A JavaFX app utilising Design Patterns Assignment

@00470183

Christopher Kearsley-Brown

Table of Contents

[Introduction 1](#_Toc102073968)

[Explanation of Software Design 1](#_Toc102073969)

[Application of Design Patterns 2](#_Toc102073970)

[Conclusion 3](#_Toc102073971)

[References 3](#_Toc102073972)

# Introduction

This is a report that will detail the design process and the code of a interactive game. The game is designed to teach an aspect of science to primary school teacher in a fun manner. The software must contain multiple design patterns in the functionality of the game.

# Explanation of Software Design

In the design phase, the initial objective was to have an idea of a game. The idea that was created was that the player is incontrol of a character who enters a labyrinth to retrieve an item and escape. Once they have retrieved this item, they would have to escape. Though this would not be an easy task as the labyrinth would contain traps and patroling enemies. If the player hits a trap, a question will appear where they have a time limit to answer a question correctly. If they answer correctly, they can continue. However, if they answer incorrectly or if the timer runs out, a game over will display. Also, if they come into contact with a patroling enemy, the game will also be over. When this happens, the players can restart the game. If the player beats the game, they will be congrated and have the option to start again.

Next was to relate this concept to science. The immune system became the perfect for this. The game’s synopsis became a virus has invaded the body has to infect the body’s cells. Once all the cells has been infected, the virus must escape to infect cells from other area of the body. White blood cells will be used as the patrolling enemies and hidden mucous membrane will be the traps. Once, a membrane has been walked through, it will reveal itself, however if it is entered again, the player will be attacked again.

These description of the game where then used to create user stories. These user stories are as follows:

* As a player, I want to be introduced and start the game so I know what is happening and I can start playing the game.
* As a player, I want to move, so I can traverse the maze.
* As a player, I want to collect/infect the cells, so I can complete the game objectives.
* As a player, I want to escape the labyrinth when I’ve completed all objectives, to complete the game.
* As a player I want to be able to restart the game so I can try again.
* As a player, I want to be able to answer a question to escape the traps.
* As a player, I want to be able to evade the enemies of the game so I am not defeated.
* As a player, I want to be congratulated so that I know I have beaten the game.

These were used to set objectives when creating the software so that the user stories can be met. The user stories would serve as the product backlog that was used to complete the software.

It is time to design the software. The first UML diagram that I created was a class diagram of the software which can be seen below.

Class Diagram Insert

The class diagram was constructed using the user stories and the description of the software. The description gave information about the objects to create and what their methods would be. Therefore classes such as Player, Item, Enemy and Trap was created and contained methods and attributes that related to them. The class diagram was than extended to show the design patterns working within the software.

Through looking at the existing classes within the class diagram and how they were going to function, it decided which design patterns should be implemented within the software. Therefore, the first design pattern to be implemented is the Singleton design pattern. The Singleton pattern only allows the creation of one instance of class while providing a global access point to the instance. As well of ensuring only one object of a class exists, it provides security to it’s variables (). The player class is perfect for the use of this design as there will only be on instance of the player. The this class also plays a pitivatal role within the software and must be easily accessible. Furthermore, the game includes restarting the game to the beginning. This will therefore usually means the recreation of the player which could lead to multiple players being initialised. The design pattern is a solution to this, ensuring that only one is initialised.

Secondly, the factory design pattern will also be implemented into the software. This patterns involves a “class that can create instances of other classes without depending on them”. “The resusable class delegates the creation to another object and access it via an interface”. “To be reusable a class that creates arbitary objects should not need to know the actual classes”. An interface will declar unimplemented abstract methods that the class will provide concrete implementations for. The factory design pattern will be used to implement the many game objects within the game. The majority of the instructions will be concatenated within the class, therefore the main class will not need to know the actual classes.

The next design pattern that will be implented is singleton. The Singleton pattern only allows the creation of one instance of class while providing a global access point to the instance. As well of ensuring only one object of a class exists, it provides security to it’s variables. The player class is perfect for the use of this design as there will only be on instance of the player and because it is central to the software, it will be interacting with the majority of the other elements of the software.

The observer pattern will also be implented within this software. This pattern “allows objects to register dependencies between objects, so that an object will notify those objects that are dependent on it when its state changes”. As this application is a game, this software is automatically going to implement this pattern to work efficiently. This is because the game will contain an EventHandler that will observe interactions from the user and then make changes and notify objects from these interactions.

Finally, the composite pattern will be implemented. This pattern “compose objects into tree structures to represent part-whole hierarchies. Composite lets clients individuals and compositions of objects uniformly”. This is most relevant for the map of the game. The map will be make of different shapes and components but will act as one when the player or enemies interact with it.

# Application of Design Patterns

When creating the software, as previously stated, the product backlog was followed to create the software. This meant that the first implmentation of the design patterns was singleton in the creation of the player to move within the software. The singleton is incredibly simple to implement. It requires following a precdure of declaring a static field varaible instance that will be used to store the Player object and setting it to be null. The player constructor will then be private meaning classes cannot construct a class in the normal manner. A getInstance() method is declared. This is the alternative used to in instantiated the Player object. It set a instantiated Player object to instance and returns to the class that called the method. However, there is to check to see if instance has already been instantiated and if it has, it will just return the current Player object. Ultimately meaning there will always just be one player within the software. The code is shown in the screenshot below.

Graphical user interface, text, application

Description automatically generated

As stated before within the design, the factory class was used to create the GameObjects within the software. All objects apart from the map were a subclass of a GameObject class. This is because the map had the composite design pattern implemented within and did not use the inherited methods that came from it. The outliner within the game objects that were create using was player. This was because there are many instances of the other game objects where there is only one player. Also, there are a number of superclass methods that are utilised by the other game objects. It was still created via the factory as it was a game object. However, the rest were created in an abundence which made it perfect for the factory to take charge of the creation.

This

# Conclusion

The

# References

* <https://refactoring.guru/design-patterns/prototype>
* https://refactoring.guru/design-patterns/singleton