SWEN30006 Software Modelling and Design

Project 2: PacMan in the TorusVerse

1. *Overall*

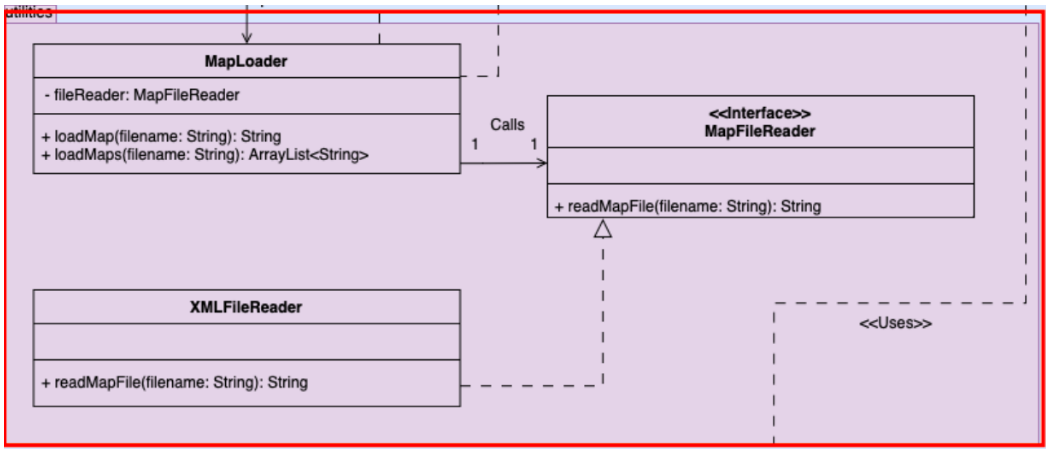
Our team has implemented a variant of PacMan game, developed an editor and a test application. Since the new variation, PacMan in TorusVerse, has granted users' access to editor as well, 2 provided codebases are combined with controller initiate all the settings and games. TorusVerse will be developed based on its original simple version, it will overall focus more on the maps provided for the game and include features of portals that allow characters move from one place to another directly. In addition, the game will support multiple levels, so player can progress from one level to the next.

2. *Design of Editor*

2.1 Read Map File

For Design of editor, the "Controller" class is set as a starting point for the application, if no input argument, then a blank grid is shown for users to edit and make their own maps. If a folder name or a specific file name is used as an input to the controller, the controller will load the map(s) using a "MapLoader".

A **strategy pattern** is used for reading map files to take into the consideration of the format of different file names (specific strategy pattern class diagram is show below). Although for this project, only '.xml' files are allowed to store the information for maps, maybe in further extension, different types of files are allowed to store the information for maps such as pdf, csv, html etc. In such cases, more file reader classes corresponding to different types of files can be created and implement the "readMapFile" method in the "MapFileReader" interface, designed open for extension while closed to modification.


utilities
MapLoader
- fileReader: MapFileReader
+ loadMap(filename: String): String
+ loadMaps(filename: String): ArrayList<String>
Calls
1
1
<<Interface>>
MapFileReader
+ readMapFile(filename: String): String
XMLFileReader
+ readMapFile(filename: String): String
<<Uses>>

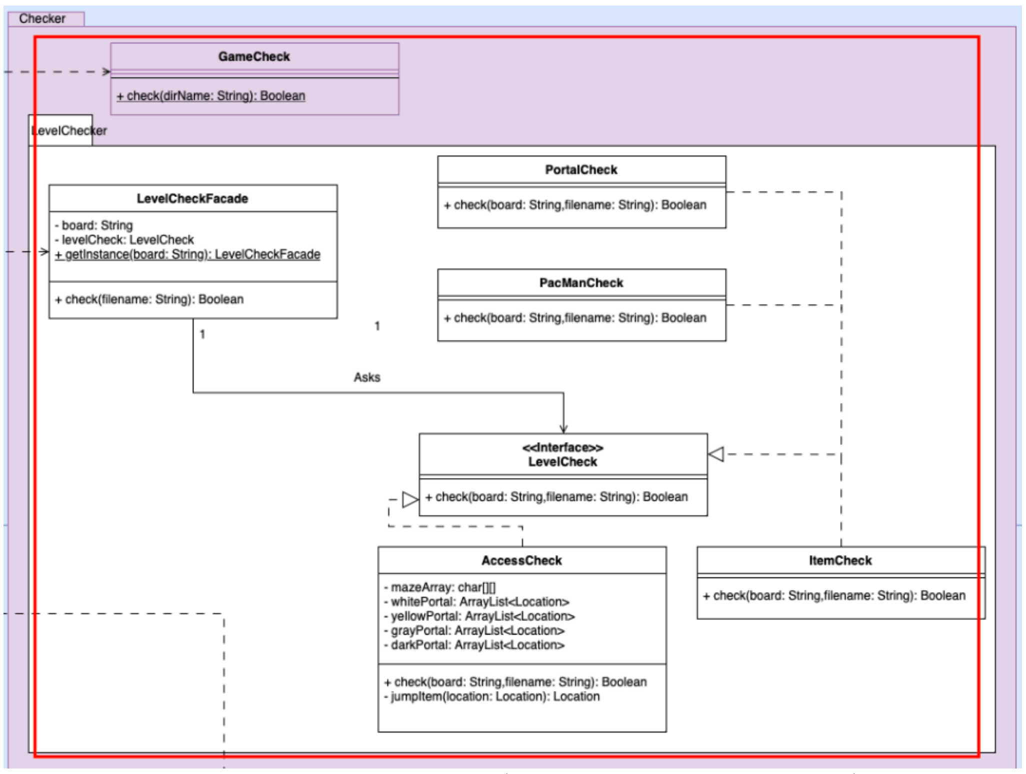
Partial class diagram of map reader
"MapFileReader" as strategy for the pattern and "XMLFileReader" as a concrete example

Originally the controller class would be a context to the “MapFileReader” strategy, but “MapLoader” here is used as an intermediate object to creates an **indirection** between the controller and the “MapFileReader” to avoid direct contact between them and **reduce coupling**, in the case that the method of the strategy gets changed and modification needs to be made to the method in controller class. Here two methods “loadMap” and “loadMaps” are in the “MapLoader” class to deal with a specific game map and a folder of game maps respectively.

Before loading the maps, game checking and level checking are required to check the validity of the game map paths as well as game itself. To address the requirement, game checking function is set as a class with one static method “check” while level checking is designed using a Façade pattern to apply various aspects of checking.

2.2 Game Check

According to the below class diagram of the Checker package, “mapLoader” class will call the **static method** in “GameCheck” to check for the map folder as well as whether maps are well-defined, it is designed not necessary to instantiate a “GameCheck” class in “mapLoader” every time game check needs to be applied as that would **save some space memory**.



Checker
GameCheck
+ check(dirName: String): Boolean
LevelChecker
LevelCheckFacade
- board: String
- levelCheck: LevelCheck
+ getInstance(board: String): LevelCheckFacade
+ check(filename: String): Boolean
PortalCheck
+ check(board: String,filename: String): Boolean
PacManCheck
+ check(board: String,filename: String): Boolean
1
1
Asks
<<Interface>>
LevelCheck
+ check(board: String,filename: String): Boolean
AccessCheck
- mazeArray: char[][]
- whitePortal: ArrayList<Location>
- yellowPortal: ArrayList<Location>
- grayPortal: ArrayList<Location>
- darkPortal: ArrayList<Location>
+ check(board: String,filename: String): Boolean
- jumpItem(location: Location): Location
ItemCheck
+ check(board: String,filename: String): Boolean

class diagram of Checker package (include game check and level check)
“AccessCheck”, “PortalCheck”, “PacManCheck” and “ItemCheck” as 4 aspects for checking

2.3 level Check

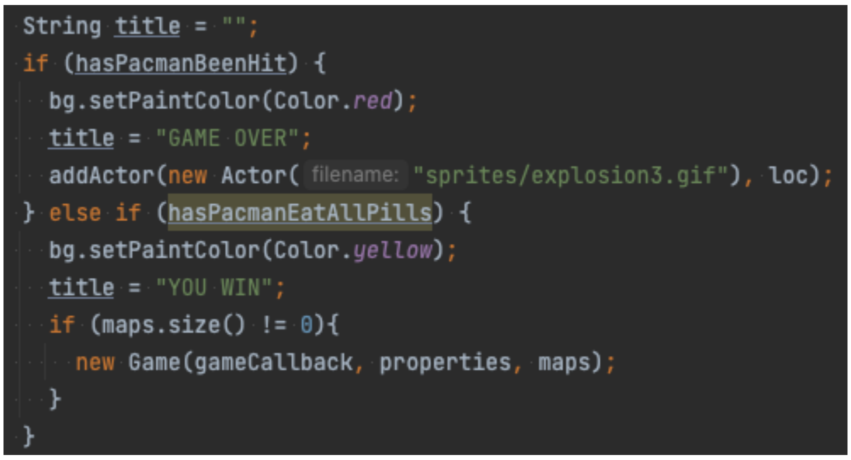
For every map in the folder, level check is applied to each one of them. **Façade** is used here **set up as singleton** to wrap the level checking subsystem into one unit and calls all the level checking in the “check” method, it also plays the role similar to the “MapLoader” in the previous diagram, which **decouples** the relations between “MapLoader” and various checking classes, to avoid direct changes to the “MapLoader” class if the methods for these checking change.

The subsystem inside is designed using a **strategy pattern** to allow for further extension as when more monsters or features are added into the game, new checking maybe required as to the number of monsters, the type of the monsters is valid or not etc, and they can all designed to be a new class and implements the “LevelCheck” interface. In this way, the size of the subsystem is increased while no further modifications are required to the existing rules and classes.

If the game check or the level check fails, different responses are made in the “Controller” class, and if they are passed, new game will be created for users to play the game on the map(s) in the folder. Then we’ve come to the design of the Pacman game itself.

3. *Design of Pacman Game (Autoplayer)*

The input parameters to the “Game” have added one more variable “maps”, an array list of strings with each string represent a single map information. After this, “PacManGameGrid” will be called in the game constructor with one string (one map) at a time as additional input to draw the current level of map. If the current map is finished and the size of “maps” is not zero, a new game will be created for user to play the game on the next map in the folder. (Demonstrated in the below code snippet)



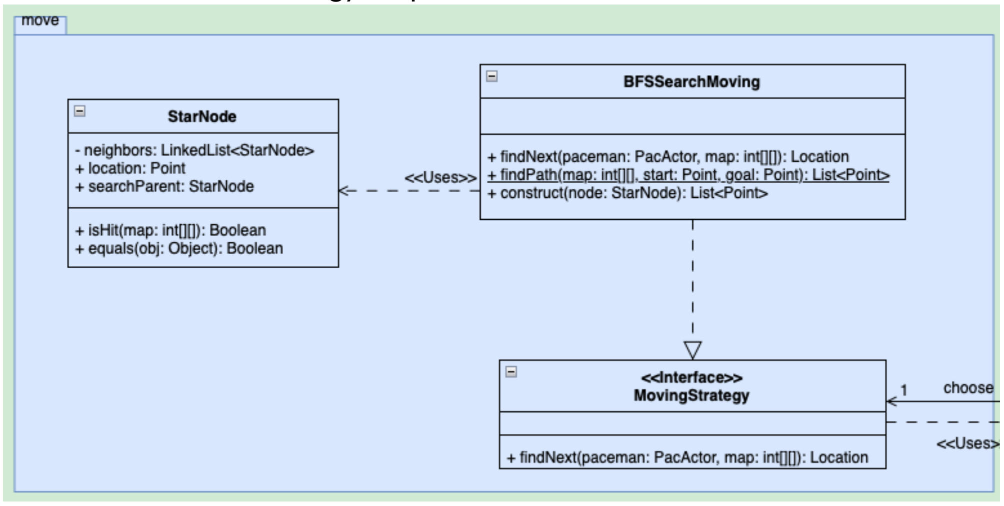
```
String title = "";
if (hasPacmanBeenHit) {
  bg.setPaintColor(Color.red);
  title = "GAME OVER";
  addActor(new Actor( filename: "sprites/explosion3.gif"), loc);
} else if (hasPacmanEatAllPills) {
  bg.setPaintColor(Color.yellow);
  title = "YOU WIN";
  if (maps.size() != 0){
    new Game(gameCallback, properties, maps);
  }
}
```

The new feature of portals in the game is added inside the game class as they are all part of the game features just like gold and pills. All the rest are pretty much the same as the original design.

Then, since the Pacman can be moved in auto mode, a **strategy pattern** is used for the design of Pacman’s auto moving strategy. From the diagram below, “MovingStrategy” is designed as an interface with is implemented by “BFSSearchMoving”, since only BFS is implemented as a moving strategy for Pacman in this project. If more moving strategy is comed up in the future, random moving, A star, DFS etc, they could all be designed as a new class and implements the “MovingStrategy” interface, so Pacman can choose with strategy to follow in the “act” method in “PacActor” class.

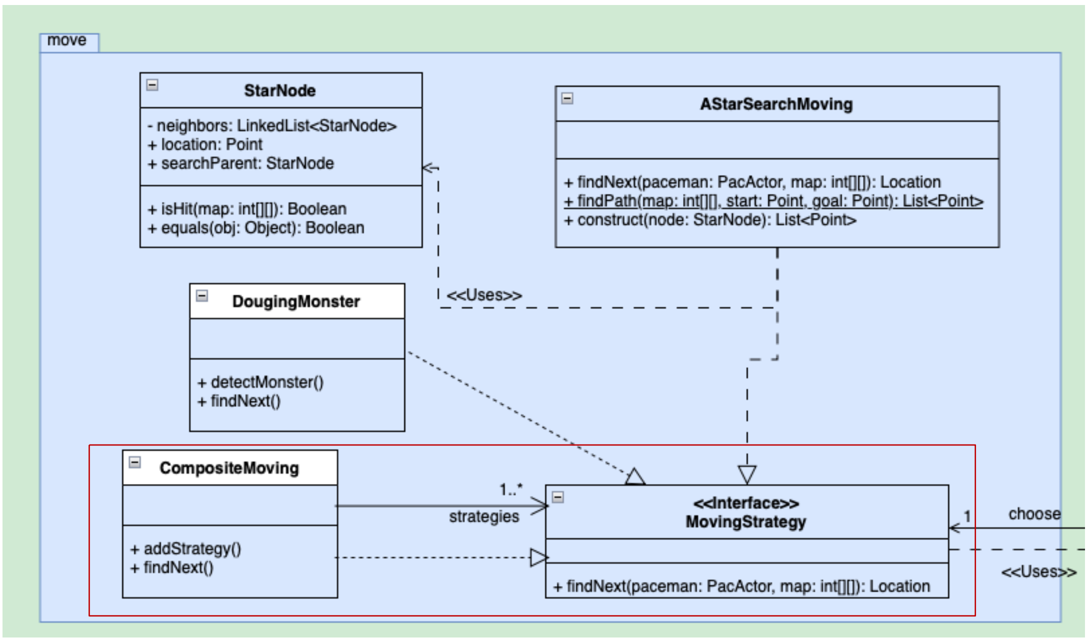
BFS move stands for Breath First Search, it gradually searchs for all the neighbours from Pacman one step each round until it find the location of the closest item to Pacman, during the search process, it will stored all the searched location into a linked list so when it finally reach to the goal, it will search through the linked list to find the path to that closest item location, and make the first move. The the process is repeated until Pacman eats the item, and it finds a new target to perform BFS again.

It is the only strategy used here as it's easy to implement and unlikely to have missing targets. However, it could be slower as it has to search for all the neighbours of all the location it expands to, if time requirement is considered for Pacman to finish the game, other faster algorithms like A star, DFS search could be considered, but then again, they could all be added into the strategy for pacman to choose.


move
StarNode
- neighbors: LinkedList<StarNode>
+ location: Point
+ searchParent: StarNode
+ isHit(map: int[][]): Boolean
+ equals(obj: Object): Boolean
<<Uses>>
BFSSearchMoving
+ findNext(paceman: PacActor, map: int[][]): Location
+ findPath(map: int[][], start: Point, goal: Point): List<Point>
+ construct(node: StarNode): List<Point>
<<Interface>>
MovingStrategy
+ findNext(paceman: PacActor, map: int[][]): Location
1
choose
<<Uses>

class diagram of Move package

This also address the extension problem as how to modify or use the current autoplayer design to have autoplayer work in the presence of monsters and pills. In this case, a **composite pattern** could be applied (demonstrated below) as the player will choose the moving pattern as usual to try and eat all the items, but an extra "dougingMonsters" strategy will be implemented as well and added into the composite class using "addStrategy" for player to avoid the monsters, and if in the future more features like Pacman can attack monsters are considered, "attackMonster" can also be implemented as a concrete example for moving strategy and be added into the composite class. In such way, Pacman can choose to move using "CompositeMoving" strategy and contains all the features required. This design is open for extension with more features and very closed to modification as no need to make changes to the current strategy.



move
StarNode
- neighbors: LinkedList<StarNode>
+ location: Point
+ searchParent: StarNode
+ isHit(map: int[][]): Boolean
+ equals(obj: Object): Boolean
AStarSearchMoving
+ findNext(paceman: PacActor, map: int[][]): Location
+ findPath(map: int[][], start: Point, goal: Point): List<Point>
+ construct(node: StarNode): List<Point>
<<Uses>>
DougingMonster
+ detectMonster()
+ findNext()
CompositeMoving
+ addStrategy()
+ findNext()
1..*
strategies
<<Interface>>
MovingStrategy
+ findNext(paceman: PacActor, map: int[][]): Location
1
choose
<<Uses>>

class diagram of Move package, with the addition of the composite pattern labelled on the diagram.

To adjust to this change and let Pacman to make corresponding actions in the future, in the “act” method in the “PacActor” class, it can check whether there are any monsters on the current map and if there are, the auto mode will implement the “CompositeMoving” strategy and if there are no monsters, it will just choose any other specific moving strategy to finish the map.