SWEN30006 Project 2 Report

Workshop [Thurs 11.00] Team 2

Weng Jae Chin - 1255326

Aurelia Iskandar - 1251512

Xin Yu Chuah - 1255380

This report entails the changes made to the design of the PacMan in the TorusVerse project and analyses the design decisions made with respect to design patterns and principles. This report aims to identify design alternatives and justify the chosen design decisions for PacMan in the TorusVerse.

Design of Editor

Firstly, the editor for Pacman in the TorusVerse is designed with a Driver class for switching game modes and changing levels, a PortalFactory class and PortalStore class for storing and managing the portal logic, and classes for Level and Game Checking.

Driver

The Driver class is designed using the Singleton Pattern, which ensures that only one instance of the Driver class can exist at any given time. This allows only one point of entry for the control for switching game modes and is the single wrapper that manages the multiple threads that are created and destroyed or hidden when switching between game modes and levels. Additionally, Observer Pattern is employed to ensure that the Driver is notified by the Game and Controller classes of when to switch modes. The Driver is a subscriber that listens for events from two publishers: the Game and the Controller. The Driver listens for events from the Game class, such as when PacMan dies or when the player wins, to know when to switch from test mode to edit mode or to change levels, and it also listens for events from the Controller, such as when the “Start Game” button is pressed. Since the Driver class has the necessary information required to determine which game mode should be active and running, this means the class utilizes the Information Expert principle as well. The class also has high cohesion as it has focused and well-defined responsibilities and simplifies maintainability and enhancement.

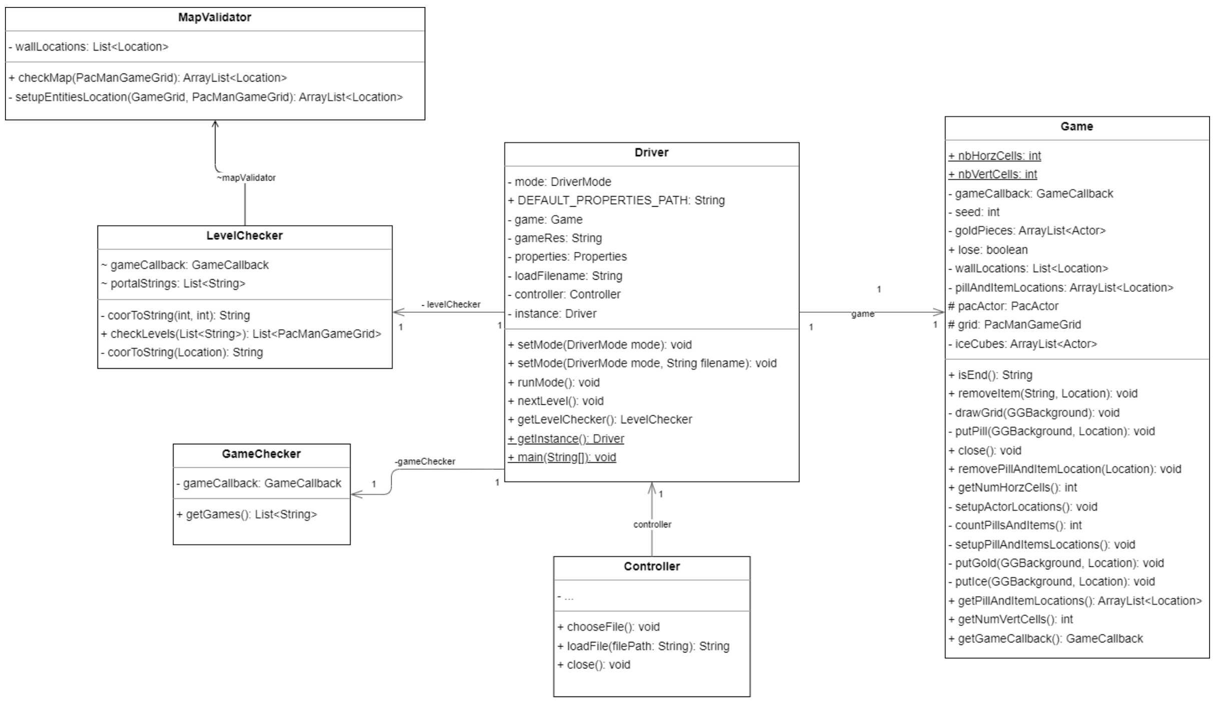
A key issue found was that long-running while loops could not be run as it would disrupt JFrame's rendering and cause the UI to stay stagnant and not update when a while loop is running. To combat this, we refactored the Game class by removing the do-while loop within the Constructor and moving the game ending checks into an `isEnd()` function that is called within the `act()` function of Game, which is a function that is automatically called by the JGameGrid Library.

Level and Game Checking

Two new classes were created to handle level and game checking: LevelChecker and GameChecker. These classes were made solely for the purpose of doing the level and game checks described in the specification; with highly specialised purposes, these classes follow the Pure Fabrication GRASP Principle. Furthermore, another Observer Pattern relationship can also be seen between the LevelChecker and Driver as the LevelChecker informs the Driver of when errors are found so that the Driver can change modes accordingly. In addition to that, we also created a MapValidator to further abstract the function of checking if all pills and gold cells are accessible to PacMan. This ensures that the LevelChecker is not too bloated with features and makes the classes more focused, understandable and manageable, thus reducing coupling.

PortalFactory and PortalStore

The PortalFactory class is responsible for creating portals. This class follows the Creator pattern, which delegates the responsibility of object creation to a separate class. As such, the PortalFactory also employed the Creator Pattern and is a Singleton to ensure that only one instance of the factory can be created to ensure consistent object creation, resource management and centralised configuration of portals. An additional class, PortalStore, was also created to store and manage the portal logic, namely the collision and Actor teleportation. This is a demonstration of the Pure Fabrication GRASP Principle as these two classes were created for highly specific purposes and tasks. Additionally, the PortalStore class provides functionality to maintain multiple portals, such as storing and searching for the complementary portal to teleport the actor to. This ensures that individual classes that want to use portals do not have to access and manage portals directly in a low-level manner and are instead able to interface with them at a higher abstraction level through this class, which is a demonstration of Indirection and Pure Fabrication GRASP principle. These two classes also allow for low coupling as it abstracts Portal-related creation and logic out of the Game class, which makes it less dependent on other classes and also ensures that additional functionality or features related to portals can be easily added and scaled.


MapValidator
- wallLocations: List<Location>
+ checkMap(PacManGameGrid): ArrayList<Location>
- setupEntitiesLocation(GameGrid, PacManGameGrid): ArrayList<Location>
~mapValidator
LevelChecker
~ gameCallback: GameCallback
~ portalStrings: List<String>
- coorToString(int, int): String
+ checkLevels(List<String>): List<PacManGameGrid>
- coorToString(Location): String
- levelChecker
1
1
Driver
- mode: DriverMode
+ DEFAULT_PROPERTIES_PATH: String
- game: Game
- gameRes: String
- properties: Properties
- loadFilename: String
- controller: Controller
- instance: Driver
+ setMode(DriverMode mode): void
+ setMode(DriverMode mode, String filename): void
+ runMode(): void
+ nextLevel(): void
+ getLevelChecker(): LevelChecker
+ getInstance(): Driver
+ main(String[]): void
1
game
1
1
Game
+ nbHorzCells: int
+ nbVertCells: int
- gameCallback: GameCallback
- seed: int
- goldPieces: ArrayList<Actor>
+ lose: boolean
- wallLocations: List<Location>
- pillAndItemLocations: ArrayList<Location>
# pacActor: PacActor
# grid: PacManGameGrid
- iceCubes: ArrayList<Actor>
+ isEnd(): String
+ removeItem(String, Location): void
- drawGrid(GGBackground): void
- putPill(GGBackground, Location): void
+ close(): void
+ removePillAndItemLocation(Location): void
+ getNumHorzCells(): int
- setupActorLocations(): void
- countPillsAndItems(): int
- setupPillAndItemsLocations(): void
- putGold(GGBackground, Location): void
- putIce(GGBackground, Location): void
+ getPillAndItemLocations(): ArrayList<Location>
+ getNumVertCells(): int
+ getGameCallback(): GameCallback
GameChecker
- gameCallback: GameCallback
+ getGames(): List<String>
-gameChecker
1
1
1
controller
Controller
- ...
+ chooseFile(): void
+ loadFile(filePath: String): String
+ close(): void

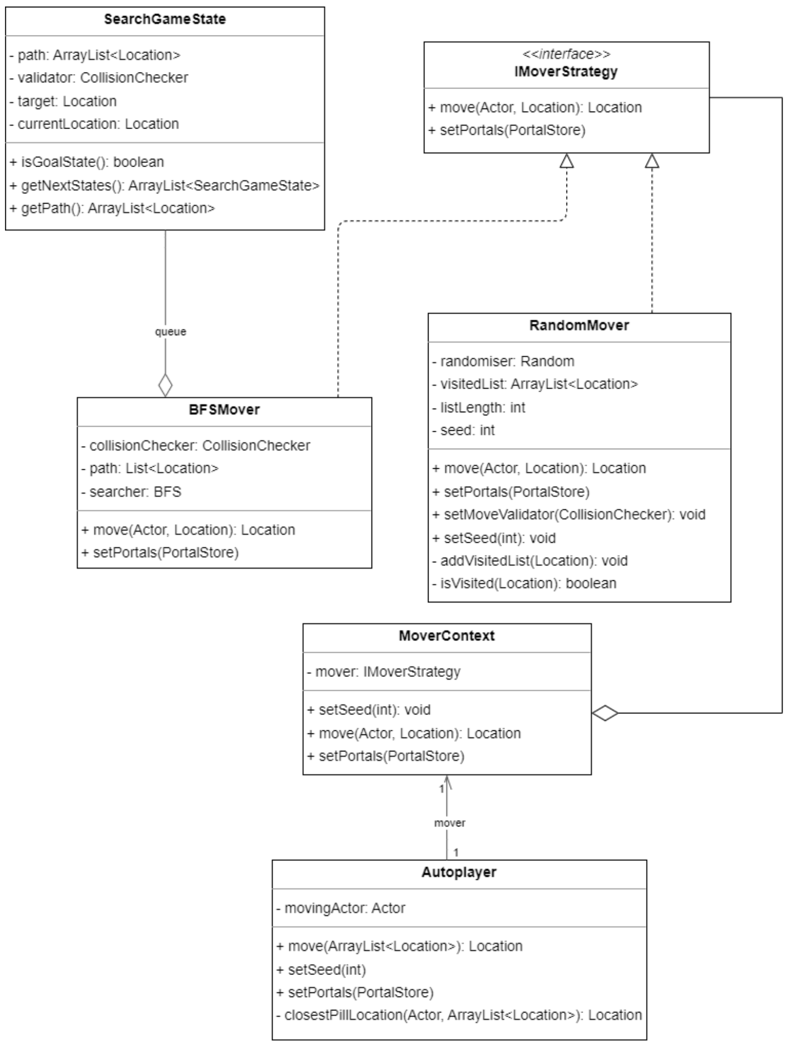
Autoplayer

The design of the autoplayer for PacMan in the TorusVerse involves several classes that work together to allow the autoplayer to navigate the game map and interact with the items or monsters accordingly. These classes include BFSMover, SearchGameState, RandomMover, IMoverStrategy, MoverContext, and Autoplayer.

The Autoplayer class is responsible for controlling the movement of PacMan. It uses a MoverContext object to determine how PacMan should move. The MoverContext class can use 2 different movements (BFSMover and RandomMover). The BFSMover class uses a breadth-first search algorithm implemented by the BFS class to find the shortest path to the target. If no path is found, such as BFSMover being stuck in a dead-end, the RandomMover class is used to move PacMan at random until a path is found.

The design of the autoplayer utilizes the GoF Strategy pattern, which allows the algorithm for PacMan's movement to be determined during runtime. This allows the MoverContext to have the ability to switch between using a BFSMover and a RandomMover depending on the situation. By utilizing the strategy pattern, new automover approaches can easily be implemented without modifying too much of the existing classes. This easy extension also demonstrates the GRASP principle polymorphism.

In addition, the classes involved in the design of the autoplayer demonstrate high cohesion and low coupling. This is due to the fact that each class included in the design of autoplayer has a well-defined responsibility and interacts with other classes through well-defined interfaces. For example, all code for deciding where PacMan should move next is within implementations of IMoverStrategy, while MoverContext selects the appropriate mover strategy. The MoverContext also exhibits the Creator principle from GRASP, as it has the ability to create instances of mover strategies. Additionally, the MoverContext class uses Pure Fabrication and Indirection. This means that it creates objects that do not represent real-world entities but are used to facilitate interaction between other objects. MoverContext facilitates the interaction between Autoplayer and IMoverStrategy by abstracting the logic of managing movement strategies away from Autoplayer. The diagram below illustrates the design of autoplayer, and the use of Strategy pattern.


SearchGameState
- path: ArrayList<Location>
- validator: CollisionChecker
- target: Location
- currentLocation: Location
+ isGoalState(): boolean
+ getNextStates(): ArrayList<SearchGameState>
+ getPath(): ArrayList<Location>
<<interface>>
IMoverStrategy
+ move(Actor, Location): Location
+ setPortals(PortalStore)
queue
BFSMover
- collisionChecker: CollisionChecker
- path: List<Location>
- searcher: BFS
+ move(Actor, Location): Location
+ setPortals(PortalStore)
RandomMover
- randomiser: Random
- visitedList: ArrayList<Location>
- listLength: int
- seed: int
+ move(Actor, Location): Location
+ setPortals(PortalStore)
+ setMoveValidator(CollisionChecker): void
+ setSeed(int): void
- addVisitedList(Location): void
- isVisited(Location): boolean
MoverContext
- mover: IMoverStrategy
+ setSeed(int): void
+ move(Actor, Location): Location
+ setPortals(PortalStore)
1
mover
1
Autoplayer
- movingActor: Actor
+ move(ArrayList<Location>): Location
+ setSeed(int)
+ setPortals(PortalStore)
- closestPillLocation(Actor, ArrayList<Location>): Location

Extensions

The current implementation of autoplayer does not take into account that eating an ice cube freezes the monsters for a defined duration. If we want to handle this scenario, one way to do this is by creating another class to store the states of monsters (similar to PortalStore), and by having the Autoplayer class hold an instance variable of this class. This would allow it to provide monster states to implementations of IMoverStrategy through MoverContext. Another method to handle this would be by having an implementation of IMoverStrategy also hold monster states as an instance variable and have the algorithms take monster states into account when determining where PacMan should move next.

Assumptions

In our software project report, we have made several assumptions that guide our development process. Firstly, we assume that error logs for each level of the software will be recorded in the log.txt file. Moreover, in order to optimise the performance and avoid potential issues with Swing and AWT modules, we have chosen not to use a single thread for panel changes. Instead, we employ different threads to handle transitions between the view and test modes, as recommended by the GameGrid official documentation. We also do not store the map that we are currently editing if we load another map, the map will be replaced by the loaded map and will be lost forever if not previously saved. Lastly, our design model does not include the 2D-Map-Editor components that were unchanged.