SWEN30006 Report – Project 2

Part 1: Design of editor

Firstly, the argument will be passed from the Driver class, and the Driver will determine the type of the argument and create a controller accordingly. If no argument is provided, the controller will initialize an empty editor, allowing the user to freely edit. If an argument is a .xml file, the controller will initialize an empty editor and load the map from the file, enabling the user to continue editing based on the loaded map. If a folder name is passed as the argument, the controller will use the RunMap class to validate the folder and its contents. If the folder is invalid, the program will exit; if it's valid, each map will be created and stored in an ArrayList. Finally, the controller will use the Game class to run the maps in the list sequentially, starting the game.

We aim to delegate the majority of the content processing to the Controller, enabling it to act as an information expert. This approach ensures that the Controller class serves as an intermediary between the connected classes, promoting high cohesion and low coupling.

Another crucial design aspect involves utilizing static methods when working with the GameCheck class (for folder detection), the LevelCheck class (for file detection), and the CheckPortals class (for checking the presence of two portals). This modification allows the RunMap class to verify file correctness without needing to instantiate the relevant classes. Instead, the methods within these check classes can be directly utilized, resulting in more concise and intuitive code.

For folder detection, we will utilize the GameCheck class to perform the following checks: first, verify if the given argument corresponds to a folder; then, check if the required files exist in the folder. Additionally, we will validate each file name, ensuring that they are .xml files and there are no duplicates. The method will return true if all files are valid; otherwise, it will return false.

During the file detection process within each folder, the LevelCheck class performs three checks. Firstly, it verifies if the file contains only one Pacman. Secondly, it examines whether there are precisely two portals of each type on the map. Finally, it utilizes the BFS algorithm to determine if Pacman, starting from its initial position, can access every pill, ice, and gold on the map. If these conditions are met, the method returns true, indicating a valid file.

To receive and store each valid map, we will utilize the XMLToMap class. It will extract all the relevant information from each .xml file and create a PacmanMap class to store it. The PacmanMap will contain the map's width and height, the names, and positions of all characters (such as TX5, Pacman, and Troll), the names and positions of all portals, and the String representation of the entire map. In the XMLToMap class, we will extract the name and position of each tile and append it to a String. When reading the actor's or portal's name, we will create an ActorPosition class and add it to an ArrayList that stores all actors along with their positions. Finally, the XMLToMap class will create a PacmanMap and return it to the RunMap class.

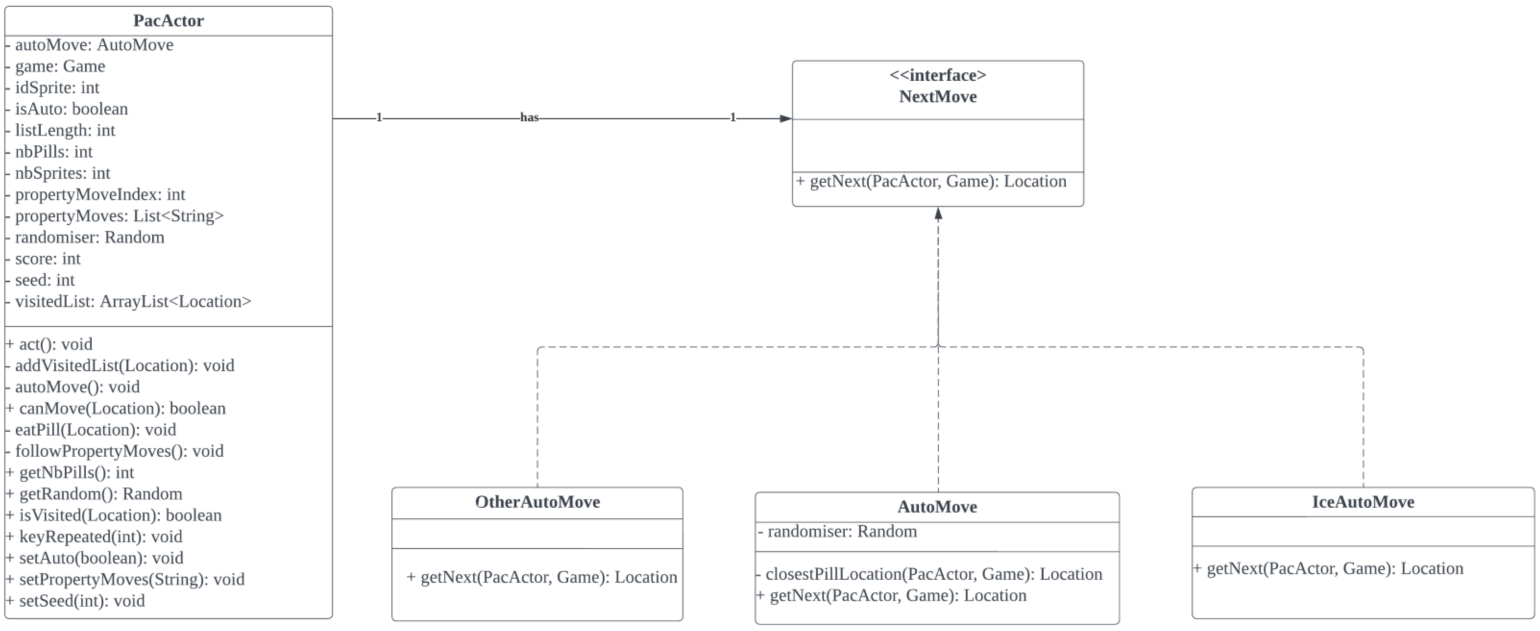
Lastly, in terms of running the game. First, a map will be created based on the string representation of each map. The location information of actors and monsters in each map will be extracted and set in the map. The game will start, and if Pacman encounters a monster, the game will end. If Pacman eats all the pills, ice, and gold in the map, the next map in the folder will be automatically loaded, allowing the user to continue the game. If the current map is the last one in the folder, a "You Win" message will be displayed, and the game will end.

Part 2: Design of autoplayer

Taking future development into consideration, we have applied the strategy pattern to the autoplayer. In our design, there is a strategy interface called "NextMove", which identifies the methods used for Pacman's movement. A specific strategy class

"AutoMove" implements this interface. If we need to add or adjust any function of the current auto-move strategy, we can create a new class called "NewAutoMove" that also implements the NextMove interface. The new algorithms would be placed in the "NewAutoMove" class. By doing this, we can add new strategies without modifying the code for Pacman or the existing "AutoMove" class. This approach adheres to the Open-Closed Principle, which encourages us to design our code to be open for extension but closed for modification.

Figure1: strategy pattern


PacActor
- autoMove: AutoMove
- game: Game
- idSprite: int
- isAuto: boolean
- listLength: int
- nbPills: int
- nbSprites: int
- propertyMoveIndex: int
- propertyMoves: List<String>
- randomiser: Random
- score: int
- seed: int
- visitedList: ArrayList<Location>
+ act(): void
- addVisitedList(Location): void
- autoMove(): void
+ canMove(Location): boolean
- eatPill(Location): void
- followPropertyMoves(): void
+ getNbPills(): int
+ getRandom(): Random
+ isVisited(Location): boolean
+ keyRepeated(int): void
+ setAuto(boolean): void
+ setPropertyMoves(String): void
+ setSeed(int): void
1
has
1
<<interface>>
NextMove
+ getNext(PacActor, Game): Location
OtherAutoMove
+ getNext(PacActor, Game): Location
AutoMove
- randomiser: Random
- closestPillLocation(PacActor, Game): Location
+ getNext(PacActor, Game): Location
IceAutoMove
+ getNext(PacActor, Game): Location