applied sciences

MDPI

Article

Design Patterns for Mobile Games Based on Structural Similarity

Ghulam Rasool [1], Yasir Hussain [2], Tariq Umer [1], Jawad Rasheed [3], Sook Fern Yeo [4,5,*] and Fatih Sahin [6]

[1] Department of Computer Science, COMSATS University Islamabad, Lahore Campus, Lahore 54000, Pakistan
[2] Department of Computer Science, Virtual University of Pakistan, Lahore 51000, Pakistan
[3] Department of Software Engineering, Nisantasi University, Istanbul 34398, Turkey
[4] Faculty of Business, Multimedia University, Melaka 75450, Malaysia
[5] Department of Business Administration, Daffodil International University, Dhaka 1207, Bangladesh
[6] Department of Computer Engineering, Nisantasi University, Istanbul 34398, Turkey
* Correspondence: yeo.sook.fern@mmu.edu.my

Abstract: Software design patterns have a proven impact on the quality of software applications and the development process of an application. The success of design patterns in the software industry has attracted mobile game developers and researchers to apply patterns in the context of mobile games. Researchers have already proposed different frameworks and design patterns, but they are not truly beneficial for game developers. The high-level taxonomies can be adjuvant while proposing useful design patterns. The existing taxonomies for mobile games do not consider different parts of a game that outline top-level structure. In this paper, we propose a new taxonomy that emphasizes the top-level structure for identifying new design patterns for mobile games. We propose five novel generic design patterns that might be applied to the development of mobile games and other software applications. The presented design patterns are, in a true sense, programming patterns that outline top-level generic classes and interfaces, and that could be the basis for the development of new games. We developed four demo games by using these patterns for the realization of taxonomy and design patterns.

Keywords: design patterns; mobile games; taxonomy of games; consumer product; product reusability; game interaction

Citation: Rasool, G.; Hussain, Y.; Umer, T.; Rasheed, J.; Yeo, S.F.; Sahin, F. Design Patterns for Mobile Games Based on Structural Similarity. *Appl. Sci.* **2023**, *13*, 1198. https://doi.org/10.3390/app13021198

Academic Editor: Paweł Weichbroth

Received: 6 December 2022
Revised: 6 January 2023
Accepted: 8 January 2023
Published: 16 January 2023

1. Introduction

Games have become an essential field of research due to the evolution and advancement of technology. Computer games are widely used for learning, skill enhancement, entertainment, and engagement. Computer games have roots in simple handheld devices such as feature phones. The use of games on mobile devices has been increasing, but the hardware space for mobile devices requires more attention from researchers. Mobile devices have transformed into smart devices (a.k.a., smartphones) from simple communication devices, attracting almost all the technologies that were intended for Personal Computers (PCs). Today, any popular application running on a PC is expected to have a mobile version. Mobile devices seem incomplete if they do not offer features for fun and leisure. The trend of playing games on smartphones is continuously increasing. Over the last one and a half decades, mobile games have dramatically increased in popularity, and they have become a source of unprecedented growth in revenue for the mobile game industry. It is estimated that 50% of the global game market will be from mobile games until 2020, (https://www.gameprime.org/2018/01/global-game-market-prediction-2020, accessed on 16 March 2020). The revenue of the global mobile game industry was over $115 billion in 2019, (http://mediakix.com/2018/03/mobile-gaming-industry-statistics-market-revenue/#gs.M2_ISeU, accessed on 16 March 2020). Industry professionals and academic researchers have argued to separate the field of mobile games from other software fields (e.g., calculator games and PC games [1–4]) and other disciplines (e.g., films, stories). Terms like Ludology [5,6], Gameplay [7], and Funware [8] have already come into existence.

Design patterns have a proven impact on software quality [9], development productivity [10], maintenance [11] and continuous involvement in almost all the disciplines of software since they were first proposed by Gamma et al. [12]. The proper application of design patterns improves the quality and increases the productivity of developers up to 40% [13]. By definition, a design pattern is a well-documented template to be used in developing new products rather than reinventing the wheel. A design pattern can be a behavior or a structure frequently occurring in related products. A pattern can be as simple as Singleton with only one class, or it can be complex such as Observer, Mediator including Singleton and other patterns in its structure. The set of 23 design patterns presented by Gamma et al. [12] is notably essential, but still, there is a need to think beyond them. The application of design patterns for developing mobile games is receiving attention from academia, developers, and researchers. Although there is significant progress, there is space to contribute by identifying new design patterns, in particular, for developing mobile games.

In most of the cases, game developers do not follow patterns; rather, they develop their games using personalized techniques and foundation classes [14]. Such developers are unaware of the true benefits of using design patterns for solving recurring design and development problems. Some design patterns cover programming aspects; however, they do not give a starting point for the developers, or they are not intended for mobile games. We derived new design patterns based on a new taxonomy that emphasizes the top-level structure of mobile games to classify them. The degree of differences and similarities in different parts of a game devise a category of a game. We propose design patterns for each of the identified categories.

We also propose a top-level design pattern for a complete game that includes menus and characters. It might include all four distinct patterns depending on the requirements of game developers. The motivation and discussion on four patterns are presented in Section 4. The top-level pattern is applied in our four demo games discussed in Section 5. Although, these design patterns may be adapted to any programming language, we prefer to use Java Micro Edition (ME) for sample source code examples and library classes. Java ME is popular for feature phones. The proposed design patterns are introduced for the first time based on the proposed taxonomy. The major contributions of this paper are as follows:

- A new taxonomy for the categorization of design patterns in mobile games;
- Derivation of five design patterns based on the proposed taxonomy;
- Evaluation of design patterns on four demo mobile games.

The rest of the paper is organized as follows. In Section 2, we provide a detailed review of related work from the literature. In Section 3, we present a new taxonomy for mobile games that is the basis for our design patterns for mobile games. The proposed design patterns are discussed and presented in Section 4. We present a case study evaluating the proposed design patterns in Section 5. The conclusion and future work are discussed in Section 6.

2. Literature Review

A general review of computer and video games related to learning was presented by Mitchell et al. [15]. Similarly, a systematic review of empirical evidence on computer games and serious games was presented by Connolly et al. [16]. Some researchers have turned their attention towards design patterns and their applications regarding the development of games for educational purposes and learning outcomes [17–38]. We discuss the previous research contributions in two categories. Research for the common vocabulary of games includes works from Church [1], Costikyan [2], Bjork et al. [3], and Davidsson et al. [17]. These works do not directly influence our work, but they are important because they can be the basis for comprehensive frameworks. In addition, they emphasize bringing all games (naturally including mobile games) into one vocabulary to be used by game designers, developers, and players.

The second category includes research contributions that are related and are the basis for this work. The authors of these papers discuss the identification or use of programming patterns directly (unlike ones that cannot be programmed and only define design space). We include works from Hui [39], Narsoo and Mohamudally [40], Narsoo et al. [41], Ilja [42], and Nystrom [43] under this category.

2.1. Research for Common Vocabulary of the Games

Church was one of the pioneer game designers that paid attention to common design vocabulary for games. He emphasized that, like other disciplines such as sports and movies, game designers and players should express and discuss games in terms of that vocabulary [1]. Church suggested developing a shared framework, and he named that framework FADTs (Formal Abstract Design Tools). Church explained the term "Formal" because it gives a precise definition, and "Abstract" because it focuses on underlying ideas, not specific game constructs. The terms "Design" and "Tools" are self-explanatory. FADTs cannot be directly used to develop games.

Church [1] argued his work would motivate game developers and researchers to discover new ways for successful design and possibly expand the FADTs framework. Church's work is no doubt impressive as this was the first attempt to propose a shared vocabulary. Later, researchers have some degree of influence from his work, and they have references to his work (e.g., [44]). Despite this, FADTs have not been expanded; instead, researchers have proposed their own ways to talk about game design, gameplay, and mechanics (e.g., [3,44]). It is clear from the nature of FADTs that they cannot be implemented using code—a feature that our paper emphasizes. However, there may be exceptions.

Costikyan [2] also emphasized on a common vocabulary for computer games because they all share aspects that keep them apart from other software design. The author formalizes the terms Interaction, Goal, Structure, and Struggle to make a complete functional definition of a game. These terms can be used to discuss the vast variety of games from board games to sports and from real-world games to web-based multiplayer games.

All games need Interaction in any form, especially the player and the game. A game is simply not a game if interaction with the player is not required. Thus, interaction is considered at the top for games by Björk et al. [3]. However, the authors explained that games require interactions for some purpose. The interaction will be meaningless if there is no Goal to be achieved by the player. Hence, nearly all types of computer games have their goals. These goals may be set by the player if the game requires it. The next term in the author's list is Struggle. This suggests the player has to struggle to achieve his/her objectives in the game. The structure of a game covers codified game rules, the interaction of game rules, tables, algorithms, modules, and software. Although, the structure of a computer game is invisible to the player, he/she agrees to achieve the goal through the structure of the game. The authors also used the term Endogenous Meaning, which means that the game's structure creates its own meanings. For example, an image of an arrow has no real value, but in the computer game "Bows and Arrows" the same image becomes valuable.

Combining all these terms, Costikyan [2] finalized the functional definition of a game as "an interactive structure of endogenous meaning that require players to struggle towards a goal". Finally, the author asserts that the functional definition of a game and taxonomy of a game presented by Marc LeBlanc et al. [44] together can be beneficial for designing games. Whereas the common terminology emphasized by Costikyan [2] applies to games, tools discussed under the title of design patterns by other researchers have gained much more acceptance in the industry.

Björk et al. [3] broadened Kreimeier's idea [45] of games in terms of design patterns. They collected about 300 game design patterns for all types of games. However, they redefined game design patterns as descriptions of recurring interaction elements relevant to gameplay. By design patterns, they do not mean only some frequently occurring problems

requiring a similar solution, but also frequently occurring game mechanics. These can be thought of as general tools to be used in different ways by different stakeholders in industry and academia rather than problem-solving tools during game development. The authors have, therefore, kept interaction (between players and between players and other game components) a foundation for collecting patterns, which is an essential element in games.

Meanwhile, they have given importance to the structural framework of games, which consists of game instance, game session, and play session [46]. The authors have created their interaction-centric model from two independent, but interchangeable parts. The first is a structural framework that describes the components of a game, while the second is game design patterns that describe the interaction of players.

Unlike the known definition of patterns as "problem-solving" tools, the authors perceive their identified patterns as a tool that could be used in various ways. The uses that are listed are idea generation, development of game concepts, pre-production process, identifying competition and patent issues, problem-solving during development, analyzing games, categorizing games and genres, and support to explore new platforms and mediums.

The applications of the design patterns discussed above are essential, but we do not see any potential use of these patterns for game developers. For example, the pattern "Paper Rock Scissors" means guess what the opponent will do and then play to oust him based on the guess. This pattern occurs in different games such as checkers, chess, and card games. Although the pattern is the same, checkers and paper rock scissors (if a video game is developed) will have an entirely different code (e.g., classes and methods), and it is much too abstract to be a programming pattern. Thus, the knowledge of this pattern may not be helpful to avoid reinventing the wheel.

2.2. Research on Design Patterns Related to Computer and Mobile Games

Hui [39], a professional mobile application developer, presented a framework of four design patterns to develop interactive content on mobile devices. These patterns are targeted for J2ME mobile applications. One crucial factor in Hui's work is that he has given due consideration to constraints posed by the limitation of mobiles such as small memory and screen size. The author examined Cascading Menu, Wizard Dialog, Pagination, and Slide Show design patterns.

Cascading Menu pattern is a scaled-down form of the popular Model-View-Controller architecture pattern. A user can insert new menu items without complication by using this design pattern. Wizard Dialog pattern is used to implement a mobile-suited version of desktop Wizard (set of dialogs, e.g., during the installation of software). This pattern is designed using another pattern Mediator, which works like a middleman controlling some related components. The Pagination pattern is used to divide a large number of menu items into smaller groups to show them on a small screen in the form of pages. Pagination loads or creates and displays only the current page, thus saving memory. Finally, the Slide Show pattern is used to switch between different displays of an application in the form of a slide show. A sort of Cascading pattern can also be observed in mobile games. The difference is that this pattern suggests that each slide should be a separate displayable object, whereas, in mobile games, common practice (and maybe a requirement to save memory) is that the slides are made from a single displayable (i.e., Game Canvas, a subclass of Canvas) using paint(), repaint(), and/or update() methods.

Several games may use the patterns outlined by [39] or customized versions of those patterns. The limitation of this framework is that it addresses the interaction of an application with the user only. Moreover, common and desired practices for games are different from other applications, thus making these patterns either unsuitable or insufficient for most games.

Narsoo and Mohamudally [40] introduced the notion of a one-function design pattern for mobile services that use J2ME (now Java ME). They took two different applications as test cases; Phone Book and Reminder. These two applications perform similar functions on different types of data. They present a similar interface while their internal workings in

private methods and classes are hidden from the user. For example, both can add, delete, edit, and search their respective data, which would be the contact names and numbers for the phone book, and reminder time and message for a reminder. Thus, they have the same pattern concerning their interfaces. Authors call this an integrated behavior structure because this structure suggests integrating all tasks into one pattern. Applications using the integrated framework may customize different arguments.

The other option that authors suggested is a one-function structure in which a pattern performs a single task. For example, now there will be four patterns, Add, Erase, Edit, and Search, for each of the Phone Book and Reminder applications. Moreover, Add and Edit are similar; thus, a single pattern with an additional parameter would be better. There may be several other functions that can be used by applications like Chat, SMS, and Calls Management. The integrated behavior structure for a phone book looks like the class shown in Figure 1.

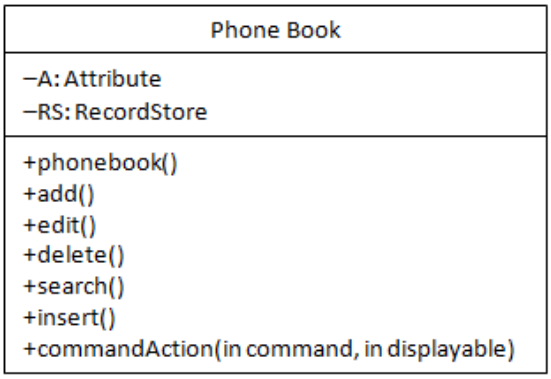

Figure 1. Integrated behavior structure for Phone Book [40].

For the "Reminder" application, everything is the same except the class name, and the constructor will be "Reminder". As mentioned in [40,47], larger classes were undesirable for mobile applications as they consume more memory, consequently an expensive item for small devices. Figure 2 shows the main class representing the Edit design pattern.

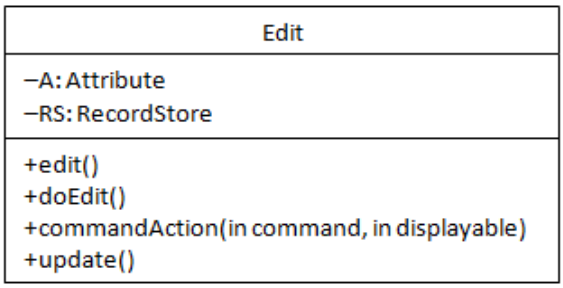

Figure 2. One-function structure for Edit [40].

The Edit pattern can be used in any application where the edit function is required. For example, it can be used to modify existing entries from the Phone Book as well as set time and date of an entry from Reminder. Its implementation in mobiles is a form with a retrieved item displayed in a text field that allows the user to enter new entries and save them. This pattern applies to any J2ME application with or without customizing to accept specific parameters.

Though the authors identified only a few design patterns, there may be tens of them. However, the authors have given us another starting point for further research. Following their work, we can think of such patterns for mobile games. For example, we can think of a "Move Avatar" pattern, which could be used to move a game avatar using arrow keys. That pattern could be used in any game where the avatar has to be moved using arrow keys. There can be numerous such patterns.

In this paper, the objective is to develop similar programming patterns at the generic level without going into detail. For example, all those games in which some levels are different only in the number of enemies will have the same structural design pattern. (However, the demo games developed along with this paper also consider some detail-level patterns. Thus, demo games can be used as templates to develop similar games.)

Narso et al. [41] demonstrated the use of design patterns for mobile games using J2ME. Initially, they used design patterns for single-player Sudoku game. Then, they demonstrated that the same design patterns could be used for other board games with little modification in interfaces. At the top, they have used Model-View-Controller (MVC) architecture. Then, the patterns such as Change Screen, Singleton, Game State Observer, and Game Memento have been used preferring interfaces rather than concrete classes wherever possible. The use of interfaces assures that the same set of patterns may be used for other board games. Figure 3 shows a top-view diagram of this structure.

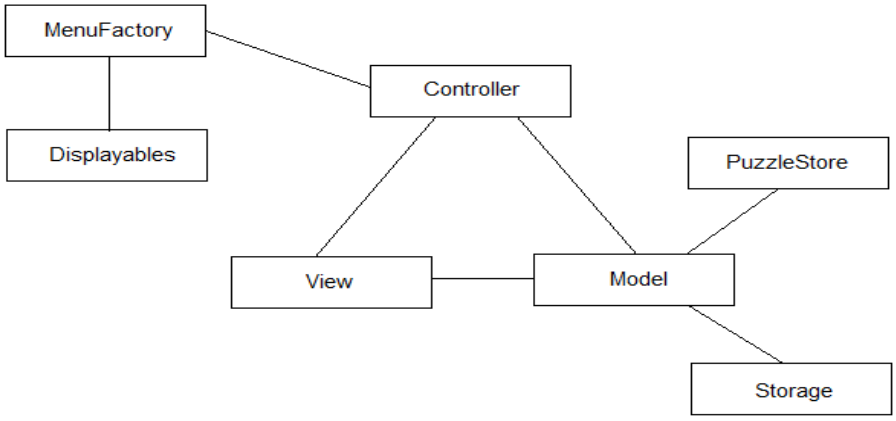

Figure 3. MVC architecture for Sudoku [41].

The model part has been implemented as an interface GameModel, leaving implementation for concrete classes of games. SudokuGameModel is a concrete implementation of this interface. It has interactions with Storage-related classes. The view part has been implemented by GameView, BoardView, and CellView classes. Moreover, its draw() method is open for extensions, thus enabling safe modifications for other games. The GameMidlet class (entry point into any java MIDlet) and GameController interface constitute Controller part. Again, GameController is an interface to achieve loose-coupling and high-cohesion in modules. The concrete implementation in the Sudoku game is SudokuGameController.

Patterns used in this sample game that may also be used for other games are Change Screen, Game Memento, Game Controller Choice, Game State Observer, Drawing Template, and Singleton. All of these patterns are self-explanatory.

The pattern suggested by [47] separates displayable objects for a menu. This style supports the system's standard menu interface, but another practice is to implement a graphical menu (including images), which is one of the playability heuristics as mentioned by [47]. The MVC-based architecture can be reused with little modifications for other board games. The authors have outlined three example cases: GameCanvas.

(1) Same Game with Different Settings: Standard Sudoku has 9×9 cells, while the authors have implemented an option of Junior Sudoku, which has 4×4 cells. The changes required to implement this option are to make the game model independent of puzzle size (using Strategy pattern) and that the game store class has to implement an interface in which both puzzles should be implemented in concrete classes.
(2) Number Game Puzzle: Almost the whole structure along with its internal relations are preserved except changing the text menu and other classes to reflect this game. Additionally, the main model class SudokuGameModel has to be replaced with new a class (which should also implement GameModel interface) containing mechanics related to the new game.

(3) Two Player Based Game: The controller interface will be changed accordingly. Additionally, a new model has to be used, or the model interface could be adjusted for a two-player game.

Similar to [40], this work is another motivation for working on mobile applications especially games. Their work is concrete evidence of the advantage of using design patterns for games for small devices. The advantage is in the form of reusability and ease of modifications and extensions. However, unlike our objective of the paper, the discussed architecture and patterns consider details of board games only. Thus, they may not be generic enough to apply to other types of games. The authors focused only on J2ME.

Ilja [42] showed that design patterns could be effective solutions to problems arising during mobile game development. Moreover, he asserted that traditional design patterns such as Model-View-Controller and Observer are still lively and can be used for the development of modern games. To validate his argument, he developed a small game. He claimed that he was new to game programming, as the problems that arise during the development, the appropriate and useful design patterns, were selected naturally, examined and used. As a result, the code was well structured, organized, and maintainable. The four patterns used were Model-View-Controller, Observer, State Machine, and Singleton. The paper also includes a discussion of two mobile game development frameworks: Cocos2d and Unity3D. The author used them as third-party game-engines for the case-study. The author discussed these patterns elegantly mentioning advantages, disadvantages, and alternative solutions. Additionally, the work shows that said middleware game-engines implement those patterns either directly or indirectly. These frameworks support C++ and Objective C languages. Thus, the author also discusses if these languages have a built-in capability to support these patterns. The paper concludes that design patterns are still useful in mobile games development, but the essential thing is to examine and decide if the use of a certain pattern will be useful in a particular situation.

This work proves only the validity of patterns in modern game programming as was its objective, but the author did not identify any new pattern.

Nystrom [43], a professional game developer, presented an online book on games programming patterns. His experience in game programming and dealing with good code practices for years resulted in a treasure for game developers especially ones who are new to this field. He asserts that twenty years old GoF patterns [12] are suitable for other disciplines of software, but they have to be revised to adapt them for games because games constitute a different discipline. Nystrom admits the validity of twenty years old GoF design patterns and regards his book as a continuation of their work, but for the context of games. He included GoF patterns (Command, Flyweight, Observer, Prototype, Singleton, and State) in his book and explained how they are applicable for games. A substantial part of his book discusses new programming patterns that he identified in codebases of games. These are 13 patterns divided into four categories; Sequencing, Behavioral, Decoupling, and Optimization. It seems unnecessary to summarize each pattern here, but it is reasonable to discuss each group in brief along with one or two patterns from the group as an example.

Sequencing patterns act as tools to implement sequences in games in such a way that everything in the game world is just correct according to the game's clock and whatever is rendered in the screen is right. Double Buffer, Game Loop and Update Method patterns belong to this group. All these patterns are of critical importance for any game. Double Buffer allows a graphics' individual pixels to be modified as desired but will be rendered on the screen atomically (as a whole). For this purpose, two separate buffers are maintained for modification and rendering. When the modification process completes, the buffers are swapped. Fortunately, unlike many other languages (e.g., C++) Java ME's Canvas class has built-in support for double buffering. Game Loop suggests a loop continuously running during gameplay. In each iteration, user input has to be entertained, the state of the game has to be updated, and the game has to be rendered. Usually, this pattern is accompanied by the Update Method pattern. This pattern is used to simulate each object to update its state processing a single frame of behavior at a time. This is done by calling an update

method on each object in collection continuously in the game loop. These three patterns are used in every mobile game knowingly or unknowingly. (Our demo games also use all of these patterns.)

Behavioral patterns consist of patterns that are used to define and refine the creation of game entities and screenplays, which tell the entities what to do. Subclass Sandbox, Type Object, and Bytecode patterns belong to this category. Subclass Sandbox suggests "defining behavior in subclasses using a set of operations provided in parent class".

Decoupling patterns are used to decouple unrelated parts of a game. The Component pattern is used to allow a single entity to span in multiple domains, keeping them isolated by placing the code for each in its component class and reducing the entity to a simple container class, which contains those components. The other two patterns in this category are Event Queue and Service Locator. The Service Locator pattern is used to give a global access point for services without coupling users to concrete classes of those service interfaces.

Optimization patterns are used to optimize the code to enhance the performance of the game. This category includes Data Locality, Dirty Flag, Object Pool, and Spatial Partition. Out of these, first and last are very uncommon and may be very difficult to implement for mobile games developed using Java ME. The Dirty Flag pattern is used to avoid unnecessary work postponing until it is required. This is done by the use of flags (usually Boolean).

The author has collected all these patterns to achieve mainly one thing: decoupling, which results in easily maintainable and modifiable code. He also emphasized performance, but not at the cost of unorganized code. Therefore, he proposed to use Singleton only where any other method is too much costly. This is what one of the original authors of 23 design patterns also suggested in an interview [48]. The author has considered the gameplay and fun part of games only, leaving their overall structures.

2.3. Summary

The literature review discussed above has different scopes, and due to this, a concrete comparison is not possible. A common vocabulary is emphasized in [1,2] for games, but with different concepts. Church [1] coined the term "Formal Abstract Design Tools" to propose his framework, but Costikyan [2] opted to differentiate games from other disciplines using game-specific terms—Intent, Goal, Struggle, Structure, and Endogenous Meaning. Björk et al. [3] also emphasized a common vocabulary, but in terms of design patterns, which define games' design space and game mechanics. Davidsson et al. [17] continued the work of Bjork et al. [3] for only mobile games. Narsoo et al. [40] identified four basic interactive design patterns for services in mobile applications, which may be used for mobile games as well. Nystrom [43] collected important patterns related to game programming. These patterns are equally applicable to mobile games with or without a little customization. Hui [39] presented design patterns to develop interactive content on mobile devices. The author targeted mobile applications developed in J2ME. Both directly or indirectly talked about user interfaces, but the work in [39] is more specific to them. The authors of [41,42] practically used design patterns for the development of mobile games. Narsoo et al. [41] used MVC at the top level and some new patterns in the implementation. He also validated them by using those patterns in similar games. Ilja [42] did not identify any new pattern; instead, he tried to prove patterns as solutions to problems that occur during the development of mobile games. The author in [42] explained the motivation of the concept using popular GoF design patterns as examples.

3. Taxonomy Based on Structural Similarity

Taxonomies play a significant role in recognizing the relationships among different objects. The researchers in the literature rely on different parameters for the classification of games that include GUI, source, and platform of games. The taxonomies proposed in the literature are not generic. Examples of taxonomies include [49–54]. These taxonomies

do not take into consideration the overall structure of the game and structural similarities among different parts of a game. As a result, they are not beneficial for game developers.

The design patterns proposed in this paper are based on a new taxonomy. The proposed taxonomy is based on the structural similarity of games. Here, we summarize the concepts and the categories of mobile games that we used in this paper. A game can be comprised of levels (or stages) and parts. If a game has levels, it means that they have some order, and a player has to play a game in that particular order. The levels vary with respect to difficulty, and all of them are parts of the main gameplay. On the other hand, if a game has different parts, they may not have an order, and not all of them are part of the main gameplay.

3.1. First Category—Unrelated Levels

In the first category, a game has levels with significant difference among them and all of the levels are "unrelated", having different game logic making them dissimilar. For example, in one level, navigation (i.e., arrow) keys may be used to set a target, while in another level they may be used to have the avatar avoid enemies. Taking another example, in one level, firing an enemy gives a bonus, while in another level catching a diamond does this. If S is the set of all levels in a game and F_i is the set of all functionalities of an arbitrary level i, the following implication must be true.

$$\forall m \in S, \forall n \in S \; ((m \neq n) \rightarrow (F_m \neq F_n))$$

Figure 4 illustrates a game where F_X denotes a set of functionalities of an arbitrary level X. The demo game "ShootDown" belongs to this category. Figure 5 shows the distinguishing properties of all three levels of this game.

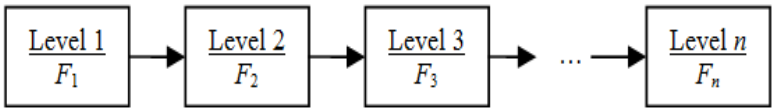

Figure 4. Sample level-based game.

Level 1	**Level 2**	**Level 3**
Avatar: Target Fire shoots immediately **Enemy:** Plane **Keys:** Left, Right, Up, Down, Fire	**Avatar:** Target Fire moves from Cannon **Enemy:** Helicopter **Keys:** Left, Right, Up, Down, Fire	**Avatar:** Tank Fire Moves from Tank **Enemy:** Smart Helicopter that can drop bombs **Keys:** Left, Right, Fire

Figure 5. Properties of all three levels of the "ShootDown" demo game.

3.2. Second Category—Related Levels

In the second type of games, subsequent levels have similarities in contrast to the first category above. In this "related levels" category, the games have differences in levels, but the levels are related. For example, the third level in a game is similar to the second level except that it adds some other functionality. As shown in Figure 4, a game belongs to this category if any of these two cases are true:

- At least two levels are similar, but there is another level which is not Similar to either of two levels, such as if X, Y, and Z are any three levels of a game:

$$(X \sim Y) \wedge (Z \sim X) \wedge (Z \sim Y)$$

- At least two levels are Related, i.e., there should be some levels X and Y having F_X and F_Y sets of functionalities, respectively, such that the following condition is met:

$$(F_X \subset F_Y) \vee (F_Y \subset F_X)$$

The majority of mobile and desktop games fall into this category. Examples include the Block Breaker mobile game by GameLoft and DX Ball by Michael P. Welch, Explode by Amidos, SnowBowl on FlashFang.com, Blue Box by MyPlayYard Games, and Sliding Cubes by MyPlayYard Games.

In Block Breaker and DX Ball, a ball is bounced off the pad to hit and break blocks placed in the space in various arrangements. The games have levels. In subsequent levels, a new obstacle and new types of blocks are introduced. As long as a new feature is not introduced in a level, the only difference is in the arrangement of blocks. Hence, a level may be related to another based on obstacles and types of blocks (e.g., some blocks break with a single hit and some blocks require two or more hits while others may not be breakable). Figure 6 shows the snapshots of different levels of Block Breaker Unlimited 3.

Figure 6. Related levels of Block Breaker Unlimited (from GameLoft).

3.3. Third Category—Similar Levels

In the third category, games have multiple levels, and there are minor differences among them. Thus, they are Similar, and no new functionality is added in the subsequent levels. They differ only in parameter values. Considering Figure 6, the game will be of a Similar Level type if,

$$F_1 = F_2 = F_3 = \ldots = F_N$$

The above equation shows that features at different levels are similar. We explain this with an example. In most versions of the "Color Bridge" game, boxes of different colors are scattered over a grid of rows and columns. The goal is to join the matching colors by bridges in such a way that they do not intersect with each other. There is not any new obstacle or feature introduced in subsequent levels; hence, all the levels seem to be instances of a single class. Thus, this game falls in the Similar Levels category. Figure 7 shows snapshots of two levels.

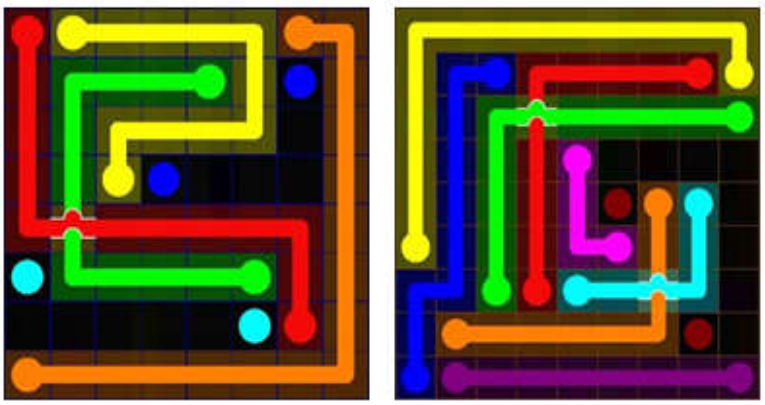

Figure 7. Similar levels of the Color Bridge game.

3.4. Fourth Category—Different Parts

In the fourth category, games do not have levels; instead, they have different parts. This category can be further divided into two types:

(1) Different Parts with Different Purposes: In the first type of games, different parts serve different purposes. All parts have a significant difference among themselves. Usually, one part presents the main game, while others play a supportive role. Parts can be switched either by clicking on specified buttons or only when some specific location or time is reached.
(2) Different Parts with Similar Purposes: Games of this category have different parts for a similar purpose, i.e., in most parts, a player has to achieve a similar goal. Thus, certain parts may not have significant differences. Only adjacent parts can be switched. "Seedling" is an example of this type of game.

A popular game, Haste-Makes-Waste falls in the first type of games in this category. It has an Introductory part, a Main Game part (where the turtle is projected), and a Shop Part where certain powers can be bought. Figure 8 illustrates the flow of the game.

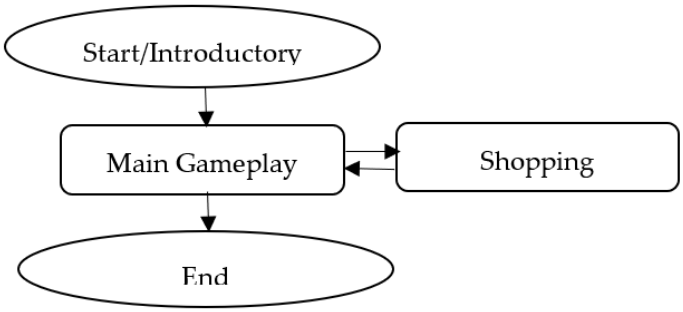

Figure 8. Switching among different parts.

The "Pinata Hunter" is another example of a game that falls into the first type. It is an interesting game consisting of two essential parts. In the main part, a hanging pinata has to be hit with a stick and candies coming out of the piñata are collected to earn money. The candies have different values. In the other part, the earned amount is spent on buying powers such as a better stick, a wider bag to collect more candies, and gloves to save hands from spam. Other than these parts, there is a Trophy Room part. There is a simple introductory part as well. As we can see, each part has a different structure and logic.

"Seeding" is an example of the second type of games in this category. It has different locations. The avatar can only enter the adjacent locations. All the locations have game-related tasks. Thus, each of the locations can be viewed as a separate part.

3.5. Fifth Category—Single Platform

In the fifth category, a game has no levels and no parts; the whole game runs on a single platform. The difficulty level varies depending on the state of the game.

Most board games or puzzles such as Sudoku, Draughts (also called Checkers), and Chess fall into this category. The Sudoku game is played throughout on a single screen consisting of grids of cells. The player selects a cell and writes a number (single digit) in each move. However, it is important to note that a single platform never means simplicity. The implementation of game logic can be complicated. This is true for other board games such as Chess.

The race games such as the Beach Rally mobile game fall into this category. Beach Rally is a 3D game. The track is surrounded by fields and trees. In fact, no new trees are passed by during the race. Instead, the same pattern of trees is continuously repeated. Thus, the whole game is played in a single game space or platform. The structure of a game highlights similarities and differences among different parts of a game. Existing taxonomies do not consider structure at all. For example, all Skill and Action games do not have a similar structure. We present a comprehensive structure of our taxonomy in Figure 9.

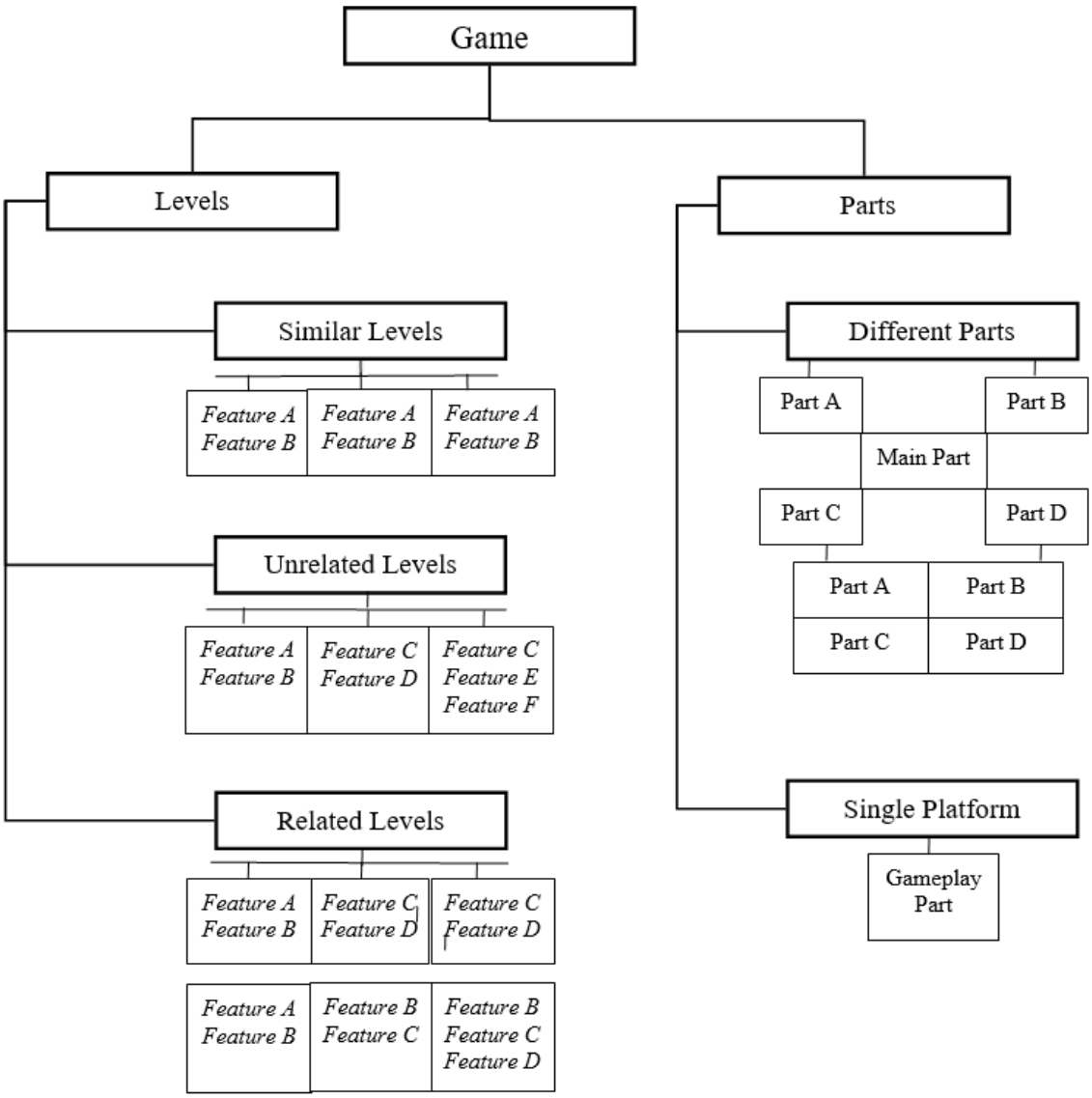

Figure 9. Structure of proposed taxonomy.

4. Design Patterns Based on Structural Similarity

Here, we present design patterns for each of the categories discussed in the previous section. A variant of GoF format is opted for the description and presentation of design patterns. We keep the category and name of a design pattern the same. The applicability of a design pattern is described in a few sentences just after the title. Intent explains the rationale of the pattern. Motivation describes a typical scenario, in which a pattern may be used. As the participants, we present a list of classes and interfaces with their short descriptions. A generic diagram of the pattern outlines the structure of the pattern showing classes with key attributes and methods. A sample code in Java ME is also given. Most of the code snippets have been taken from demo games. To save space, we include interface (and public) methods only, which may be used as a template to start developing a game (with a high-level of abstraction). After discussing each pattern, we present the top-level design pattern, which includes other elements of a game, such as menus, menu options, and other characters.

4.1. Common Patterns

All design patterns that we present here have some common roles (classes, interfaces). Therefore, all these patterns may be thought of as implementation variants of common

design patterns. Instead of talking about those common participants over and again, we prefer to present that common design pattern first.

(1) Participants: Updateable is the interface that should be implemented by all classes, which needs to be updated continuously. These classes include levels, menus, and menu options. The updateable interface is central to all design patterns presented in this paper.
   - AbstractUpdateable implements Updateable. It is the abstract superclass of all Updateable classes. It provides frequently used properties of TheCanvas and utility methods such as wrapping text and cropping images. This class can serve another purpose; including the default implementation of all Updateable methods. It may also serve as an adapter for subclasses that require only Updateable methods.
   - UpdateableManager creates and manages Updateable including paused Updateable.
   - TheCanvas class represents the canvas of the game and is responsible for running the game thread and calling update() method on the current Updateable object. In update() method, calculations and drawings are performed using the Graphics object of TheCanvas. This class receives key events and passes them to the current Updateable for appropriate actions. In Java ME, this class extends GameCanvas.

(2) Diagram: Figure 10 shows the class diagram of the common pattern.

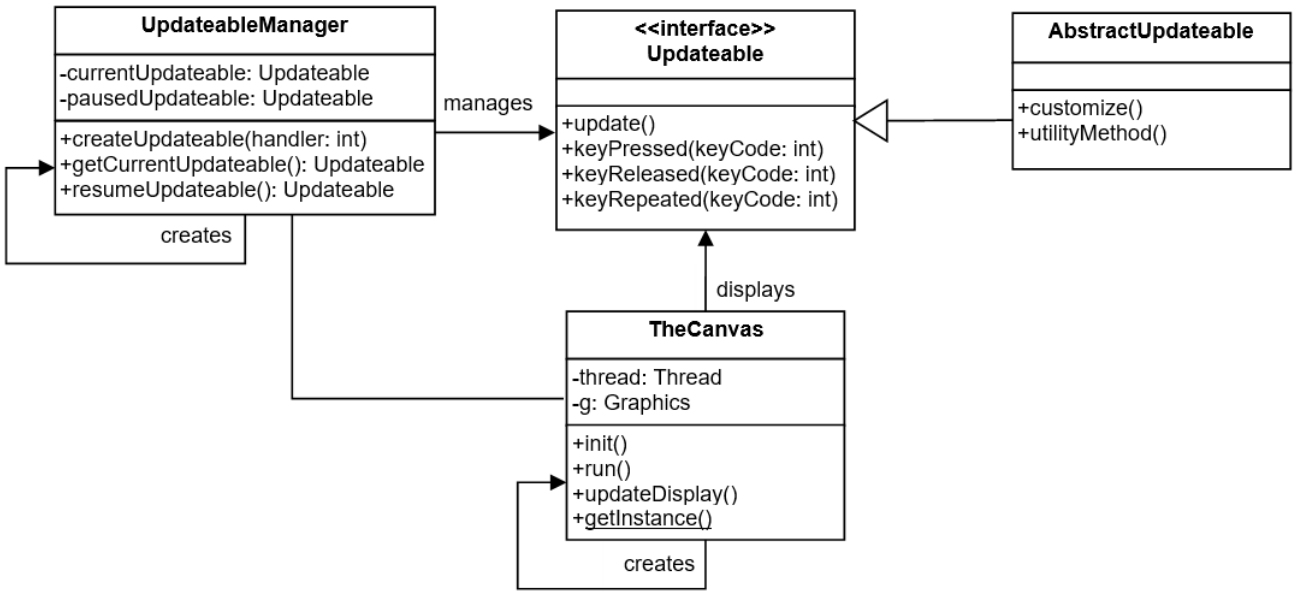

Figure 10. Common classes for all patterns.

(3) Sample Code: Important methods, properties, and other tokens are shown in bold in the sample code in the Appendix A. Actually, only update() and at least one of key event methods (i.e., keyPressed(), keyReleased(), and keyRepeated()) are essential while others may be added if necessary. init() is intended to initialize the updateable before starting for the first time. destroy() may be used to release all resources claimed by the Updateable when it is stopped or, in case memory is too scarce, when it is paused.

The AbstractUpdateable class provides attributes of TheCanvas class invoking static getters on the instance of TheCanvas. These attributes include canvasWidth, canvasHeight, and the most important Graphics object. It seems unsuitable to provide static access to them instead of just including a reference to TheCanvas object that could be used to access those members. The reason is that they are frequently used by all the levels and parts (subclasses of AbstractUpdateable).

TheCanvas class shows a splash screen for a specified duration. After that, the thread continuously calls the update() method on the current Updateable.

The update() method draws the appropriate drawings using the static Graphics object declared in AbstractUpdateable class. Note that GameCanvas.getGraphics() returns a Graphics object associated with the canvas but with protected access. Therefore, to give default access, TheCanvas class provides a static method getGraphicsObj().

4.2. Unrelated Levels

This design pattern may be used for level-based mobile games in which all the levels are different from each other and/or few factors may be common.

(1) Intent: Separate classes with different functionalities for each level and let the super class have common functionality and control.

(2) Motivation: Most mobile games have levels. When a user completes a level, the more challenging level starts next. This process continues until the last level is completed or the player fails sometime during the play session. The levels in some games are not similar and differences outnumber similarities. These differences may be in the game logic, game characters such as an avatar, non-playing characters, opponents, and rules for scoring.

(3) Implementation: In this situation, it is appropriate to implement each level in a different class. If, somehow, there is some similarity or common functionality in all the levels that may be implemented in the superclass of all game levels—GameLevel.

(4) Diagram: The diagram in Figure 11 shows the complete Unrelated Levels pattern. As it is obvious, only a few classes, GameLevel, Level1, Level2, . . . , LevelN, have to be added to Common Pattern. Ellipsis (. . .) between Level2 and LevelN shows that there can be any number of levels, all extending GameLevel directly (Use of ellipsis in this way is not standard in UML, but here we want to clarify our point. We will refrain from this in later diagrams, though).

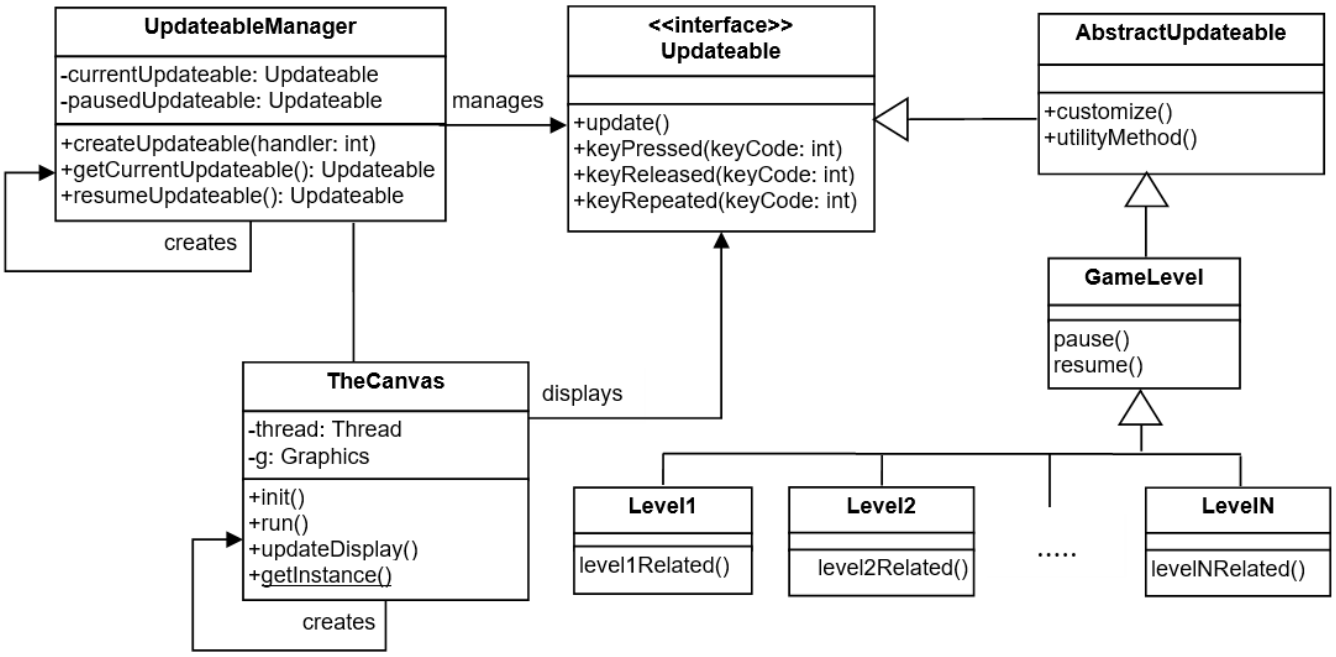

Figure 11. *Unrelated Level* pattern.

(5) *Participants:*

- Updateable, AbstractUpdateable, UpdateableManager, TheCanvas: These participants have already been explained under the Common Pattern category.

- GameLevel is the abstract superclass with subclasses at individual levels. It should be a subclass of AbstractUpdateable. It contains common properties and methods with general implementations such as pause() and resume() methods.
- Level1, Level2, . . . , LevelN are subclasses of GameLevel. The respective level's specific functionalities are implemented in these classes. The UpdateableManager class instantiates these levels based on the level's handler.

(6) Sample Code: The Appendix A shows sample code for the GameLevel class. Though not necessary, the GameLevel class may implement all methods of Updateable interface with common to all functionalities. The levels' classes may call these methods from the same overridden method. For example, update() of GameLevel may be called from update() of Level2 using super.update(). Most of the members in GameLevel are likely to have protected access.

As we can see, the update() method has been defined in GameLevel, and an individual level class may not define it by itself. However, in most cases, each level should update its calculations and graphics itself.

We want to clarify that both keyPressed() and listenKeys() methods are intended to take action when keys are pressed. keyPressed() is Updateable's proxy method for TheCanvas.keyPressed(). It is intended for single key presses, e.g., selecting an option, pausing and resuming the game, and enabling power during gameplay. On the other hand, where keys' state (comprising of all keys collectively) is required, listenKeys() serves the purpose; it captures the state and takes action accordingly. For example, when the avatar has to move up, the "Up" key has to be pressed, when it has to move right, the "Right" key has to be pressed, and when it has to move diagonally towards the top-right, both the "Up" and "Right" keys have to be pressed simultaneously.

(7) Consequences: There can be overlapping similarities among different levels. For example, in a game, a "firing" action can be required in the second and third levels only and "fighting" in the third and fourth level only. In this situation, one can argue that the game cannot have an Unrelated Levels pattern because of those similarities. If similarities are minimal as compared to differences, we may still apply this pattern. The reason is that keeping it in the second category to use the Related Levels pattern requires multiple inheritance. Java (and other modern languages) does not support multiple inheritance (though possible through interfaces, but without code reusability, and many other restrictions).

(8) Relationships: One way to address overlapping similarities is to implement each such similarity in a separate class and instantiate it in the appropriate level class. For the above example, firing and fighting features would be implemented in separate classes and instantiate them in only those level classes which use them. This way, we would prefer composition over inheritance. By separating the commonalities among levels from the intrinsic behaviors of levels, we tend to use the Strategy pattern [55] with the Unrelated Levels pattern. If, somehow, a level has to be added that is mostly similar to an existing level; there will not be any justification to not use the Related Levels pattern (in its original form), even though, because of the majority of dissimilar levels, the Unrelated Levels pattern may be preferred.

4.3. Related Levels

This design pattern may be used for games having related but not identical or similar levels.

(1) Intent: This pattern lets classes with different features extend their functionality from Super Class and lets classes with minor differences extend level classes.

(2) Motivation: In some games, the levels are just slight variants of preceding levels. They usually add very little or no new functionality and vary only in the values of some parameters. In most cases, the levels can be classified based on similarities in levels.

(3) Implementation: Classes for differing levels will extend GameLevel, but classes for slightly differing levels will extend an appropriate similar level rather than extending

GameLevel directly. For example, a game has three different levels "Level One", "Level Two", and "Level Three". Another level, "Level Four", is similar to "Level Three", except that it is more difficult because of the increased number of obstacles (or includes another type of obstacle). In this case, classes for Levels 1–3 will be implemented in respective classes directly extending GameLevel, but classes for Level 4 will extend the Level 3 class. A level can be extended by some other levels. Most of the class members will have protected access modifiers to ensure that subclasses can access them. There can be any number of level classes at any level of inheritance (from GameLevel).

AbstractUpdateable has already been described under Common pattern. Any individual design pattern has to be substituted to a Common (top level) pattern through this interface to build a complete pattern.

(4) Diagram: Figures 12–14 present class-diagrams of different variants of this design pattern.

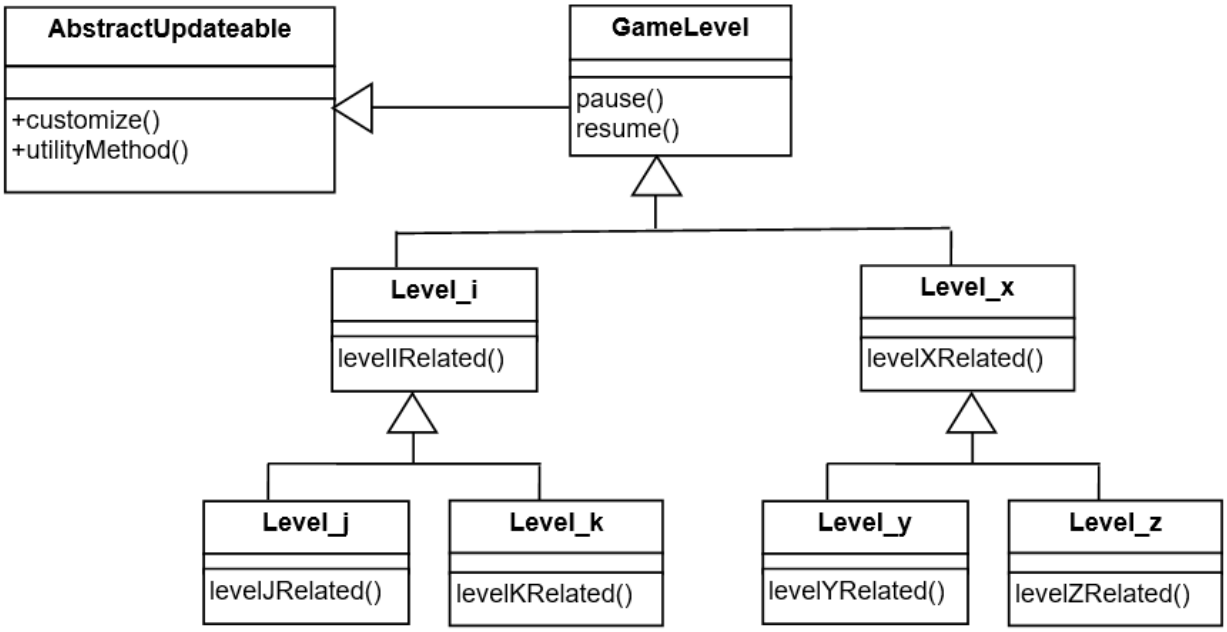

Figure 12. Related Levels pattern with default approach.

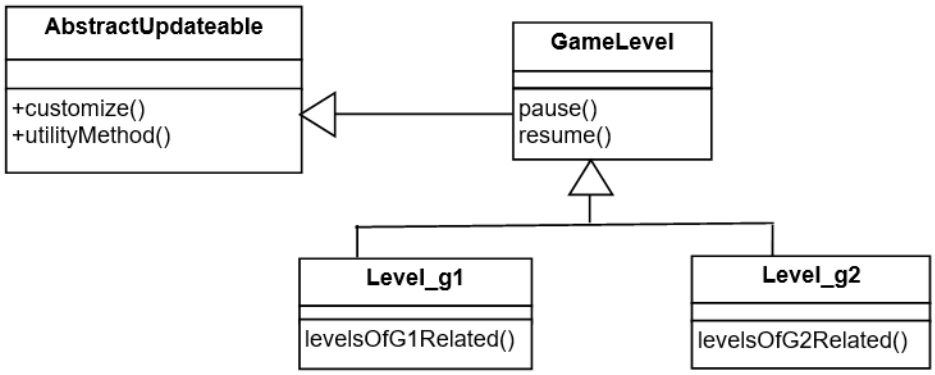

Figure 13. First alternative approach for implementing *Related Levels* pattern.

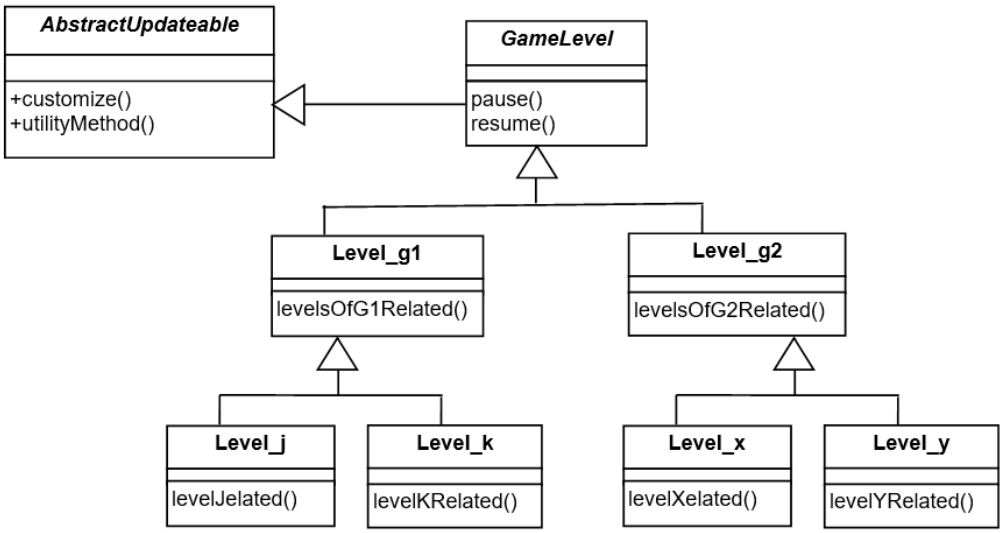

Figure 14. Second alternative approach for implementing *Related Levels* pattern.

(5) *Participants:*
- GameLevel is the abstract superclass of all the level classes. This is similar to the one discussed under Unrelated Levels pattern.
- Level_i, Level_j, Level_k, Level_x, Level_y, Level_z are classes for arbitrary levels.

(6) Sample Code: The Appendix A shows the default approach of Related Levels pattern that is a trade-off between two alternative approaches just discussed above. The GameLevel class may not be much different from the one mentioned in the last design pattern. For this reason, we skip this class. The sample code is for individual-level classes.

It is clear from the sample code that no variables are declared to store enemy-related properties in LevelOne. Level three introduces enemies; therefore, LevelThree class declares variables to store enemy related information. Suppose, LevelFour does not introduce any new features, but includes all those that are present in LevelThree. For example, it just increases the number of stars and the number of enemies. LevelFour extends LevelThree in this case. Even LevelFour can add its functionality as the design does not restrict it from doing that.

(7) Consequences: We can use any one of two approaches to refine the design. In both approaches, similar or related levels are categorized into groups.
- In the first approach, a separate concrete class can be used for each group instead of using separate classes for individual levels. A level will be nothing more than an instance of the class for the respective group. The customization of levels will be done by parameters to a constructor and/or through setters. This refinement is shown in Figure 13.
- In the second approach, a separate abstract class can be used for each group. Then, each level class belonging to the group will extend that abstract class. Not to mention, each group's abstract class will extend GameLevel. This approach will make each level more independent—a plus point, but this will increase the number of classes. Figure 14 shows this approach.

As we mentioned under "Second Category—Related Levels" in the previous section, there are two cases which can place a game in this group. Referring to those cases, if all the levels of a group are similar, the first alternative approach will be suitable; otherwise, the second approach is preferable.

The first alternative approach makes each level more independent and is better to organize levels, but it results in a large number of classes even if the levels do not have

critical differences. Suppose we have to add a new level which does not fit in any group, a new group has to be created first (i.e., a new group class will have to be created extending GameLevel), then a concrete class representing the new level will have to be created. Another point (either positive or negative) is that all level classes lie on the same level of inheritance.

In the second approach, making levels as objects instead of classes, we limit them to functionalities defined in the respective class representing the group. Thus, to add new functionality in a level, a separate group class has to be created. This is in contrast to the original and first design alternative in which we implement the new functionality in an already existing class for the level. For this reason, this alternative may not be preferable in most cases.

The design pattern in its original form (i.e., not refined) is a tradeoff between the bulk of classes at a single level of inheritance of first design alternative and the too-compact design with closed openness of the second alternative. This form is a bunch of inheritance relationships; GameLevel is at the top, and it is extended by classes of levels that add new functionalities. These functionalities should be, however, different from those implemented by classes of the same level of inheritance. For example, if LevelX and LevelY are direct children of some common parent, it should mean that both of these classes implement new functionalities that are different from each other.

For the addition of a new level, there are three possibilities:

- If it has functionalities other than those implemented by LevelX and LevelY, the new class LevelZ will be a direct subclass of GameLevel.
- If it is similar to any existing level, it will extend the class of that existing level.
- If it contains all functionalities of an existing level in addition to any new ones, it will extend that existing level.

This suggests that inheritance can go down to any level. Thus, this pattern is usually open to adding new levels.

(8) Relationships: This pattern is an extension of *Unrelated Levels*. In a game having unrelated levels, if some new level similar to a previous level has to be added, the new level will have to extend the previous one. Thus, the game would no longer belong to the Unrelated Levels category, instead, to the Related Levels category. The relationship between two patterns may also be judged if the individual level in the Unrelated Levels (see Figure 12) is compared with the group in the Related Levels' first alternative approach (see Figure 13).

4.4. Similar Levels

This pattern may be used to develop mobile games consisting of similar levels. The difference is only because of the different values of parameters.

(1) Intent: GameLevel class has major control and any variation may be controlled through parameters.
(2) Motivation: Most games offer levels to the players. All levels are entirely similar regarding graphics, game actions, key functions, and other features. The difficulty usually increases with each next level because of more, but the same obstacles to overcome or less time to catch a bonus. All levels seem to be instances of one screen.
(3) Implementation: This scenario suggests that only one concrete class should be there to represent any of the game levels. This class could implement the whole game logic. All levels are nothing more than instances of this class. Thus, this pattern does not require the creation of a separate class for each level. The customization of levels may be done by parameters to the constructor and/or through setter methods. Another option to customize a level is to implement an automatic technique. For example, the number of enemies in each next level would be increased by 20%. A variable should be there to store the number of levels. Access to this variable can be given to

the player through an input field in Settings to let him/her set the number of levels himself/herself.

(4) Diagram: Figure 15 explains class diagram of the Similar Levels pattern.

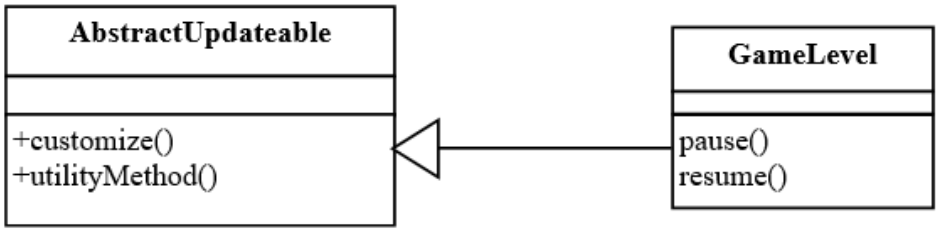

Figure 15. Similar Levels pattern.

(5) Participants: GameLevel is the only concrete class that is used as a template to create levels. This class extends AbstractUpdateable.

(6) Sample Code: The Appendix A presents sample code for GameLevel class. The code in the Appendix A shows how instances of levels created by following code in the UpdateableManager class.

(7) Consequences: This pattern is the simplest from all patterns presented above. Because of its simplicity, it does not have any design options or tradeoffs. Other than classes and interfaces common to all patterns, the only essential class is GameLevel, which, unlike all other patterns, is a concrete class. This does not mean that the pattern is only for simple games. There is a potential for GoF and other patterns to take part in the implementation of such additional complexities.

4.5. Different Parts

This pattern may be used to develop mobile games with different parts, where one part usually implements the main game environment.

(1) Intent: This pattern suggests that game logic should be divided into different parts and communication between parts may be through an interface.

(2) Motivation: In some games, the game logic is divided into different parts. The player has to enter and exit these parts at certain points. These parts are different enough to handle them differently. In the first type, one part usually represents the main game environment where a player has to deal with enemies with the help of friends to achieve the goal. Other parts are supporting the main gameplay in one way or the other. In the second type, the main game environment is partitioned where each part represents a different location or scene. Though, each part has main game-related tasks, they are sufficiently different from others to be implemented in a separate class. Adventure games are most likely to have this pattern.

(3) Implementation: As each part is different from all others, a separate class would be used for each part. All of these classes will directly extend AbstractUpdateable because there is nothing common among the parts. It is quite possible that parts have to communicate with each other especially while leaving one part to entering another. Certain factors might be shared in this communication. For example, in a game, coins may be earned in the main part of the game, which may be spent in the shop part. Similarly, the powers or weapons bought in the shop would be used in the main game part. To address this sharing, a class with shared features is required. An appropriate name for this class may be GameStatus, since the status of the game is shared.

(4) Diagram: Figure 16 represents class diagram of the Similar Levels pattern.

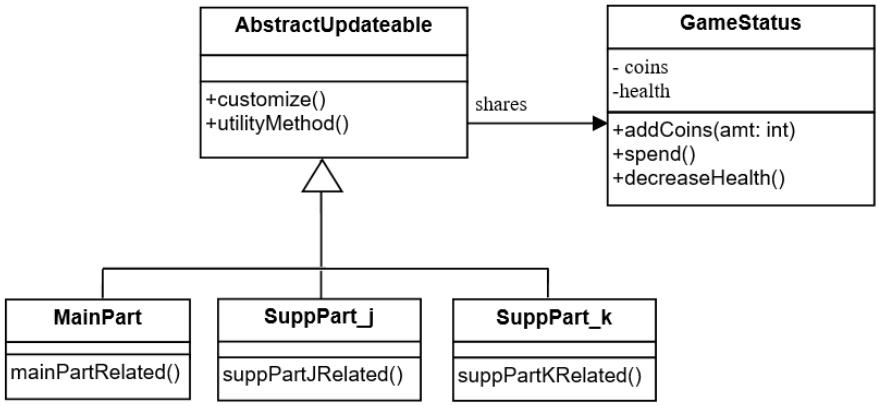

Figure 16. Class diagram of Different Parts pattern.

(5) Participants:
- MainPart class includes the implementation of the main game logic and environment.
- SuppPart_j, SuppPart_k (an arbitrary number of parts) are classes implementing each supporting part.
- GameStatus class stores the status of the game, which is shared by different parts.

(6) Sample Code: The Appendix A depicts sample code for the Different Parts pattern.

(7) Consequences: Each part of a game can have entirely different logics and design. The diagram and sample code shown here are minimal. In practice, there can be a group of classes for each part. This design pattern does not take details and complexities of a part into account. The pattern does not limit the complexity of a part as long as it can be identified as a part.

(8) Relationships: Although Different Parts pattern seems a simple variation of Unrelated Levels, yet there is a definite difference between them. In Unrelated Levels, we talk about levels, while in Different Parts we talk about parts.

4.6. Single Platform

This pattern may be used to develop mobile games that have a single game area or environment/platform.

(1) Intent: This pattern suggests a single platform for the development of games.

(2) Motivation: Board games such as Checkers and Sudoku have a single environment. A single screen (usually static) shows the board. These games may not have any levels. The complete game from start to completion is played in one virtual place or platform.

In some other types of games, though screens representing game environment seem dynamic, they actually have a single platform; thus, start and completion take place in that platform. There may be levels, but the transition between them is not clear.

(3) Implementation: In both scenarios, a single class may be used to implement the platform. If a platform is too complicated to be implemented in one class, a group of classes may be used. For example, one class implements cells, another class implements color patterns, and one class represents, for example, the mainboard. Unlike other patterns discussed above, it is not important that all of these classes should extend a common superclass.

In fact, there can be a single class serving the purpose of both GameGraphics and GameLogic. On the other hand, there can be a number of classes to make up the game's graphics part and a number of classes to implement the game's logic.

(4) Diagram: Figure 17 presents the class diagram of the Single Platform pattern.

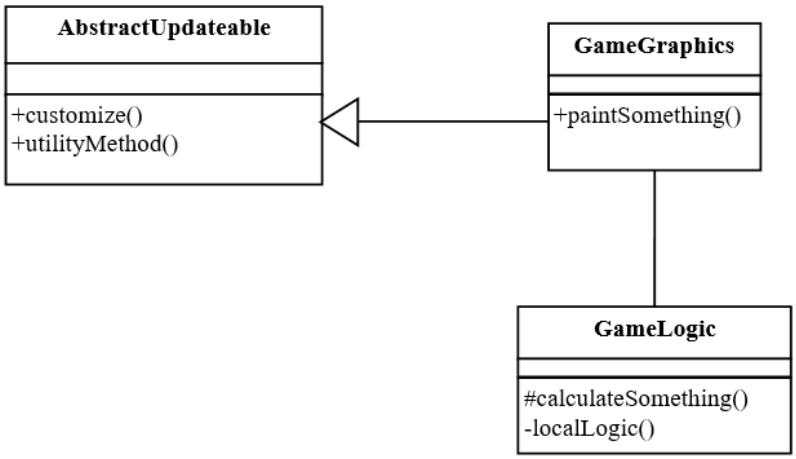

Figure 17. Single Platform pattern.

(5) Participants:
- GameGraphics class contains an implementation of view of the game. All drawings may take place in this class.
- The GameLogic class contains an implementation of game logic, which includes calculations and decisions.

(6) Sample Code: The Appendix A shows sample code for this pattern. GameDesign class will be very similar to the GameLevel class of the Similar Levels pattern, except there will be only one instance of this class unlike a separate instance of GameLevel for each level. GameLogic is supposed to implement game logic (similar to Model in MVC architecture). As mentioned above, there may be more than one class for this purpose. Additionally, it is possible that there is no separate class for game logic at all. The GameDesign class is doing everything instead. However, this will tend GameDesign to be a God Class, which is undesirable. Consequences: A quite simple game consisting of a single platform may have one class implementing both graphics and logic of the game (instead of using separate classes for both). However, this pattern's simplicity does not mean that all games having this pattern will be simple. A game's overall structure also depends on the game's logic besides the top-level view. For example, a puzzle may consist of a single screen, but the implementation of the puzzle logic may require thousands of lines of code. Hence, even if there is at least one essential class as a participant of the pattern, there may be other supporting classes. Thus, naturally, this pattern is open to including new patterns in its structure.

(7) Relationships: This pattern can lead to MVC architecture. Indeed, it may be a MVC pattern, if MIDlet is considered as Controller, GameLogic class as Model, and Updateable, AbstractUpdateable, and GameDesign as classes belonging to the View part.

4.7. Top Level Pattern of a Complete Game

All the individual design patterns discussed above focus on game levels/parts or the main game platform. They do not discuss other features such as game characters and menus.

An object that can be drawn on canvas and is a part of a game is called an Entity. A game can have a number of entities. Each entity is represented by an image.

Entity class encapsulates the features of an entity. Most likely, the Entity should be a direct subclass of Sprite in Java ME. The Sprite class includes handy methods such as collidesWith(), setLocation(), and setVisible(). All characters (e.g., Enemy, Pad, Fire, etc.) may extend the Entity class to add their behaviors. If a character does not have its own overridden features, it can be a direct instance of an Entity.

Although menu and menu options are not part of the gameplay and game engine, they are still an important part of any game. Each design pattern is just a substitute for

one main pattern. This pattern is shown in Figure 17. There may be a number of design patterns for menus and menu options. In our case, classes for menu and all menu options should implement the Updateable, the same interface implemented by classes of all levels or parts in our design patterns. For convenience, they can extend the AbstractUpdateable class instead. All menu option classes should extend the abstract class MenuOption.

The Unrelated Levels design pattern enclosed in a dashed box has been substituted as an example. Any individual design pattern can be substituted likewise. Figure 18 represents the complete generic design pattern for mobile games. All our demo games use this pattern. The diagram also shows how Entity classes can be substituted to an individual pattern. Any extended or concrete level can have this relationship with Entity depending upon requirements. Here, the GameLevel class has been included to show how the Entity is connected to a design pattern (as GameLevel is a participant in three of our patterns). For the Different Parts pattern, classes implementing parts, and for Single Platform pattern, GameDesign class will be in a relationship with the Entity instead of GameLevel class.

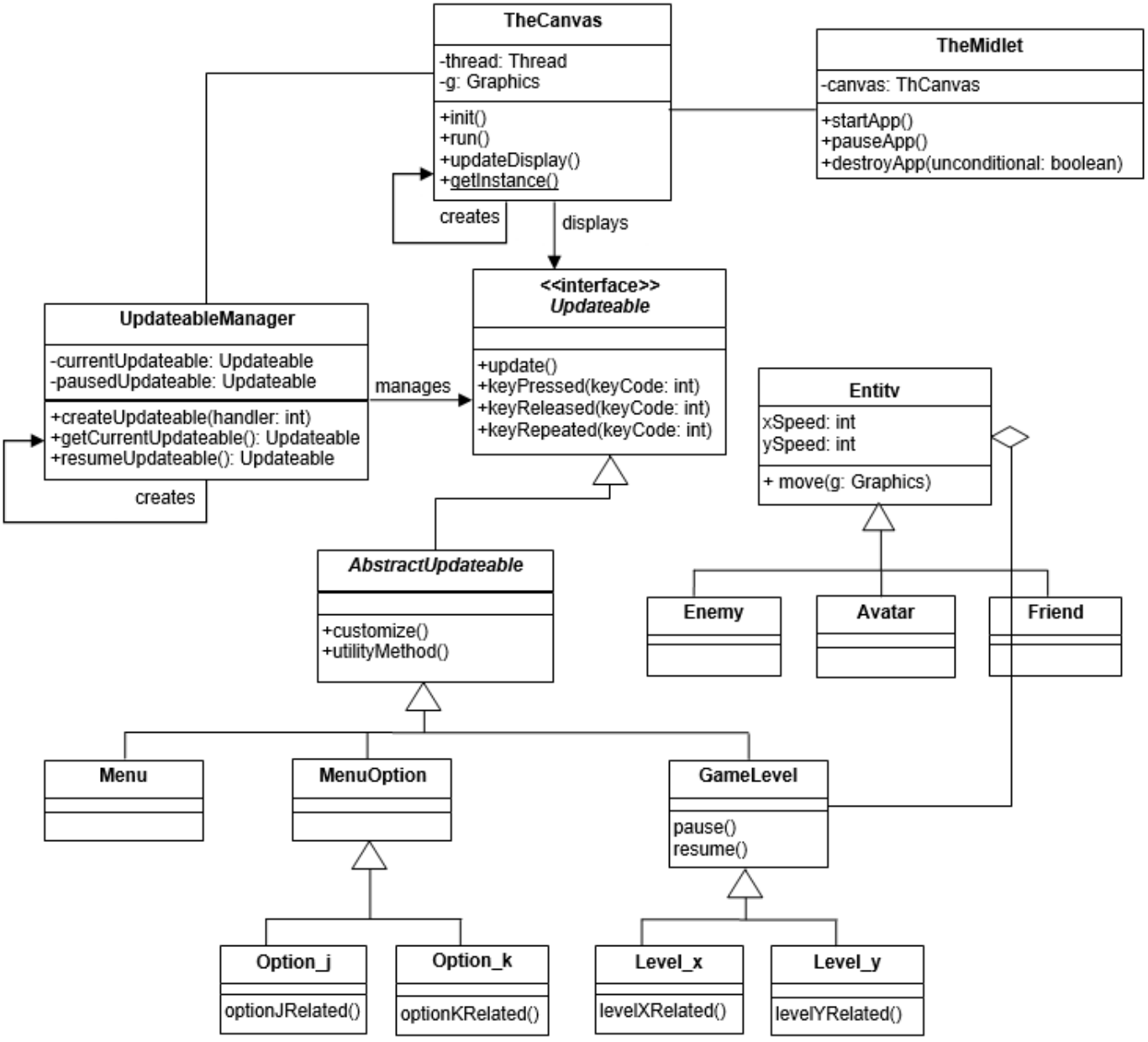

Figure 18. Integration of an individual pattern into a complete game.

As we mentioned earlier, we have developed four demo games for mobiles using the first four categories. They are simple and in accordance with the diagrams presented in this paper. The games are:

- ShootDown uses Unrelated Levels pattern.
- GrabStars uses Related Levels pattern.
- Avoid uses Similar Levels pattern.

OnJourney uses Different Parts pattern.

5. Case Study (Demo Games)

To validate the proposed taxonomy and design patterns, we have developed four demo games, namely Avoid, ShootDown, GrabStar, and OnJourney, as a case study using first four design patterns (i.e., similar levels, unrelated levels, related levels, and different parts). The fifth pattern, "single platform", is pretty simple having only one or two essential classes, and we did not apply that pattern in our demo games. These games are evaluated by two graduate students studying at COMSATS University Lahore Campus. In the first phase of the case study, we investigated how already developed games can be categorized based on our proposed taxonomy. We realized that a number of already developed games can be placed under any one of the categories proposed by our taxonomy as shown in Table 1.

Table 1. Mapping of existing games with categories of taxonomy.

Games	Category/Level
Color Bridge	Similar Levels
Block Breaker, DX Ball	Related Levels
Seeding by NEWGROUNDS, Haste-Makes-Waste	Different parts
Sudoku, Draughts, Beach Rally	Single platform

In the second phase, we present summarized information about four demo games that we developed to realize the concept of design patterns.

5.1. Design Pattern: Similar Levels (Game: Avoid)

5.1.1. Scenario

This game consists of levels. In all levels, fireballs are continuously falling at random places and the Avatar is a pad. The player has to move it left and right to avoid the fireballs. To complete a level, a player has to avoid all fireballs until the level time finishes. However, if a fireball hits the pad, one life is deducted. When all lives are deducted, the game is over. Four-pointed stars also fall at random times. If a player obtains the star, a bonus of a hundred points is awarded.

Figure 19 demonstrates the snapshot of a sample level of the Avoid game.

Figure 19. *Avoid* demo game.

5.1.2. Justification

As we can see, no new functionality is introduced at any level. The difficulty of each next level increases only by adding more fireballs and varying level time. Thus, all levels are similar, and they are just instances of one concrete class named as GameLevel. The properties that vary level by level can be set using parameters to the constructor.

5.1.3. Consequences of Using the Design Pattern

This pattern is the simplest of all the design patterns. Due to its simplicity, it does not have any design options or tradeoffs. Other than classes and interfaces common to all patterns, the only essential class is GameLevel which, unlike all other patterns, is a concrete class. There is a potential for GoF and other patterns to take part in implementing such additional complexities.

5.2. Design Pattern: Unrelated Levels (Game: ShootDown)

5.2.1. Scenario

This game consists of unrelated levels. In the first level, planes are flying down. The player has to target and fire them within a given time limit. A player can move the target in any direction. When a plane comes into the target, it can immediately be fired. In the second level, enemy helicopters have to be targeted and fired from a fixed anti-aircraft gun (a.k.a., cannon in the game). In the third level, an anti-aircraft tank (a.k.a., tank in the game) can fire only vertically. Thus, to shoot down the smart helicopters that can drop bombs, the tank has to be adjusted by moving it. When either level time finishes before hitting all enemies in a level or the tank comes under bombs twice, the game is over. Figure 20 presents snapshots of different levels.

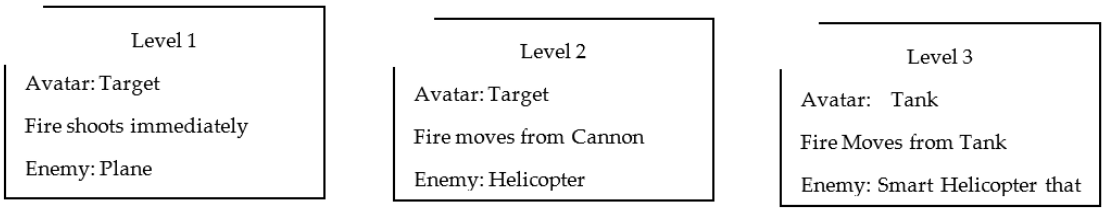

Figure 20. Different levels of the ShootDown demo game.

5.2.2. Justification

We can see from the snapshots of Figure 20 that all three levels are different from each other. The common factor in all of them is that there are enemies that have to be targeted and fired. The type of enemy, how to target it, and how it is hit when the player presses the fire button is different; hence the implementation of each level is different from others. In first and second levels, the target is the avatar. A player can control it using arrow keys. In the third level, the avatar is the tank. A player can move it horizontally using left and right keys. In the first level, when a player presses fire, the plane under target is immediately hit. In the second level, fire moves from the cannon towards the target. While in the third level, fire moves vertically upward from the tank. In the third level, unlike other levels, helicopters can drop bombs, which have to be avoided by moving tank away from them. It is important to mention that images for each level are also different. With these differences among different levels, we can justify saying that this game has an Unrelated Levels pattern. There are common factors for all the levels. These include Level Complete, Level Failed, Pause, and Calculate Score. All these levels would be implemented in GameLevel, which is the abstract superclass of all levels.

5.2.3. Consequences of Using the Design Pattern

One can argue that first and second levels have similarities such as target. Thus, why should they not share a common superclass that implements the "target" feature? (i.e., does it not have a Related Levels pattern instead?) First, the similarities are minimal as compared

to the differences. As this is a demo game, we have tried to make things simple. There is a possibility in a full-scale game that levels bear negligible similarities while having essential differences. Second, whereas the "target" feature is common in the first and second levels, the "fire" feature is common in the second and third level. The question arises of how to address this overlapping of similarities in a language like Java, where multiple inheritance is not possible (though possible through interfaces, but without code reuse, and many other restrictions)? Hence, we cannot put this game and all such games (having too much overlapping of similarities among levels) under the Related Levels category. One way to address these types of similarities or commonalities is to implement each such similarity in a separate class and instantiate it in the appropriate level class. Through this way, we may prefer composition over inheritance. By separating commonalities among levels from intrinsic behaviors of levels, we tend to use the Strategy pattern [55] along with the Unrelated Levels pattern. Figure 21 shows this pattern with a sort of Strategy pattern used.

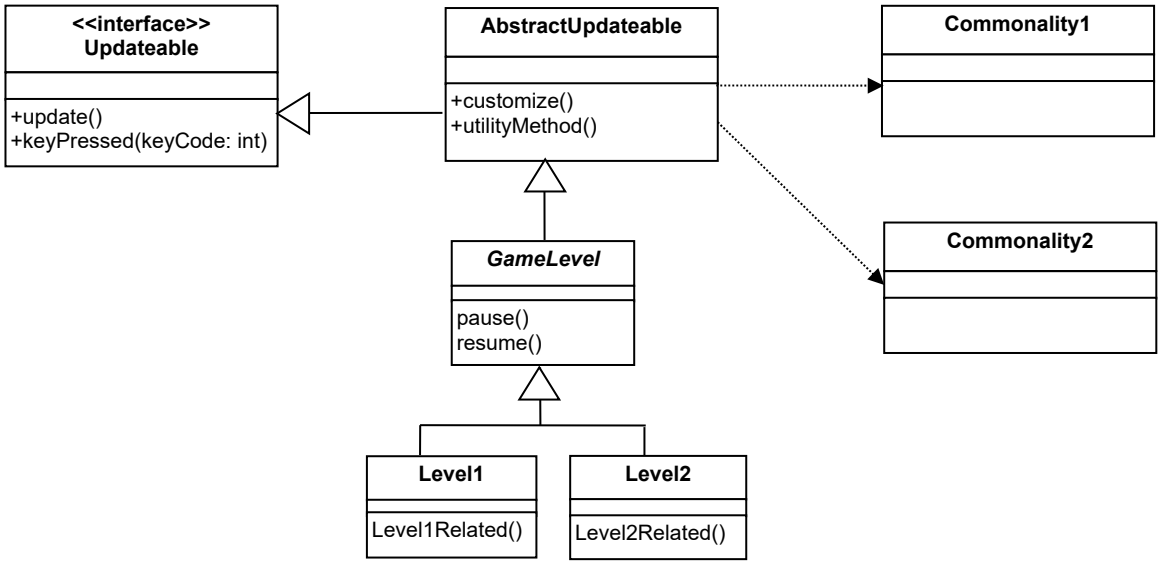

Figure 21. Unrelated Levels pattern with Strategy pattern.

5.3. Design Pattern: Related Levels (Game: GrabStar)

5.3.1. Scenario

This game is developed based on the concept of related levels. The player has to collect all stars, which are continuously moving on the screen. To collect stars, an arrow has to be moved and collide with them. All the stars have to be collected within a specified time; otherwise, the game will be over. In the third level, in addition to collecting stars, the player has to avoid cross-shaped enemies. Colliding with an enemy will decrease one life, and when two lives are lost, the game is over. The fourth level is similar to level three. Figure 22 shows the snapshots of different levels.

5.3.2. Justification

The default and common functionality in all levels is to collect stars by moving the arrow towards them and colliding with them. This functionality is implemented in the abstract superclass GameLevel. The first and second levels do not add any new functionality; thus, their respective classes are subclasses of GameLevel in its simplest forms. They customize attributes such as "level time" and "number of stars". The third level introduces enemies, which should be avoided. Thus, the Level 3 class overrides the paint() method to add this new functionality, but it also calls super.paint() to include the default functionality. Besides, new data members (such as noOfEnemies) are added with protected access (i.e., subclasses may access them). The fourth level is the same as the third

level except the number of enemies is increased, and some other properties are changed. Hence, this game correctly has a Related Levels pattern.

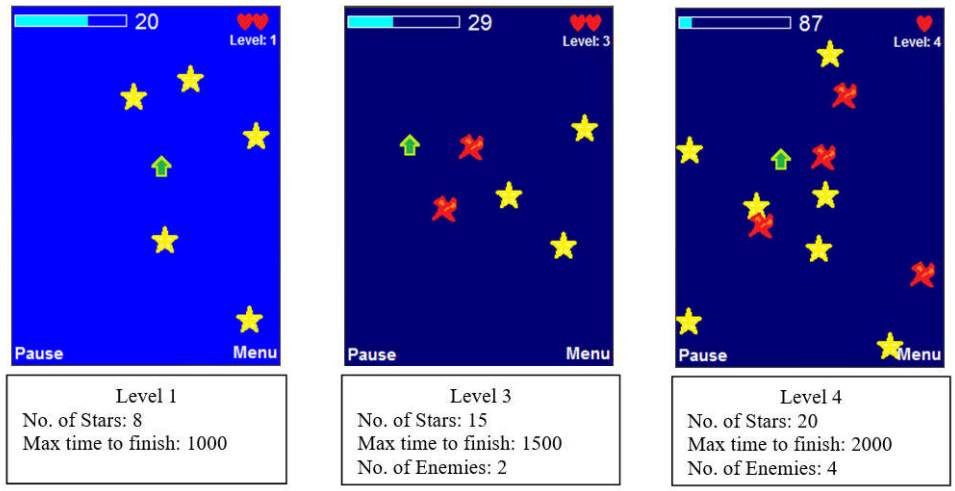

Figure 22. Related levels of GrabStar demo game.

5.3.3. Consequences of Using the Design Pattern

The consequence of using this design pattern are discussed in Section 4 (C Related Levels).

We can infer from discussion that the inheritance can go down to any level. Thus, this pattern is usually open for adding new levels. In one option, all classes of related or similar levels are direct subclasses of one common abstract class, which we call "group class". This group class is, in turn, a direct subclass of GameLevel. This alternative makes each level more independent and is better to organize levels, but it results in a large number of classes even if levels do not have critical differences. For example, if we want to add a new level which does not fit in any group, a new group has to be created first (i.e., a new group class will have to be created extending GameLevel), then a concrete class representing the new level needs to be created. Another point (either positive or negative) is that all level classes lie on the same level of inheritance.

The second alternative is that all similar levels are just instances of one concrete group class. Making levels objects instead of classes, we limit them to functionalities defined in the respective class representing the group. Thus, to add a new functionality in a level, a separate group class has to be created. This is in contrast to the original and first design alternative in which we implement the new functionality in an already existing class for the level. For this reason, this alternative is not preferred. In one sense, this alternative is an enhanced version of Unrelated Levels; if we want to add a new level similar to an existing one in Unrelated Levels, would not it be better to create just a new instance of the existing level class? However, to do this, the existing level class should have some interface to customize its attributes (e.g., level time and level bonus)

The original design option of this pattern is a trade-off between the bulk of classes at the single level of inheritance of first design alternative and overly compact design with close openness of the second alternative.

5.4. Design Pattern: Different Parts (Game: OnJourney)

5.4.1. Scenario

This is an adventure game developed on the concept of different parts. The hero named Bidiro has to set out on a journey. Before the player starts playing, guidelines for the journey are given. Bidiro is the avatar in the main game part. Once the game starts, Bidiro starts walking towards his destination. While walking, he can get coins by jumping at specific locations. Walking decreases health. There are shops beside the road at certain destinations. Bidiro can enter in a shop to buy food from earned coins to regain health. He

can also hire a taxi or a bicycle if he can afford them. Minimum hiring will take him up to the next shop. By riding a bicycle, he can go faster with less decrement in health. Similarly, using a taxi can let him travel even faster with a minimal decrement in health. However, Bidiro can earn coins only when he is walking. The goal is to reach the destination as early as possible. Figure 23 presents the snapshots from different parts of the demo game.

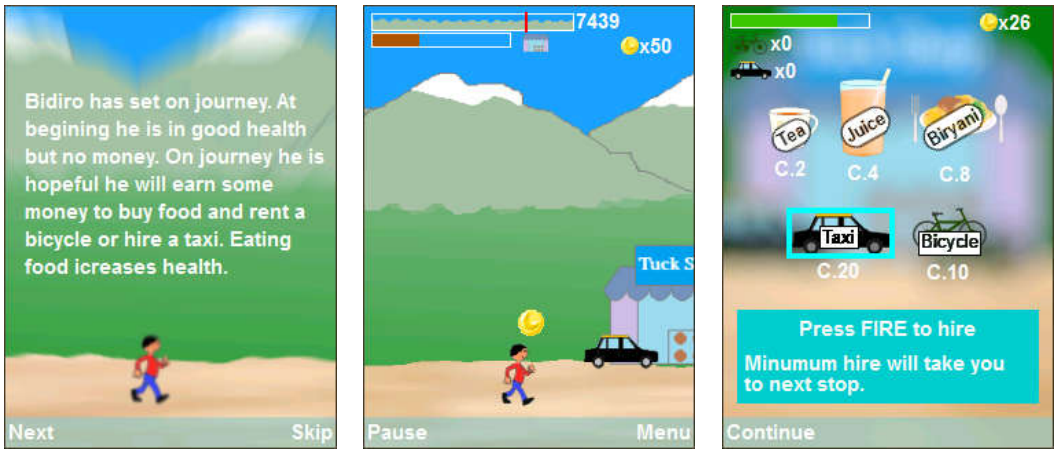

Figure 23. Different parts of OnJourney demo game.

5.4.2. Justification

OnJourney is a perfect example of the Different Parts pattern. All parts have nothing in common except some images used as icons. The Introductory part is about an overview of the game and guidelines for the players. This part has nothing to do with gameplay since it does not take any input about the upcoming Main part from the player. The Main part is where the player can earn money, utilize health and any resources he/she bought from that money to reach the destination at earliest. At specific points, he/she can enter the shop, where the Shop part begins. Exiting the Shop part leaves the player back in the Main part. When he/she reaches the destination, the End part begins. Each part has been implemented in a separate class. The Shop and Main parts share things; coins earned in Main are spent in Shop, and food bought in Shop (and eaten) increases the health, which will help him travel further in Main, and taxi and bicycle hired in Shop can be used to go faster in Main. As suggested by the design pattern, the shared things are addressed using the Status class. An instance of this class is created in the MainPart class, and a reference to it is made available in the ShopPart class. The Status class contains variables for storing coins, health status, and the number of taxi and bicycle hires and methods to use these resources.

5.4.3. Consequences of Using the Design Pattern

Each part of the game can have entirely different logic and design. The diagram and sample code shown here are minimal. In practice, there can be a group of classes for each part. This design pattern does not take details and complexities of a part into account. The pattern does not limit the complexity of a part as long as it can be identified as a part. Although the Different Parts pattern seems a simple variation of Unrelated Levels, there is a definite difference between them. In Unrelated Levels, we talk about levels, while in Different Parts we talk about parts. In a play session, one has to play a game level by level, where each next level is either more challenging or lengthier, while the game parts represent different locations or scenes of the game environment. In level-based games, one may not enter into any level in a single play session; he can enter only the next level. On the other hand, when a game has different parts, there may be two possibilities: either the player can enter any adjacent part or can enter only a specific part through some command or any other way.

6. Conclusions and Future Work

Software design patterns are used to develop quality applications. Consequently, their application for the development of mobile games is very beneficial to attain the benefits of easy evolution and reusability. Besides, design patterns improve the development process by providing already tested and proven solutions to frequently occurring game design problems. Design patterns can be identified and reused in various parts of mobile games such as menus, placements of non-player-characters, scoring, and other entities. Genres of mobile games also have an impact on the design patterns used in them. Thus, all games belonging to a specific genre would have some commonality in design patterns occurring in them.

In this paper, mobile games have been categorized based on a new taxonomy that considers top-level structural similarity without considering the details of the games. By structure, we mean how different parts of a game are related to each other and how they are placed together to form a whole game. Each pattern is a part of one design pattern that includes other game entities, menus, and menu options as well.

Based on taxonomy, we introduce five novel design patterns dedicated to mobile games: Unrelated Levels, Related Levels, Similar Levels, Different Parts, and Single Platform design patterns. We performed a pragmatic validation of the proposed design patterns and demonstrated their applicability through the implementation. To validate the proposed design patterns, we developed four demo games, namely Avoid, ShootDown, GrabStar, and OnJourney. The source code of these games can be used as templates to develop new games. Together with the base taxonomy, design patterns are programming patterns outlining core classes and interfaces of games' top-level structure. Hence, game designers and developers may use these patterns for the development of games.

The proposed taxonomy can be enhanced by including new factors in the future. Currently, the proposed design patterns and developed games are evaluated by two graduate students. There may be threats to external validity of our proposed design patterns and their applications for all type of mobile games. In order to mitigate this threat in the future, we plan to evaluate proposed design patterns and developed games from academia and the software industry on a large scale. Based on the feedback from academia and industry, the existing design patterns may be modified to reflect any enhancement in the classification and taxanomy. More categories may be added to encompass a large number of existing games. We also plan to empirically study the benefits of applying design patterns in mobile games development. This can be done through a direct comparison for the games that follow or do not follow design patterns during development of mobile games.

Author Contributions: Conceptualization, G.R., Y.H., T.U. and J.R.; methodology, G.R., Y.H., T.U. and J.R.; software, G.R., Y.H., T.U. and J.R.; validation, G.R., Y.H., J.R., S.F.Y. and F.S.; formal analysis, G.R., Y.H., T.U. and J.R.; investigation, G.R., Y.H., T.U. and J.R.; resources, G.R., S.F.Y. and F.S.; data curation, G.R., Y.H., J.R., S.F.Y. and F.S.; writing—original draft preparation, G.R., Y.H. and J.R.; writing—review and editing, G.R. and J.R.; visualization, G.R., Y.H., J.R., S.F.Y. and F.S.; supervision, G.R. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A

Appendix A.1. Sample Code of Common Pattern

```
public interface Updateable
{
void init();
```

```
void update();
void keyPressed(int keyCode);
void keyReleaed(int keyCode);
void keyRepeated(int keyCode);
void destroy();
}
public abstract class AbstractUpdateable implements Updateable, GameConstants
{
protected static canvasWidth = TheCanvas.getInstance().getWidth();
protected static canvasHeight = TheCanvas.getInstance().getHeight();
protected static Graphics g = TheCanvas.getInstance().getGraphicsObj;
 . . .
//protected methods such as setDelay() etc.
//Utility methods such as cropImage() etc.
}
public class TheCanvas extends GameCanvas implements Runnable, GameConstants
{
 . . .
private Updateable currentUpdateable;
private Thread runner;
private final Graphics g = getGraphics();
 . . .
public void start()
{
//show splash screen...
runner.start();
}
public void run()
{
while(go) {
if(splashCtr < SPLASH_DELAY) {
//show splash screen
}
else {
currentUpdateable.update();
flushGraphics();
}
Thread.sleep(currentDelay);
}
}
public void keyPressed(int key)
{
currentUpdateable.keyPressed(key);
}
Graphics getGraphicsObj()
{
return g;
}
 . . .
}
public class UpdateableManager implements GameConstants
{
private Updateable currentUpdateable;
private Updateable pausedUpdateable;
```

```
...
public Updateable createUpdateable(int handler)
{
...
switch(handler) {
case MENU_SCR:
currentUpdateable = new Menu();
break;
...
case LEVEL_ONE:
currentUpdateable = new LevelOne();
break;
case LEVEL_TWO:
currentUpdateable = new LevelTwo();
break;
...
}
return currentUpdateable;
}
public Updateable createUpdateable(int handler, boolean pause)
{
if(pause) {
pausedCtrl = control;
pausedUpdateable = currentUpdateable;
}
return createUpdateable(handler);
}
public Updateable resumeUpdateable()
{
control = pausedCtrl;
currentUpdateable = pausedUpdateable;
return currentUpdateable;
}
...
}
```

Appendix A.2. Sample Code of Unrelated Levels Pattern

```
public abstract class GameLevel extends AbstractUpdateable
{
//protected data members...
...
public void update()
{
if(!paused) {
paint();
if(!levelCompleted && listenKeys) {
listenKeys();
}
}
}
protected void paint()
{
if(!levelCompleted && !failed)
{
```

```
createScoreboard();
if(timeElapsed > timeToFinish) {
failed = true;
}
 . . .
}
else if(levelCompleted) {
levelCompleteScreen();
}
 . . .
}
public void keyPressed(int keyCode)
{
//actions to key-events which are common to all levels...
}
protected void listenKeys()
{
//Actions based on key states, common to all levels
}
//protected methods such as pause() and resume()...
//other game specific methods...
}
public class Level1 extends GameLevel
{
private Image targetImg;
private Image jetImg;
//Other level-specific attributes
...
protected void customize()
{
setDelay(gameDelay);
timeToFinish = 400;
...
}
public void init()
{
//initialize variables...
}
protected void paint()
{
super.paint();
//level-specific drawings...
}
protected void listenKeys()
{
//level-specific actions against key-states...
}
//Other methods...
}
```

Appendix A.3. Sample Code of Levels of Related Levels Pattern

```
public class LevelOne extends GameLevel
{
//protected variables...
```

```
...
protected void customize()
{
noOfStars = 3;
timeToFinish = 600;
...
}
//other protected methods which could be overridden be some extending
// class...
}
public class LevelThree extends GameLevel
{
protected int noOfEnemies; //new introduced variable
//other protected variables...
...
protected void customize()
{
noOfEnemies = 4;
//other customizations...
...
}
public void update()
{
super.update();
//new introduced feature related code...
...
}
//other protected methods...
}
public class LevelFour extends LevelThree
{
...
protected void customize()
{
noOfEnemies = 6;
...
}
//Other protected methods...
}
```

Appendix A.4. GameLevel Class of Similar Levels Pattern

```
public class GameLevel extends AbstractUpdateable
{
public static int levelNo;
//Other variables...
public GameLevel(int timeToFinish, int noOfEnemies, int noOfStars, boolean first)
{
this.timeToFinish = timeToFinish;
this.noOfEnemies = noOfEnemies;
this.noOfStars = noOfStars;
...
}
public void update()
{
```

```
if(!paused) {
paint();
if(!levelCompleted && listenKeys) {
listenKeys();
}
}}
private void paint()
{
//platform and other characters drawing...
 . . .
}
public void keyPressed(int keyCode)
{
 . . .
}
//Other private methods...
}
```

Appendix A.5. Sample Code Creating Instances of GameLevel in Similar Levels Pattern

```
Updateable createGameLevel(int timeToFinish, int noOfEnemies, int noOfStars)
{
currentUpdateable = new GameLevel(timeToFinish, noOfEnemies, noOfStars, false);
return currentUpdateable;
}
```

Appendix A.6. Sample Implementation of Different Parts of Different Parts Pattern

```
public class IntroPart extends AbstractUpdateable
{
private int pages = 2;
private int page = 1;
private String story;
//Other private variables . . .
public void init()
{
story = "Bidiro has set on journey. At begining...";
}
public void update()
{
if(page == 1) {
 . . .
}
}
public void keyPressed(int keyCode)
{
}
//Other private methods...
}
public class MainPart extends AbstractUpdateable
{
//private variables specific to this Main Part...
 . . .
public void init()
{
//Initializations...
```

```
}
public void update()
{
//update appropriate values of variables after calculations...
paint();
}
private void paint()
{
//draw graphics
}
public void keyPressed(int keyCode)
{
//Actions against key-events...
}
//Other private methods...
}
```

Appendix A.7. Sample Code Implementing Single Platform Pattern

```
public class GameDesign extends AbstractUpdateable
{
//Game Design related variables...
private GameLogic game;
 ...
public GameDesign(MIDlet midlet)
{
 ...
}
public void update()
{
//update appropriate values of variables related to drawings...
paint();
}
private void paint()
{
//Drawings...
}
public void keyPressed(int keyCode)
{
//Actions against key-events...
}
}
public class GameLogic
{
//Game-logic related variables
 ...
public void updateState(int row, int col)
{
//Update values based on moves and calculations...
}
//Game-logic private methods...
}
```

References

1. Church, D. Formal Abstract Design Tools. Gamasutra Game Developer Magazine. 1991. Available online: http://www.gamasutra.com/view/feature/131764/formal_abstract_design_tools (accessed on 10 December 2013).
2. Costikyan, G. I have No Words & I must Design: Toward a Critical Vocabulary for Games. In Proceedings of the computer games and digital cultures conference, Tampere, Finland, 6–8 June 2002; pp. 9–33.
3. Björk, S.; Lundgren, S.; Holopainen, J. Game Design Patterns. In Proceedings of the Digital Games Research Conference, Utrecht, The Netherlands, 4–6 November 2003; pp. 180–193.
4. Clearwater, D. What defines video game genre? thinking about genre study after the great divide. *J. Can. Game Stud. Assoc.* **2011**, *5*, 29–49.
5. Juul, J. First Use of "Ludology": 1951. The Ludologist Online Magazine. Available online: http://www.jesperjuul.net/ludologist/first-use-of-ludology-1951 (accessed on 3 November 2014).
6. Frasca, G. Ludology Meets Narratology: Similitudes and Differences Between (video) Games and Narrative. Originally published in Finnish as Ludologia Kohtaa Narratologian in Parnasso, 3, 1999. English Version. 1999. Available online: http://www.ludology.org (accessed on 5 December 2022).
7. Fabricatore, C. Gameplay and Game Mechanics Design: A Key to Quality in Videogames. In Proceedings of the OECD-CERI Expert Meeting on Videogames and Education, Santiago, Chile, 17–18 October 2007.
8. Takahashi, D. Funware's Threat to the Traditional Video Game Industry. Venturebeat. 2008. Available online: http://venturebeat.com/2008/05/09/funwares-threat-to-the-traditional-video-game-industry (accessed on 3 November 2014).
9. Ampatzoglou, A.; Frantzeskou, G.; Stamelos, I. A methodology to assess the impact of design patterns on software quality. *Inf. Softw. Technol.* **2012**, *54*, 331–346. [CrossRef]
10. Nuruzzaman, M.; Hussain, A.; Tahir, H.M. Towards Increasing Web Application Development Productivity through Object-Oriented Framework. *Int. J. Future Comput. Commun.* **2013**, *2*, 220. [CrossRef]
11. Alghamdi, F.M.; Qureshi, M.R.J. Impact of Design Patterns on Software Maintainability. *Int. J. Intell. Syst. Appl.* **2014**, *6*, 41. [CrossRef]
12. Gamma, E.; Helm, R.; Johnson, R.; Vlissedes, J. *Design Patterns Elements of Reusable Object-Oriented Software*, 1st ed.; AddisonWesley Professional: Indianapolis, IN, USA, 1994; ISBN 0201633612.
13. Nucleus: Nucleus Research Report: Microsoft Patterns and Practices. August 2009. Available online: http://msdn.microsoft.com/en-us/practices/ee406167.aspx (accessed on 5 December 2022).
14. Doran, J.P.; Casanova, M. *Game Development Patterns and Best Practices*; Packt Publishing: Birmingham, UK, 2017.
15. Mitchell, A.; Savill-Smith, C. The Use of Computer and Video Games for Learning. A Review of the Literature. Available online: http://www.mlearning.org/docs/The%20use%20of%20computer%20and%20video%20games%20for%20learning.pdf (accessed on 5 December 2022).
16. Connolly, T.M.; Boyle, E.A.; MacArthur, E.; Hainey, T.; Boyle, J.M. A systematic literature review of empirical evidence on computer games and serious games. *Comput. Educ.* **2012**, *59*, 661–686. [CrossRef]
17. Davidsson, O.; Peitz, J.; Bjork, S. Game Design Patterns for Mobile Games. *Proj. Rep. Nokia Res. Cent. Finl.* **2006**. Available online: https://www.scribd.com (accessed on 7 January 2023).
18. Ampatzoglou, A.; Chatzigeorgiou, A. Evaluation of object-oriented design patterns in game development. *Inf. Softw. Technol.* **2007**, *49*, 445–454. [CrossRef]
19. Dondlinger, M.J. Educational video game design: A review of the literature. *J. Appl. Educ. Technol.* **2007**, *4*, 21–31.
20. Dickey, M.D. Game design and learning: A conjectural analysis of how massively multiple online role-playing games (MMORPGs) foster intrinsic motivation. *Educ. Technol. Res. Dev.* **2007**, *55*, 253–273. [CrossRef]
21. Kelle, S.; Klemke, R.; Specht, M. Effects of game design patterns on basic life support training content. *J. Educ. Technol. Soc.* **2013**, *16*, 275–285.
22. Gestwicki, P.V. Computer games as motivation for design patterns. *ACM SIGCSE Bull.* **2007**, *39*, 233–237. [CrossRef]
23. Schmitz, B.; Klemke, R.; Specht, M. Mobile gaming patterns and their impact on learning outcomes: A literature review. In Proceedings of the 21st Century Learning for 21st Century Skills, Saarbrücken, Germany, 18–21 September 2012; pp. 419–424.
24. Hahbudin, F.E.; Chua, F.F. Design patterns for developing high efficiency mobile application. *J. Inf. Technol. Softw. Eng.* **2013**, *3*, 1–9.
25. Kelle, S.; Klemke, R.; Specht, M. Design patterns for learning games. *Int. J. Technol. Enhanc. Learn.* **2011**, *3*, 555–569. [CrossRef]
26. Lameras, P.; Arnab, S.; Dunwell, I.; Stewart, C.; Clarke, S.; Petridis, P. Essential features of serious games design in higher education: Linking learning attributes to game mechanics. *Br. J. Educ. Technol.* **2017**, *48*, 972–994. [CrossRef]
27. Ni, Q.; Yu, Y. Research on Educational Mobile Games and the effect it has on the Cognitive Development of Preschool Children. In Proceedings of the Third International Conference on Digital Information, Networking, and Wireless Communications, (DINWC) 2015, Moscow, Russia, 3–5 February 2015; pp. 165–169.
28. Pombo, L.; Marques, M.M.; Carlos, V.; Guerra, C.; Lucas, M.; Loureiro, M.J. Augmented Reality and Mobile Learning in a Smart Urban Park: Pupils' Perceptions of the EduPARK Game. In Proceedings of the International Conference on Smart Learning Ecosystems and Regional Development, Aveiro, Portugal, 22–23 June 2017; pp. 90–100.
29. Laine, T.H. Mobile Educational Augmented Reality Games: A Systematic Literature Review and Two Case Studies. *Computers* **2018**, *7*, 19. [CrossRef]

30. Zsila, Á.; Orosz, G.; Bőthe, B.; Tóth-Király, I.; Király, O.; Griffiths, M.; Demetrovics, Z. An empirical study on the motivations underlying augmented reality games: The case of Pokémon Go during and after Pokémon fever. *Personal. Individ. Differ.* **2018**, *133*, 56–66. [CrossRef]
31. Papadakis, S. The use of computer games in classroom environment. *Int. J. Teach. Case Stud.* **2018**, *9*, 1–25. [CrossRef]
32. Keogh, B.; Richardson, I. Waiting to play: The labour of background games. *Eur. J. Cult. Stud.* **2018**, *21*, 13–25. [CrossRef]
33. Braham, A.; Buendía, F.; Khemaja, M.; Gargouri, F. User interface design patterns and ontology models for adaptive mobile applications. *Pers. Ubiquitous Comput.* **2022**, *26*, 1395–1411. [CrossRef]
34. Takoordyal, K. *Beginning Unity Android Game Development*; Apress: New York, NY, USA, 2020.
35. Khan, M.; Rasool, G. Recovery of Mobile Game Design Patterns. In Proceedings of the 2020 21st International Arab Conference on Information Technology (ACIT), Giza, Egypt, 28–30 November 2020; pp. 1–7.
36. Flores, N.; Paiva, A.C.; Cruz, N. Teaching Software Engineering Topics Through Pedagogical Game Design Patterns: An Empirical Study. *Information* **2020**, *11*, 153. [CrossRef]
37. Ganesh, A.; Ndulue, C.; Orji, R. The design and development of mobile game to promote secure smartphone behaviour. In Proceedings of the CEUR Workshop Proceedings, College Station, TX, USA, 19–20 August 2021; pp. 73–87.
38. Glaser, N.; Schmidt, M. Systematic literature review of virtual reality intervention design patterns for individuals with autism spectrum disorders. *Int. J. Hum.–Comput. Interact.* **2022**, *38*, 753–788. [CrossRef]
39. Hui, B. Big Designs for Small Devices. JavaWorld.com. 2002. Available online: http://www.javaworld.com/javaworld/jw-12-2002/jw-1213-j2medesign.html (accessed on 17 December 2013).
40. Narsoo, J.; Mohamudally, N. Identification of Design Patterns for Mobile Services with J2ME (Santa Rosa, USA). *Issues Inf. Sci. Inf. Technol.* **2008**, *5*, 623–643.
41. Narsoo, J.; Sunhaloo, M.S.; Thomas, R. The Application of Design Patterns to Develop Games for Mobile Devices Using Java 2 Micro Edition (Zurich, Switzerland). *J. Object Technol.* **2009**, *8*, 153–175. [CrossRef]
42. Ilja, A. Use of Design Patterns for Mobile Game Development. Bachelor's Thesis, Department of Computing Science, Umea Universitet, Umea, Sweden, 2012.
43. Nystrom, R. *Game Programming Patterns*, 1st ed.; Genever Benning: Seattle, WA, USA, 2014; ISBN 978-0990582908. Available online: http://gameprogrammingpatterns.com (accessed on 14 November 2014).
44. Hunicke, R.; LeBlanc, M.; Zubek, R. *MDA:* A Formal Approach to Game Design and Game Research. In Proceedings of the Challenges in Games AI Workshop, 19th National Conference of Artificial Intelligence, San Jose, CA, USA, 25–29 July 2004; pp. 1–5.
45. Kreimeier, B. The Case For Game Design Patterns. Gamasutra Game Developer Magazine. 2002. Available online: http://www.gamasutra.com/view/feature/132649/the_case_for_game_design_patterns.php (accessed on 17 November 2014).
46. Björk, S.; Holopainen, J. Describing Games: An Interaction-Centric Structural Framework. In Proceedings of the Level Up-1st International Digital Games Research Conference, Utrecht, The Netherlands, 4–6 November 2003; pp. 4–6.
47. Korhonen, H.; Koivisto, E.M.I. Playability Heuristics for Mobile Games. In Proceedings of the 8th International Conference on Human-Computer Interaction with Mobile Devices and Services, MobileHCI'06, Espoo, Finland, 12–15 September 2006; pp. 9–16.
48. O'Brien, L. Design Patterns 15 Years Later: An Interview with Erich Gamma, Richard Helm, and Ralph Johnson. 2009. Available online: http://www.informit.com/articles/article.aspx?p=1404056 (accessed on 19 November 2014).
49. Lindley, C.A. Game Taxonomies: A High-Level Framework for Game Analysis and Design. *Gamasutra Game Developer Magazine*. October 2003. Available online: http://www.gamasutra.com/view/feature/2796/game_taxonomies_a_high_level_.php (accessed on 10 May 2015).
50. Crawford, C. *The Art of Computer Game Design*; Osborne/McGraw-Hill: Berkeley, CA, USA, 1984.
51. Elverdam, C.; Aarseth, E. Game Classification and Game Design: Construction through Critical Analysis. *Games Cult.* **2007**, *2*, 3–22. [CrossRef]
52. Kickmeier-Rust, M.D. Talking Digital Educational Games. In Proceedings of the 1st Int. Open Workshop on Intelligent Personalization and Adaptation in Digital Educational Games, Graz, Austria, 14 October 2009; pp. 55–66.
53. Dahlskog, S.; Kamstrup, A.; Espen, A. Mapping the game landscape: Locating genres using functional classification. In Proceedings of the 4th Digital Games Research Conference, Graz, Austria, 2–4 December 2009.
54. Klabbers, J.H.G. The Gaming Landscape: A Taxonomy for Classifying Games and Simulations. In Proceedings of the 1st Digital Games Research Conference, Utrecht, The Netherlands, 4–6 November 2003; pp. 54–68.
55. Freeman, E.; Robson, E.; Bates, B.; Sierra, K. *Head First Design Patterns*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2004.