Implementing Game Mechanics with GoF Design Patterns

Xeni-Christina Kounoukla[1], Apostolos Ampatzoglou[2], Konstantinos Anagnostopoulos[1]

[1] Department of Computer Science, Mediterranean College, Thessaloniki, Greece
[2] Department of Mathematics and Computer Science, University of Groningen, Groningen, Netherlands
x.kounoukla@mc-class.gr, a.ampatzoglou@rug.nl, kanagnostopoulos@medcollege.edu.gr

ABSTRACT

Implementing game mechanics (i.e., game rules and logic) inherently involves high volumes of required complexity, which in turn leads to the introduction of accidental complexity (e.g., long methods, code repetition, etc.). Thus, usually games suffer from poor quality, i.e., attributes such as maintainability and flexibility are weakened. A possible solution for this shortcoming is the reuse of well-known software engineering practices, such as GoF patterns. GoF are not the only patterns that are applicable in game development: game mechanics represent recurring problems in game design and accompanying solutions. However, these patterns are rather abstract and no guidance on their implementation is provided. The aim of this study is to introduce basic instantiations of game mechanics with GoF patterns, which can potentially increase their usability in practice. To this end, nine mappings were identified and a case study on OSS games was performed to explore the applicability of the approach. Combining these two types of patterns is expected to provide various benefits: (a) the game mechanics will be accompanied with sample implementations that can be reused, to act as a starting point for source code development; (b) these implementations will obey to good design principles—therefore their maintainability will be safeguarded; and (c) the fact that game mechanics are recurring, guarantees the applicability of the proposed implementations in various games.

Categories and Subject Descriptors

D.2.10 [**Software Design**]: Methodologies
D.2.11 [**Software Engineering**]: Software Architectures

Keywords

Game development; design patterns; game mechanics; reuse

1. INTRODUCTION

Developing games is substantially different from classical software engineering (SE), in the sense that in most of the cases, games have a limited lifecycle due to their ***shrunk product time to market***. As a result, many games suffer from poor design and weakened software quality attributes (e.g., maintainability) [1]. Therefore, the ***need for software engineering methodologies for game development*** has been steadily growing over the last years and has evolved into a field of great interest [1]. A software engineering technique that has been validated as valuable in game development is design patterns. In the literature, the term pattern is used to characterize any recurring solution to a common problem. In the context of game development, patterns appear in two major forms: (a) ***GoF (Gang of Four) patterns***, which are introduced at the detailed-design and implementation phase to solve common object-oriented design issues [6], and (b) ***game design patterns*** (also known as game mechanics), which correspond to reusable parts of game logic [5]. Each one of these two types of patterns introduces different benefits, and is useful in the game design and implementation. On the one hand, ***GoF patterns*** are the most-known set of patterns. The application of GoF patterns has proven to be beneficial concerning design-time quality attributes [2]. On the other hand, ***game mechanics*** [5] constitute a collection of design choices available for a variety of games. These choices can correspond to recurring parts of gameplay, which is undoubtedly the most essential part of game design. However, they are solutions described at a higher level; thus, there is a lack of guidance on how to implement them.

PCI '16, November 10-12, 2016, Patras, Greece
© 2016 ACM. ISBN 978-1-4503-4789-1/16/11…$15.00
DOI: http://dx.doi.org/10.1145/3003733.3003779

2. METHODOLOGY

This study aims at combining the aforementioned types of patterns, by ***introducing basic instantiations of game mechanics through GoF patterns***. The applicability of the approach is investigated through a proof-of-concept approach that aims at identifying existing game mechanics instances that are implemented with GoF patterns in OSS games. To investigate the opportunities of implementing game mechanics with GoF patterns, we have performed an exploratory study on ten OSS games. The goal of this study is to ***analyze*** GoF patterns instances ***for the purpose of*** characterization ***with respect to*** the implementation of game mechanics ***in the context of*** OSS games. To achieve this goal, we have set the following sub-objectives: (a) define a list of game mechanics that are candidates for implementation with GoF patterns, (b) explore various GoF patterns so as to identify a list of candidate implementations for each game design pattern, (c) provide exemplar mappings between game and GoF design patterns, and (d) perform an exploratory empirical study on OSS games as a proof-of-concept to identify real-world cases, when such mappings occur. To this end, two RQs have been formulated:

RQ_1: What are the possible mappings between game mechanics and GoF design patterns?

To answer RQ_1, we accomplished sub-objectives (a) to (c), by proposing exemplar mappings between game and GoF patterns. For that, we first defined a list of game mechanics that are candidates for implementation with GoF patterns. Secondly, we explored GoF patterns to identify a list of candidate implementations for each game mechanic. Finally, we created exemplar class diagrams of the identified mappings between GoF and game mechanics patterns. The method that was used for answering this question was an informal literature review on various sources—ranging from academic studies to grey literature (websites, blogs, etc.).

RQ2: Are these mappings occurring in practice?

To gather data for answering RQ_2 we decided to use a software engineering repository, named *Percerons*, which documents de-

sign pattern occurrences [4]. From the repository we selected ten random games (in which we expected that the explored game mechanics could be implemented—i.e., games with similar purposes with those explained in Section 3), and catalogued all GoF pattern occurrences. These occurrences have been manually investigated to explore if the GoF patterns are implementing a game mechanic. We note that no formal case study processes have been set, since the goal of this empirical study is purely exploratory.

3. RESULTS

3.1 Game Mechanics with GoF Patterns

Turn based Games with Template Method. The intent of the *Template Method* pattern is to define the skeleton of an algorithm in a method. Some steps are deferred to subclasses. The pattern lets subclasses redefine certain steps of an algorithm without affecting the algorithm's structure [6]. *Turn-based Games* is a game mechanic that is applicable to games in which the players take turns to make their move [5]. 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. In the example of Figure 1, `Monopoly`, `GameOfLife`, and `Chess` classes extend the abstract class `Game` and reflect different turn-based games. Each one of them carries different implementations for `initializeGame`, `nextMove`, `endOfGame`, and `changeTurn` operations, according to the different logic of each turn-based game. This pattern facilitates the easy addition of new types of turn-based games (to the game engine that creates them) that will reuse the existing infrastructure for implementing their game loop.

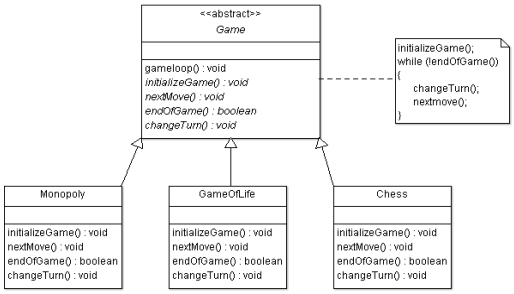

Figure 1 - Template Method / Turn-based Games

Agents with Strategy. The intent of the *Strategy* 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 run-time [6]. *Agents* are game 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 mechanics pattern [5].

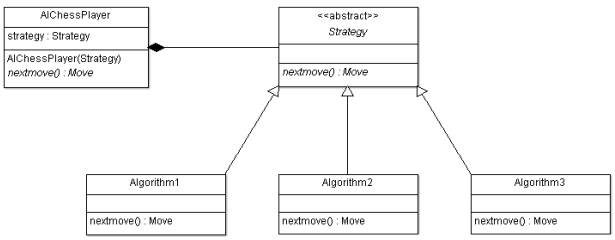

Figure 2 - Strategy / Agents

The class diagram of Figure 2 was created to reflect the mapping of these two patterns. The example consists of the abstract class `Strategy` that is responsible for the behavior of an `AIChessPlayer` of the chess game. Its concrete subclasses define different implementations of the AI algorithm (subclasses: `Algorithm1`, etc.), which decides the next move of the player depending on the game level. The class that plays the Context role in the GoF pattern is the `AIChessPlayer` class and it represents the AI player (agent). Using this structure, if along maintenance a new algorithm has to be integrated in the game, it can be incorporated without altering the code (conform to the Open-Closed Principle).

Power-Ups with Visitor. The *Visitor* 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 [6]. The *Power-Ups* game mechanic concerns game elements that give an advantage to the player, when they are picked-up by him/her [5]. The class diagram created to illustrate this mapping is presented in Figure 3 and 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 attributes speed and visibility change accordingly to the type of power-up visitor. The structure of the visitor allows the extension of the game with new power-up functions, conforming to the Open-Closed Principle.

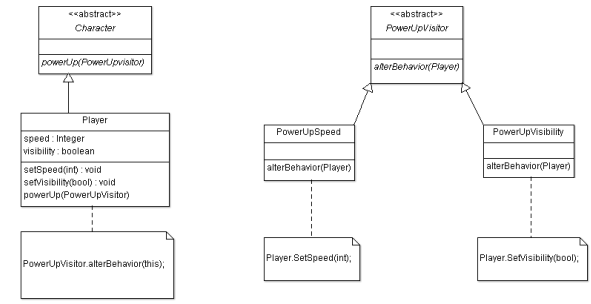

Figure 3 - Visitor / Power-Ups

Game World with Composite. The *Composite* pattern provides 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 [6]. The game mechanic *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 [5]. The class diagram of this mapping (see Figure 4) 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), avatars, houses and ho-

tels (individual objects). Along maintenance, the addition of new components on the board is provided through the mechanism of the pattern.

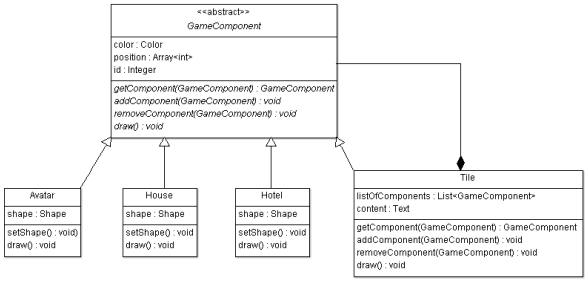

Figure 4 - Composite / Game World

Levels with State. The intent of the *State* pattern is to allow an object to change its behavior every time it's 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 [6]. 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 [5]. In the related class diagram (Figure 5), the class `Game` plays the context role of the State pattern occurrence, 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. A `Game` object changes its behavior (through the `gameFunction` operation) according to its state. Similarly to Strategy, State allows the addition of extra levels along maintenance without altering the code.

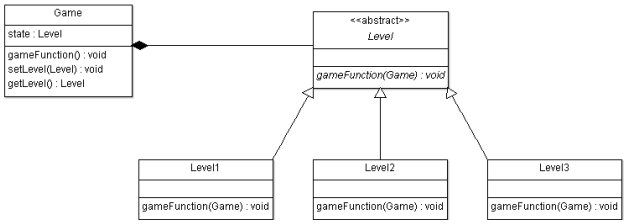

Figure 5 - State / Levels

Progress Indicator with Observer. The intent of the *Observer* 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 [6]. *Progress Indicator* refers to a game element that gives the player information about his current progress [5].

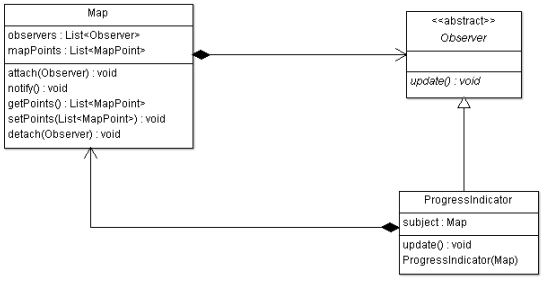

Figure 6 - Observer / Progress indicator

The sample implementation of this mapping refers to a simple game map, which consists of several checkpoints. 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. The structure of the pattern allows the easy extension of the game over two different axes: (a) the addition of map types (subclasses in the `Map` hierarchy), and (b) the addition of new concrete observers (i.e., indicators that are based on the state of the map).

Units with Abstract Factory. The *Abstract Factory* 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 [6]. 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* [5]. In the implementation of this mapping (see Figure 7), `Units` are considered as game elements which represent different types of soldiers (`Infantry`, `Horseman`, `Bowman`) that are equipped with a `Shield`, `Bow`, or `Sword`. `SoldierFactory` and `EquipmentFactory` are the concrete classes derived from the abstract `AbstractFactory` class. They define implementations of operations `getSoldier` and `getEquipment`. The former class is responsible for the creation of a soldier, whereas the latter for creating its equipment.

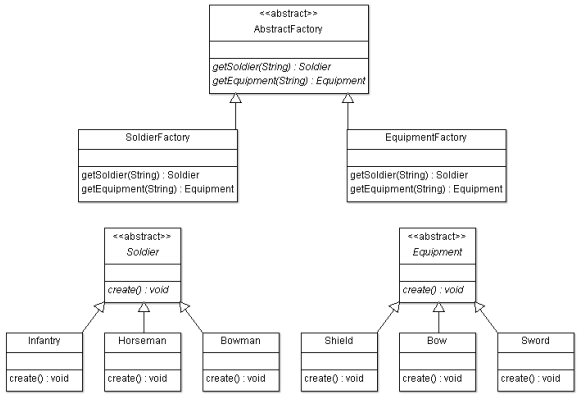

Figure 7 - Abstract Factory / Units

Movement with Strategy. The game mechanics 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 [5].

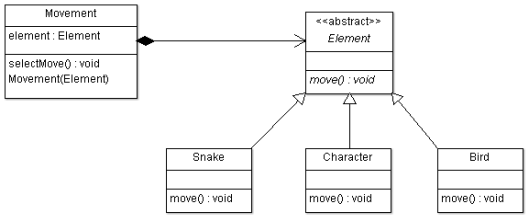

Figure 8 - Strategy / Movement

In Figure 8, we considered a game where the 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 animates 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 instance is played by the `Movement` class. The extension axis of this pattern is the addition of new moveable elements with their own animations.

Varied Gameplay with State. *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 [5]. Due to the fact that Varied Gameplay constitutes in practice a very large game mechanic pattern, a simple example was considered (see Figure 9): a human player is able to play the game either against a human opponent or an AI opponent. 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` is the context. In this case, apart from the obvious extension axis (addition of new types of games), the pattern provides the opportunity for developers to group similar functions (other than `gamePlay`) into meaningful clusters.

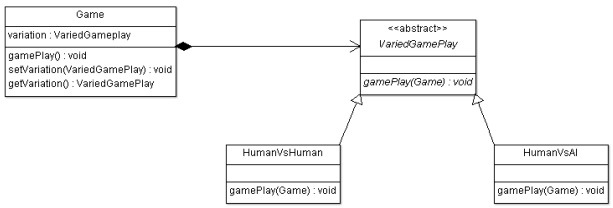

Figure 9 - State / Varied Gameplay

3.2 Occurrences in OSS Java Games

In this sub-section we answer RQ_2, by investigating if we are able to identify the aforementioned mappings in real games. The results of the proof-of-concept empirical study, i.e., which of the aforementioned mappings have been identified in the explored games, are outlined in Table I. As presented in Table I, according to the conducted empirical study, the frequency of occurrences of mappings between game mechanics and GoF design patterns is satisfactory. From the nine mappings that we presented in Section 3.1, four were identified in real pattern occurrences (approximately 44%). However, no mapping has been identified in more than one game. Nevertheless, we need to acknowledge that towards the aforementioned results a significant role has been played by: (a) the abstractness of game mechanics descriptions, and (b) the lack of any guidance on their implementation. Thus, we expect that any mapping that has been identified was unintentional, or based on the personal expertize of the open source game developer.

Table I - Occurrences of Mappings in OSS Java Games

Mappings	#Occurrences	OSS Game
State / Levels	0	
State / Varied Gameplay	1	Infothello
Strategy / Agents	1	Infothello
Observer / Progress Indicator	0	
Visitor / Power-Ups	0	
Strategy / Movement	1	Arcadiban
Composite / Game World	0	
Abstract factory / Units	1	DragonChess
Template Method / Turn-based Games	0	

Therefore, the findings of this pilot case study have proven that the mapping proposed in Section 3.1 can be identified in practice and that there is a potential in promoting the systematic use of GoF patterns for implementing game mechanics. In this direction of work, we plan to replicate this study in the opposite way, i.e., to catalogue all expected game mechanics in one OSS game, identify their implementations, and check if it involves any pattern. Using such a research setting would provide us with evidence (i.e., precision and recall) on the existence of such mappings. Nevertheless, such a validation was out of the scope of this manuscript.

4. DISCUSSION / CONCLUSIONS

In this section we discuss implications to researchers and practitioners. On the one hand, game researchers' ***body of knowledge on patterns has been expanded***. Furthermore, software engineering researchers could further investigate these mappings, since they provide potential solutions to the game design field. Nevertheless, having a greater number of specific implementations on how to instantiate a game design patterns will increase their usability in practice. Additionally, researchers could further investigate the varying effect of GoF patterns when they are employed in the implementation of game mechanics, so as to reach more concrete conclusions and bring further empirical evidence.

On the other hand, practitioners have been provided with ***standardized solutions for frequent game design problems***. In particular, the mappings between game and GoF patterns, which 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. Based on the qualities that the game engineers are most interested in; 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. As a parallel benefit, the level of ***games' maintainability is expected to increase***. This side-benefit can be provisioned by the introduction of GoF design pattern instances in the source code of games (based on the literature GoF patterns have a proven positive effect on maintainability [3]). The applicability of GoF patterns in game will be safeguarded by the fact that they implement mechanics, which are by definition applicable in many games. In other words, GoF patterns are expected to provide documented extension axes for the most usual changes along games' maintenance, since they differentiate between versions w.r.t. changes of the same type (e.g., animation, terrains etc.).

REFERENCES

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

[2] Ampatzoglou A., Charalampidou S. and Stamelos I., "Research state of the art on GoF design patterns: A mapping study", *Journal of Systems and Software*, Elsevier, 86 (7), 2013.

[3] Ampatzoglou A., Chatzigeorgiou A., Charalampidou S., and Avgeriou P., "The Effect of GoF Design Patterns on Stability: A Case Study", *Transactions on Software Engineering*, IEEE, 41 (8), pp. 781-802, 2015.

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

[5] Bjork S. and Holopainen J., “Patterns in game design”, *Charles River Media*, 2005.
[6] Gamma E., Helms R., Johnson R., and Vlissides J., “Design patterns: Elements of reusable object-oriented software”, *Addison-Wesley*, 1995.