Part 1: The Current Design

Our first step was to scrap the skeleton code for the pacman game and plug in our implementation from *PacMan in the Multiverse*. This was done as we believe our code was extensible and had a high level of cohesion.

It included elimination of the high coupling and lack of cohesion of the skeleton. We did this by adding a class called GameManager to manage Game, as this class was extremely bloated. We also added abstract parent classes for the relevant actors, GameItems and GameCharacters, increasing cohesion. Through doing this we also utilised information expert to ensure classes only had access to information relevant to their individual operation. In the end, we could make a Game, simply manage that game with the GameManager. Also creation and handling of actors was much more simplified and clean. The main issue with this design was its inability to support multiple levels, and the fact that GameItems did not account for varying collision behaviour with GameCharacters.

We then took out all of the extensions *PacMan in the Multiverse* added to the base code. This was the base we built *PacMan in the TorusVerse* from. There was much that needed to be refactored and added to accommodate the features and additions, this will be discussed in the next section.

Along with this the 2D map editor was taken and added to the pacman game so starting up with a file as input opens the editor for design of the levels.

Note: The Domain design model in Figure 6 shows a very simple design for the system with new features added. New features are described below.

Part 2: Adding new features

2.1 Levels

A multilevel system has been implemented for *PacMan in the TorusVerse*. This was done through a refactoring of the Game and GameManager classes we had in our previous design. A level class was made containing all necessary information to run a level. The GameManager class now contains and manages an array list of levels which are made in and played by Game. This simplified the design a lot. Previously we considered creating a new Game for each level and handling the making of these levels in Driver but instead it seemed to make more sense for Game to only be created once and have multiple levels. This increased cohesion as it let Driver only handle what was necessary to drive the game. Additionally we made GameManager a singleton as there will only ever be one instance of it. Allowing global access and simplifying argument calls in all functions.

For the level creation itself we used the decorator design pattern as we wanted to be able to dynamically add in game items and characters at runtime. To do this we made Level abstract and added a single concrete implementation of level. Then we made an abstract LevelDecorator with two implementations, LevelCharacterDecorator and LevelItemDecorator. By doing this, game items and game characters can be dynamically added and drawn, simplifying this functionality.

Finally to allow multiple levels to be played consecutively the Game class cycles through each level one by one until there are no levels left to play, the game then pauses and displays the win message. “YOU WIN”.

2.2 Auto Player

The auto player that is used for the game is a simple AI that will choose the next move that gets pacMan closer to the closest Pill/Gold. It uses a BFS from the current pacMan position and spreads out until a Pill or Gold is found and back propagates to the first move that was needed to reach that Pill/Gold. When applying the auto player, the strategy design pattern was used. Strategy was used because if a new improved auto player was made, then all that is needed to implement it is a class that implements the strategy interface and to switch the auto player to use the newly created auto player, see figure 3 . For example if the auto player was to focus more on the presence of monsters and ice (ice being able to freeze monsters) then you make a class called MonsternIceAutoPlayer and implement the AutoPlayer interface and now the game can easily switch to that auto player by setting the autoplayer to the MonsternIceAutoPlayer which is able to be defined as a AutoPlayer type. Figure 3 shows this by the AdditionAutoPlayer being the newly updated auto player. The strategy pattern for auto player also allowed for new auto players to be as complicated and packed (lots of methods) as they want without interfering with the pacMan which is implementing the auto player. This supports high cohesion.

2.3 Game and Level Checking

When implementing the level checking, each check that was required by the specs was divided into their own class. Doing this allowed for the use of the composite design pattern. All the checks implemented the LevelRequirement interface which allowed for one LevelRequirement variable to include as many checks as needed. Using the composite pattern for checking allowed for very easy extension in adding new level checks. All that is needed is for someone to make a new class that implements the interface and alter the level checking method from the interface. However, adding more and more checks can get messy, so the factory design pattern was used so one class can hold the creation of all checks and future checks that could be added. Due to multiple classes potentially requiring a level checker that may or may not be different to each other, this factory was designed as a singleton so that every class can easily add checks to their level checker.

When a level does fail, This could mean that the level is impossible to complete. For this reason the system will not load the level, and if the level that failed is a multiple level game, then the game will skip this level and move onto the next one.

Before a level fails, the log is updated. The format of the level errors is one line letting the reader know that the next lines are the errors for the specified level. This is followed by either the level error (the problem that occurred) or a message saying that no errors were found with the level. This format was used because it is much easier for the user to read and find errors relating to a certain level if needed.

Game checking is similar to Level Checking. The difference is that these are checked before the game runs. If there is an error with no maps being found, or maps having the same sequence, then the game exits and logs the error to the logfile. The Factory pattern is used here as it implements the LevelRequirement interface, and this is done in the Driver class.

2.4 Portals

Portals have been added to the game. Portal pairs of the same colour pair up and any game character that walks on one teleports to the other. Whilst playing the game the portals are a little too big

and could use a resize but as this did not affect the functionality of the game and this project was primarily concentrated on design we did not fix this.

We decided that a portal should be classified as a GameItem as all game items are immobile and can also collide with the player or monsters. To do this we made them inherit from GameItem. Although now we see that pills, ice and gold all interact differently to the portals. To fix this strategy pattern was utilised as the behaviour when portal is collided with is different to the three others. An interface, ICollisionStrategy, was made with a method to handle collisions. Both EdibleCollisionStrategy and PortalCollisionStrategy implement this interface with their versions of what collision does. This simplified our implementation and allowed extensibility. PacActor or Monster can now just call “collide()” to interact with the game items instead of handling eating and teleporting separately.

To make portals work it was crucial the levels were refactored as discussed earlier. During the refactoring, the original function in Game, “drawGrid” added and drew all actors to the game at once. To pair up the portals these two steps needed to be split up, this is because the portal pairing works by going through all portals and storing its matching colour. As a result all the portals can be added to the game, pairing can occur and then they can be drawn.

Part 3: Application and UI

3.1 Additional Buttons

Two buttons were added to add a better and easier user experience when wanting to control the game from the controller.

3.1.1 Start Game Button

This was added so the user can easily, once a level is shown on the map editor screen, play that game, unless the level checker fails of course. The user has access to play that game, either win or lose, and be brought straight back to the controller. This was done by creating a new thread (using Swingworker) to run the game on and dismissing the thread once the game was complete.

3.1.2 Auto play Game Button

In addition to the Start game button, if the user would like to play the game currently on the controller screen with auto player playing for them, they can easily press the “auto player” button. This prints a text to the terminal letting the user know that the auto player will be playing for the next game. Once the game is complete, the auto player is turned off and must be turned back on (auto player button pressed) if the user also wants it auto playing for another level they may choose.

3.2 Start Up

Startup required checking the directory if it was a file, or a folder, or none - for starting up in Edit Mode of the given map, Test Mode or Edit Mode with no current map respectively.

To load up a given map in Edit Mode, I created the XMLParser class, which reads in an XML file, grabs the width and height, and the individual rows which are converted into a String to be passed on to the Game class which would draw the Grid from the given String. The XMLParser class uses constant Strings from the Constants class, which contains every item’s String representation.

Summary

PacMan in the TorusVerse was created in such a way to enable the extensibility of more levels, game items, monsters and more. This was done by utilising GRASP principles to keep high cohesion, low coupling and ease of use. Additionally through using strategy, composite, factory, decorator and singleton design patterns we have made our code more comprehensible and solved all major problems encountered in the design process. Strategy pattern solved our issue of easily adding more auto player algorithms and our issue of differing collision behaviour in game items. Composite pattern helped us fix being able to add multiple level and game checks. We used the Factory pattern to create different level checks that can easily be added on to the level checker. The Decorator enabled easy level creation and grid changes at runtime and the Singleton pattern ensured that only a single instance of the GameManager and the LevelRequirementFactory existed. Our new design has provided a cleaner, neater model which is able to handle extensions in an elegant and simplified way for future instalments of the franchise.

Static design model

The design model below shows some plane classes that are extended into other figures so the model is easy to read and understand. For example the GameItems class is extended in Figure 2.

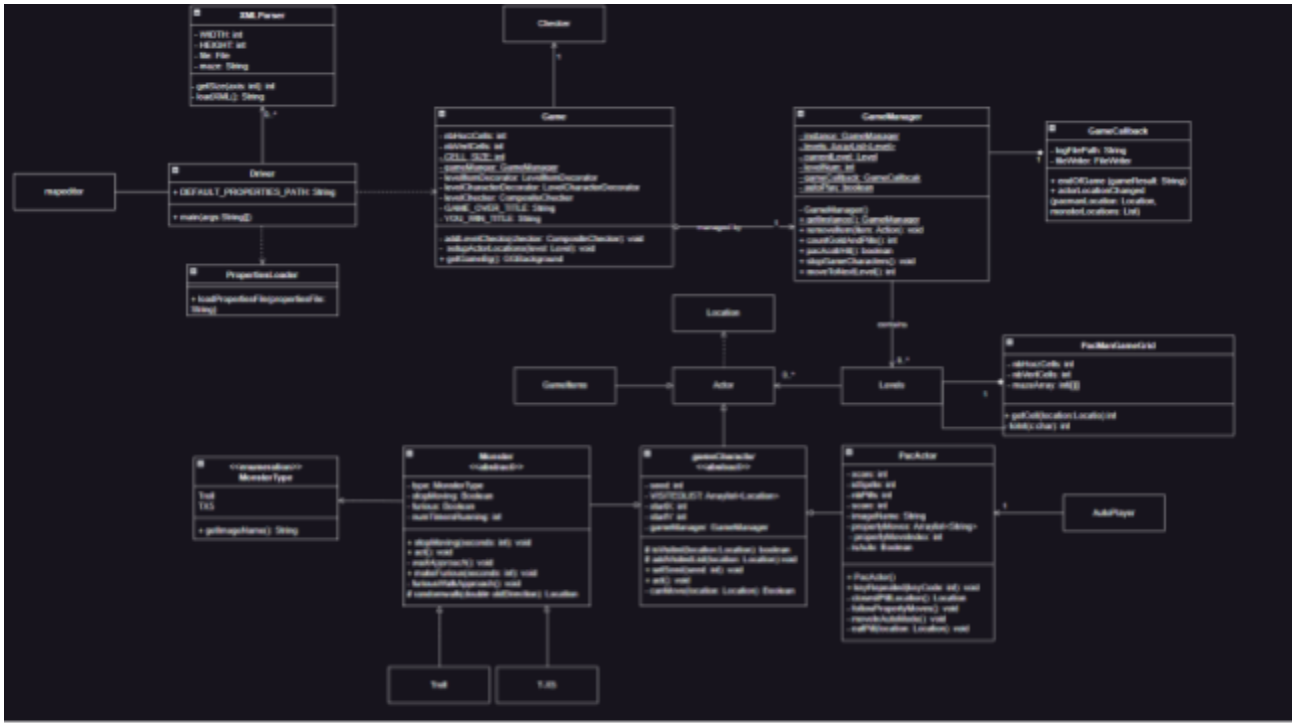

Figure 1

Design model Game Items

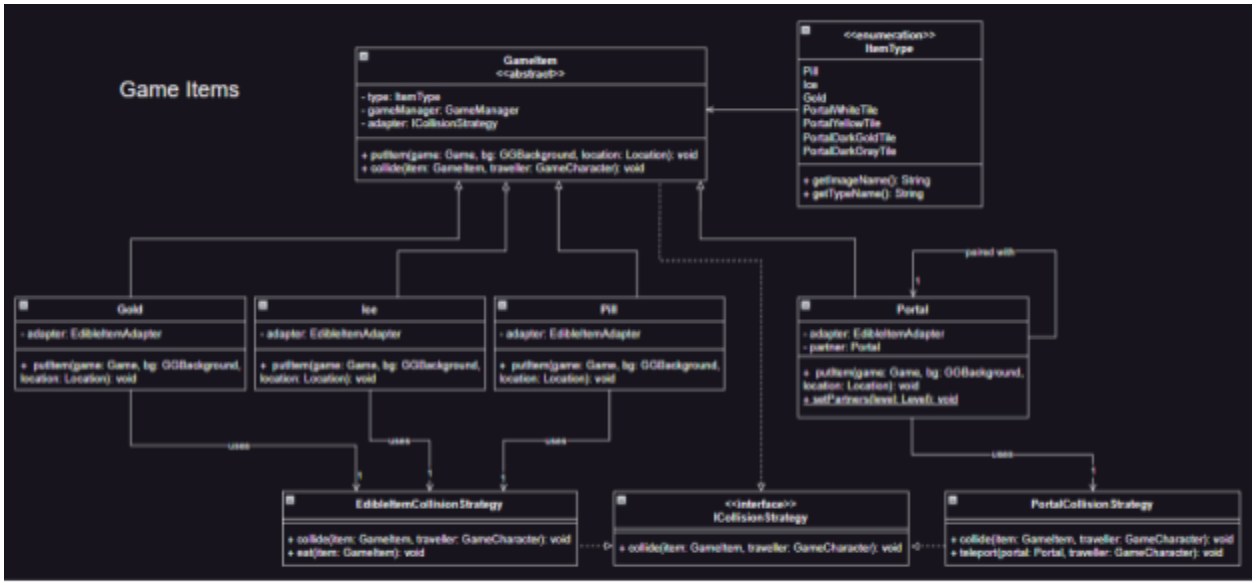

Figure 2

Design model Auto Player

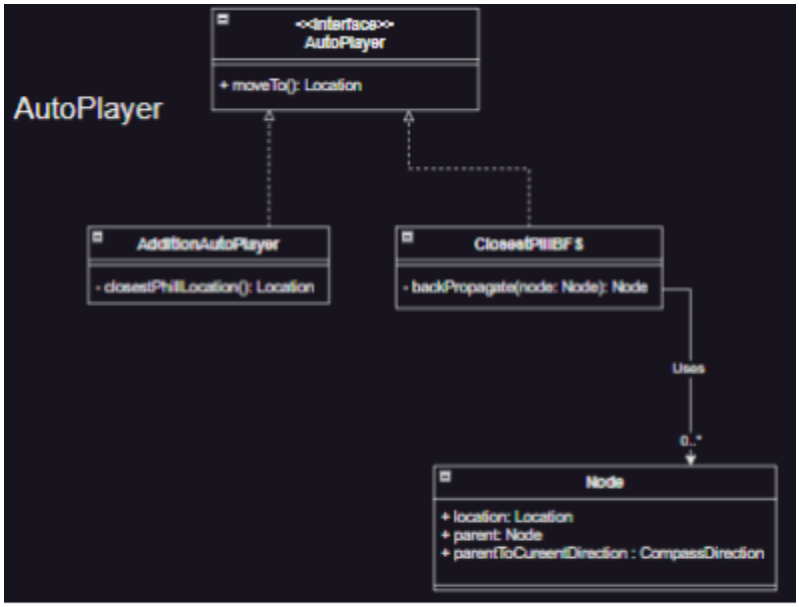

Figure 3

Design model Level Checker

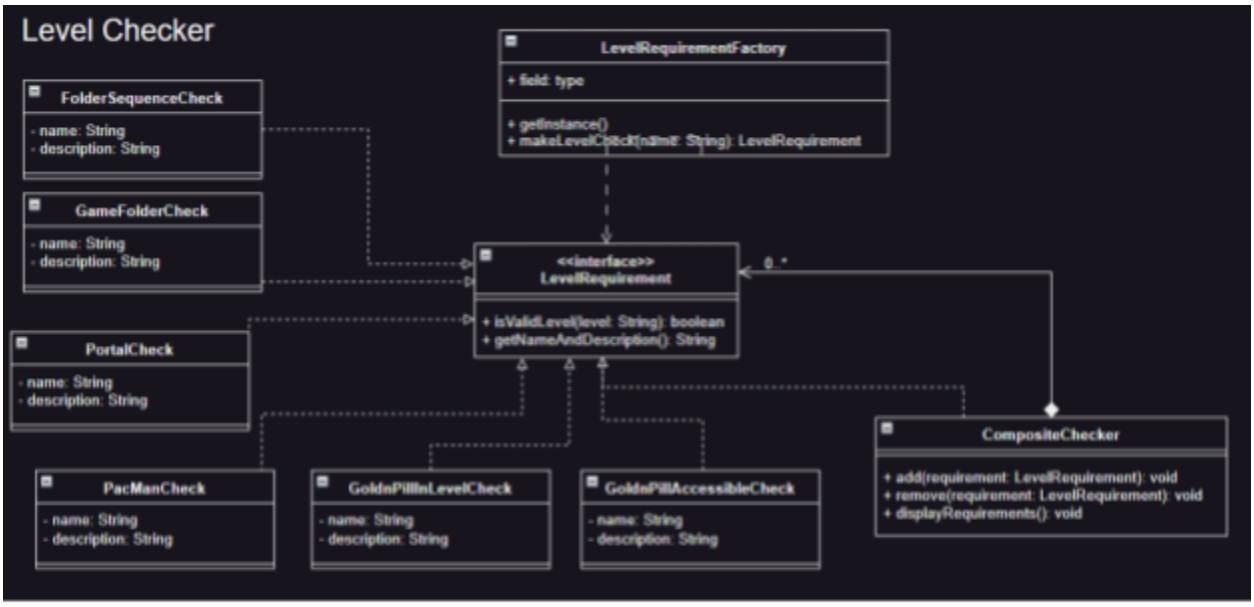

Figure 4

Design model Levels

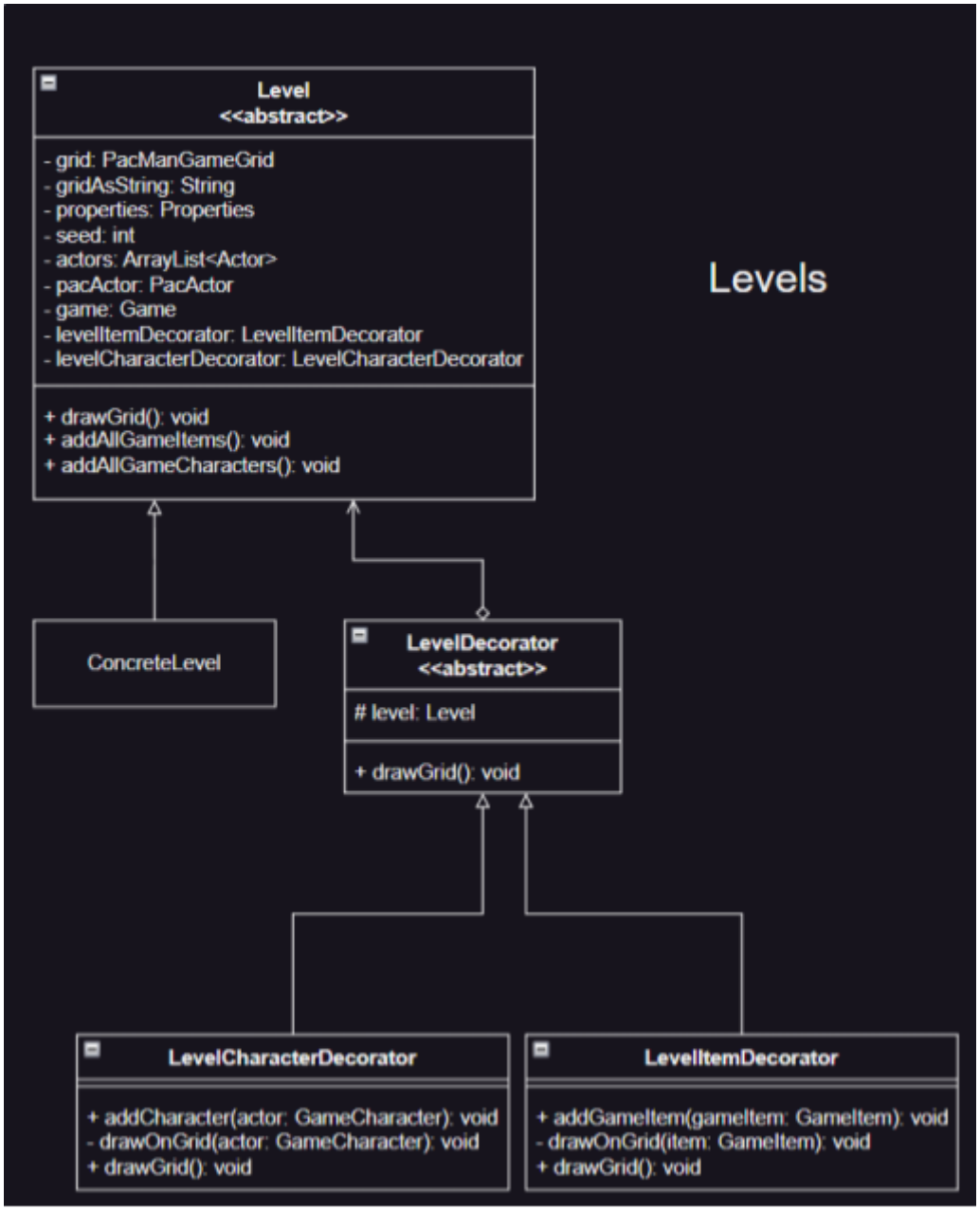

Figure 5

Domain design model

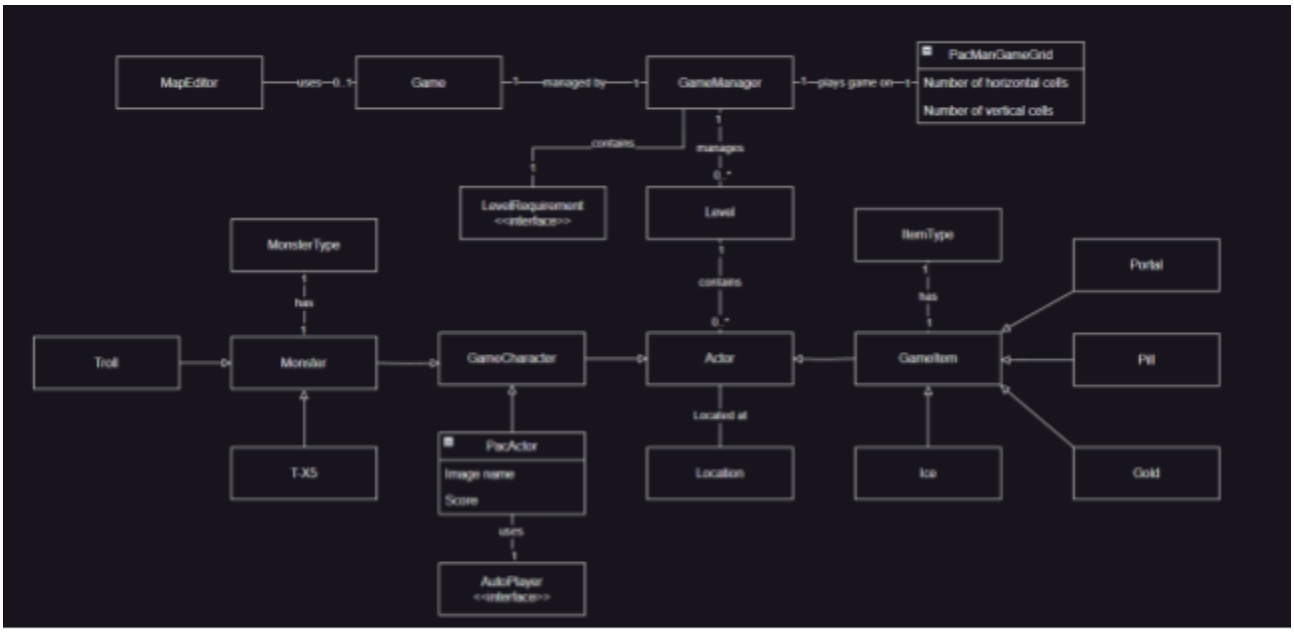

Figure 6