OULUN AMMATTIKORKEAKOULU

Eleonoora Kalliokoski

DESIGN PATTERNS IN GAMES

Use of Software Design Patterns in a Game Development Project

DESIGN PATTERNS IN GAMES

Use of Software Design Patterns in a Game Development Project

Eleonoora Kalliokoski
Bachelor's Thesis
Autumn 2023
Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Information Technology

Author(s): Eleonoora Kalliokoski
Title of the thesis: Design Patterns in Games
Thesis examiner(s): Janne Kumpuoja
Term and year of thesis completion: Autumn 2023 Pages: 27 + 1 appendix

This thesis studies the use of software design patterns in game development through both a practical and theoretical lens. The main goal was to provide examples of design patterns as used in game development as well as analyse some factors that may affect their suitability to a given project.

The theoretical background of the thesis focuses on different sources on software design patterns and their practical implementations, with some sources concerning game development in particular. Rather than go in depth with all established design patterns, the thesis briefly summarizes most basic patterns and provides more thorough descriptions and examples of a smaller number of patterns that are especially relevant to the discussion.

To provide some practical examples, the thesis work included the development of a small game prototype. This prototype was developed on the Unity game engine using C#. The general structure of the prototype is described as part of the thesis. The scope was kept deliberately small, with the goal of producing a short but playable game experience with the main focus of the thesis being on the theoretical background and discussion.

With the exception of audio elements, which were acquired as pre-made assets, the game project was developed entirely within the context of this thesis. This included the basic concept and ideation, level and puzzle design, programming, and basic graphics. The development of the prototype was done concurrently with the background research.

The finished prototype game is in the “escape room” genre, where the player must solve puzzles and use items to open a final exit and win the game. As a prototype, the game is short, consisting of only two item puzzles and two lock puzzles. Player interactions are done with a mouse, and navigation from one screen to the next happens via on-screen buttons. The player is also able to pick up items within the game space and use them on game objects.

The prototype is then analysed on a structural and code level, identifying design patterns where they appear as well as patterns that could have been included without changing the premise or scope of the game. Some consideration was also given to the game engine and the ways it facilitates the inclusion of design patterns and the use of object oriented programming in general.

In the discussion section more patterns are brought into consideration, particularly in the context of game scope and genre and how these may affect the choice and usefulness of design patterns.

The final conclusion is that while design patterns can be useful in the development of games, there are several factors that affect the choice and usage. It was also concluded that the small scope of the prototype limits the extent to which it can include different patterns, and a larger project could showcase more patterns within the context of a single game.

Keywords: design patterns, software development, game development, code structure

CONTENTS

1 INTRODUCTION

The purpose of this project is to study software design patterns and how they can be applied in game development in the context of a small scale project. The thesis consists of four main elements: theoretical background, practical development, analysis, and further discussion.

The theoretical background is gathered from various works describing and discussing software design patterns and their usage. The purpose of this section is to establish the general idea of design patterns as they are used in software development as well as provide descriptions and hypothetical examples of a few patterns in particular for the purposes of analysis and discussion.

The practical development section consists of the development of a small videogame project. The project starts from a basic concept developed to the point of a playable prototype. The scale of the prototype is kept deliberately small, as the main focus is on the design patterns that can be demonstrated through it. The gameplay and general elements of the finished prototype are described as part of the thesis.

The analysis section of the thesis goes into further detail on the prototype and its code structure. Besides identifying design patterns that were used in the program, the section also suggests some alternative solutions that could have been used to accomplish the same goals.

Finally, the discussion section further describes the ways design patterns could be used in game development and how different types of projects may benefit from different patterns. The main focus is on matters of project size and game genre and how they may affect what design patterns are reasonable to include in development.

2 THEORETICAL BACKGROUND

Software design patterns were first proposed as a concept by the so-called “Gang of Four”, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides in their 1996 book “Design Patterns: Elements of Reusable Object-Oriented Software”. They cited as an inspiration the work of Cristopher Alexander, who described patterns used in architectural design. They presented design patterns as solutions to common problems that could be utilized regardless of language, application domain, or other factors. (Gamma et al. 1996, chapter 1.)

In their original book the Gang of Four created a catalog of 23 basic patterns divided into creational, structural, and behavioral patterns (Gamma et al. 1996, chapter 1.7). The names and definitions of these patterns are still in use, although the number of patterns described by various authors has since expanded. Currently the website Software-Pattern.org lists 85 different software design and architectural patterns in 13 different categories, including the 23 original ones (Software-Pattern.org).

At its most basic, a design pattern consists of a problem and a potential solution; the details of the implementation are outside the scope of design patterns. This allows them to be flexible and reusable, providing developers with tools for commonly encountered situations.

While the reusability of design patterns is one of their main advantages, there are other reasons why knowledge of established patterns can be beneficial to a developer. First, it saves time as the programmer does not have to solve a problem from first principles, but can focus on finding an appropriate implementation of an existing solution. They improve the readability of code and documentation by creating clear sections and identifying structures that may be familiar to other developers. Direct communication between developers is also improved if all parties are using the same terminology, allowing them to identify and describe potential solutions without having to explain the planned code in excessive detail. As such, design patterns are not only a tool, but also a way of facilitating communication. (Doran & Casanova 2017, What are design patterns.)

The original main categories were creational, structural, and behavioral patterns. Subsequently the concept of concurrency patterns has been introduced to describe patterns used in multi-threaded programming, along with other smaller categories (Software-Pattern.org). For the purposes of this

thesis only some of these patterns will be described in detail as they will be relevant to the discussion. Other patterns included in the original catalog are briefly summarized under their relevant categories.

2.1 Creational patterns

Creational patterns are related to different ways of creating objects. It should be noted that the decision not to create an object and use an existing instance instead can also be part of a pattern, as in the case of the Singleton pattern. The Singleton and Object Pool patterns will be discussed in more detail with examples.

2.1.1 Singleton

The most basic design pattern, the Singleton ensures there is only one instance of a class, with a global point of access to allow it to be used from anywhere in the software (Project Management Institute 2020, The Singleton). This means that every piece of code that references the class in question is accessing the same instance of it, regardless of where or when in the program the reference occurs.

While this can be the result of programmer choice, where the developer simply takes care not to create an instance of a class in multiple parts of the code, the pattern can also be reinforced programmatically. For example, a call to the class in question may include a check to see if an instance already exists. If this is the case a new instance is not created, and the reference is directed to the existing class instance.

As the Singleton pattern means a class gets only created once and can be accessed from anywhere in the project, it is a good choice for classes where global, repeated access is required but multiple copies may result in errors, redundancy, or performance issues. A potential example of the pattern in game development could be a player controller. In this case, access to the controller is required in all playable sections of the game, but multiple instances may result in problems such as duplicated or imprecise inputs. This makes a singleton a good choice.

2.1.2 Object pool

A more recent addition to the design pattern vocabulary, the Object Pool pattern aims to optimize memory usage by recycling instances of frequently needed classes rather than constantly destroying and recreating new instances. This minimizes the performance load of initializing new instances and makes memory usage more predictable, which can help with optimization. The Object Pool also manages the total number of instances by dynamically creating new instances as needed while destroying any overflow. (Baron 2021, Optimizing With The Object Pool Pattern.)

Since it mainly concerns the reuse of multiple instances of the same class, the Object Pool is most useful when dealing with object classes where instances are requested and then discarded repeatedly. If objects are never reused after being discarded, storing them in memory is an unnecessary drain on resources. Similarly, if only a small number of instances are being reused infrequently, the impact on performance from maintaining the Object Pool can outweigh the benefits.

As videogames by their nature are dynamic pieces of software, they can often benefit from using the Object Pool pattern. An example of this would be a game where the player tries to shoot at continuously spawning enemies while trying to avoid their attacks. Such a game could use object pools to manage both the enemy instances and the projectiles used by the player and the enemies. These types of objects tend to be identical instances created in large numbers and with short lifespans, which makes them an excellent use case for the Object Pool pattern. Furthermore, because they are such a large part of the game content, optimizing memory usage in these cases can have significant effects on the general performance.

2.1.3 Other creational patterns

Other creational patterns introduced in the original GoF book include the Abstract Factory, Builder, Factory Method, and Prototype. The Abstract Factory, Builder, and Factory Method patterns all provide interfaces for creating objects with different approaches to how strictly the construction process is limited to particular classes or implementations. The Prototype pattern provides a defined template for the creation of new objects. (Gamma et al. 1996, chapter 3.)

2.2 Structural Patterns

Structural patterns are related to larger structures formed by classes and objects and how they relate to each other. For this paper, the patterns examined in more detail are the Decorator and the Composite.

2.2.1 Decorator

The Decorator pattern can be used as an alternative to subclasses by attaching more components to an object in a dynamic manner (Project Management Institute 2020, The Decorator). As the Decorators can be used at runtime and combined to layer their effects, they are more flexible than pre-defined subclasses and can be more responsive to the needs of the software.

The downside of this flexibility is that using multiple Decorators on the same object may lead to conflicts as their effects interact in unpredictable or unwanted ways. This can be mitigated by introducing rules or restrictions on the kind of Decorators that can be used simultaneously or in which order they are applied on the object, at the cost of some of the flexibility.

One example where the Decorator pattern may be used in game development is the idea of status effects. Status effects are usually temporary effects that can be applied to characters during gameplay. These can change the parameters of the affected characters such as health or applied damage, add or remove usable skills, or disable the characters entirely until removed. As these effects can typically be applied to many different characters but affect all of them in similar ways, using Decorator patterns to store the status effects rather than coding them into each character class separately not only saves time and effort, but minimizes the chance of errors or inconsistencies.

2.2.2 Composite

The Composite pattern allows multiple related elements, or leaf objects, to be treated as parts of a hierarchical system. This system can be treated as a single unit by accessing the highest element, also called a node. The elements are arranged in a tree structure that may have multiple layers.

The main purposes of using the Composite pattern are organization, navigation, and minimizing conflicts between objects. (Project Management Institute 2020, The Composite.)

In game development, the Composite pattern can be used to handle commanding units in a war strategy game. In some games of the genre, the player may choose to give tasks to individual fighters or select several such units to form larger groups. These groups may consist of several different types of objects and be different sizes, and the groups can be assigned the same tasks as individual characters. The Composite pattern, which allows the program to treat leaf and node members in identical ways, works well for this type of application.

2.2.3 Other structural patterns

Other structural patterns included in Design Patterns include the Adapter, Bridge, Façade, Flyweight, and Proxy. The Façade is conceptually similar to the Composite in that it provides a higher-level interface to use several subsystem interfaces in a unified manner. These interfaces may not be incompatible; the purpose of the Façade is to simplify the use of the subsystems. The Adapter pattern, on the other hand, changes the interface of a singular class to allow it to interact with another class with an incompatible interface. The Flyweight pattern supports handling large numbers of similar objects. The Bridge is used to decouple an abstraction from one or multiple implementations, which allows the two to vary without affecting each other. Finally, the Proxy is used to allow controlled access to an object where it would be difficult or otherwise unwanted to let other objects interface directly. (Gamma et al. 1996, chapter 4.)

2.3 Behavioral patterns

Behavioral patterns describe ways in which objects communicate and behave in relation to each other. While structural patterns define the way objects connect to each other in a hierarchy, behavioral patterns are concerned with information and requests and how they are directed, relayed, and responded to. The patterns described in more detail here include Command, Memento, Observer, and State.

2.3.1 Command

The Command pattern encapsulates a request as an object that can then be passed between objects. This allows the requests to be recorded for later access. A record of requests not only means they can be delayed and fulfilled at a later point, but they can be referred to after execution. This means the actions can be undone or executed multiple times, which would not be possible without storing information about the requests. (Baron 2021, Implement a Replay System with the Command Pattern.)

As the Command pattern allows requests to be stored and accessed at a later date, it has several possible applications in game development. For example, some puzzle games allow the player to go back a set number of moves. Strategy games where multiple players decide their actions at the same time, which are then executed after all actions have been chosen, also require a record of the different requests.

2.3.2 Memento

Where the Command pattern saves a request, the Memento pattern is used to save the internal state of an object into an external object to allow the internal state to be restored later. The Memento only stores the state with no additional information, and it is only accessed by the object it originates from. This allows the recording and restoration of the state without violating the encapsulation of the originator. (Project Management Institute 2020, The Memento.)

As a similar pattern to the Command, the Memento pattern can also be used to implement undo functionality. This is especially useful in cases where simply reversing the used commands is not enough to return to an earlier state. For example, any actions with an element of randomness cannot be accurately reversed without knowledge of the previous state.

2.3.3 Observer

The Observer pattern links objects together in a one-to-many relationship so that when the state of one object changes, the change is relayed to all its dependents and they are updated accordingly. The observer itself does not implement any functionality; its role is purely relaying information. Both

passing the data to the dependents and them updating in response to it happens automatically when the state of the original object changes. (Doran & Casanova 2017, The Observer pattern explained.)

As the Observer pattern passes information in a one-to-many arrangement, it can be used for different types of event managers in game development. For example, a player character may have an internal variable tracking its current health. This is displayed in a visual tracker in the UI, as well as changing the character's appearance once it falls below a certain level. Rather than have the player's internal health function track these changes and request updates, an Observer pattern can be employed. In this case the Observer would monitor the player character's health and pass any updates to both the tracker object and the render system of the character. These objects then implement the necessary changes as part of their regular update cycle, without the need for a request from the originating script.

2.3.4 State

The State pattern allows an object to have several different behavioral models that change based on its internal state (Baron 2021, An overview of the State pattern). The behavior patterns are all part of the object's functionality, but different behaviors are activated or enabled depending on the active state. This removes the need to modify the object at runtime, as all required functionality is encapsulated within it.

An example of the use of the State pattern in game development is the use of states to control the tool used in a farming game. In many games in the farm simulator genre there is only a limited number of tools, each of which has its own functionality. This can be implemented through the State pattern by using an internal state to record which tool, if any, the player has equipped. This state is then used to determine which action is taken when the player gives a tool input and can also be used to choose the appropriate animation.

2.3.5 Other behavioral patterns

The behavioral pattern category in Design Patterns also includes the Chain of Responsibility, Interpreter, Iterator, Mediator, Strategy, Template Method, and Visitor patterns. The Chain of

Responsibility pattern allows requests to be passed along to several recipients, any of which can potentially handle the request. The Interpreter pattern is used to define the grammar of simple languages and use the resulting rules to interpret sentences in this language. The Iterator pattern can access the individual elements of an aggregate object one by one without the need to access the internal structure. The Mediator can be seen as a many-to-many version of the Observer pattern, as it encapsulates the interactions of a set of objects, directing requests and minimizing direct connections between classes. The Visitor pattern is used to define and perform operations on a set of classes without the need to modify the target classes.

Finally, the Strategy and Template Method patterns concern algorithms. The Strategy pattern allows a set of related algorithms to be used interchangeably, while the Template Method pattern defines the basic skeleton of an algorithm and then allows subclasses to modify individual steps of the algorithm without affecting its actual structure. (Gamma et al. 1996, chapter 5.)

3 GAME PROTOTYPE ANALYSIS

In this section, the game project is broken down on both the structure and code level. The purpose of this is to identify some of the design patterns within the code as well as evaluate the importance of these patterns and consider potential alternatives.

3.1 Premise and technical details

A short videogame prototype was developed as a practical development exercise of this thesis. As the main purpose of this prototype is to provide practical examples of some of the design patterns, the scope of the project was kept deliberately small to allow more focus for the theory and its applications in the thesis.

The game is a short single-player experience in the “escape room” genre of games, where the player is expected to use clues and items in the game environment to solve puzzles and open a final exit. All interaction with the game happens with mouse movements.

The game was made using the Unity engine, with code written in C#. Git was used for file and version management.

While the game takes place in a 3D environment within the game engine, all objects and other game elements are graphically 2D. This choice was made to limit the scope of the project, as 3D assets are more time-consuming to create and manage. These 2D graphics were created specifically for this project and are limited to the game elements that require interaction.

All audio used in the project consists of pre-existing assets with applicable licenses for the project.

3.2 Visual structure and gameplay

The main appearance of the game consists of three elements: a view of the game space, the navigation arrows, and the inventory tab. Due to the small scope of the project, the in-game space consists of one room with a limited number of objects and one door.

The inventory tab is located at the bottom of the screen and displays any items that the player has collected. The items can be rearranged within the inventory or dragged out of it to be used on objects in the game space. This tab can also be hidden to allow a full view of the game space (figure 1).

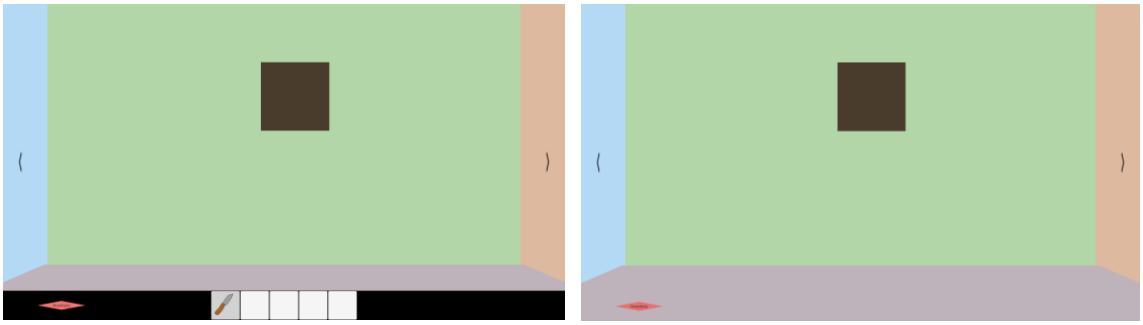

FIGURE 1: Inventory tab visible vs hidden.

The navigation arrows are used to navigate inside the game space. The player cannot move; parts of the game space are accessed by switching between cameras showing different areas. In default view, the side arrows are used to switch the view to each of the four walls of the room. There are three hotspots where the player can move into a close view, in which case the side arrows are hidden and a new arrow is added to allow the player to move back to the general view (figure 2).

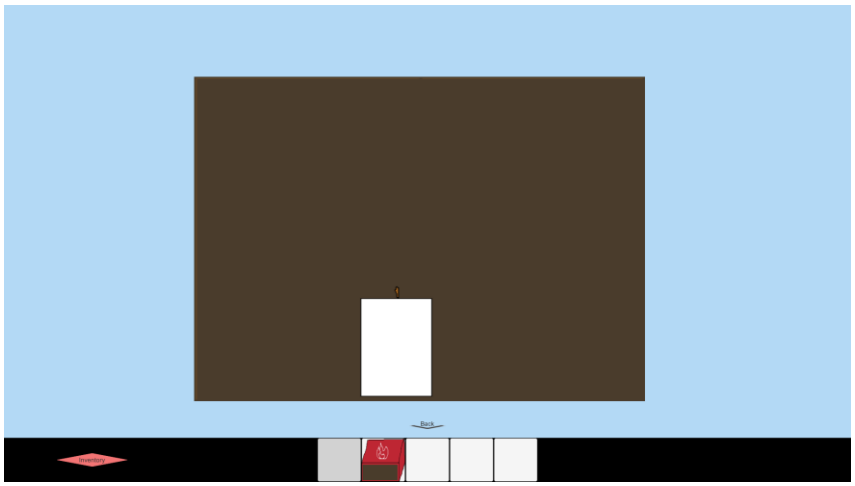

FIGURE 2: A close view with new navigation arrow visible.

The main part of the view is the game space, which includes the objects the player can interact with as well as visual clues. The interactable objects fall into three categories: combination locks the player manipulates directly, items that can be picked up and added to the player's inventory, and item puzzles where the player can use their collected items to cause a change within the game space (figure 3).

The player's goal is to solve the puzzles, progress through the game, and find a key that can be used to open the door. When this is done, the player is brought to the victory screen, which marks the end of the game.

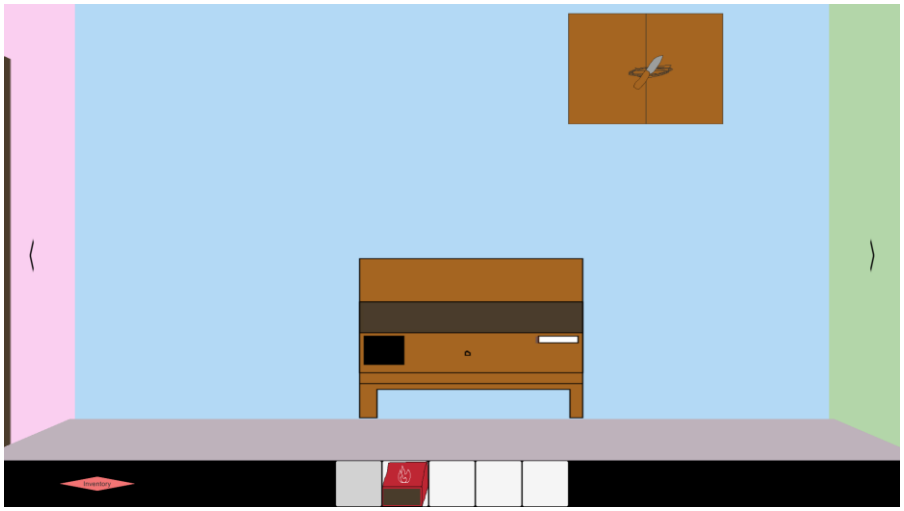

FIGURE 3: Using an item on an object.

3.3 Effects of using Unity

As with all game development projects, the choice of game engine and editor has an effect on the structure of the game and various choices made during the development process. Most importantly, the way Unity structures projects around individual game objects and the treatment of those objects as collections of components makes it easy to take an object-oriented approach on projects.

The editor also makes it relatively easy to link objects and scripts to each other directly in the editor view without the need to either hard-code these connections into the scripts or use functions to find the necessary components (figure 4). This facilitates the division of code into separate scripts and classes. It also makes the objects more reusable, as they are not tied to a specific set of other objects. These aspects make it easier to make the projects modular.

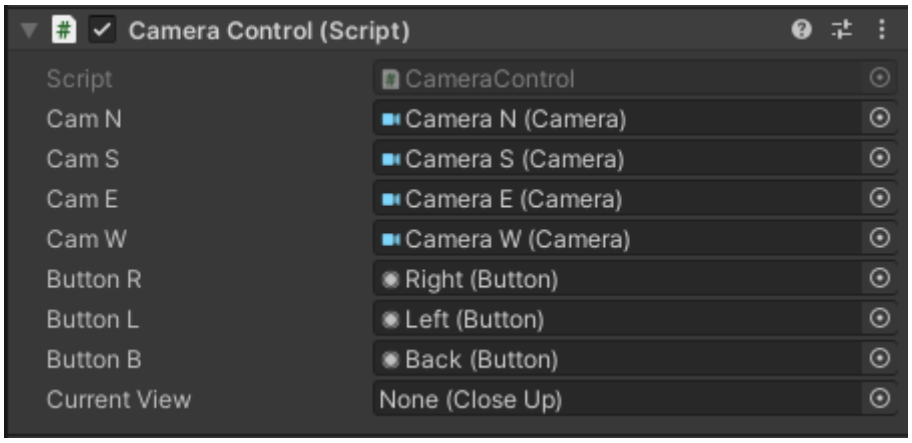

FIGURE 4: A screen capture of a component inside the Unity editor.

3.4 The inventory manager and the Singleton

The largest single script in the project is the inventory management system. This script contains the functions keeping track of the items the player has acquired, adding and removing them as appropriate, as well as displaying them in the inventory tab. It also handles moving the items within the inventory and marking items as active to make it possible for objects to check if the correct item is being used on them. While item objects in the game space handle being added to the inventory from their end, this is done by passing the item data into the addition function within the manager, rather than directly interfacing with the inventory array.

This is an example of the Singleton pattern, as there is only one such manager in the entire game. While this is in part due to the small scope of the project, it would likely remain the same even in a bigger project. In a game where object interactions are a big part of the gameplay, an inventory system is required through all parts of the game, so a continuous, singular item list is necessary. Having multiple scripts adding or subtracting items directly or having several sources tracking the currently active item runs the risk of duplicating items, incorrect interactions, and other potential errors.

Each item is established as its own item class, which is used to handle the interactions. If the project was extended, more items could be added simply by creating more classes for each. As long as the item classes are included, both the inventory system and the interaction script can handle them. Due to the small scope, the inventory system could also be done as a pre-determined list of items, where whether the player has acquired an item or not is recorded as a Boolean value. With this approach, the inventory could be set up with a specific location for each item, with spaces left empty

or filled as appropriate. An example of this system can be seen in the Super Nintendo Entertainment System game The Legend of Zelda: A Link to the Past, where the in-game inventory leaves missing items as empty spots while the file choice screen shows silhouettes of items that have not been acquired (figure 5).

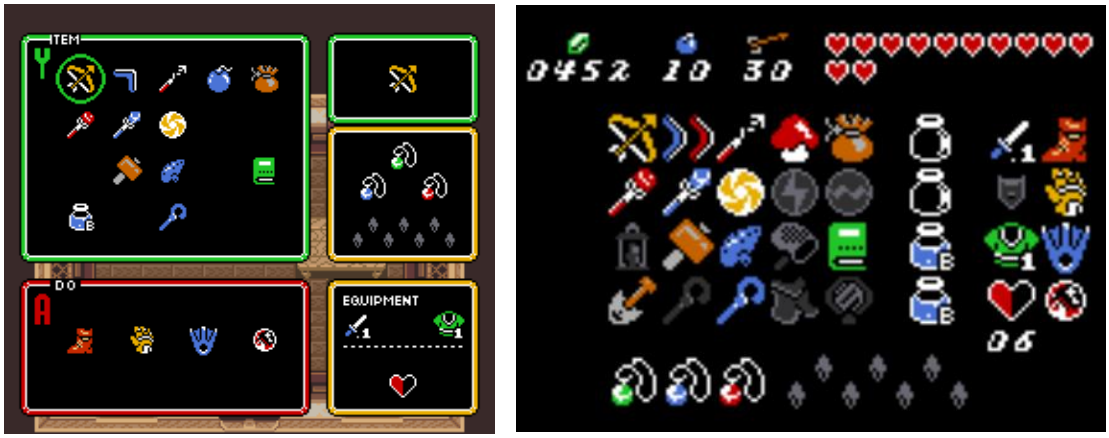

FIGURE 5: In-game inventory and loading screen inventory of the same save file.

Despite the small number of items the decision was made to create the extendable inventory array. This was done to follow the principles of reusable code mentioned in Design Patterns, as part of the purpose of using design patterns is to allow code to be expanded and recycled as necessary (Gamma et al. 1996, chapter 1).

The camera controller is another singular class, changing the active camera whenever the navigation arrows are used. There is another script to move to the close-up views where appropriate; however, it only changes the active UI elements and then passes the appropriate reference to the camera control script, rather than switching cameras directly:

```
public void closeUp()
{
    prevCamera.enabled = false;
    newCamera.enabled = true;

    if (layers == 0)
    {
    rightButton.gameObject.SetActive(false);
    leftButton.gameObject.SetActive(false);
    backButton.gameObject.SetActive(true);
    }
```

```
        controller.setCurrent(this);
    }
```

However, the camera controller is not a good example of the Singleton pattern. In a larger project there would likely be several camera controller classes, at least one for each level of the game, as there would be no point in carrying information about cameras in other scenes throughout the game. In a similar way many other classes in the project technically fall into the Singleton pattern only due to the small scope of the project.

3.5 Combination locks and the Composite pattern

There are two combination locks in the game. One requires the player to input a four-digit numerical code, while the other expects a visual combination of bars of varying heights. Within the code this combination is also abstracted as numbers, although the player does not see these numbers.

One method of implementing these lock objects would be to treat each lock as a singular unit within the code, with the same script handling input of each digit and the verification process. Instead, they are structured as a variation of the Composite pattern.

Both locks have the same basic code structure, consisting of a parent object that holds the other elements, four objects corresponding to each digit of the code and one handling the verification process. The digit objects respond to the player's input, cycling through the potential values with each click and updating the visuals to show the change to the player. The verification object runs a function when interacted with, requesting the value of each digit, combining them into an array, and passing this on to the check script.

The check script is the same for both lock objects and is located in the main event manager. The script accepts an attempted combination and the correct solution as parameters, turning the arrays into strings and comparing the results. The process iterates over the arrays automatically and could be used to check combinations of any length. The script then passes a result of true or false back to the verification object, which can then react accordingly if the combination is correct. The check script does not at any point interact directly with the digit objects, treating the verification object as a representation of the entire combination.

FIGURE 6: An example of the object hierarchy.

Another minor example of the Composite pattern in the game is the display of the inventory tab. The item slots and item sprites are considered parts of the inventory tab element within the hierarchy of the UI elements. When the inventory button is used to either hide or display the inventory tab, the entire structure is moved up and down within the screen space. Rather than use commands to move each element individually, though, the code only affects the main inventory tab. The other elements move along with the main structure.

This is another example where the functionality of Unity as an editor and engine facilitates the use of design patterns. The editor makes it relatively simple to arrange elements to form these hierarchies. In this case, the inventory slots are assigned as children of the main inventory space within the editor, without the need to establish this structure through code (figure 6).

3.6 Potential patterns

Due to the small scope of the project, there is only a limited number of patterns that could be used in the code. However, even so there are a couple of other design patterns that could have been used in the development of the project.

The camera control system could have used an observer object to pass information about the currently active camera from the navigation system to the camera controller. The Observer pattern could also be used to trigger the appropriate sound effects when items are acquired or used correctly. In the current game, the objects send a signal to the appropriate audio source when appropriate.

Hiding and displaying the inventory tab could have been done through use of the State pattern. In the current implementation the inventory tab is moved outside the screen space and back to its original position when the appropriate button is pressed. This accomplishes the two goals of hiding

it from view and making it impossible to interact with, as the player cannot reach objects outside the screen space.

```
public void OnClickInv()
    {
        if (expanded == true)
        {
            rectTransform.anchoredPosition = new Vector2(xAxis, yAxis);
            expanded = false;
        }
        else
        {
            expanded = true;
            rectTransform.anchoredPosition = new Vector2(xAxis, yAxis2);
        }
    }
```

An alternative solution would be to give the inventory tab two states, active and inactive. This way the inventory tab would be disabled whenever the state is changed to inactive and made accessible again when it is changed back to active. The current code does not meet the definition of the State pattern as the functionality and behavior of the inventory does not actually change when it is hidden, it is simply moved out of the player's reach when needed.

4 DISCUSSION

While the previous chapter analysed the prototype and the design patterns that were or could have been included in it, the purpose of this chapter is to expand the discussion beyond the individual project. The main questions considered are how the scale and genre of the project would affect the patterns that are beneficial to the development.

4.1 Design patterns and scale

It should be noted that this project, while including many aspects of a traditional game development process, is not an accurate representation of the typical game project even at a small scope. The premise of the project as a prototype for showcasing and discussing design patterns not only means the playability of the end result is a secondary concern, but also that many of the typical stages of a game development process were simplified or entirely omitted. For example, *Practical Game Design* by Adam Kramarzewski and Ennio De Nucci (2018) identifies 13 distinct phases in a game development process, most of which can be further broken down, along with numerous roles and aspects that were not relevant to this project despite being central to most real world game projects, such as monetization and promotion. Any changes from the original design were made without need for discussion or negotiation and the final prototype is only slightly more advanced than a minimum viable product (MVP).

The minimal need for communication and discussion in particular contrasts with the idea that design patterns provide a vocabulary and framework for discussion between developers (Doran & Casanova 2017, What are design patterns). While knowledge of patterns was useful in conceptualizing parts of the code, there was no concern for potential miscommunication or clarification.

Furthermore, while an effort was made to design the code to be reusable and modular where possible, this is mostly theoretical as the scope of the game did not allow the repetition of elements within the project. As such, the aim of reusability could be seen as an unnecessary constraint rather than a time investment that will have benefits later in the development process.

With some of the less complicated patterns, it is possible to include variations of patterns without consciously planning to do so. On the other hand, there is also the potential of identifying similarities to patterns after the fact, which can then guide further iteration. In these situations it should be considered whether adapting code to incorporate pre-established design patterns will give any benefits or become pointless busywork. In general, design patterns will likely be more useful when they are intentionally made part of the design from the beginning.

Where the small scale of the project might decrease the usefulness of patterns, this also suggests that a larger project would benefit from design patterns more. Aside from facilitating communication within a larger team as well as reusability of code, there are clear situations where particular design patterns would be useful in a larger or more complex project. For example, patterns that concern the organization and interaction between classes will become more beneficial as the number of elements involved increases. Communication between two objects is naturally simpler to organize than the connections between twenty, making patterns such as Mediator and Adapter more useful in larger scale projects.

4.2 Design patterns and genre

Another factor that might affect which design patterns are useful would be the genre of the game. In the case of the prototype, there were very few active objects at any time, with little need for creating or destroying new instances at runtime. While such elements could be included as a minigame or other deviations from genre conventions, there are other types of games where these behaviors would be practically required.

As mentioned during the theory section of the Object Pool pattern, games that feature shooting, particularly in the “bullet hell” genre, typically feature large numbers of dynamically created and destroyed object instances in the form of projectiles and enemies. This would make patterns that deal with organizing large groups of similar or even identical objects, such as Object Pool and Flyweight, more useful even in a small scale project. Other useful patterns such as Factory and Prototype focus on creating new instances.

The usefulness of patterns such as Command and Memento for programs that require an undo functionality was already discussed in a previous chapter. This makes them well suited for genres

such as puzzle and strategy. Any randomized or procedurally generated elements would also necessitate ways of preserving information for future use. For example, a game with randomized levels that also has a level of persistent progression could use a Memento pattern to save the general progression while allowing levels to be regenerated as needed.

Games with multiplayer aspects can also make use of these information patterns. Saving states and commands is especially important in turn-based games, where actions will be necessarily delayed. In addition, games with online connections will have a certain amount of delay simply due to the time it takes for information to be passed along.

Genres such as RPG can benefit from the Decorator pattern. These games often allow players to change their character classes, which can be handled by using a Decorator to add new functionality to the character. It could also be used for equipment that grants temporary changes to player values. Simply preserving the original values in a Memento pattern or another structure is not sufficient as the base value may change during the time the equipment is in use.

Aside from the already mentioned use case of changing active tools, the State pattern is useful for most genres that feature dynamically changing animations. For example, a platformer will often have separate animation cycles for character actions such as walking, running, jumping, and standing still. Simply triggering a new animation when an action is taken would lead to errors when an action takes longer than the designated animation or when two animations overlap. If the chosen animation is instead based on the player's current state, it will change as appropriate.

It can be concluded that just like the scale of the project, the genre and other content of a game project will affect the type of design patterns that are useful. Meanwhile other patterns may not be helpful or even complicate the design needlessly with no true benefits. Therefore, there is no particular set of design patterns that would be universally useful in game development, and their use should be considered on a case by case basis.

5 CONCLUSION

The original aim of this thesis was to analyse the usage and potential of software design patterns in a small scale game development project as well as consider how factors such as project scope and game genre may affect the choice of design patterns. The conclusion is that while design patterns can be useful in game development and provide solutions to common problems, not all design patterns are useful in every project, and may even cause unnecessary complications if their use is not considered before implementation.

As a learning experience the thesis was successful. Besides the practical experience gained from completing the prototype, the background research provided a solid understanding of design patterns and their use in software development. Analysing the prototype and its code structure was a very interesting exercise, particularly when it came to considering alternative solutions. This gave a new perspective on the structure of the program and the particular choices involved in developing the prototype and its functionality.

The small scale of the development project, while appropriate for the scope of the thesis, did limit the factors that could be analysed. In particular, the idea in several sources that design patterns provide a type of vocabulary to facilitate documentation and communication between developers as well as clarify code structures was left as a hypothetical exercise as the team only consisted of one developer. Likewise, the limited size of the prototype also means that the number of patterns that could be incorporated into the project was very limited.

The development of the game prototype should also not be considered as a typical example of a full game development project. A full commercial game product is often developed by a team of people, with the largest game studios employing hundreds of people working on different aspects of the game. This prototype, while playable, was the work of a singular developer with a heavy focus on the programming aspect. However, since the thesis was likewise focused on code and design patterns this should not impact the quality of the work.

With more time and resources, it could be interesting to design and develop a game that showcases most, if not all the original 23 patterns in its code. Not only would this provide practical examples in

the specific context of videogames, but it would also allow further discussion on the kind of factors that make specific patterns useful or even detrimental in the context of a given project.

Another potential avenue of study would be the use of design patterns as a means of communication between developers. As the concept of design patterns has been established for decades, research could be done into whether developers currently use the vocabulary provided by these patterns and their definitions when communicating with other developers. Conversely, studies could be conducted on the question of whether introducing the language of design patterns into a pre-established development team can produce clearer or more effective communication.

REFERENCES

Doran, John & Casanova, Matt 2017. Game Development Patterns and Best Practices. Birmingham: Packt Publishing. Search date 17.11.2023. https://learning.oreilly.com/library/view/game-development-patterns/9781787127838/

Gamma, Erich, Helm, Richard, Johnson, Ralph & Vlissides, John 1996. Design patterns: Elements of reusable object-oriented software. Boston: Addison-Wesley. Search date 17.11.2023. https://learning.oreilly.com/library/view/design-patterns-elements/0201633612/

The Legend of Zelda: A Link to the Past 1991. Videogame. Director Tezuka, Takashi. Screen capture. Nintendo.

Baron, David 2021. Game Development Patterns with Unity 2021. Second edition. Birmingham: Packt Publishing. Search date 17.11.2023. https://learning.oreilly.com/library/view/game-development-patterns/9781800200814/

Project Management Institute 2020. The Design Patterns Companion. Newtown Square: Project Management Institute, Inc. Search date 17.11.2023. https://learning.oreilly.com/library/view/the-design-patterns/9781628256581/

Software-Pattern.org. Software Design and Architectural Patterns. Search date 17.11.2023. http://software-pattern.org

GAME DIAGRAM APPENDIX 1