SWEN30006 Project 2

Rohit Sandeep Ambakkat 1200197
Thaya Chevaphatrakul 1167144
Tran Thanh Han Ha 1472202

The design of the Editor/Tester for PacMan in the Toruverse has been implemented in a manner such that it is easily extendable and it is robust. To address the various requirements that have been provided we make use of various GOF design patterns that give us pre-existing solutions to some common design problems. The use of these design patterns was essential in the development of our design as they provide us with scalability and flexibility. Additionally, we also incorporate the various GRASP patterns into our design as they provide us with an ideal foundation or guideline for assigning various responsibilities that need to be carried out within our design.

Our design's primary aim is to provide high cohesion and low coupling. However, there may be a tradeoff present within the design and we may have had to make adequate moderation as within a program such as this the Game Class will be bound to have a higher degree of coupling due to it being responsible for a large chunk of the operation.

Design of Editor/Tester

1. Start-Up:

The initial skeleton code provided had the Editor and Tester divided into two separate programs. However, the requirements required us to have them merged and the entry point to be the **<<Driver>>** class. The start-up behaviour also had to alter depending on the arguments provided to the program itself as different arguments led us to run different sections of the program. As such we chose to implement this start-up behaviour via the use of a Factory Method pattern in the class **<<EditorTesterFactory>>.** We initiate this class in the **<<Driver>>** class and call its function **createGameGridObject** to help us with the creation of the appropriate object which is either a controller or an instance of a game depending on whether the user has passed a folder, file or nothing as an argument to the program. The factory method pattern is an ideal solution for this particular requirement as the problem it poses is that we have to determine what class is going to be ideal for the creation of the **<<Game>>** and **<<Controller>>** instances especially when we have such complex creation logic. The Factory Method pattern utilizes pure fabrication and the creator pattern to separate the responsibility of the complex creation logic allowing our design to be much more cohesive as a singular class is now responsible for this instantiation. It also promotes lower coupling as now the **<<Driver>>** is not directly related to both the **<<Game>>** and **<<Controller>>** but rather we rely on an intermediary. This design is an ideal choice for the Editor/Tester as it keeps it open to protected variation and if there were to be additional start-up logic added it would be quite simple to extend.

2. Portals:

To develop the portals which trigger an event which is the movement of the character sprite that lands on its location to the corresponding portal we utilize the observer design pattern. This is an ideal design pattern to utilize as various objects such as PacMan or the monster are interested in observing the state changes or events triggered by the portal to perform a necessary reaction. To implement this design pattern we create a **<<Portal>>** class within the **<<Game>>** class due to the information expert and creator principle which is used to allow each of the portals on the map to notify other listeners such as PacMan or monsters to changes in its state. The **<<Portal>>** class also creates an instance of the **<<PortalManager>>** class which is used to manage the subscribers to the **<<Portal>>** class within the game. The classes that listen or observe the state of the **<<Portal>>** within the game inherit the **<<PortalListener>>** interface which allows them to create their unique response to changes in the state of the portal which for our implementation is changing their current location to the location of the corresponding portal. The design we have currently implemented for the portals is an ideal choice as it provides low coupling due to the classes such as **<<PacActor>>** and **<<Monster>>** being disconnected directly from the **<<Portal>>** class itself and only being aware of the interface. It also provides our design with high cohesion as the Actor classes are responsible for their response to the events created by the Portal while the **<<Portal>>** class is responsible for managing its observers. It offers our design-protected variation as well for any additional actors that interact with the portals we can easily allow them to observe the state changes in the portal. If we were to implement any unique response to the portal that differs from the usual, we can easily do so as we simply have to alter the implementation of the method that is being inherited from the **<<PortalListener>>** interface.

3. Autoplayer:

Within the original design, PacMans movements were directly implemented into the **<<PacActor>>** class itself. However, to implement Autoplayer for PacMan we have chosen to instead rely on the Strategy Pattern. The Strategy pattern is a suitable choice for the problem that is presented with the inclusion of the Autoplay feature. The Strategy pattern allows us to freely choose which approach to PacMans movement to take depending on the input from the property file and it allows us to keep all the related algorithms regarding PacMans movement coupled. To implement the pattern we create a common interface that each strategy implements which is **<<MovementStrategy>>**. We utilize the original logic that was present in the code to determine whether **<<PacManActor>>** should create an instance of the **<<AutoMove>>** class which contains the algorithm if PacMan is to move by itself or whether it should create an instance of **<<RegularMove>>** class. The strategy pattern provides our code with high cohesion as each differing strategy is well contained within its class and is linked together via a common interface. It also creates lower coupling as **<<PacActor>>** does not have to rely on a concrete implementation of the strategy and can simply utilize the strategy which suits their needs. The pattern also successfully makes use of the GRASP pattern of polymorphism to allow us to deal with alternative behaviours. Additionally, it ensures that our design utilizes protected variation if a developer wished to add an additional movement algorithm or wanted to alter the functionality of an existing algorithm it is quite straightforward as they simply have to create another class that implements the common interface or edit the specific class.

4. Level Checking

To store and utilize the various checks required for each of the levels created we have chosen to employ a combination of the composite and strategy pattern within our design. This is an ideal choice as similar to the design choice made with the Autoplayer it allows us to choose which checks to implement and keeps these related algorithms well encapsulated. We also use the Composite Design pattern as it allows us to treat a large number of varying checks or rules as if they were objects of the same class and provides a singular access point to all of the checks. The composite pattern is also ideal in allowing developers to modify which checks are utilized and potentially add or remove certain rules. To implement these patterns into our design we initially create the strategy pattern and so we create a common interface **<<LevelCheck>>** which each check such as **<<PacManCheck>>** implements. To utilize the Composite design pattern in concert with the Strategy pattern we also create a **<<CompositeLevelCheck>>** class which each check class extends. Each check can then be added into an Array within **<<CompositeLevelCheck>>** allowing them to be accessed easily. The design choice we have made regarding level checking provided us with high cohesion, lower coupling, and polymorphism similar to that of the Autoplayer. The use of the composite pattern also provides an additional use of polymorphism as we can call each check without differentiating between their specific classes. The design choice also provides protected variation as we can easily add another check and add it to the **<<CompositeLevelCheck>>**.

5. Game Checking

In order to implement Game Checking we don't employ any particular GOF design pattern. We instead use the GRASP pattern pure fabrication to create the class **<<GameCheck>>** which provides our design with high cohesion. Using pure fabrication keeps all logic regarding checking the game folder contained within its own class which a class such as the **<<EditorTesterFactory>>** can utilize. Our simple design choice also ensures protected variation if a designer chooses to add additional game checking as we can simply extend the functionality of the **<<GameCheck>>** class.

6. Multiple Maps

To implement the cycling through multiple maps within the game we also do not employ any particular GOF design pattern but instead use GRASP patterns to assign the responsibility. As the **<<Game>>** class is the class that contains the map and the information regarding the creation of the maps, we utilize the creator and information expert patterns and assign the responsibility to **<<Game>>.**

7. Edit and Test Mode Features

To ensure that level checking is completed on each level of a game folder we employ a facade design pattern via the class **<<LevelCheckFacade>>**. This is an ideal choice as it provides the **<EditorTesterFactory>>** which is responsible for creating an instance of the game if we pass through the folder, an easy interface to the complex logic involved with the checks. Using the Facade pattern hides this complexity from the Game's side of the program. The use of the facade pattern also provides lower coupling via indirection as the factory does need to directly link to the LevelChecks. The facade pattern also encourages high cohesion as all the necessary functionality that is required by the factory from the editor subsystem is all located within a singular class.

To allow an instance of **<<Controller>>** class to create an instance of **<<Game>>** class and vice versa we employ the adapter pattern. The adapter pattern allows us to take the information from both classes and transform them into a format that the other class can accept. Within our design, we create a common interface which is **<<ProgramAdapter>>** that is utilized by the two adapters within our system which are **<<GameAdapter>>** and **<<EditorAdapter>>.** When the game intends to create an instance of the editor it creates an instance of an **<<EditorAdapter>>** and passes through the required arguments and vice versa. The adapter pattern provides many advantages to our design such as lower coupling via indirection as both classes do not directly interact with each other but rather interact with the interface. It also potentially promotes high cohesion as all the logic regarding the communication between both classes is kept well contained

Design of Autoplayer

To modify our current autoplayer design to have it function in the presence of monsters and pills we can have another Strategy class that can be named **<<DefensiveStrategy>>** that extends our current **<<AutoMoveStrategy>>.** Within this class, we can implement the logic we wish to occur if a monster was approaching PacMan such as attempting to move toward the closest ice cube and override the original implementation of moveInAutoMode(). The information about the ice cubes can be passed from the game to Pacman and then directly to the new strategy class. To determine when we want to employ this **<<DefensiveStrategy>>** we can utilize the observer pattern. We can make PacActor a subscriber or listener to the **<<Monster>>** class and when their location is within a certain range of the **<<PacActor>>** we can notify the class to switch to the **<<DefensiveStrategy>>.** This approach that has been designed is suitable for this extension as it utilizes our existing Strategy design pattern which allows us to employ the polymorphic behaviour to switch between strategies when necessary. It also continues to promote low coupling and high cohesion as with the observer pattern the PacActor and Monster are not directly linked. In order to visualize the potential design we can refer to the Simple_Extension_Design_Model.

Assumptions:

- When PacMan dies the game window does not close
- There will only be one or no argument passed through to the program which will be either a folder containing the maps or a XML file.
- There will be a maximum of 10 maps within the game
- There can only exist one set of one type of portals
- The properties file will only make use of PacMan.isAuto and the seed
- The level check is performed exclusively when the Save or Load button on the controller GUI is pressed or prior to testing a game folder.
- The outputs of the game and level check will be outputted to another file rather than Log.txt
- Teleportation of actors is exclusive to the portals
- If an XML file is passed through as an argument it will not be empty