled (through advanced AI) by the game system*. Despite their popularity among practitioners, game mechanics usually are not accompanied by any **guidance**, on how this requirement will

[1] http://www.gamesindustry.biz/articles/2015-04-22-gaming-will-hit-usd91-5-billion-this-year-newzoo

be **designed or implemented** *(Need-B)*. We note that having sample implementations is of paramount importance for more complex game mechanics (e.g., *Agents*) that impose a "heavy" game logic, compared to more trivial ones (e.g., *Score*—updating a score every time that the game state changes).

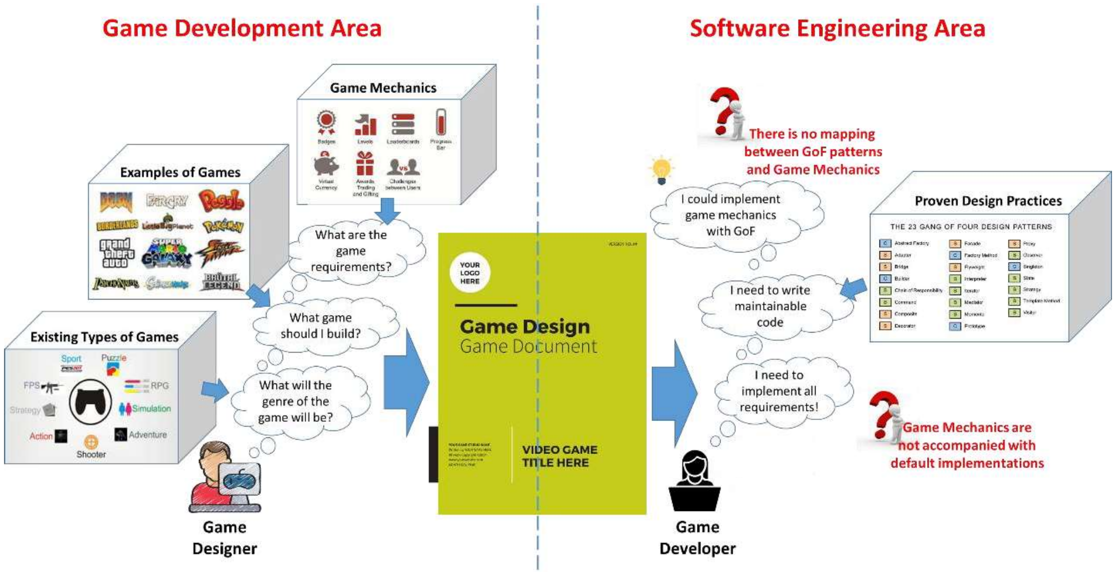


Figure 1: Motivation Overview

The aforementioned needs are visualized in Figure 1. It is worth mentioning that game development involves cross-functional and inter-disciplinary teams consisting of game designers, game / software developers, graphical artists, audio engineers, educators, and many others [4][7]. Additionally, in the last years the game development software engineering process: (a) adopts more agile practices, since the players are involved in the development stages through play-testing; and (b) is iterative as well as on-going, in the sense that community feedback plays a crucial role in games' improvement and assessment [8][7]. We note that in Figure 1, we simplify the process (for illustration purposes) and present an isolated view of the cross-functional and interdisciplinary teamwork, in a waterfall-like manner (despite the more agile, but complex practices that are used in practice). So, suppose a game designer that first decides the type of game that is going to be developed (***game genre***)—e.g., *Sports Game*; next he/she needs to explore existing successful games of this genre (e.g., *FIFA©, PRO Evolution Soccer©, etc.*) to "*borrow*" ideas, tentative requirements, and good practices. Proven game design practices can also be found in the literature or the web (***game mechanics***)—e.g., the *Score game mechanic* must be applied in any *Soccer game*. Based on this search, and his/her own ideas on the game, he/she develops the *Game Design Document (GDD)*, which is the end artifact of the conceptual phase of game design; and the input of the actual development process. However, the GDD is a living document and needs a recursive process of documentation [8] as it is not always an easy task the translation of game mechanics to functional requirements. Given the GDD, the software developer, compiles a list of requirements that need to be implemented. By considering that game mechanics are recurring, the game developer would highly benefit in terms of effort, by identifying default implementations for them, which he/she would tailor to the needs of the specific game, instead of "*re-inventing the wheel*" in every game mechanic instantiation. Given the need for continuous changes in games, the developer should seek proven practices (such as ***design patterns*** [9]) for his/her design and source code implementations. Thus, a promising solution to this issue is to *provide default implementations for game mechanics, using design patterns* to reduce development effort through reuse and improve the games' maintainability.

Based on the above, in this study we propose the instantiation of game mechanics through GoF (Gang of Four) design patterns, so as to: (a) provide a template instantiation mechanism for common game requirements—promoting design reuse [9]; (b) encourage the use of proven, extendibility-beneficial solutions to common design problems [10]; and (c) solve the shortcoming of game mechanics in terms of abstractness [6]. Despite the fact that GoF design patterns have already been discussed as a solution to the design problems faced in game development [11][12][13][14][15], their applicability in practice has been demonstrated only through either small-scale examples for teaching, or on individual games (see Sections 2.2 and 2.3). A possible link between game mechanics and GoF patterns can boost their applicability, in the sense that game mechanics are documented recurring solutions that have emerged from real-life games. Therefore, if a fraction of game mechanics (in the place that they exist) are implemented with GoF design patterns, the number of GoF design pattern instances in games is expected to increase.

To this end, we introduce the GAME-DP (Game Mechanics Instantiation with Design Patterns) repository. The elements of the repository are pairs of patterns: a game mechanic and a GoF design pattern, providing the full documentation of the mapping and examples of how it can be implemented in practice. The aforementioned approach can be useful in the game industry in two ways: developing game frameworks or game engines (e.g., implementing mechanics, such as *Game World* [6]) and developing specific game titles, either from scratch or by using a game engine. Since we acknowledge that in the current state the development of games through engines is the most common practice, and that when using engines developers use already pre-built components or prefabs, in this paper we present a detailed illustration (see Section 4.3) on how GAME-DP elements fit their use, through a real-world scenario.

To empirically evaluate GAME-DP, in this paper we have conducted an ***experiment***, to understand if the instantiation of game mechanics through GoF patterns is beneficial in terms of: (a) software extendibility and (b) design reusability. The results suggested that indeed GoF design patterns can be mapped to game mechanics, and that they are perceived as useful by software developers. The rest of the paper is organized as follows: in Section 2, the necessary background information is introduced, i.e., background on game mechanics, the use of design patterns in game development and in education, whereas in Section 3 the related work, i.e., papers on the mapping of different types of patterns. In Section 4 we present the GAME-DP approach: (a) the methodology for mapping game mechanics and patterns, and (b) examples of such mappings; and an illustrative example in a real-world OSS game, built with Unity. In Section 5, we present the design for our experimental validation, whose results are presented in Section 6. In Section 7 we provide the discussion of the results, and in Section 8 the threats to validity. Finally, in Section 9, we conclude the paper.

2. Background Information

2.1 Game Mechanics

Game mechanics have been introduced by Bjork et al. [6] as a set of patterns that act as descriptions (employing a unified vocabulary) of re-occurring interactions in game scenarios and gameplay. According to Adams and Dormans [16] game mechanics could be described as a non-typical language that facilitates the communication between team members (e.g., artists, developers, etc.) [6]. LeBlanc [17] proposed the MDA (Mechanics/Dynamics/Aesthetics) framework which is a layered model that bridges the gap between the relationship of rules and player’s experience. Callele et al. [18] captured game mechanics and their associated experience requirements, which were treated as functional requirements. Daneva [8] investigated gameplay requirements, which were also perceived as functional requirements. Additionally, Bjork and Holopainen [6] supported that “the way to recognize patterns is playing games, thinking games, dreaming games, designing games and reading about games”. Therefore, the proposed patterns are based on Bjork's concept [6] and are collected from interviewing professional game programmers, analyzing existing games and transforming game mechanics. An example of such a pattern is the *Paper – Rock – Scissors pattern* that is well known in game design community (also as *triangularity*). This pattern is used when there are three discrete states (or options for a player) and option `A` defeats option `B`, option `B` defeats option `C` and option `C` defeats option `A`. Game mechanics are usually documented in the game design document, which in many cases is the only document available in the game development lifecycle [6]. Other approaches have been also developed to document game mechanics such as the Game Ontology Project [19] that utilizes a hierarchical structure to describe elements of games from players' perspective, and The 400 Project [20] that presents a list of practical rules of thumb that governed game design from professional game designers' perspective.

Lately, game mechanics have been applied in research efforts in various application domains. As serious games gain more and more interest, lately researchers have introduced the meaning of Serious Game Mechanic (SGM) which incorporates components such as methods, activities, tasks, objectives for learning aspects. For instance, Arnab et al. [21] introduced SGMs that are linked with pedagogical practices: by promoting mechanics of learning to connect game mechanics directly a player's actions. In another example, game mechanics have been explored by Schmitz et al. [22] as a way to achieve the learning outcomes of educational games.

2.2 GoF Design Patterns in Game Development

Despite the fact that object-oriented design patterns (DP) are designed to provide a common structure for solving common software development problems [9], the use of a software design pattern is still at low rate in video games. Until the middle of 90s, game developers did not aim to produce reusable code since every program has been written from scratch in assembly language [23]. Later, reusable coding has proven to be one of the most important issues in game development because games became far more complex and their production process more time consuming. To alleviate this problem, frameworks and game engines have been created. A framework is a collection of classes that can be widely reused and integrated with other components [24][14]. Usually, they implement mechanisms that occur in many games, such as input handling, file handling (texture, models, audio etc.), 3D rendering etc. Game engines are programs that provide developers the potential to design game levels, handle player and opposition behavior, by using scripting languages and powerful GUIs. As it is easily understood, if frameworks and game engines are "well-structured", they can be maintained without extreme effort and be transformed so that they can fit as many game genres as possible. To achieve this expectation, game developers should use software engineering techniques and methodologies. The evaluation of object-oriented design pattern application in computer games has been investigated by Ampatzoglou and Chatzigeorgiou [24] and Nguyen et al. [15] analyzed existing systems and examined how the application of patterns affects the game structure and maintainability. The analysis indicated that patterns improve complexity and coupling of the game and cohesion of the code, although affecting the project's size by increasing its lines of code. In addition to that, Nguyen et al. [15] attempted to create a game that was based on patterns. The results suggested that design patterns should be considered an efficient way of properly achieve abstractions and decoupling in games. Regarding game engines, Polančec and Mekterović [25] developed a MOBA (Multiplayer Online Battle Arena) game using patterns in Unity, while Bucher [26] introduced a set of patterns also in Unity to reduce code complexity and allow scalability.

2.3 Design Patterns and Games in Education

Design patterns play an important role in teaching and learning software design, that is quite challenging since students not only have to understand the theory, but also to use the patterns effectively [27].

The added value of teaching design patterns in students (novice developers) is that design patterns encapsulate the knowledge and experience of experts, and therefore they "help novice perform more like an expert" by reducing the learning time [28]. One way to engage students in the teaching of design patterns is to apply them in game development, which is very popular among the youth. To this end, in this section, we present studies that are on the intersection of patterns and games, in the context of education. Da Cruz Silva et al. [27] demonstrated an exemplar for teaching design patterns using a well-known game, namely: Angry Birds. Results were positive, proving that the exemplar was useful and could be promising for teaching design patterns. Additionally, Gestwicki and Sun [29] suggested a teaching methodology for design patterns through a game. Although their sample was small, students understood the problem firstly and then applied the corresponding design pattern. Gómez-Martín et al. [30] used a family of abstract strategy games to teach design patterns. The feedback was valuable as the approach helped students to gain a close to reality experience. However, several studies deal with the difficulties of teaching design patterns. For instance, Pillay [31] highlighted the learning difficulties faced by 22 undergraduate students in learning design patterns; such as "identifying which design pattern to use". Ghazarian [32] also mentioned that students misused design patterns, reporting this observation in a preliminary work.

2.4 Effect of GoF Design Patterns on Quality

In this section, we present a sample of studies that explore the relation of GoF design patterns to quality attributes. The specific research direction is vast, and in that sense only a limited number of studies can be presented, giving priority to studies that explore the quality attributes of interest: i.e., maintainability, correctness, and reusability; as well as, the employed research method: i.e., using an experimental setup.

Vokac et al. [48] and Prechelt et al. [10], have conducted controlled experiments concerning the maintainability of systems with and without design patterns. Prechelt et al. [10] considered Abstract Factory, Observer, Decorator, Composite and Visitor, while the subjects of the experiment have been professional software engineers. The results of the experiment suggest that it is usually useful to apply a design pattern rather than the simpler solution. The dilemma of employing a design pattern or a simple solution is best answered by the software engineer's common sense. The experiment by Vokac et al. [48] is a replication of the aforementioned experiment and therefore uses the same patterns and similar subject groups. In addition to the former experiment, the authors increased experimental realism in the sense that subjects have used a real programming environment instead of pen and paper. The results suggest that design patterns are not universally beneficial or harmful with respect to maintenance, since each design pattern has its own nature. In the discussion of the paper conclusions concerning each design pattern are solidly presented.

Ampatzoglou et al. [49] investigated the correlation between 12 design patterns and correctness. For that, they performed a case study involving 94 software projects in the game application domain. In this study, information was collected regarding bug tracking and pattern instances from each version of every software. During the analysis, each pattern was analyzed separately in order to identify correlations between the number of defects and pattern instances. The results of the study suggest that specific design patterns are related to higher defect frequency, although the presence of pattern occurrences (without examining each pattern separately) seems not to be correlated with such a frequency.

Gatrell and Counsell [46] investigated the effect of 11 design patterns on correctness by analyzing a commercial project written in C# (with ~266KLOC). For that, pattern-participating (PP) classes were manually collected and compared against non-pattern-participating (NPP) classes over a two-year period, correlating them with the fault history provided by the source control system, aiming at finding fault-prone classes. The results of the study suggest that PP classes are more fault-prone than NPP classes, as well as that this is related to both a higher number and the size of changes in NPP classes. Additionally, the authors characterized Adapter, Template Method and Singleton as the most fault-prone patterns.

Finally, Aversano et al. [47] investigated the relationship between correctness of pattern participants and the scattering degree of concerns that communicate with them. For that, occurrences of 12 design patterns were extracted from several snapshots of three open-source projects, and the correctness was measured in terms of code defects. The results of this study suggest that patterns that induce crosscutting concerns (i.e., implemented across several classes spread along the system) are correlated to a higher number of defects in their participants.

3. Related Work

This section, discusses related work to this paper, i.e., other studies that propose the implementation of game mechanics with GoF design patterns. First, Kounoukla et al. [33] introduced nine (9) basic instantiations of game mechanics with GoF patterns. In particular, they implemented `Turn-based Games` with `Template Method`, `Agents` with `Strategy`, `Power-Ups` with `Visitor`, `Game World` with `Composite`, `Levels` with `State`, `Progress Indicators` with `Observer`, `Units` with `Abstract Factory`, `Movement` with `Strategy`, and `Varied Gameplay` with `State`. Since the current study is an extension of the study by Kounoukla et al. [33] we reuse all these sample instantiations as our demonstrations in Section 4. We note that in the original study [33], we had only described the mappings theoretically (without code examples) and we have not assessed their effect on extendibility and reusability, since we have identified only three (3) instances in open-source games.

In a parallel research attempt, Qu et al. have performed two relevant studies [34][35]. In the first one, they demonstrate how the `Game State Management` mechanic could be implemented with `State`, whereas in the second the implementation of `Fast Enemies Clone` with `Prototype`, `Enemy State and Behavior` with `Strategy`, `Weapon Management` with `Decorator`, and `Creative Control` through `Builder`. Another research work implemented game mechanics based on existing game design patterns for cooperative video games [36]. Specifically, they implemented the `Complementarity` game design pattern with `Additive Abilities, Shared Goals` with `Shared Final Outcome, Shared Puzzles` with both `Synchronized Tasks` and `Sequential Tasks` and finally `Shared Objects` with `Shared Space`. By extending the related work, outside the domain of computer games and game mechanics, one can identify white papers and academic literature discussing how architectural patterns can be implemented with GoF design patterns. For instance, Bashiiui [37] discusses how the `Pipes and Filters` architectural pattern can be implemented through the `Decorator`. However, since such studies are considered indirect related work, they are not reported in this section.

Based on the above, this paper advances the state-of-the-art, in several ways, since:

- it provides GAME-DP, i.e., a novel repository of mappings between GoF design patterns and game mechanics, which does not exist in the current SoTA;
- GAME-DP apart from the mappings that are identified in the literature [33][34][35], includes instances that have been identified in OSS games (see right side of Table 1);
- in contrast to Kounoukla et al. [33] and Qu et al. [34][35] it provides a full-fledged empirical validation on the proposed mappings;
- compared to studies presented in Section 2.2, it focuses on the application of GoF patterns on implementing game mechanics, and not random spot of the game source code;
- compared to studies presented in Section 2.3, the paper is not exploring the application of GoF in games for learning purposes, but in real-world development and reuse tasks; and

- compared to studies presented in Section 2.4, the paper presents the effect of patterns in development and reuse tasks, in the context of game development.

Therefore, this paper is the first study, in the context of game development; which focuses on the mapping between game mechanics and GoF design patterns, evaluating the efficiency of using these mappings when performing reuse and development tasks (in the form of a full-fledged empirical study).

4. Game Mechanics Implementation with Design Patterns: The GAME-DP Repository

The research method used for developing the proposed solution is design science [38], and in particular an engineering cycle. According to Wieringa [38] to solve a practical problem the following steps need to be performed:

- ***Problem investigation***: The problem that we explore has been extensively introduced in Section 1, see (*Need-A)* and (*Need-B*). As main stakeholders we have identified game developers and designers; whereas as effects of the phenomenon (lack of guidance on how to implement game mechanics) the poor extensibility and reusability of the software.
- ***Treatment design***: The proposed treatment is the linking of game mechanics to GoF patterns, which by definition treat reusability and extendibility. To explore the tentative benefits of the proposed treatment, we have developed several mappings between game mechanics and GoF patterns.
- ***Treatment validation***: To validate the efficiency of the treatment we have performed a controlled experiment as presented in Sections 5 and 6.
- ***Treatment implementation***: To implement the proposed treatment, we have developed the Game Mechanics Implementation with Design Patterns (GAME-DP) repository[2], which has been integrated in the Design Pattern Repository (DePRE) of EXTREME[3] (previously known as Percerons). EXTREME is a software engineering platform [39] created by one of the authors of this paper, with the aim of facilitating empirical research in software engineering, by providing: (a) indications of componentizable parts of source code [40], (b) quality assessment [41], and (c) design pattern instances [39]. The Design Pattern Repository includes instances of GoF design patterns from approximately 600 open-source projects (~150 of which are games—accounting to more than 4,500 pattern instances identified by SSA pattern detector [42]); and through this research endeavor it now contains links of GoF design patterns to game mechanics.

For the rest of this section, we present GAME-DP and an illustrative application of a GAME-DP element in a Chess open-source game that used the Unity game engine.

4.1 Repository Schema & Data Description

The success of the GAME-DP repository depends on the ease with which game developers can satisfy their needs, outlined as follows: (a) identify the requirements of interest and the relevant game mechanic; (b) understand the default implementation of the mechanic; and (c) exploit the GoF pattern underlying structure for improving the extendibility and reusability of the software. To ensure the above, we have designed the repository based on the aforementioned axes, as shown in Figure 2.

[2] https://extreme.se.uom.gr/percerons/searchM.php

[3] http://extreme.se.uom.gr

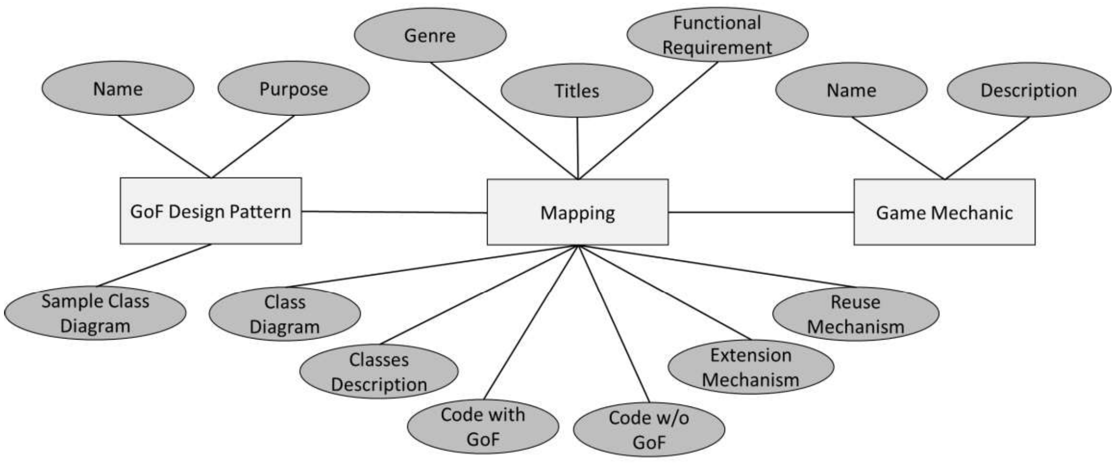


Figure 2: Conceptual Mapping between a Game Mechanic and a GoF Design Pattern

Each element of the repository corresponds to one mapping between a game mechanic and a GoF design pattern. The properties of each element can be split into three parts. The first part corresponds to the basic details of the ***Game Mechanic*** (*i.e., name, description*); the second part, corresponds to the ***GoF Design Pattern*** description (i.e., *name, purpose, sample diagram*); the third part acts as a ***Demonstration for the Mapping***, providing the following information: (a) *game genre*; (b) *example game titles*; (c) example of the *functional requirements* to be implemented—game mechanic; (d) the *class diagram of the solution*; (e) a *detailed presentation of classes and methods involved in the pattern*; (f) link to the *implementation with the pattern*; (g) link to the *implementation without the pattern*; (h) exploitation of *extension mechanisms*; and (i) exploitation of *reuse mechanisms*.

4.2 Method for Storing and Retrieving Data in/from the Repository

The mappings that are stored in the repository have been extracted based on two sources: the related literature and open-source games. First, we have explored related work (see Section 2.3) and we catalogued the default implementations of 13 game mechanics [33][34][35]. To extract mappings from open-source software, we have applied two strategies (as outlined in the bullet list below):

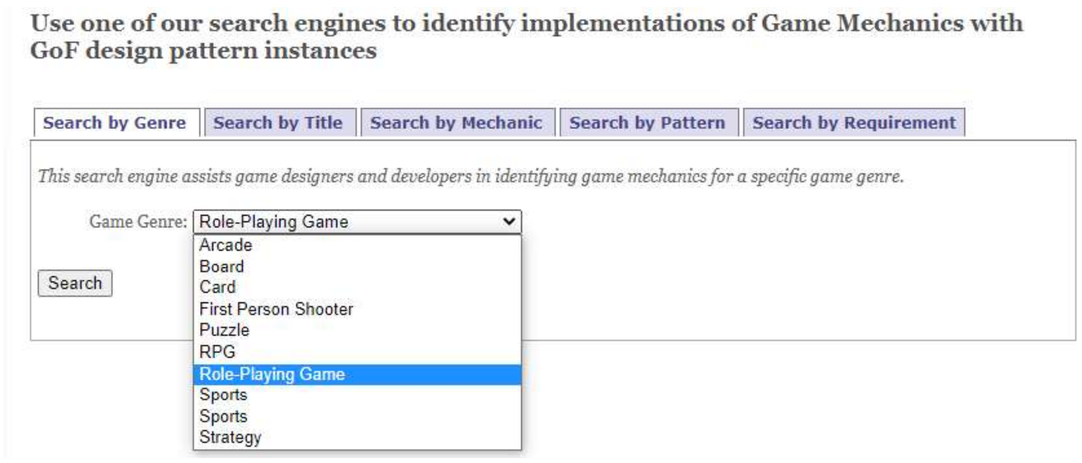
- ***Forward Identification of Mappings***. In a forward engineering manner, we start from game requirements (i.e., game mechanics) and then proceed with the identification of the implementation (GoF design patterns). To apply this process, we explored five OSS games (namely: BloodBowl, FreeCol, Pacdasher, Flesh Snatcher, Open Imperium Galactica—links to the repositories of these projects are provided in Appendix B) and identified five opportunities for such mappings. We note that the identification was not exhaustive, i.e., we have not systematically searched the complete source code or requirements of the game, but we have rather opportunistically identified requirements that correspond to game mechanics, tracked them to the source code, and checked if a pattern is applied. If a pattern was applied, we retained the current implementation, if not, we have refactored the source code to a GoF design pattern. The identification of mechanics was performed by the third author, by playing the game, after getting acquainted with the game mechanics' catalogue.
- ***Reverse Identification of Mappings***. In reverse engineering, we start from "*later*" development artifacts (e.g., source code) and try to map them into "*earlier*" artifacts. To populate our repository, we used the existing dataset of DePRE (in EXTREME) [39]. From the repository we selected two of the games (namely: FreeCol and Pacdasher) that were used in the Forward Identification of Mappings step, and catalogued all GoF pattern occurrences. These occurrences have been manually investigated to explore if they are implementing a game mechanic.

In the end of the aforementioned process, the repository consisted of 23 mappings (see Table 1).

Table 1: GAME-DP stored elements

Literature Patterns (no code)	OSS-Based Identified Mappings (with code)
• Agents with Strategy	• Limited Set of Actions with Command
• Creative Control through Builder	• Game World with Builder
• Enemy State and Behavior with Strategy	• Traverse with Strategy
• Fast Enemies Clone with Prototype	• Symmetric Information with Observer
• Game World with Composite	• Reward with Visitor
• Game State Management with State	• Single Player with State
• Levels with State	• Enemies with Flyweight
• Movement with Strategy	• Units with Flyweight
• Power-ups with Visitor	• Manoeuvring with State
• Progress Indicators with Observer	• Pickups with Factory
• Tiles with Composite	
• Turn-based Games with Template Method	
• Units with Abstract Factory	
• Varied Gameplay with State	

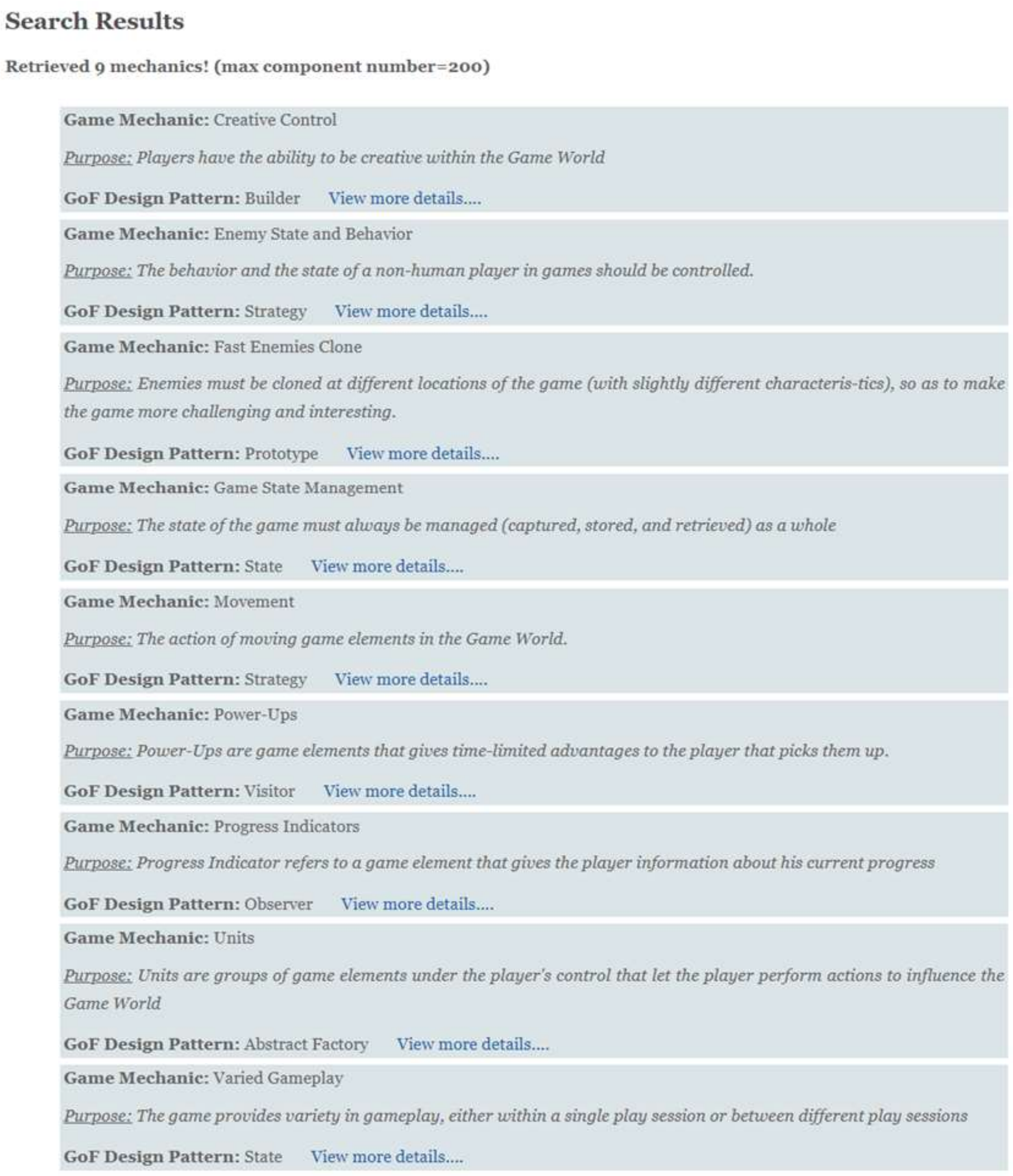
To retrieve the data from the repository, we have created a web interface, inside EXTREME that queries the GAME-DP repository, providing the end-user with five options: (a) Search by Game Genre; (b) Search by Game Title; (c) Search Game Mechanic; (d) Search Requirement; and (e) Search GoF Pattern. Through the first and second option, the user can see all the game mechanics that are stored in the repository for a given genre or title. Then, upon the selection of the mechanic, the mapping to the GoF pattern is displayed—these options are expected to be used in the first step of game design (see Figure 1). The third and the fourth option, are expected to be used by the developer when requirements and game mechanics have been fixed and are ready to be implemented. Finally, the fifth option is expected to be used by software developers, mostly for exploratory purposes, to answer questions such as: "*How could I use the Bridge design pattern in game development?*". Example screenshots of the repository, are presented in Figures 3-5.


Use one of our search engines to identify implementations of Game Mechanics with GoF design pattern instances

This search engine assists game designers and developers in identifying game mechanics for a specific game genre.

Game Genre: Role-Playing Game

Figure 3: Search Options



Search Results

Retrieved 9 mechanics! (max component number=200)

Game Mechanic: Creative Control

Purpose: Players have the ability to be creative within the Game World

GoF Design Pattern: Builder View more details....

Game Mechanic: Enemy State and Behavior

Purpose: The behavior and the state of a non-human player in games should be controlled.

GoF Design Pattern: Strategy View more details....

Game Mechanic: Fast Enemies Clone

Purpose: Enemies must be cloned at different locations of the game (with slightly different characteris-tics), so as to make the game more challenging and interesting.

GoF Design Pattern: Prototype View more details....

Game Mechanic: Game State Management

Purpose: The state of the game must always be managed (captured, stored, and retrieved) as a whole

GoF Design Pattern: State View more details....

Game Mechanic: Movement

Purpose: The action of moving game elements in the Game World.

GoF Design Pattern: Strategy View more details....

Game Mechanic: Power-Ups

Purpose: Power-Ups are game elements that gives time-limited advantages to the player that picks them up.

GoF Design Pattern: Visitor View more details....

Game Mechanic: Progress Indicators

Purpose: Progress Indicator refers to a game element that gives the player information about his current progress

GoF Design Pattern: Observer View more details....

Game Mechanic: Units

Purpose: Units are groups of game elements under the player's control that let the player perform actions to influence the Game World

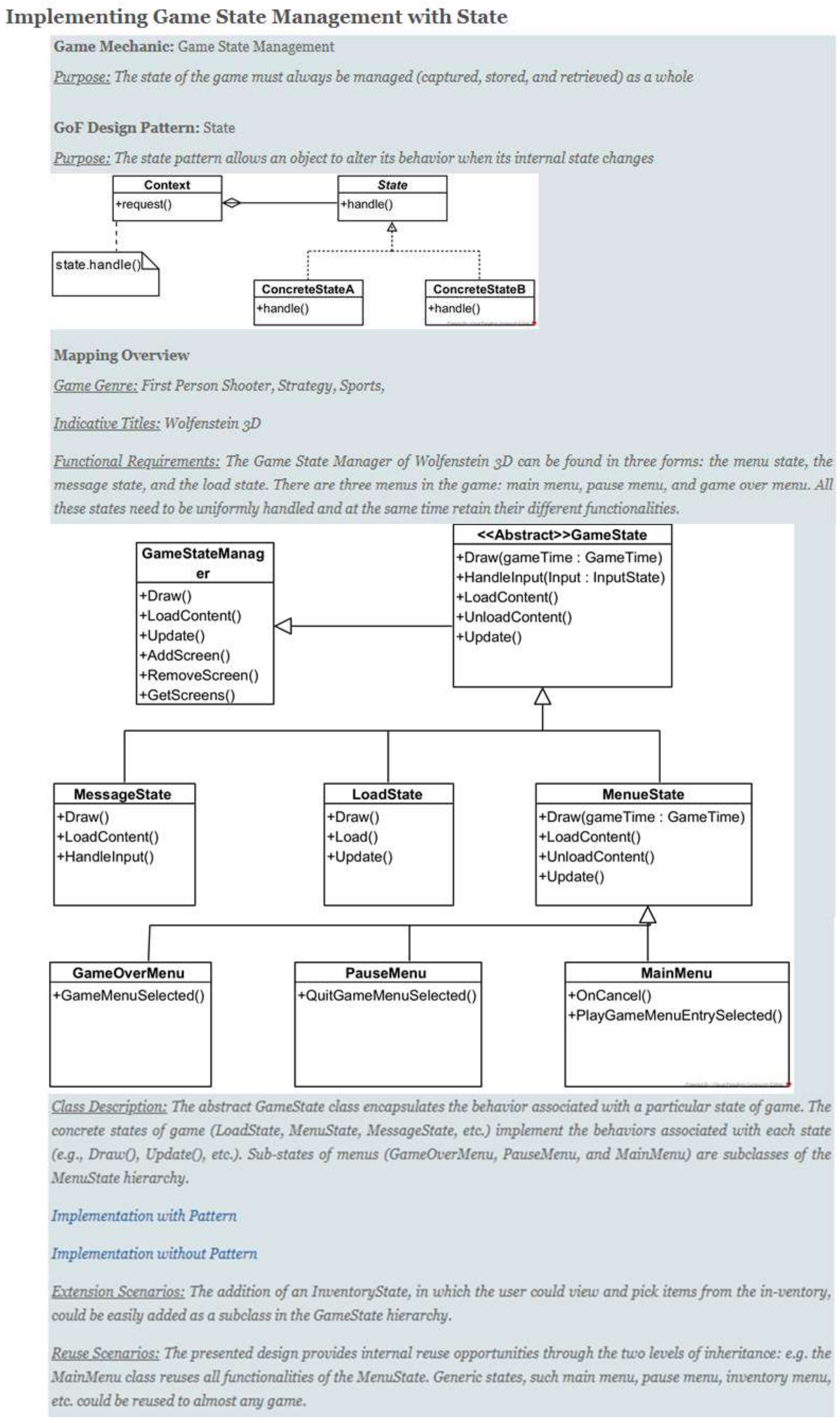
GoF Design Pattern: Abstract Factory View more details....

Game Mechanic: Varied Gameplay

Purpose: The game provides variety in gameplay, either within a single play session or between different play sessions

GoF Design Pattern: State View more details....

Figure 4: Results Overview



Implementing Game State Management with State

Game Mechanic: Game State Management

Purpose: The state of the game must always be managed (captured, stored, and retrieved) as a whole

GoF Design Pattern: State

Purpose: The state pattern allows an object to alter its behavior when its internal state changes

Mapping Overview

Game Genre: First Person Shooter, Strategy, Sports,

Indicative Titles: Wolfenstein 3D

Functional Requirements: The Game State Manager of Wolfenstein 3D can be found in three forms: the menu state, the message state, and the load state. There are three menus in the game: main menu, pause menu, and game over menu. All these states need to be uniformly handled and at the same time retain their different functionalities.

Class Description: The abstract GameState class encapsulates the behavior associated with a particular state of game. The concrete states of game (LoadState, MenuState, MessageState, etc.) implement the behaviors associated with each state (e.g., Draw(), Update(), etc.). Sub-states of menus (GameOverMenu, PauseMenu, and MainMenu) are subclasses of the MenuState hierarchy.

Implementation with Pattern

Implementation without Pattern

Extension Scenarios: The addition of an InventoryState, in which the user could view and pick items from the in-ventory, could be easily added as a subclass in the GameState hierarchy.

Reuse Scenarios: The presented design provides internal reuse opportunities through the two levels of inheritance: e.g. the MainMenu class reuses all functionalities of the MenuState. Generic states, such main menu, pause menu, inventory menu, etc. could be reused to almost any game.

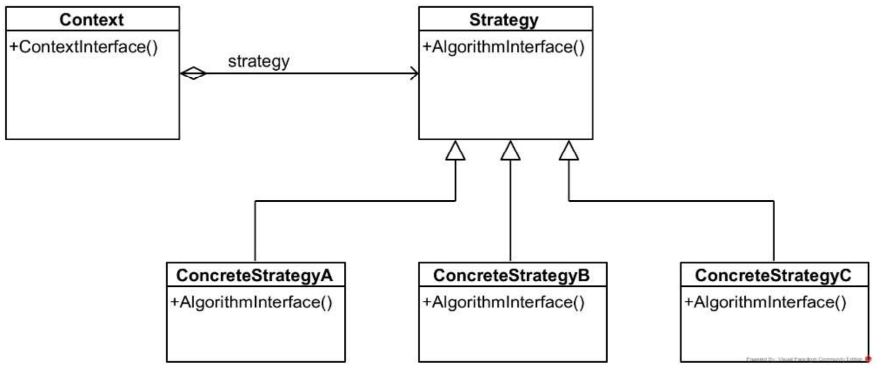
Figure 5: Mapping Overview

The complete documentation of the GAME-DP elements is available on the online repository and Appendices A and B.

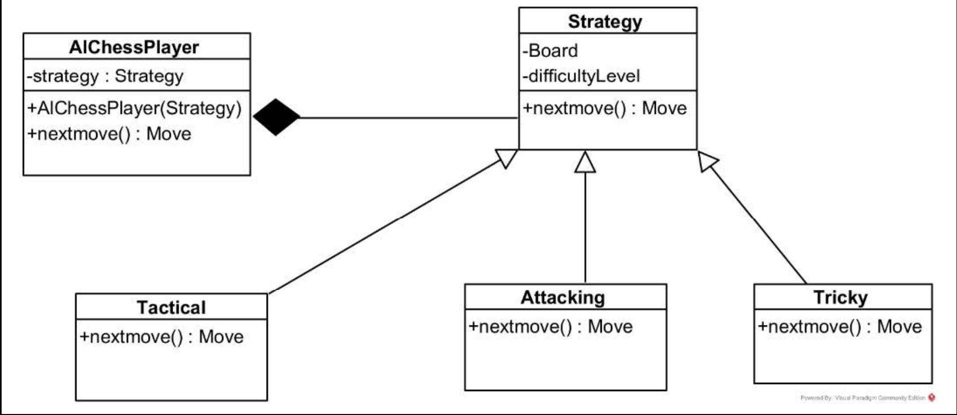
4.3 Illustrative Example with Unity

We demonstrate the use of GAME-DP repository, by refactoring to pattern [43] a Chess game, developed with Unity[4]. By browsing among the mappings stored in GAME-DP, we have been able to retrieve 3 game mechanics related to Chess: `Turn-based Games`, `Agents`, and `Varied Game Play`. Among them, we preferred to apply Agents, since it was the one with the most intense game logic. The corresponding GAME-DP element is presented in Table 2.

Table 2: GAME-DP element for chess game

Part A: Game Mechanic	
Name	***Agents***
Description	Entities in games that take the roles of players but are controlled by the game system
Part B: GoF Design Pattern	
Name	***Strategy***
Purpose	The strategy pattern is a behavioral software design pattern that enables selecting an algorithm at runtime. Instead of implementing a single algorithm directly, code receives run-time instructions as to which in a family of algorithms to use
Sample Class Diagram	
Part C: Implementation of Game Mechanic with GoF Pattern	
Game Genre	FPS, Strategy, Board, RPG, Card, Arcade
Example Game Titles	Chess
Functional Requirement	Develop a family of algorithms that are responsible for selecting the next move of an AI-player (non-human player) in a chess game. The nextMove method returns the next move of the player, based on specific chess styles. The current design supports three chess styles: tactical, attacking, and tricky.

[4] https://github.com/SebLague/Chess-AI

Class Diagram of Solution	 Strategy -Board -difficultyLevel +nextmove() : Move AIChessPlayer -strategy : Strategy +AIChessPlayer(Strategy) +nextmove() : Move Tactical +nextmove() : Move Attacking +nextmove() : Move Tricky +nextmove() : Move
Code Explanation	The abstract class Strategy is responsible for the selecting the next move of an AIChessPlayer. The concrete strategies that implement the next move selections (i.e., implementations of the AI algorithm) are implemented in the sub-classes: Tactical, Attacking, and Tricky. The strategy takes into account other parameters, such as the state of the Board and the difficultyLevel, which are attributes in the superclass. The class that plays the Context role is the AIChessPlayer class and it represents the agent of the game mechanic.
Reuse Opportunities	The same structure could be used in all board games such as backgammon. For example, reusing the Strategy class in another game, would facilitate the reuse of the Board class, which is mandatory for selecting the next move of the computer player.
Extension Mechanisms	Using this structure, if along maintenance a new chess style (e.g., Positioning, Dynamic, etc.) has to be integrated in the game, it can be incorporated by adding a new subclass, without altering the code (conformance to the Open-Closed Principle).

By exploring the source code of the selected Chess-AI implementation in GitHub, we have identified that the selection of the next move is located in the `Assets.Scripts.Core.AI.Evaluation`, `Assets.Scripts.Core.AI.MoveOrdering`, and `Assets.Scripts.Core.AI.Search` classes. The identified parameter (`targetDepth`) for the depth of "*thinking*", while selecting the next move is located in `StartSearch()` method, the evaluation of the move in the `Evaluate(Board board)` method, whereas the ordering of all possible moves in `OrderMoves(Board board, List<Move> moves)`. The application of the pattern enables the creation of three hierarchies of these core AI strategies that would enable the run-time alternation of the way the AI player: (a) can ***evaluate*** the ***board*** condition (e.g., by applying different weights to the pawns that are active in the board), (b) can ***order*** the available ***moves*** (e.g., by assigning different scores to pawns promotion or threat level), and (c) can ***search*** the available ***moves*** (e.g., by increasing the look-ahead option). Upon the development of the corresponding scripts in C# relying on the inheritance and polymorphism mechanisms, imposed by the Strategy pattern, the corresponding behavior is called in the `Assets.Scripts.Core.GameManager` class. The strategy of the AI player can be differentiated either after a specific number of moves, or if the evaluation of the board condition for the AI player goes below a certain threshold (i.e., starts or is getting closer to losing).

5. Experimental Setup

This section presents the design of the experiment that we have conducted for evaluating the GAME-DP repository. The experiment was designed and conducted, based on the guidelines on experimental software engineering by Wohlin et al. [44].

Research Goals and Research Questions. The goal of the experiment was to assess the potential benefits offered by the GAME-DP repository, in terms of: (a) software extendibility; and (b) software reusability (stemming from *Need-A*, see Section 1). Thus, we have set two main research questions:

***RQ*$_1$**: Can the use of GAME-DP repository lead to more efficient (fast and correct) game extension?
***RQ*$_2$**: Can the use of GAME-DP repository facilitate (fast and correct) reuse in game development?

Experimental Design. To answer the aforementioned questions, we have designed an experiment, with 12 low-medium experienced game developers (each one developed 1-3 games); having however, medium-high experience in software development (3-10 years of development background, 55% of whom are senior professional software engineers). The selection of this type of participants was based on our belief that the mapping between game mechanics and GoF patterns will be more beneficial for novice developers [10][28][39]. The implications of this choice are discussed as tentative threats to validity in Section 8. The research questions have been answered using a quantitative pre-post analysis, supported by quotes from participants (mostly for interpretation and discussion purposes). The pre-post analysis is used to assess the effect of a "*treatment*" on a specific problem, by comparing some indicators before and after the application of the treatment (into different groups). In the case of our research the treatment is the use of the GAME-DP repository (including two factors: the application of the pattern per se, and the provided documentation). To perform the analysis, we split the participants into three groups. Since we want to control the two aforementioned factors (use of GoF pattern, and use of GAME-DP elements), we need to create a group for each possible combination of factor variable scores. Thus:

- ***For Group-1***: we provide a source code implementation of the mechanic that *does not use any GoF Design Pattern*. Therefore, this implementation *is not a GAME-DP element.*
- ***For Group-2***: we provide a source code implementation of the mechanic that *uses a GoF Design Pattern*. However, the relevant *GAME-DP element is not made accessible to the developer*, so the developer is not aware of the GoF pattern (or he/she is, based on his current knowledge).
- ***For Group-3***: we provide an implementation of the mechanic that *uses a GoF Design Pattern*; and we make the relevant *GAME-DP element available to the developer*. So, in this group, the developer makes full use of all information related to the GoF pattern and the game mechanic.

Pre-Post Analysis investigates for differences among G1, G2, and G3. To explore these differences, each participant is provided with two tasks: (a) an extension task (related to RQ$_1$), as an example see Figure 6; and (b) a reuse task (related to RQ$_2$), see Figure 7—the full task list is provided in Appendix C. We note that to reduce the time required to fulfil the tasks, and involve practitioners to as many tasks as possible, we have preferred to detach the experiment from the use of a game engine; i.e., we selected OSS games developed from scratch. The threats to validity raised by this decision are discussed in detail in Section 8.

The experiment was organized as a half-day workshop, divided into four parts, summarized as follows:

- the first part included a pre-interview of participants, so as to evenly distribute them into the aforementioned groups. Each pre-experiment interview lasted for 10 minutes and during this period the researchers got details on the experience of the participants on programming, and GoF design patterns.
- in the second part, the participants were given the task descriptions, and were given 20 minutes to understand the tasks, and discuss questions with the researchers.
- in the third part, the participants were given the corresponding source code, and were given 2 hours and 15 minutes, for completing the 3 tasks.
- in the fourth part, the participants were given 20 minutes to fill a post-experiment questionnaire—see Appendix C.

Task Description: Each player in BloodBowl can move inside the field, block his enemy, throw (i.e., pass) the ball, and attempt to steal the ball.

We ask you to implement a new possible move (namely: `jumpToBlock`) in which a player attempts to block the opponent from a further distance: not by standing in his way, but by jumping. The logic of the move is the same as block, but: (a) it can be executed when the defender is 2 or 3 blocks away from the attacker; and (b) has 50% and 33% chances of being successful and injuring the blocked player (stunned). Upon an unsuccessful `jumpToBlock` attempt, the blocker gets injured (KO). The starting class for this task is: `MoveActionState`. The correction guidelines (i.e., how we graded the correctness of the solutions) are presented below:

- The developer successfully spots the parts that need to be updated (`if` statements in G1, classes and methods in G2 and G3) (3 points)
- The developer successfully implements the `execute` functionality (3 points)
- The developer successfully implements the `isLegal` functionality (3 points)
- The developer successfully implements the `endsTeamTurn` functionality (1 points)

Figure 6: Example of an Extension Task

Task Description: Suppose a Poker game. During the bet handling of poker each player can execute 3 actions: folding, calling, or raising the bet. The folding is valid when it's the player's turn, whereas its execution will end up in de-activating him for the round and ending his turn. The calling is valid when it's the player's turn and has the amount of cash in his hand, whereas its execution will end up in increasing the pot and ending his turn. Finally, raising the bet is valid when it's his turn, has raised the amount of cash in his hand and it is lower or equal to the maximum amount the rest of the active players have, whereas its execution will end up in increasing the pot and ending his turn. By reusing the knowledge from the example in the mapping of the Limited Set of Actions game mechanic, develop a class Player that holds all necessary actions and implement the action execution and checking if it is valid. The code of the `Player` class is provided below. The correction guidelines are presented below:

- Skeleton of classes (2 points) / Reusing the provided template (3 points)
- The developer implements the `execute` (2 points) and `isLegal` functionality (3 points)

```
public class Main {
   public static void main(String[] args) {
   }
}
class Player {
   HashMap<String, Command> commands;
   boolean active;

   public Player() {
       commands = new HashMap();
       //add code here
   }
   public void executeCommand(String commandKey) {
       //add code here
   }
   //void changeTurn(): changes the turn
   public void changeTurn() {
       System.out.println("Turn changed");
   }
   //returns true if it's player's turn
   public boolean isPlaying() {
       return true;
   }
   //returns true if the amount is smaller than the highest player with cash
   public boolean canRaise(double amount) {
       return true;
   }
   //returns true if the amount is below the holding cash of the Player.
   public boolean cashInHand(double amount) {
       return true;
   }
   public void setActive(boolean b) { //sets the Players state in current round
       active = b;
   } }
interface Command {}
```

Figure 7: Example of a Reuse Task

In Table 3, we present the task assignment and the group distribution of participants. Participants are anonymized and referred as P1-P12 (from the most experienced to the least experienced one). In this way, we guarantee that each group consists of the same number of participants of the 1st (1-3) to 4th (9-12) quartile of experience. To achieve this a round-robin algorithm was used.

Table 3: Assignment of Participants into Groups

Mapping	**Extendibility Task**	**Reuse Task**
Limited Set of Actions with Command	P1 / P2 Group-1 / Group-2	P11 / P12 Group-1 / Group-2
Game World with Builder	P5 / P6 Group-3 / Group-2	P7 / P8 Group-2 / Group-3
Traverse with Strategy	P4 / P2 Group-1 / Group-2	P10 / P9 Group-1 / Group-3

Mapping	Extendibility Task	Reuse Task
Symmetric Information with Observer	P12 / P8 Group-1 / Group-2	P5 / P6 Group-1 / Group-2
Reward with Visitor	P1 / P3 Group-2 / Group-3	P11 / P12 Group-1 / Group-2
Single Player with State	P10 / P9 Group-1 / Group-3	P4 / P3 Group-1 / Group-3
Enemies with Flyweight	P11 / P7 Group-2 / Group-3	P1 / P2 Group-3 / Group-2
Units with Flyweight	P3 / P4 Group-1 / Group-3	P9 / P10 Group-1 / Group-3
Manoeuvring with State	P6 / P5 Group-1 / Group-3	P8 / P7 Group-2 / Group-3

For each one of these tasks, the following variables have been recorded:

[V1] Group (G1 / G2 / G3)

[V2] Task ID (#mapping from Appendix B)—implying: GoF pattern and the Game mechanic

[V3] Task Type (Extension / Reuse)

[V4] Time to Complete Task

[V5] Correctness score (based on the correction guidelines—see Figure 6 and 7)

[V6] Responses to interview questions

Variables [V1] to [V5] are going to be used for answering the research questions. As a means of analysis, we will use basic descriptive statistics, visualization methods, and hypothesis testing to explore the existence of statistically significant differences (in [V4] and [V5]) between groups (G1-G2 and G2-G3), organized by research question ([V3]). A sample examined hypothesis is presented below:

$H_{0(1)}$: The mean ***time*** needed to complete an *extension task* from participants in `Group G1`, are not different from the time needed to complete an extension task from participants in `Group G2`.

In addition to the above quantitative analysis, we have provided participants with a questionnaire aiming to describe the obtained benefits in a qualitative manner—the questionnaire is presented in Appendix C (and stored in [V6]). At this stage it is necessary to clarify that all tasks satisfy several constraints, to guarantee that the experiment environment is controlled. For instance, code development has been performed under the supervision of the first and third author. Additionally, full details of task requirements have been provided; thus, the subjects were not able to deflect from the experiment scenario. All subjects were allowed to use Eclipse for compiling and completing the development tasks.

Conducting the Experiment. According to Wohlin et al. [44], the experiment operation phase consists of three steps: preparation, execution and analysis. The preparation phase included the organization of the workshop and the recording of variables [V1]—[V3] in the dataset. During experiment execution, the researchers have collected and validated data concerning the time-related variable [V4]. On the completion of the experiment, two researchers (first and third author) have corrected the solutions provided by the experiment participants (concerning correctness variable [V5]). The guidelines used while evaluating objective correctness are presented in Appendix C. The two correctness scores were aggregated as an average (in case their difference was lower than two grades), or a third grader (second author) has been involved to resolve the difference. The conflict process was not initiated at the grading stage of this experiment, probably due to the clear correction instructions. Finally, the first and the second author surveyed the participants to extract the values of the variable concerning quality and understandability benefits [V6].

6. Results

In this section we present the results of our experiment, organized by research question. We note that the timely and successful (getting all correctness points—we remind the reader that correctness was assessed by the researchers using a pre-existing grading schema, see Figures 6 and 7 for examples) completion rate for extension or reuse tasks (by considering the dataset as a whole—without discrimination to groups) was 16.67%; whereas the average completion time for extension tasks was 36.33 minutes and the average completion time for reuse tasks was 34.38 minutes. By exploring the completion time by participant, 55% of the developers completed their task ***faster*** when using the input from *Group-1 (no pattern / no documentation)*, whereas 65% of the developers produced the ***more correct*** solution when using input from *Group-3 (pattern and GAME-DP documentation)*. We remind the reader that based on Table 3, pairwise comparisons have taken place in the experiment. It is noteworthy, that only 1 developer produced his/her most correct solution using input from Group-1. The quick production of a solution when belonging to Group-1 can be explained by the fact that in this group, the participant has to read no documentation and a (usually) shorter non-pattern solution [43], enabling the reduction of time required for "*understanding*" the code, which according to Kosti et al. [45] is more time consuming than the actual implementation of the change. Nevertheless, the feeling of a "*quicker*" understanding the task among *Group-1* participants is deceptive, since the quick understanding leads to more error-prone solutions. Next, we elaborate on the results, organizing them per research question. A detailed interpretation of the results, as well as research and practical implications are provided in a consolidated form in Section 6.

5.1 Effect of using GAME-DP repository elements on extendibility

In this section, we discuss the effect of using the GAME-DP elements on the extendibility of game implementations. In Figure 8, we present the time required to reach a solution on the extension task, as well as the correctness of the submitted solution, denoting the group of each participant. The average time required to produce a solution was 29.16 minutes for *Group-1 (no pattern / no documentation)*, 40.33 minutes for *Group-2 (pattern / no documentation)*, and 39.50 minutes for *Group-3 (pattern and GAME-DP documentation)*. On the contrary, the correctness of the submitted solution was on average: 5.33 (out of 10) for *Group-1*, 6.50 for *Group-2*, and 8.16 for *Group-3*.

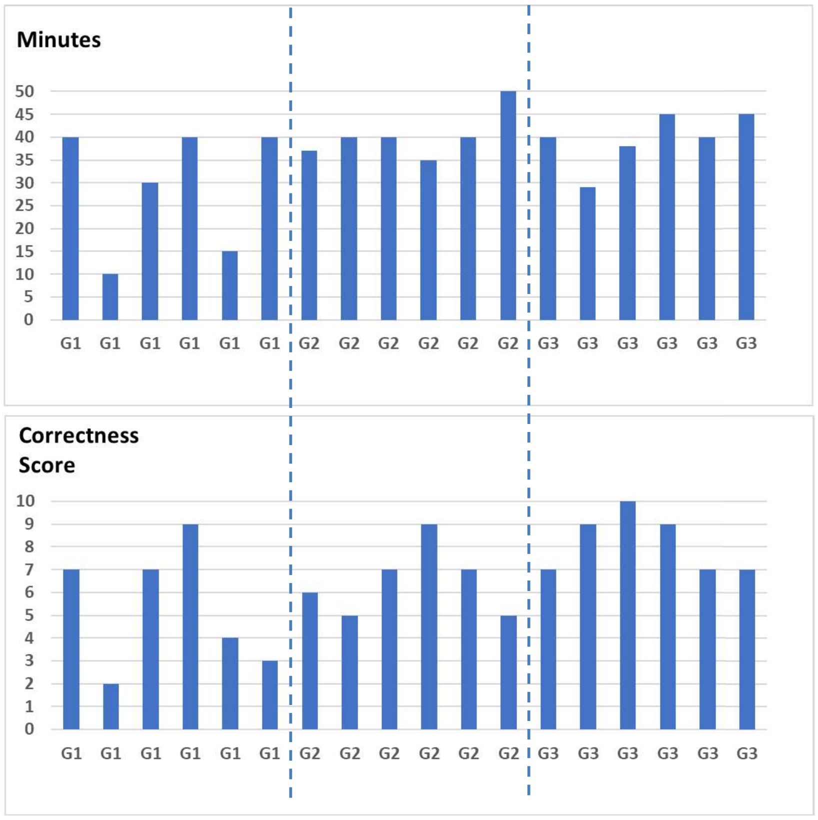


Figure 8: Completion Time and Correctness Level for Extension Tasks

The Friedman's analysis of variance (ANOVA)[5] among the groups suggested that there is no statistically significant difference among the three groups, neither in terms of time to complete (sig: 0.112) nor in terms of their correctness (sig: 0.083). Nevertheless, a post-hoc analysis has revealed that: (a) in terms of ***time to complete*** all differences are ***not statistically significant***; whereas (b) in terms of ***correctness***, *Group-3* provides ***statistically significantly more correct solutions*** ($p<0.05$ for the comparison to Group-1; and $p<0.05$ for the comparison to Group-2). These differences (or their absence) can be visually depicted through the boxplots of Figure 9. A tentative explanation on the differences in terms of correctness was provided by one of the participants: "*If the existing code is structured correctly in the first place (notice from author: as it is when a pattern is there), making future alterations without negatively impacting the code correctness is easier*". By visually inspecting Figure 9, we can observe that the group with the highest ***variation*** (long line or box) in terms of ***required time*** or ***correctness*** is Group-1, ranging from "*too high*" to "*too low*". This finding suggests that factors other than the group (e.g., developers' experience) are able to characterize the effort to complete the task, as well as the quality of the obtained solution.

[5] We note that we preferred to execute a non-parametric test, since the preconditions for parametric tests were not fulfilled by our dataset.

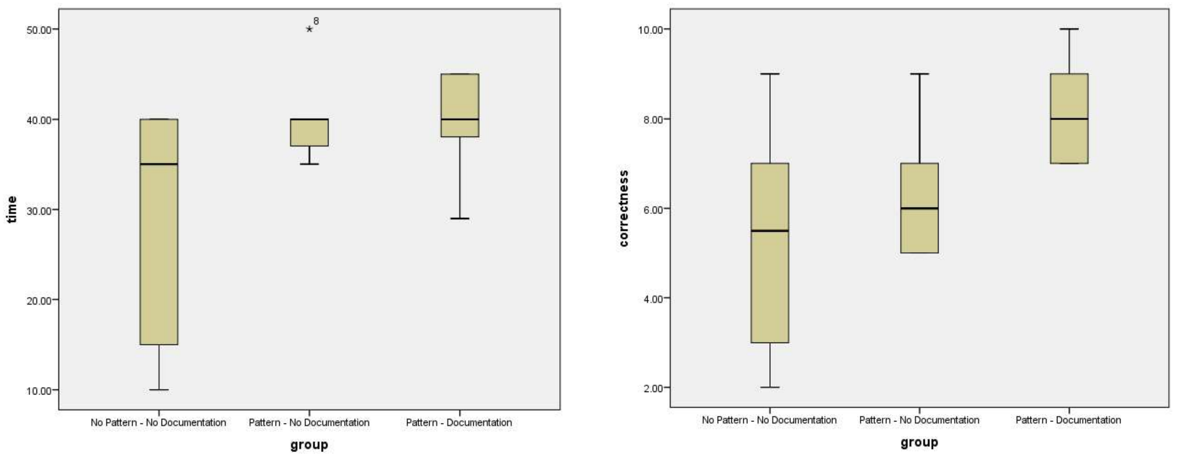


Figure 9: Boxplots on the differences in terms of time and correctness (extension tasks)

The most usual errors that have been identified in *Group*-1 and *Group*-2 had to do with incorrect implementation of the game mechanic. For instance: P2 (belonging to *Group-2*) was not able to understand the `Rewards` mechanic, and thus has not implemented properly the point system; *or* P4 (belonging to *Group-1*) was not able to correctly understand the implementation of `Traverse` and could not implement correctly the `GoalDecider` function. On the other hand, developers of *Group-3* produced errors that were more related to programming decisions, e.g., P4 has implemented a solution with a `HashMap` (unnecessarily complicating the solution) without saving them if they are not printed.

Extension tasks are completed within a similar timeframe, regardless of the level of information that is provided to developers (in the sense that there are no statistically significant differences). In some cases, developers that are assigned simple starting code and are not provided with any documentation are faster in delivery. However, the solutions that they provide are of lower levels of correctness, compared to those that are provided with documentation and a pattern-based solution.

5.2 Effect of using GAME-DP repository elements on reusability

In this section, we follow the same structure as in Section 5.1, focusing on the reuse task. In Figure 10, for each participant (denoting only the group that he/she belongs to) we present the completion time for each reuse task and the achieved correctness score. The average completion time for reuse tasks was 31.33 minutes for *Group-1 (no pattern / no documentation)*, 37.66 minutes for *Group-2 (pattern / no documentation)*, and 34.16 minutes for *Group-3 (pattern and GAME-DP documentation)*. The correctness score was: 4.16 (out of 10) for *Group-1*, 5.33 for *Group-2*, and 7.83 for *Group-3*.

An interesting observation from Figure 10 (when compared to Figure 8), is that 3 participants failed to complete the reuse task (as shown by the lack of a correctness bar), even though they used the complete timeframe given for the task (in Figure 8 we can observe that all participants submitted a solution—even of poor quality). This finding suggests that the reuse (of design rationale) tasks are mentally more intensive compared to extension tasks. Another interesting observation is that for reuse tasks the time required for submitting a solution from participants of *Group-1* and *Group-3,* is similar. However, still the correctness levels of submissions from *Group-3* are substantially higher, suggesting that the benefit from using the GAME-DP elements for reuse tasks (in terms of correctness) come with less cost (additional time to solve the problem) compared to extension tasks.

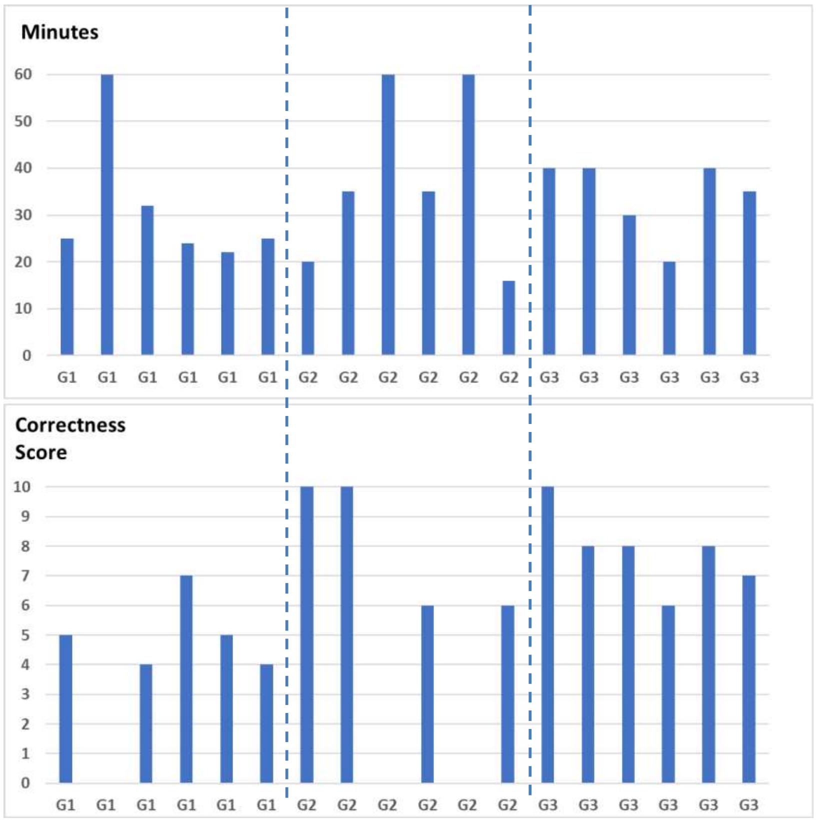


Figure 10: Completion Time and Correctness Level for Reuse Tasks

Based on the Friedman's analysis of variance (ANOVA), we have reached the conclusion that the aforementioned differences are not statistically significant among all groups, for neither required effort to submit a solution (sig: 0.755), nor for the correctness of the solution (sig: 0.134). Nevertheless, similarly to our results on extension tasks, the correctness levels of *Group-3* are higher at a statistically significant level (based on post-hoc analysis—$p<0.05$ for the comparison to Group-2; and $p<0.01$ for the comparison to Group-1). This result is visualized in the boxplots of Figure 11, in which we can also observe that for *Group-2*, the completion time and the quality of the solution are completely unpredictable, ranging from "*very low*" to "*very high*" values (i.e., for correctness: from no submission to completely correct ones). This finding suggests that knowledgeable participants gain from the use of patterns, even without documentation (e.g., the solution of P2 scored 10 out of 10, and was submitted in 20 minutes), whereas less experienced developers "*get lost*" in the complexity of the given initial solution (e.g., P11 and P12 failed to submit a solution)[6]. The errors that have been reported in reuse tasks are related to the understanding of the game mechanic for *Group-1* ("*There was no method used for storing the unit*") and *Group-2* ("*There is no need to traverse the list of goals*") and more related to the programming aspect for *Group-3* ("*For the linking of the two new states it would be better to use implements instead of extends*").

[6] We remind that the index after P, in the codes of participants denote their experience: 1 is the most experienced developer, and 12 is the least experienced one

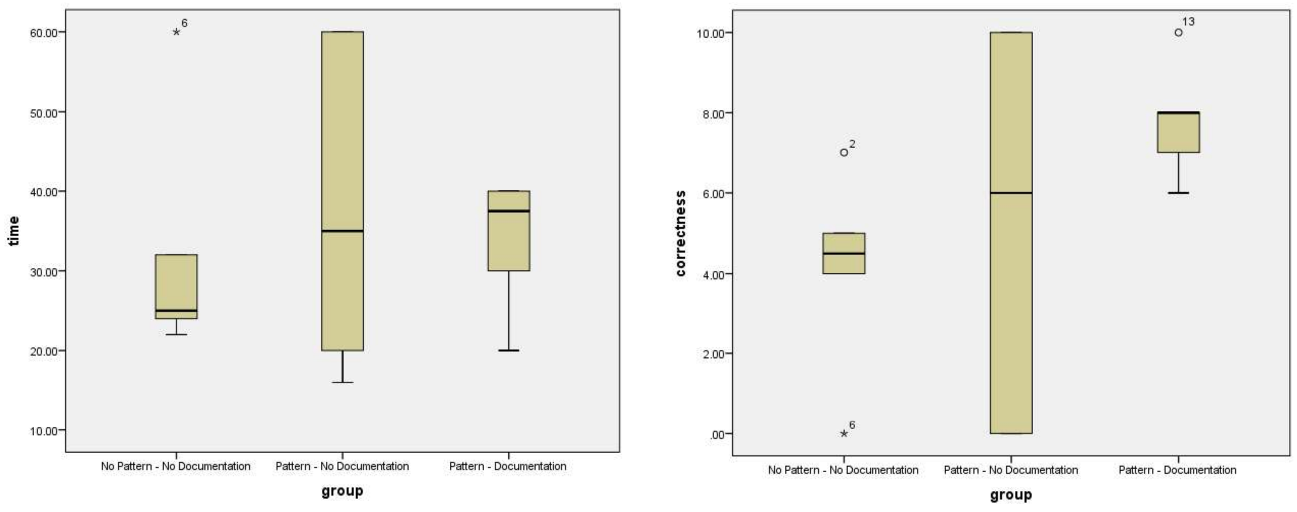


Figure 11: Boxplots on the differences in terms of time and correctness (reuse tasks)

Reuse tasks are completed in a similar timeframe for the group that is given a non-documented non-pattern starting point and the group that is given a documented pattern starting point, but the quality of the solutions differs significantly. Thus, the use of GAME-DP repository seems even more appropriate for reuse tasks. The benefit obtained by the use of patterns without any documentation can lead to very diverse effects, based on the level of knowledge / experience of the developer.

7. Discussion

In this section we discuss the findings of the paper, placing special emphasis on the interpretation of the results, and the implications that they bring in academia, and the game development industry.

Interpretation of Results. The main findings of this study can be organized and be discussed, from three perspectives: (a) comparison among groups—effectiveness (fast and correct execution of tasks) of GAME-DP repository; (b) comparison among tasks—differences in effectiveness (fast and correct execution of tasks) in reuse and extension; and (c) the role of the developer in the effectiveness (fast and correct execution of tasks) of GAME-DP repository. The main findings are elaborated and interpreted below.

- *Effectiveness of GAME-DP repository*: GAME-DP elements has been evaluated, based on two criteria: time to complete a task and correctness of the examined solution. With respect to correctness, the solutions that relied on GAME-DP elements documentation were of better quality, compared to both pattern-based and non-pattern-based solutions without documentation. This finding (which is statistically significant) suggests that the provision of documentation facilitates the in-depth understanding of the existing structure of the software, as well as of the task itself (in the sense that the game mechanic describes the requirement). Especially with respect to game mechanics, all participants agreed that the documentation was detailed enough (e.g., "*The provided documentation was quite resourceful, so it was easy to understand the game mechanics, since it was very concise*"). With respect to GoF design patterns, all developers agreed on the usefulness of the documentation, since they believed that the documentation saved a lot of time: "*If the documentation was absent, a great deal of time would have been wasted, trying to understand the whole concept*" or "*Despite my lack of experience on the specific technology, the documentation described it in a way that is understandable even on an amateur and on someone with zero knowledge*". This in-depth understanding, guarantees the correct execution of the task, at least in terms of logic, since programming errors can creep into the code in all cases (it is mostly related to the experience of the developer, rather than the design or documentation). On the other hand, regarding time completion, the results (suggesting that the use of documentation and the provision of a pattern-based initial solution has not shrunk development time) are perceived as counter-intuitive, but can still be interpreted, based on the design of the study. In particular, the results of the study suggested that developers that have been provided with non-pattern-based initial solutions and no documentation, have finished the task earlier (however, not successfully / correctly). The combination of the aforementioned observations leads to the conclusion that the developers of this group, have not acquired an in-depth understanding of the software structure, or the problem. However, since the problem space was limited (they were given some classes to update), they could complete the task without "*wasting*" time to identify the parts of the code that need to be updated—which would be the main benefit of such the GAME-DP documentation. Additionally, the developers that were provided with GAME-DP documentation have invested some time to read the documentation and understand the structure and the requirements, correctly and in depth. This has led to an increase in the time required to submit the solution, but also at the improvement of the quality of the solution. We believe that in real-world problems, the time required to complete an extension task without a documentation would have been substantially higher, and reduce the overhead caused by the time invested in reading the documentation.

- *Effectiveness in different tasks*: Given the above, it becomes clear that the use of GAME-DP repository yields an important benefit in terms of software correctness, but it does not come without a cost, in the sense that it slightly increases the development time. However, this extra cost introduced by using GAME-DP has proven to be smaller for reuse tasks, compared to extension tasks. A tentative explanation for this is that the reuse task is more demanding in terms of understanding the structure and the mechanics of the game, in the sense that new code needs to be developed, by replicating a design rationale, compared to perform a small change in an existing small-scale system. Therefore, the effort benefit obtained by understanding the design rationale that was offered by using GAME-DP elements, has eliminated the time investment in reading the documentation. Similarly to before, we believe that in larger-scale systems, this difference would have been even larger.
- *The developer confounding factor:* While designing the study, one of the most important confounding factors that we had to treat was the 'developer' factor—as raised by one of the participants: "*I do not believe that documentation plays the most important role in code quality. Other factors do: such as deadlines and experience*". Despite the shuffling of developers among groups and tasks (which we believe to some extent mitigated the relevant threat to validity), we have reached conclusions that point to developers on the way that results can be interpreted. For example, we have observed that the results (completion time and correctness) were very divergent: (a) for extension tasks, in the group that non-pattern-based and undocumented starting points have been provided; and (b) for reuse tasks, in the group that pattern-based, but undocumented starting points have been provided. This divergence can be explained by the developers' experience, since we have noticed that (in the general case) "good" (timely and correct) outcomes are reached by experienced developer, whereas "poor" ones, by less experienced developers. Therefore, we can claim that the developer factor appears to be an important one in the groups that do not use the GAME-DP elements documentation, and it is to some extent eliminated when the developer is provided with explanations on the code structure and requirements. This finding further highlights the importance and the benefits obtained by the proposed mapping between GoF design patterns and game mechanics. In addition to this, the fact that for extension tasks, the use of a pattern based-solution relieves the divergence in terms of success factors, suggests that in small-scale applications a "good" design rationale can guide the developer on how to apply changes in the system. On the other hand, in the more demanding task of replicating a design rationale, the non-pattern-based group fails, the undoc-

umented pattern-based group provides solutions of highly divergent quality. Promoting the developer as a key factor for success, in the absence of useful documentation.

Implications for Practitioners. Based on the findings of this study, we encourage game developers to try to use GoF patterns (either based on their own knowledge or by consulting the GAME-DP repository), while implementing a game mechanic. Although this process might seem time consuming for developers, the empirical evidence of our study suggest that the process is beneficial in the sense that the final outcome is of better quality, and the extra cost for adopting this process is not significant. Finally, we suggest practitioners to retain in-house mapping of game mechanics and implementations (pattern- or non-pattern-based) that document how a game mechanic can be instantiated. Such in-house catalogues pertain to the benefits of GAME-DP repository, but also include the personal view of developers and the in-house accumulated experience.

Implications for Researchers / Interesting Future Work Directions. Based on the findings of this paper, we encourage researchers of the game community to intensify their work on how "good programming practices" can be applied to game development. Such research directions, apart from GoF patterns could also target refactorings, technical debt management, design principles, etc. Additionally, we encourage the further studying of game mechanics as design hotspots, which implement core functionalities of the game, and therefore, urge for the application of the aforementioned practices. As concrete future work opportunities, we propose the in-depth study of the developer experience factor while dealing with the implementation of game mechanics. Such an approach would require a replication of this experiment with more practitioners that would enable to perform statistical analysis on the influence of this factors (our findings are only based on descriptive statistics and observation of values). Additionally, we believe that it worth replicating the study in larger projects, which would enable to separately calculate the: (a) the time to identify the place in which the change will need to occur; (b) the time required to understand the code to be changed; and (c) the time to apply the change per se. Recording these different variables would enable future research efforts to specialize the impact of design patterns to a specific mental or technical activities, so as to understand the phenomenon in more depth, and in a more realistic setting. Finally, another interesting research implication from this study is the need for exploring educational approaches for bringing the concepts of object-orientation (and patterns more specifically) to game developers, along with their level of adoption.

8. Threats to Validity and Limitations

While the GAME-DP framework itself is subject to limitations (see next paragraph), the experiment conducted to assess its validity is open to factors that threaten the validity of its results. First of all, the experiment has been carried out with a specific set of engineers and a specific extension and reuse task, thus unavoidably leading to generalizability threats. The ability of the experiment to assess the usefulness of mapping between game mechanics and design patterns is subject to construct validity threats referring to the extent to which measurements actually test the hypothesis or theory of interest. While measuring the correctness and time it takes for participants to complete certain tasks with and without the use of the proposed framework is a reasonable way to assess the usefulness of the GAME-DP repository, other aspects of the proposed approach might not be fully reflected in the experiment. Nevertheless, the fact that participants managed to complete tasks in a more correct manner when using the repository underscores, at least partially, the value of documenting game mechanics/design pattern mappings. Replicability issues which are also related to conclusion validity threats stem from the fact that the experiment was carried out by researchers, especially for conducting the interviews, measuring the time and assessing the correctness of the tasks. However, this threat is partially mitigated by the detailed presentation of the experimental design which allows the full replication of the experiment by other researchers, as well as from the provision of a replication package[7]. The replication package is organized per participant. For each participant, we provide the corresponding task descriptions, and a link to the starting solution that was provided to each participant. As separate files, we provide the code per se, as well as, the dataset upon data collection, that the data analysis relied upon. We note that due to GDPR issues, we were not able to provide the solutions that were submitted by the participants (their identity would by identifiable based on their git-name). Nevertheless, we were able to provide the acquired scores in an anonymized manner. Moreover, the assignment of scores to the tasks is guided by a specific rating scheme. The conclusion validity is also threatened by the developer confounding factor since the experimental results are tied to the skills of developers that carried out the corresponding tasks as discussed in Section 7. Additionally, another threat to the validity of the experiment stems from the fact that the participants were not game development experts. We believe that this decision has not negatively affected the outcomes of the study, since none of the tasks required any prior knowledge of the internals of game engines, frameworks, or manipulation of artistic aspects of the game (e.g., graphics, sound, etc.), but only the scripts that implement the game logic. Although this decision seems to hurt the realism of the experiment (in the sense that a large portion of the complexity of game design is being hidden from participants), we believe that the impact is minimal, since even if a GoF design pattern would be introduced in a game engine-based game, the changes would only be related to the scripting language part (e.g., C# scripts for Unity, as demonstrated in Section 4.3). Finally, another threat to the validity of the findings comes from the fact that the practitioners are novice developers. In practice, novice developers would not be expected to take decisions on when to use a pattern or not, or taking architectural decisions of any kind. Nevertheless, we believe that ad/hoc decisions (e.g., to implement a specific requirements/mechanic with a pattern) can be taken from developers of any level of maturity or seniority in the company. However, we acknowledge that the results are not applicable to senior software engineers who could take system-wide decisions (i.e., to use GAME-DP repo for implementing all game mechanics with patterns).

In terms of limitations of the proposed approach, we acknowledge that the GAME-DP repository provides not a full coverage as there are other game mechanics and design pattern pairs which might have not been revealed by our investigation. However, the repository or any such set of mapping can be gradually extended so as to include the most common mechanics that a game developer will encounter. Furthermore, illustrating the mapping between a game mechanic and the corresponding design pattern, even through the use of code examples, does not suffice to support the developer in creating his own solution and that the process is prone to errors. Future work could investigate the use of automated code generation through scaffolding so as to enhance the productivity of game developers. Nevertheless, as the current state of the experiment stands the selection of GoF patterns and game mechanics that have been considered in the evaluation, sets a threat to generalizability, in the sense that evaluation could have been different, if different mappings have been selected.

9. Conclusions

Game development is a challenging task often confronted with the instantiation of recurring game mechanics. To assist game developers in reusing and extending their solutions, we introduce template instantiations of game mechanics by mapping them to GoF design patterns. The mapping is instantiated through the GAME-DP repository holding pairs of game mechanics and their corresponding pattern coupled with full documentation and code examples. Furthermore, we empirically validate such map-

[7] https://users.uom.gr/~a.ampatzoglou/aux_material/patterns_mechanics_mapping_replication.zip

pings in terms of software extendibility and reusability through an experiment with software engineers. The results revealed that when developers are assisted by the knowledge and documentation of pairings between game mechanics and game patterns in an extension or reuse task, the successful completion rate was increased at the cost of increased time to comprehend the documentation and then apply the solution. In particular, as more information is revealed through the proposed repository to the participants, the correctness of the solutions increases by 53% for extension tasks and by 88% for reuse tasks, while the time to produce a solution did not exhibit statistically significant differences (although in the general case it was shorter in undocumented, non-pattern-based solutions). At the same time all interviewed developers indicated that the documentation provided through the framework helped them to understand the rational of game mechanics. Based on the findings we can postulate that the existence of public or in-house mappings between recurrent game mechanics and implementing patterns can substantially improve the quality of the resulting code for games and can improve the communication among stakeholders involved in the complex process of game development.

Data Availability Statement

The data that support the findings of this study are openly available in the website of the authors at: https://users.uom.gr/~a.ampatzoglou/aux_material/patterns_mechanics_mapping_replication.zip, reference number: Not Applicable.

References

[1] Engström H, Marklund BB, Backlund P, Toftedahl M. Game development from a software and creative product perspective: A quantitative literature review approach. Entertainment Computing, 2018; 27:10–22.

[2] Viana R, Ponte N, Trinta F, Viana W. A systematic review on software engineering in pervasive games development. Proceedings of the Brazilian Symposium on Computer Games and Digital Entertainment, 2014.

[3] Ampatzoglou A, Stamelos I. Software Engineering Research for Computer Games: A systematic Review. Information and Software Technology, Elsevier, 2010; 52(9):888-901.

[4] Aleem S, Capretz LF, Ahmed F. Game development software engineering process life cycle: a systematic review. Journal of Software Engineering Research and Development, 2016; 4(6).

[5] Eisenberg R. Management of technical debt: a Lockheed Martin experience report. Proceedings of the 5th International Workshop on Managing Technical Debt (MTD' 13), Baltimore, USA, 2013.

[6] Bjork S, Holopainen J. Patterns in Game Design (Game Development Series), Charles River Media, Inc. P.O. Box 417 403 VFW Drive Rockland, MA United States, 2004.

[7] Politowski C, Fontoura L, Petrillo F, Guéhéneuc Y. Are the Old Days Gone? A Survey on Actual Software Engineering Processes in Video Game Industry. Proceedings of the IEEE/ACM 5th International Workshop on Games and Software Engineering (GAS), 2016; 22-28.

[8] Daneva M. Striving for balance: A look at gameplay requirements of massively multiplayer online role-playing games. Journal of Systems and Software, 2017; 134:54-75.

[9] Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns — Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

[10] Prechelt L, Unger B, Tichy WF, Brossler P, Votta LG. A controlled experiment in maintenance comparing design patterns to simpler solutions. IEEE Transactions on Software Engineering, 2001; 27(12):1134–1144.

[11] Ampatzoglou A, Chatzigeorgiou A, Samaras N. Investigating the use of patterns in open-source

games. Proceedings of the Balkan Conference on Informatics, 2007.

[12] Barakat N. A Framework for integrating software design patterns with game design framework. Proceedings of the CSIE 2019, Cairo, Egypt, 2019.

[13] Li T, Fei Y. Analysis and Design of 3D Game Engine Based on Design Patterns. Computers and Modernization, 2009; 135-142.

[14] Michou O, Vamvaka M, Ampatzoglou A. AynOmel 3D: A Pattern-Based Game Framework. Proceedings of the 11th Panhellenic Conference on Informatics (PCI '07), Patras, Greece, 2007.

[15] Nguyen DZ, Wong SB. Design Patterns for Games. Proceedings of the Special Interest Group on Computer Science Education (SIGCSE'02), Association of Computing Machinery, Cincinnati, Kentucky, 2002; 126- 130.

[16] Adams E, Dormans J. Game mechanics: advanced game design, New Riders, 2012.

[17] LeBlanc M. Tools for creating dramatic game dynamics. The Game Design Reader: Rules for Paly Anthology, MIT Press, Salen, K., Zimmerman, E. (Eds.), 2006; 438-459.

[18] Callele D, Dueck P, Wnuk K, Hynninen P. Experience Requirements in Video Games. Proceedings of the 23rd International Requirements Engineering Conference, 2015; 324-333.

[19] Zagal PJ, Mateas M, Fernández-Vara C, Hochhalter B, Lichti N. Towards an Ontological Language for Game Analysis. Proceedings of the International DiGRA Conference, 2005; 3-14.

[20] Falstein N. The 400 Project. 2021. Accessed April 5, 2021. http://www.theinspiracy.com/the-400-project.html

[21] Arnab S, Lim T, Carvalho MB, Bellotti F, de Freitas S, Louchart S, Suttie N, Berta R, De Gloria A. Mapping learning and game mechanics for serious games analysis. British Journal of Educational Technology, 2015; 46(2):391-411.

[22] Schmitz B, Specht M, Klemke R. An analysis of the educational potential of augmented reality games for learning. Proceedings of the 11th Annual World Conference on Mobile and Contextual Learning (mLearn), Helsinki, Finland, 2012.

[23] Alhusain S, Coupland S, John R, Kavanagh M. Towards machine learning based design pattern recognition. Proceedings of the 13th UK Workshop on Computational Intelligence (UKCI '13), IEEE Computer Society, Guildford, United Kingdom, 2013; 244–251.

[24] Ampatzoglou A, Chatzigeorgiou A. Evaluation of object-oriented design patterns in game development. Information and Software Technology, Elsevier 2007; 49(5):445-454.

[25] Polancec D, Mekterovic I. Developing MOBA games using the Unity game engine. Proceedings of the 40th International Convention on Information and Communication Technology, Electronics and Microelectronics, 2017; 1050-1515.

[26] Bucher N. Introducing design patterns and best practices in unity. Proceedings of the SouthEast Conference (ACMSE), 2017; 243-24.

[27] da Cruz Silva DS, Schots M, Duboc L. Fostering Design Patterns Education: An Exemplar Inspired in the Angry Birds Game Franchise. Proceeding of the XVIII Brazilian Symposium on Software Quality (SBQS'19), 2019.

[28] Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In: Broy M., Denert E. (eds) Pioneers and Their Contributions to Software Engineering. Springer, Berlin, Heidelberg, 2001.

[29] Gestwicki P, Sun FS. Teaching Design Patterns Through Computer Game Development. Journal on Educational Resources in Computing, 2008; 8(1).

[30] Gómez-Martín MA, Jiménez-Díaz G, Arroyo J. Teaching design patterns using a family of games. Proceedings of the 14th annual ACM SIGCSE conference on Innovation and echnology in Computer Science Education (ITiCSE '09). Association for Computing Machinery, New York, NY, USA, 2009; 268–272.

[31] Pillay N. Teaching design patterns. Proceeding of the Southern African Computer Lecturers Association Conference (SACLA). 2010; 1–4.

[32] Ghazarian A. Work in progress: Transitioning from novice to expert software engineers through design patterns: Is it really working?. Proceedings of the Frontiers in Education Conference. 2012; 1-2.

[33] Kounoukla X, Ampatzoglou A, Anagnostopoulos K. Implementing Game Mechanics with GoF Design Patterns. Proceedings of the 20th Pan-Hellenic Conference on Informatics (PCI '16). ACM, New York, NY, USA, 2016; 30.

[34] Qu J, Song Y, Wei Y. Design Patterns Applied for Game Design Patterns. Proceedings of the Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, Las Vegas, 2016; 351-356.

[35] Qu J, Song Y, Wei Y. Design Patterns Applied for Networked First Person Shooting Game Programming. Proceedings of the Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, Las Vegas, 2014; 1-6.

[36] Fiebelkorn N, Clark B, Sung K. Would gamers collaborate given the opportunity? Proceedings of the ACM International Conference Proceeding Series, 2018.

[37] Bashiiui H. Search Keywords - Relationship between Pipes-and-Filters and Decorator design patterns. 2006. Accessed January 15, 2021. https://www.codeproject.com/Articles/13135/Search-Keywords-Relationship-between-Pipes-and-Fil

[38] Wieringa RJ. Design Science Methodology for Information Systems and Software Engineering, Springer-Verlag Berlin Heidelberg, 2014.

[39] Ampatzoglou A, Michou O, Stamelos I. Building and mining a repository of design pattern instances: Practical and research benefits. Entertainment Computing, Elsevier, 2013; 4(2).

[40] Ampatzoglou A, Gkortzis A, Deligiannis I, Stamelos I. A methodology on extracting reusable software components from Open Source Games. Proceedings of the 16th International Academic MindTREK Conference, ACM, Tampere, Finland, 2012; 93-100.

[41] Ampatzoglou A, Gkortzis A, Charalampidou S, Avgeriou P. An Embedded Multiple-Case Study on OSS Design Quality Assessment across Domains. Proceedings of the ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM' 13), IEEE Computer Society, USA, 2013; 255-258.

[42] Tsantalis N, Chatzigeorgiou A, Stephanides G. Predicting the Probability of Change in Object-Oriented Systems. Transactions on Software Engineering, IEEE Computer Society, 2005; 31(7):601-614.

[43] Kerievsky J. Refactoring to Patterns, Addison-Wesley, 1st Edition, 1994.

[44] Wohlin C, Runeson P, Host M, Ohlsson MC, Regnell B, Wesslen A. Experimentation in Software Engineering, Kluwer Academic Publishers, Boston / Dordrecht / London, 2000.

[45] Kosti MV, Georgiadis KK, Adamos DA, Laskaris N, Angelis L. Towards an affordable brain computer interface for the assessment of programmers' mental workload. International Journal of Human-Computer Studies, Elsevier, 2018; 115:52–66.

[46] M. Gatrell, S. Counsell, Design patterns and fault-proneness a study of commercial C# software,

in: Proc. Fifth Int. Conf. Res. Challenges Inf. Sci. (RCIS ’11), IEEE, Gosier, France, 2011: pp. 1–8. doi:10.1109/RCIS.2011.6006827.

[47] L. Aversano, L. Cerulo, M. Di Penta, Relationship between design patterns defects and crosscutting concern scattering degree: an empirical study, IET Softw. 3 (2009) 395. doi:10.1049/iet-sen.2008.0105.

[48] M. Vokác, W. Tichy, D.I.K. Sjøberg, E. Arisholm and M. Aldrin, “A Controlled Experiment Comparing the Maintainability of Programs Designed with and without Design Patterns - A Replication in a Real Programming Environment”, Empirical Software Engineering, Springer, 9 (2003), pages 149-195.

[49] A. Ampatzoglou, A. Kritikos, E.-M. Arvanitou, A. Gortzis, F. Chatziasimidis, I. Stamelos, An empirical investigation on the impact of design pattern application on computer game defects, in: Proc. 15th Int. Acad. MindTrek Conf. Envisioning Futur. Media Environ. (MindTrek ’11), ACM, Tampere, Finland, 2011: pp. 214–221. doi:10.1145/2181037.2181074.