Design Patterns For Implementing
Game Mechanics

Author: Xeni-Christina Kounoukla
Supervisor: Dr. Apostolos Ampatzoglou

Thessaloniki, May 2016
Abstract

Typically, developing computer games is a very complicated task and as a result, many games suffer from poor design and weakened software quality attributes like maintainability. On the other hand, according to the software engineering literature, the application of GoF design patterns in game development is beneficial concerning certain design-time qualities. The aim of this study is to introduce basic instantiations of game mechanics through GoF patterns and evaluate the benefits of this mapping. Having specified implementations on how to instantiate a game design pattern is expected to increase their usability in practice. Nine mappings were identified and a case study on open source software Java games was performed in order to explore the applicability of the approach and evaluate the resulting benefits. Additionally, alternative, non-pattern employing implementations were developed, in order to be directly compared to the GoF pattern involving instances regarding software quality. The findings suggest that the application of GoF design patterns in the implementation of game mechanics improves certain aspects of software quality, while it might, on the other hand, weaken some other. Therefore, it is necessary for a designer to consider several factors, such as the most desired quality attributes and perform a multi-criteria decision analysis. The results provide pointers to interesting areas for future work, as well as advices on design-time decisions in game development.
Acknowledgements

Firstly, I would like to express my sincere gratitude to my dissertation supervisor Dr. Apostolos Ampatzoglou, whose expertise, patience, continuous support, and generous guidance made it possible for me to conduct this project and related research. His immense knowledge and persistent help inspired and motivated me. He has supported my effort in uncountable ways. It was a great pleasure working with him.

I am thoroughly grateful to Dr. Tasos Stylianou for his insightful comments, suggestions, and encouragement during the course of this project.

Last but not least, I would also like to thank all my Professors who put their faith in me. Their knowledge and commitment to the highest standards were a constant source of inspiration.
# Table of Contents

Abstract .......................................................................................................................... 2

Acknowledgements ...................................................................................................... 3

List of Figures and Tables .............................................................................................. 6

1. Introduction .................................................................................................................. 7
   1.1 Motivation .................................................................................................................. 8
   1.2 Aim and Objectives ............................................................................................... 8

2. Literature Review ....................................................................................................... 10
   2.1 Background Concepts ............................................................................................ 10
      2.1.1 What Is a Design Pattern? ............................................................................... 10
      2.1.1 Game Design Patterns .................................................................................... 10
      2.1.2 GoF Design Patterns ....................................................................................... 11
      2.1.3 Software Quality Attributes ........................................................................... 12
   2.2 Related Work and Contribution ............................................................................. 14

3. Research Methodology ............................................................................................... 16
   3.1 Research Questions .................................................................................................. 16
   3.2 Cases and Units of Analysis (Sampling) .................................................................. 16
   3.3 Data Collection ....................................................................................................... 17
   3.4 Data Analysis ......................................................................................................... 18

4. Findings and Results ................................................................................................... 19
   4.1 Mappings Between Game and GoF Design Patterns ............................................. 19
      4.1.1 Implementation of Turn-based Games with Template Method ...................... 19
      4.1.2 Implementation of Agents with Strategy ......................................................... 20
      4.1.3 Implementation of Power-Ups with Visitor ...................................................... 21
      4.1.4 Implementation of Game World with Composite .......................................... 22
4.1.5 Implementation of Levels with State .......................................................... 23
4.1.6 Implementation of Progress Indicator with Observer ............................ 24
4.1.7 Implementation of Units with Abstract Factory ..................................... 25
4.1.8 Implementation of Movement with Strategy .......................................... 26
4.1.9 Implementation of Varied Gameplay with State .................................... 27
4.2 Occurrences of Mappings in OSS Java Games ............................................ 28
4.3 Quality Attributes in Pattern Solutions and Non-pattern Solutions .......... 29
5. Discussion and Analysis of Findings .............................................................. 32
  5.1 Interpretation of Results ............................................................................ 32
    5.1.1 Interpretation of Results Regarding Occurrences of Mappings .......... 32
    5.1.2 Interpretation of Results Regarding Quality Assessment ................. 32
  5.2 Implications for Researchers and Practitioners ....................................... 34
6. Threats to Validity and Limitations .............................................................. 36
7. Conclusion ................................................................................................... 37
8. Bibliography ............................................................................................... 39
List of Figures and Tables

Figure 1. Template Method / Turn-based Games ........................................20
Figure 2. Strategy / Agents ........................................................................21
Figure 3. Visitor / Power-Ups ......................................................................22
Figure 4. Composite / Game World .................................................................23
Figure 5. State / Levels .................................................................................24
Figure 6. Observer / Progress indicator ..........................................................25
Figure 7. Abstract Factory / Units .................................................................26
Figure 8. Strategy / Movement ......................................................................27
Figure 9. State / Varied Gameplay .................................................................28
Table I: Occurrences of Mappings in OSS Java Games ...................................29
Table II: Implementation of Strategy / Movement vs. Alternative Implementation ...30
Table III: Implementation of Abstract Factory / Units vs. Alternative Implementation ..................................................30
Table IV: Implementation of Strategy / Agents vs. Alternative Implementation ....31
Table V: Implementation of State / Varied Gameplay vs. Alternative Implementation ..........................................................31

6
1. Introduction

Currently, computer games appear to be thriving and computer game development is considered to be one of the most modern and fast growing software industries in the worldwide economy. Typically, developing computer games is a very complicated task to accomplish, and in combination with the evolving nature of this kind of software, it requires the involvement of professionals from many different fields of computer science. In addition to this, game software engineering differentiates from classical software engineering since most of the time, computer games have limited lifecycles and are developed in a smaller time period compared to other software products. As a result, many games suffer from poor design and weakened software quality attributes like maintainability and functionality. Therefore, the interest for developing engineering methodologies for game development has steadily been growing over the last few years and has evolved into a field of great interest (Ampatzoglou and Stamelos, 2010).

In the software engineering literature, the term pattern is used to characterize any recurring solution to a common software design problem. In the context of computer game development, patterns appear in two major forms: (a) GoF (Gang of Four) design patterns that are introduced at the detailed-design and implementation phase, so as to enhance games' maintainability (Gamma et al., 1995), and (b) game design patterns, that correspond to reusable parts of game logic, which is the part that holds the game's story. The latter are also called game mechanics (Bjork and Holopainen, 2005). This study includes the introduction of basic instantiations of game mechanics through GoF patterns, the investigation of the frequency at which game mechanics are implemented with GoF patterns in open-source software games, and the resulting potential benefits.

This paper is structured in seven sections. In the rest of this section, the rationale for this study is explained and the aims and objectives are presented. In section 2, background concepts and related work are discussed. In section 3, the methodologies used to complete this project are presented. In section 4, the results of the study are presented, while in section 5, the study's findings and their implications for
researchers and practitioners are discussed. In section 6, possible threats to validity and limitations are analyzed. Finally, section 7 is the conclusion.

1.1 Motivation

Since the computer game industry is one of the most strong and profitable ones, the need of useful tools for game design is critical. In 2005, Bjork and Holopainen introduced game design patterns, which are, in fact, a tool for understanding, analyzing, and creating games. More specifically, game design patterns is a collection of design choices possible in games and they correspond to recurring parts of gameplay, which is undoubtedly the most essential part of game design. Game design patterns are frequently occurring in many games since, by definition, they represent recurring issues in game mechanics implementation (Bjork and Holopainen, 2005).

However, at this point, game design patterns lack sample implementations. Therefore, although a solution is described in a higher level there is a lack of guidance at the implementation level. Having specified implementations on how to instantiate game design patterns is expected to increase their usability in practice.

The application of GoF design patterns in game development has been proven to be beneficial concerning design-time qualities like maintainability (Ampatzoglou and Chatzigeorgiou, 2007). Therefore, the mapping of game and GoF design patterns is expected to be beneficial to both researchers and practitioners since using such a mapping would increase the design-time qualities of the software. By taking into account the frequency of occurrence for game design patterns and the enhanced quality that GoF patterns offer, an increase in the overall quality is expected.

1.2 Aim and Objectives

The aim of this study is to evaluate the benefits of implementing game design patterns with GoF design patterns. To achieve this goal, we have set the following objectives: (a) define a list of game design patterns that are candidates for implementation with GoF, (b) explore various GoF patterns so as to identify a list of candidate implementations for each game design pattern, (c) provide exemplary mappings between game and GoF design patterns along with the corresponding code, (d) perform a case study on OSS games so as to explore the applicability of the approach,
and finally, (e) evaluate the effect GoF design patterns have, when game mechanics are instantiated through them, by comparing pattern solutions to non-pattern, alternative ones.
2. Literature Review

The goal of this section is two-fold. Firstly, background concepts, regarding game design patterns, GoF design patterns, and software quality assessment models are described. Additionally, related work concerning the topic of this study is presented along with the points in which this study differentiates from it.

2.1 Background Concepts

2.1.1 What Is a Design Pattern?

In the 1970s, Alexander, et al. formulated the concept of design patterns in their book "A Pattern Language: Towns, Buildings, Construction" based on their experience of architectural designs. They describe it as follows: "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice." (Alexander, Ishikawa and Silverstein, 1977). In 1995, to fit the needs of software engineering, the concept of patterns was transformed by Gamma, et. al., the so-called GoF (Gang of Four, Gamma, Helms, Johnson and Vlissides). Software design patterns are regarded as a way of codifying expert knowledge schemas that have been proven to provide efficient solutions to common recurring problems that are encountered in software design practice (Gamma et al., 1995).

2.1.1 Game Design Patterns

The existence of a common language is essential to the discussion of different aspects of gameplay. A solution to this lack of terminology was offered by Bjork and Holopainen in 2005 in their book "Patterns in Game Design", with the introduction of game design patterns. Each one of these patterns describes in a high, abstract level, a recurring part of the interaction possible in games and combined together with other patterns, they describe the possible storyline in a game. In other words, game design patterns describe design choices or emergent features that reoccur in many games. They constitute a useful guide that aims to support not only analytic work, but also creative design work. However, the use of game design patterns does not guarantee
creation of good or interesting games; their goal is to enhance game developers' thinking, creativity, and communication.

The template used for the description of game design patterns consists of the name of the pattern, its core definition, and general description. Furthermore, it consists of directions on how to apply the pattern, and a description of its consequences in gameplay. Finally, there is a section that lists the relations that exist between the described pattern and other patterns.

The collection of game design patterns is categorized based upon the component framework. As a result, there are game design patterns for the following: (a) game elements, (b) resources and resource management, (c) information, communication, and presentation, (d) actions and events, (e) narrative structures, predictability, and immersion patterns, (f) social interaction, (g) goals, (h) goal structures, (i) game sessions, (j) game mastery and balancing, and (k) meta games, re-playability, and learning curves.

To sum up, game design patterns are an excellent guide that promotes game designers' inspiration by helping them avoid missing possible ideas or getting stuck in the same thoughts. They constitute a creative design tool that enables design verification, problem-solving in interaction design, and game analysis (Bjork and Holopainen, 2005).

2.1.2 GoF Design Patterns
Most developers will admit that object-oriented software is hard to design. However, experienced programmers do not solve every problem from first principles, but instead, they reuse solutions that have worked well in the past. GoF design patterns refer to the design patterns introduced in 1995 by Gamma, et. al. The 23 GoF design patterns that were catalogued, are regarded as one of the most influential and important source for object-oriented design theory and practice in the field of software engineering. Since then, it is hard to find an object-oriented system that does not make use of them.
In particular, the GoF design patterns provide flexible, elegant and reusable solutions to specific design problems that commonly occur to object-oriented systems. By expressing proven techniques, they prevent developers from using design alternatives that might compromise reusability and maintainability of the software. In general, GoF design patterns promote programming to an "interface" and not an "implementation" and favor composition over inheritance.

Most of the times, GoF patterns are classified based on their purpose which, in fact, reflects what a pattern does. As a result, a pattern can be classified as creational, structural, or behavioral. Creational patterns concern the process of object creation and they promote flexibility in deciding which objects need to be created for a given case. On the other hand, structural patterns deal with the composition of classes or objects. Finally, behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility. Design patterns are also classified based upon another criterion, which is called scope and specifies whether the pattern is applied on class or object level (Gamma et al., 1995).

2.1.3 Software Quality Attributes
In this study, QMOOD was used as a reference model for discussing the effect of design patterns on software quality. QMOOD is an improved hierarchical model for object-oriented quality assessment, with lower-level metrics that are well-defined only in terms of design characteristics.

The first level of QMOOD describes six quality attributes, which are abstract concepts that describe desirable characteristics of software design. They are listed below along with a small description:

- **Functionality**: reflects classes that satisfy needs.
- **Effectiveness**: reflects the satisfaction of the desired functionalities.
- **Understandability**: reflects the ease of learning.
- **Extendibility**: reflects the ability of the system to incorporate new requirements.
• **Reusability**: reflects the ability of the solution to be reapplied to a new problem.

• **Flexibility**: reflects the ability of the product to be adapted to a new problem.

The third level of QMOOD describes eleven software design metrics that state exactly what to count in class definitions and relationships. They are listed below along with their descriptions:

• **Design Size in Classes (DSC)**: this metric counts the number of classes in the design.

• **Number of Hierarchies (NOH)**: this metric counts the number of class hierarchies in the design.

• **Average Number of Ancestors (ANA)**: this metric value signifies the average number of classes from which a class inherits information.

• **Data Access Metric (DAM)**: this metric value represents the ratio of the number of private or protected attributes to the total number of attributes declared in a class.

• **Direct Class Coupling (DCC)**: this metric counts the number of different classes that a class is related to.

• **Cohesion Among Methods of Class (CAM)**: this metric reflects the relatedness among methods of a class based upon the parameter list of the methods.

• **Measure of Aggregation (MOA)**: this metric is the ratio of the number of methods inherited by a class to the total number of methods that are accessible by member methods of the class.

• **Number of Polymorphic Methods (NPM)**: this metric value signifies the number of public methods in a class.

• **Class Interface Size (CIS)**: this metric counts the number of public methods in a class.

• **Number of Methods (NOM)**: this metric value represents the number of all the methods defined in a class.
Quality attributes are calculated by equations of weighted values of certain software design metrics (Bansiya and Davis, 2002).

### 2.2 Related Work and Contribution

Even though when GoF patterns were originally introduced, they were not linked to specific quality attributes, it is a common belief among researchers and practitioners that they positively affect them. A plethora of studies and researches has been done on this topic.

In 2007, Ampatzoglou and Chatzigeorgiou evaluated the use of design patterns in game development. Their findings suggest that the application of design patterns tends to reduce complexity and coupling and increase the cohesion of the software and thus, they conclude that the use of design patterns in game programming should be encouraged (Ampatzoglou and Chatzigeorgiou, 2007).

Khomh and Guéhéneuc (2008) evaluated the effect of all GoF patterns on software quality attributes through a survey. The outcome of their study suggested that design patterns do not always have a positive impact in quality attributes and that they should be used with caution by developers, especially novice ones (Khomh and Guéhéneuc, 2008).

In 2012, Zhang and Budgen performed a systematic literature review on the effectiveness of software design patterns. What their study has shown is that design patterns have been subjected only to limited empirical evaluation. However, there was some qualitative support that the use of patterns increases maintainability (Zhang and Budgen, 2012).

Furthermore, in 2012, Ampatzoglou, Frantzeskou and Stamelos proposed an analytical methodology for comparing design patterns to alternative designs. Their study highlights existing threshold that when surpassed, design pattern becomes more or less beneficial than alternative design. The authors suggest that multi-criteria analysis is needed when deciding over the use of pattern or non-pattern designs, since typically, while patterns improve certain software qualities, they may, also, weaken some others (Ampatzoglou, Frantzeskou, and Stamelos, 2012).
In 2013, Ampatzoglou, Charalampidou and Stamelos, summarized the research state of the art on GoF design patterns, concerning the impact of pattern employment on software quality attributes. The outcome of their study confirmed the positive effect of patterns application regarding software quality. However, as they state, sometimes design patterns enhance one quality attribute in the expense of another (Ampatzoglou, Charalampidou and Stamelos, 2013).

What is more, in 2015, Ampatzoglou et al. performed a case study to investigate the effect of GoF design patterns regarding stability, which means the resistance of a certain class to changes that are propagated from other classes. Their findings indicated that classes that play only one role in a GoF design pattern occurrence are more stable than the ones with none or multiple roles. In addition to this, in their study, it was observed that the use of either association or aggregation for establishing object communication affects the stability, with classes playing the Composite role being more stable than classes playing the Aggregate role (Ampatzoglou et al., 2015).

The main difference between the related work and this study is the fact that in the software engineering literature there is lack of evidence regarding the effect of the application of GoF design patterns to specific game mechanics, as far as software quality is concerned. What is more, game design patterns describe solutions in the abstract level and there is a lack of guidance at the implementation level in the literature. This study aims to contribute to the research concerning these topics.
3. Research Methodology

In order to evaluate the benefits of implementing game design patterns with GoF design patterns, a case study was performed on Java open source software (OSS) games. The case study has been designed and presented according to the guidelines provided by Runeson et al. (2009).

3.1 Research Questions

In order to achieve the goals, which were stated in section 1, three research questions (RQs), that serve as a guide to the study design and the reporting of findings, were formulated:

*RQ1:* What are the possible mappings between game and GoF design patterns?

*RQ2:* How often these mappings occur in practice in OSS Java games?

*RQ3:* Are there any differences, with respect to software quality attributes, between pattern solutions and non-pattern solutions, as far as the implementation of game mechanics is concerned?

3.2 Cases and Units of Analysis (Sampling)

The cases and units of analysis that were used to answer the aforementioned research questions are extracted from open source software games, all written in Java. More specifically, we analyzed 30 open source standalone games. To gather our cases the decision to use the software engineering repository, named percerons, was made. This repository documents designed pattern occurrences and it was created in 2013 (Ampatzoglou, Michou and Stamelos, 2013). In order to guarantee the data validity, game and GoF design pattern occurrence validation process was performed before data extraction. What is more, in cases where a .jar file was not included in the download source of a game, the game was excluded from the case study, because of the fact that, the tools that were used, need the .jar file of a project in order to perform design pattern detection and static quality analysis. In addition to this, in cases where the implementation of a game mechanic employed more than one GoF design pattern,
the game was also excluded from the case study, because of the fact that pattern interaction (coupling) might influence the results.

Finally, as far as the mapping between game and GoF design patterns is concerned, descriptive research methods and literature review were used.

**3.3 Data Collection**

Before proceeding to the case study on open source Java games, identification of possible mappings between game and GoF design pattern was needed to be done. First of all, a list of game design patterns that are candidates for implementation with GoF patterns was defined. The next step was to explore several GoF design patterns so as to identify a list of candidate implementations for each game design pattern. After that, exemplar class diagrams of the identified mappings between GoF and game design patterns were created, along with their corresponding sample implementations written in Java. The creation of the class diagrams was done using open source software; that is the ArgoUML tool. ArgoUML is a leading open source UML modeling tool. It includes support for all standard UML 1.4 diagrams and it runs on any Java platform. ArgoUML tool can be downloaded from the web\(^1\). The class diagrams are presented and described in section 4.

For pattern detection the tool created by Tsantalis et al. (2006) was used, along with the online tool suite, called percerons, that is provided by Ampatzoglou et al. (2012). The former tool can be downloaded from the web\(^2\), while the tool suite percerons is available online\(^3\). Both tools that were employed are capable of automatically identifying GoF design pattern occurrences in a given Java project.

---

\(^1\) [http://argouml.tigris.org/](http://argouml.tigris.org/)

\(^2\) [http://java.uom.gr/~nikos/pattern-detection.html](http://java.uom.gr/~nikos/pattern-detection.html)

\(^3\) [http://www.percerons.com/](http://www.percerons.com/)
3.4 Data Analysis

The data analysis phase of our case study involved the use of the QMOOD software quality model as a reference model for evaluating the effect of design patterns on software quality (Bansiya and Davis, 2002). Background information on this model and its quality attributes and software metrics is presented in section 2.

In order to be able to evaluate the possible benefits of game mechanics implementations with GoF design patterns, alternative, non-pattern solutions were developed. As a result, for each pattern occurrence that was extracted from cases and concerned the application of a GoF design pattern on a game mechanic, an alternative implementation, that did not employ a GoF pattern, was developed. Suggestions made by Ampatzoglou, Charalampidou, and Stamelos (2013) were taken into account to complete this task.

In order to assess the benefits of pattern solutions compared to non-pattern solutions as far as the implementation of game mechanics is concerned, the percerons client-server application was used. This application can be downloaded from the web\(^4\). Percerons client is able to perform the static quality analysis and calculate the values of software metrics and quality attributes, as they were introduced in the QMOOD model. Results are stored in a local database. On the other hand, percerons server is responsible for retrieving the results of static analysis from the local database, processing them and visualizing them through the web browser. Specific details on how to use this application exist in the webpage where it can be downloaded (Ampatzoglou et al., 2012). Finally, to determine a particular quality attribute, only the classes that are part of the GoF pattern or the game mechanic in concern, were taken into account. In all the assessed game design implementations, the value of each quality attribute is calculated as the average quality attribute value of all the participating classes.

\(^4\) http://www.percerons.com/download.html
4. Findings and Results

In this section all the relevant results of the case study that was performed, are presented. The findings are organized by research question, as it was mentioned before. It is important to be noted, that this section only concerns the presentation of the findings. An analytic discussion about the interpretation of results, and their implications for researchers and practitioners is provided in section 5.

4.1 Mappings Between Game and GoF Design Patterns

RQ1: What are the possible mappings between game and GoF design patterns?

In the following sub-sections, the mappings that were identified between game and GoF design patterns are presented along with a brief analysis to help the reader better understand the concepts and methods that are involved. For each one of the mappings, an exemplar class diagram was created to illustrate the case, along with a sample code implementation, written in Java. Class names start with an uppercase letter, whereas methods' and attributes' names start with a lowercase letter. What is more, abstract classes and methods are in italics.

4.1.1 Implementation of Turn-based Games with Template Method

The intent of the Template Method design pattern is to define the skeleton of an algorithm in a method, which is called template method. Some steps are deferred to subclasses. The pattern lets subclasses to redefine certain steps of an algorithm without affecting the algorithm's structure (Gamma et al., 1995). Turn-based games is a game pattern that refers to games in which the players take turns to make their move (Bjork and Holopainen, 2005).
The class diagram created for this mapping consists of an abstract Game class whose member function gameloop() plays the role of the template method and defines the structure of the algorithm that implements the game loop. Monopoly, GameOfLife, and Chess classes extend the abstract class and reflect different turn-based games. Each one of them, carries different implementations for initializeGame(), nextmove(), endOfGame(), and changeTurn() operations in accordance to the different gameplay logic of each turn-based game.

4.1.2 Implementation of Agents with Strategy

The intent of Strategy design pattern is to define a family of algorithms and encapsulate each one of them, making them interchangeable within the family. Strategy enables an algorithm's behavior to be chosen at runtime (Gamma et al., 1995). Agents refer to a game's entities that simulate players. In other words, Agents have the role of players but their behavior is controlled by the game system. Most of
the times, artificial intelligence algorithms implement the behavior of the Agents game design pattern (Bjork and Holopainen, 2005).

The class diagram above, that was created to reflect the mapping of these design patterns, consists of the abstract class Strategy that is responsible for the behavior of an AI player of the chess game. Its concrete subclasses define different implementations of the AI algorithm which decides the next move of the AI player depending on the level of the game. The class that plays the Context role in the GoF pattern is the AICheapPlayer class and it represents the agent (i.e. the AI player).

4.1.3 Implementation of Power-Ups with Visitor

Generally, the Visitor design pattern represents an operation that is performed on the attributes of an object. The pattern allows the definition of a new operation without the need to change the classes of the elements on which it operates (Gamma et al., 1995). The Power-Ups game mechanic concerns game elements that when they are picked-up by the player, they give him advantages. Usually, in games, these advantages are time-limited (Bjork and Holopainen, 2005).
Figure 3. Visitor / Power-Ups

The class diagram created to illustrate this mapping contains the abstract PowerUpVisitor class and its concrete classes PowerUpSpeed and PowerUpVisibility. The PowerUpSpeed class defines the implementation of the alterBehavior() method which changes the speed attribute of the character, whereas the PowerUpVisibility class defines the implementation of the alterBehavior() function that changes the visibility attribute of the character. The classes that play the Element role in the Visitor pattern, are the abstract class Character and its concrete class Player. The player’s attributes speed and visibility change according to the type of power-up the visitor reflects each time.

4.1.4 Implementation of Game World with Composite

The Composite design pattern describes composition of objects into tree structures to represent part-whole hierarchies. The group of objects is treated as if it was a single instance of an object and thus, the pattern lets clients treat composition of objects and individual objects uniformly (Gamma et al., 1995). The game design pattern Game World refers to the environment in which a gameplay or at least a part of it takes place. Usually, in Game World the spatial relationships of game elements are important (Bjork and Holopainen, 2005).
The class diagram above, that was created for this mapping, represents an example of the game board design of the actual game Monopoly. Tile class plays the role of the Composite in the pattern structure, while Avatar, House and Hotel are the Leafs. A Tile object contains its text content which is an object of a class that does not participate in the pattern, and thus, it is not depicted on the class diagram. Besides that, it contains the avatar of the player or any hotels and houses the player owns. In this sample implementation of the mapping, the game board of Monopoly is considered to be the client that handles uniformly all the game components such as tiles (composite object) and avatars, houses and hotels (individual objects).

4.1.5 Implementation of Levels with State

The intent of the State GoF pattern is to allow an object to change its behavior every time its internal state changes. When implementing this pattern, the object will appear to alter its class. The State pattern's structure closely resembles the one of Strategy pattern (Gamma et al., 1995). The Levels game mechanic refers to parts of the game in which players have the ability to act until a certain goal has been reached. Usually, the differences between Levels concern the content, aesthetics, or both (Bjork and Holopainen, 2005).
The context role of the State pattern occurrence in the related class diagram above plays the class Game, and its altering state is the attribute called state, which is of type Level. The latter abstract class represents the state (i.e. the level of the game), while the derived concrete classes, Level1, Level2, and Level3 define the different implementations of gameFunction() method. An object of type Game changes its behavior (through the gameFunction() operation) according to its state.

4.1.6 Implementation of Progress Indicator with Observer

The intent when implementing the Observer GoF pattern is to establish a one-to-many dependency between objects. As a result, when one object (subject) alters its state, all the subject's dependents that are called observers, are notified and updated automatically (Gamma et al., 1995). The game design pattern Progress Indicator refers to a game element that gives the player information about his current progress (Bjork and Holopainen, 2005).
In the implementation of this mapping, a simple game map that consists of several checkpoints is considered. The class that represents checkpoints (MapPoint) is not included in the class diagram since it does not explicitly participate in the pattern. Whenever the player reaches a checkpoint the progress indicator is updated accordingly, indicating the progress of the player's navigation through the map of the game. The class Map plays the role of the subject while class ProgressIndicator is its dependent concrete observer.

4.1.7 Implementation of Units with Abstract Factory

The Abstract Factory design pattern provides an interface that is responsible for creating families (abstract factories) of either related, or dependent objects (products), without explicitly specifying their concrete classes. The client creates a concrete implementation of the abstract factory and by using the abstract class of each factory, it creates concrete objects (Gamma et al., 1995). The game mechanic Units refers to groups of game elements that may have different actions and attributes associated with them. They are under the player's control and enable the player to perform actions that influence the Game World (Bjork and Holopainen, 2005).
In the implementation of this mapping, Units are considered as game elements that have the shape of square or circle and may be colored either in blue or red. ShapeFactory and ColorFactory are the concrete classes derived from the abstract AbstractFactory class. They define implementations of operations getColor() and getShape(). The former class is responsible for the shaping the products (Units), whereas the latter is responsible for coloring them.

4.1.8 Implementation of Movement with Strategy

The game design pattern Movement refers to the action of moving game elements in the game world. In general, Movement allows players to move game elements into desired positions and control or explore the game world (Bjork and Holopainen, 2005).
To create the class diagram above, we considered a game where the game elements snake, character and bird have the ability to move but their type of movement is different for each one of them. For instance, the snake slithers, the character runs and the bird flies, and thus, three different implementations of an algorithm that programs the action of moving, are needed. These implementations are defined in the concrete strategy classes, Snake, Character, and Bird within the move() operations. The role of the context in this pattern instance plays the class Movement.

4.1.9 Implementation of Varied Gameplay with State

The game mechanic Varied Gameplay reflects the variety in gameplay either in a single game session, or between different game sessions. For the games to be interesting, a certain level of Varied Gameplay should always be provided (Bjork and Holopainen, 2005).
Due to the fact that Varied Gameplay constitutes in practice a very large game pattern, a simple example was considered to create the class diagram. The fact that a human player is able to play the game either against a human opponent or an AI opponent is considered as a simple variation of the Varied Gameplay pattern. As a result, the class diagram consists of the abstract class VariedGamePlay (state) and its concrete subclasses HumanVsHuman and HumanVsAI (concrete states). The class Game plays the role of the context.

4.2 Occurrences of Mappings in OSS Java Games

RQ2: How often these mappings occur in practice in OSS Java games?

This sub-section tries to answer the RQ2 that was placed in section 3 and concerns the occurrences of the previously presented mappings between game and GoF design patterns, in real open source software (OSS) games written in Java. A more thorough discussion and analysis of the findings can be found in section 5. The results of the case study performed, as far as the number of mappings' occurrences is concerned, are presented in the following table. All related games can be downloaded from the percerons repository by following the appropriate download source links. Information about the online tool suite percerons can be found in section 3.
<table>
<thead>
<tr>
<th>Mappings</th>
<th>Number of occurrences</th>
<th>Java game</th>
</tr>
</thead>
<tbody>
<tr>
<td>State / Levels</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>State / Varied Gameplay</td>
<td>1</td>
<td>Infothello</td>
</tr>
<tr>
<td>Strategy / Agents</td>
<td>1</td>
<td>Infothello</td>
</tr>
<tr>
<td>Observer / Progress Indicator</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>Visitor / Power-Ups</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>Strategy / Movement</td>
<td>1</td>
<td>Arcadiban</td>
</tr>
<tr>
<td>Composite / Game World</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>Abstract factory / Units</td>
<td>1</td>
<td>DragonChess</td>
</tr>
<tr>
<td>Template Method / Turn-based Games</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>

Table I: Occurrences of Mappings in OSS Java Games

### 4.3 Quality Attributes in Pattern Solutions and Non-pattern Solutions

**RQ3:** Are there any differences, with respect to software quality attributes, between pattern solutions and non-pattern solutions as far as the implementation of game mechanics is concerned?

In this subsection, the results concerning RQ3 are presented. In order to evaluate the impact of the application of GoF design patterns on game mechanics regarding quality attributes, alternative implementations that do not employ GoF patterns were developed for each one of the mapping occurrences that were presented in the previous subsection. In the following tables the six quality attributes (according to the QMOOD software quality assessment model) of the real mapping occurrences are demonstrated along with the quality attributes concerning the alternative, non-pattern implementations.
### Table II: Implementation of Strategy / Movement vs. Alternative Implementation

<table>
<thead>
<tr>
<th></th>
<th>Arcadiban game: Strategy / Movement</th>
<th>Arcadiban game: Alternative implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functionality</td>
<td>1.54</td>
<td>3.33</td>
</tr>
<tr>
<td>Effectiveness</td>
<td>0.57</td>
<td>0.9</td>
</tr>
<tr>
<td>Understandability</td>
<td>-2.76</td>
<td>-12.78</td>
</tr>
<tr>
<td>Extendibility</td>
<td>0.43</td>
<td>0</td>
</tr>
<tr>
<td>Reusability</td>
<td>3.28</td>
<td>7.52</td>
</tr>
<tr>
<td>Flexibility</td>
<td>0.93</td>
<td>2.0</td>
</tr>
</tbody>
</table>

### Table III: Implementation of Abstract Factory / Units vs. Alternative Implementation

<table>
<thead>
<tr>
<th></th>
<th>DragonChess game: Abstract Factory / Units</th>
<th>DragonChess game: Alternative implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functionality</td>
<td>0.56</td>
<td>0.36</td>
</tr>
<tr>
<td>Effectiveness</td>
<td>0.04</td>
<td>0.22</td>
</tr>
<tr>
<td>Understandability</td>
<td>-1.91</td>
<td>-1.62</td>
</tr>
<tr>
<td>Extendibility</td>
<td>0.13</td>
<td>0.85</td>
</tr>
<tr>
<td>Reusability</td>
<td>1.28</td>
<td>0.8</td>
</tr>
<tr>
<td>Flexibility</td>
<td>-0.12</td>
<td>0.13</td>
</tr>
<tr>
<td></td>
<td>Infothello game: Strategy / Agents</td>
<td>Infothello game: Alternative implementation</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
<td>1.1</td>
<td>1.35</td>
</tr>
<tr>
<td><strong>Effectiveness</strong></td>
<td>0.36</td>
<td>0.6</td>
</tr>
<tr>
<td><strong>Understandability</strong></td>
<td>-1.69</td>
<td>-1.85</td>
</tr>
<tr>
<td><strong>Extendibility</strong></td>
<td>0.6</td>
<td>0.25</td>
</tr>
<tr>
<td><strong>Reusability</strong></td>
<td>2.21</td>
<td>3.08</td>
</tr>
<tr>
<td><strong>Flexibility</strong></td>
<td>0.45</td>
<td>1</td>
</tr>
</tbody>
</table>

Table IV: Implementation of Strategy / Agents vs. Alternative Implementation

<table>
<thead>
<tr>
<th></th>
<th>Infothello game: State / Varied Gameplay</th>
<th>Infothello game: Alternative implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Functionality</strong></td>
<td>2.02</td>
<td>2.3</td>
</tr>
<tr>
<td><strong>Effectiveness</strong></td>
<td>0.85</td>
<td>1.2</td>
</tr>
<tr>
<td><strong>Understandability</strong></td>
<td>-2.95</td>
<td>-3.53</td>
</tr>
<tr>
<td><strong>Extendibility</strong></td>
<td>0.75</td>
<td>0.0</td>
</tr>
<tr>
<td><strong>Reusability</strong></td>
<td>3.97</td>
<td>5.3</td>
</tr>
<tr>
<td><strong>Flexibility</strong></td>
<td>1.75</td>
<td>2.75</td>
</tr>
</tbody>
</table>

Table V: Implementation of State / Varied Gameplay vs. Alternative Implementation
5. Discussion and Analysis of Findings

This section of the paper discusses the results and findings of the case study. In particular, the subsections that follow, provide an analysis and interpretation of the results and a discussion about their implications for researchers and practitioners.

5.1 Interpretation of Results

5.1.1 Interpretation of Results Regarding Occurrences of Mappings

As it is shown in Table I, according to the conducted case study, the frequency of occurrences of mappings between game mechanics and GoF design patterns is relatively low. From the nine mappings, which were presented in section 4, four of them were identified in real occurrences. What is more, only one instance for each one of the identified mappings was found.

However, the relatively low frequency of mapping occurrences can be partly justified by the exclusion criteria based on which the case study was performed. More specifically, there were games in which, even though a mapping occurrence was identified, the .jar file of the project was missing, and thus, quality assessment would not have been possible. Additionally, it is observed that many game design patterns are too large to be implemented by only one GoF design pattern, which was another exclusion criterion. On the other hand, it is also important to be noted, that some game mechanics are very small, and therefore, a GoF pattern could not have been applied on their instantiation.

5.1.2 Interpretation of Results Regarding Quality Assessment

Assessing the effect of patterns on software quality attributes is a difficult task, since admittedly, in most real games, patterns interact with each other; this is called pattern coupling. Such interactions make the evaluation of their effect on quality attributes complicated. According to Tables II, III, IV, and V, the quality assessment of pattern and non-pattern implementations of game mechanics resulted in controversial findings. As far as the implementation of the game mechanic Movement with the GoF design pattern Strategy is concerned, an appreciable increase is observed with regards to understandability and extendibility of the instance, compared to the alternative
implementation. However, quality attributes like effectiveness and flexibility are substantially reduced. The non-pattern solution, is shown to be more beneficial regarding the functionality and reusability of the code.

Furthermore, regarding the comparison between the instantiation of game mechanic Units with the application of Abstract Factory pattern and the non-pattern solution that was developed, a substantial weakening of effectiveness, understandability, extendibility, and flexibility is observed in the pattern solution. However, it is important to note that the particular Abstract Factory instance is not typical but it can be rather considered as a design pattern variant one. On the bright side, the analysis suggests that the employment of Abstract Factory in the implementation results in an appreciable improvement in functionality and reusability.

The application of the Strategy design pattern in the instantiation of the game mechanic Agents has a beneficial effect on the extendibility and understandability of the implementation compared to the alternative solution. On the other hand, the quality attributes functionality, effectiveness, and even more reusability and flexibility are negatively affected by the use of Strategy pattern, but not in a significant way.

Finally, the employment of the State design pattern in the implementation of the game pattern Varied Gameplay positively affects the understandability and extendibility compared to the non-pattern involving solution. However, effectiveness, reusability, flexibility and functionality are improved in the alternative solution compared to the pattern employing instance.

Although the number of mapping cases, that are assessed with respect to design-time qualities is small, the findings of the study indicate that while certain aspects of software quality are improved when game mechanics are implemented through GoF patterns, another aspect might be compromised. Most of the times, the application of design patterns is beneficial regarding the understandability and extendibility of the code. On the other hand, effectiveness and flexibility appear to be enhanced in the alternative instantiations. However, the small number of real game occurrences does
not allow us to draw definite conclusions about the effect of GoF patterns in game mechanics.

Quality evaluation's findings are in accordance with the related work in software engineering literature, that was presented in section 2. The application of GoF patterns is not uniformly beneficial, and there are cases where the design pattern involving implementation is not the optimal solution regarding certain aspects of software quality. In particular, State and Strategy patterns, appear to have a positive effect in understandability and extendibility, as it was expected, whereas Abstract Factory pattern is shown to improve the functionality and reusability of the code, in spite of the fact that the particular implementation that was evaluated is a pattern variant and does not have the typical pattern structure. Conclusively, there is no general answer about the effects of the patterns, since their application usually enhances some quality attributes in the expense of others.

5.2 Implications for Researchers and Practitioners

Concerning implications for researchers and practitioners, the findings of this study can be helpful for providing pointers to interesting areas for future work, as well as advices on how employment of GoF patterns in game development could be beneficial regarding certain software quality attributes.

The mappings between game and GoF design patterns along with sample instantiations, that were introduced in this study, could serve as a guide for game designers and developers not only during the design phase but also on the implementation level. Based on the quality attributes that they are most interested in, software developers can select whether to apply GoF design patterns in the instantiation of game mechanics or opt for a personalised solution. It is clear that it is necessary for a designer to consider several factors, such as the most desired quality attributes, and perform a multi-criteria decision analysis.

What is more, software engineering researchers could further investigate mapping between game and GoF design patterns, since, as it is shown in this study, they can be beneficial concerning design-time qualities. Furthermore, having a greater number of
specific implementations on how to instantiate a game design pattern will increase their usability in practice. Additionally, researchers could further investigate the varying effect of GoF design patterns when they are employed in the implementation of game mechanics, so as to reach more concrete conclusions and bring further evidence.
6. Threats to Validity and Limitations

This section of the paper discusses possible threats to validity and limitations of the conducted study. Firstly, the results on the four mappings between game and GoF design patterns cannot be generalized to the rest of them, or even more, to possible ones that were not identified in this study. Additionally, another threat to validity is the fact that each mapping was extracted from a real instance of only one game and thus, its effect on the quality attributes might vary in other possible game mechanic implementations. What is more, the existence of pattern variants in the examined cases may have possibly influenced the results of the quality evaluation.

Finally, the case study was performed on Java open source software games, and thus, the results should not be generalized to games, which are developed in other object oriented programming languages, such as C++, or to games developed from other respectable and well known software companies that possibly employ a bunch of software engineering professionals. The fact that many of the games considered for investigation were developed from individual practitioners, rather than a team of high-profile professionals from different computer science fields, might have played a role in the findings of the study.
7. Conclusion

Although, nowadays, computer games appear to be thriving, game design and development face many challenges, with the most prominent ones being poor design and weakened software quality attributes. On the other hand, it is a common belief among researchers and practitioners that the application of GoF design patterns assists in the improving of software, with respect to design-time qualities. The aim of this study is two-fold: (a) introduce basic implementations of game design patterns (also known as game mechanics) through GoF patterns and (b) evaluate the possible benefits of this combination. Game design patterns correspond to reusable parts of game logic and having specified implementations on how to instantiate them is expected to increase their usability in practice. In addition to this, and by also taking into account the enhanced quality that GoF patterns offer, an increase in the overall quality is also expected.

In the first part of the study, descriptive research methods and literature review was used. Nine mappings between game and GoF design patterns were identified. Exemplar class diagram for each one of the mappings, was created, along with sample code implementations in Java. In the second part of the study, the case study research method was used. As a result, a case study was performed on open source software Java games, so as to explore the applicability of the approach and evaluate the resulting benefits. Additionally, alternative, non-pattern employing implementations were developed in order to be directly compared to the GoF pattern involving instances, regarding certain software quality qualities.

According to the results of the case study performed, four out of the total nine mappings between game design and GoF patterns were detected in real game instances. Although, concrete conclusion about the advantages or disadvantages of game mechanics implementation through the usage of GoF patterns cannot been reached because of the small number of mapping occurrences that were detected, the findings of the evaluation suggest that while the application of GoF patterns is beneficial to certain aspects of software quality, some other aspects might be compromised. State and Strategy design patterns applied to game mechanics
implementations, are associated with an appreciable improvement in the understandability and extendibility of the code, while the Abstract Factory pattern appears to enhance the functionality and reusability of the software. Nevertheless, design patterns should be used with caution during the development of games because they might impede some other, desired quality attributes.

Conclusively, the results of this study suggest that a practitioner needs to perform a multi-criteria decision analysis before proceeding to the designing phase of development. Depending on the particular needs of each game mechanic, several factors must be considered before a developer decides if GoF patterns should be employed. The most desirable quality attributes that better serve the needs of a particular game must be taken into serious account, since a simple personalized solution might actually be optimal. The results provide pointers to interesting areas for future work. In-depth investigation of the varying effect of the application of GoF design patterns in the implementation of game mechanics will result in games with increased overall software quality.
8. Bibliography


