SWEN30006 Report - Project 2 PacMan in the TorusVerse

fri-11:00 team 2

Part 1: Editor

To satisfy the requirement of creating a single application in the task, we started by merging the ‘mapeditor’ package in 2D-Map-Editor-master into Pacman. We have also added a new sub-package, 'mode', to implement the different modes of the system and modified the class 'Driver' correspondingly to control various modes and their interaction with other modules based on the Controller principle of GRASP to decrease coupling. The 'Mode' class in this sub-package is an interface. According to existing requirements, there are currently three subclasses, 'EmptyEditMode', 'MapEditMode,' and 'TestMode', to implement the interface class ‘Mode’ to match the rules of different modes. If new test rules need to be added later, the system can be easily extended by adding subclasses in package ‘mode’, which fits the low coupling principle in GRASP.

During initialization, the game must accept arguments from the command line. How to handle input independent from the game domain becomes a significant task. According to the information expert principle, the Driver class can be fully responsible for receiving inputs. However, implementing in such a way will result in low cohesion as the Driver will not only have responsibility for starting the game but also for handling input, which results in loose integration. To solve this, we created a facade class called ArgsHandler, responsible for handling command line argument inputs and returning what mode the editor should start. Through indirection, compared to letting the Driver handle input, the handler class made the Driver class more cohesive and effortless for expansion and management. Apart from employing indirection and facade, the ArgsHandler class uses the singleton pattern. A singleton class can ensure only one instance is initialized, as having multiple ArgsHandler is pointless in our design.

Before starting the game, we employed composite and strategy patterns in GoF to do the level check and game check of the game. In the sub-package 'checker', two strategy pattern classes, 'GameCheckerStrategy' and 'LevelCheckStrategy' each encapsulate the check items that need to be performed before the game starts, class

'SingleNameChecker' and 'DuplicateChecker' for 'GameCheckerStrategy'; 'PairPortalChecker,' ''SinglePacmanChecker' and 'TwoGoldPillChecker' for 'LevelCheckStrategy'. The different algorithms implemented are written in independent strategy subclasses, which reduces the dependencies between classes and meets the low coupling principle in GRASP, making the system more flexible, scalable, and easy to maintain. And if a game folder needs to be loaded, we use the composite pattern to deal with the combination problem, 'CompositeGameChecker' for 'GameCheckerStrategy', and 'CompositeLevelChecker' for 'LevelCheckerStrategy'. In this way, there is no need to distinguish whether a game folder or a single map needs to be loaded. Unified processing gives the combination structure high cohesion, and the internal combination makes each small function independent, which also conforms to the low coupling principle.

Finally, according to the information expert principle of GRASP, we create an independent class, 'EditLogger' to process the log file that needs to be reported if the check fails. Notably, the EditLogger class is implemented with the singleton pattern. With the singleton pattern, classes can have single, global access to the Editlogger class with low complexity, as having multiple loggers is unnecessary and might also raise problems of having race conditions with numerous loggers trying to write the same file.

After passing the check, we load the map file through the package 'file'. The file package is designed to include classes related to file operations. We created a ‘Loader’ facade class to process file readings in our implementation. Though information experts principle, classes like ‘Driver’ should be responsible for file handling as they have enough information. However, doing so will decrease cohesion and violate the single responsibility principle. By creating a facade Loader class, classes that need to read a map, namely the driver and game, can be more cohesive, less coupled, and more extendable in the future. For example, if file read logic needs to be changed in the future, only the Loader class needs to be changed instead of modifying all classes that need to read files.

Furthermore, regarding the editor GUI, our team utilized the existing editor ‘Controller’ class to facilitate the controller GRASP principle. That is, instead of linking multiple individual classes from the 2D-Map-Editor code base to the original

Pacman code base, only the controller class is connected. Moreover, the controller can act as an intermediate class between the GUI and the game-playing domain, which will benefit future extensions and maintenance by minimizing coupling. By acting as an intermediate object, the controller class also uses the indirection principle to avoid direct coupling, making the game domain and the GUI domain more cohesive.

Figure1. The way to call GUI

Part 2: New Portals

Project 2 adds a new rule, 'portal,' that Project 1 didn’t have. Two portals are a pair. When a Pacman or a monster falls into a portal, it will appear from another portal corresponding to this portal.

To implement portals, we add a new package, 'tiles/portal', with the classes 'Portal' and 'PortalComposite' according to the composite pattern to handle more than one pair of portals that may appear in the game. Since different pairs of portals are distinguished by color, we created an enumeration class,' PortalColor', to set the portal color so that portal colors can be determined without needing more details for further extensions, which fits the Pure Fabrication and Information expert principle of GRASP.

By having a composite class, portals in the class can be accessed by a common interface of operations, which allows the game to treat all portals as a single portal object, leading to a more extendable design.

Part 3: Autoplayer

To implement the autoplayer, we first focus on enhancing the original design of the monsters. In the initial design, the game only supports having one monster for each type. However, such a design could be more scalable and easier to extend. Considering these constraints and meeting the requirements of handling multiple monsters in the future, our team applied the composite strategy. Through the application of the composite strategy, all monsters in the game can be treated as a whole by using common interfaces; therefore Pacman can easily know where monsters are if needed. With Pacman having monsters' locations in hand, the autoplayer can easily be extended to be smarter to handle the monsters in the future.

Moreover, we also created a package 'Autoplay', which contains the strategy pattern to provide an extendable autoplayer design with the potential for further improvement. There is an interface class, 'PacAutoPlayStrategy', with the current requirement class 'NoMonsterStrategy' to implement the scene of the Pacman auto test with no monster. Here, we follow the walking approach of Pacman we modified in Project 1 to implement it and find and eat each gold/pill. In the previous implementation (project 1), the path-finding algorithm used by Pacman was not only unintelligent but also outdated: it will not be able to find routes when portals are introduced and have tiles enclosed in walls. With the aim of enabling the Pacman to reach all pills and gold on the map, we empowered the Pacman with graph searching algorithms so that it can eat all pills automatically, even if they are only accessible via portals. We also have a possible extension class, 'HasMonsterStrategy' for further expanding the implementation of the interface. Adding more strategies to implement the PacAutoPlayStrategy interface will be fine for further possible extensions. Moreover, future developers also can use the composite strategy to combine multiple strategies into one. In addition, our implementation also allows software professionals to use patterns like decorator if

they want to reuse some of the existing strategies. For example, if the game design team decides that Pacman will eat ice after eating pills and gold, they can reuse the current NoMonsterStrategy via the decorator pattern.

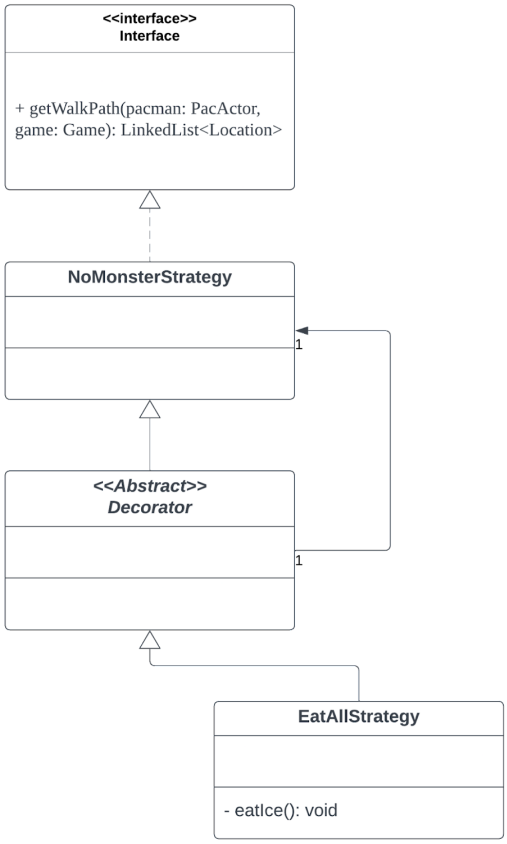

Figure 2. Possible extension for autoplayer

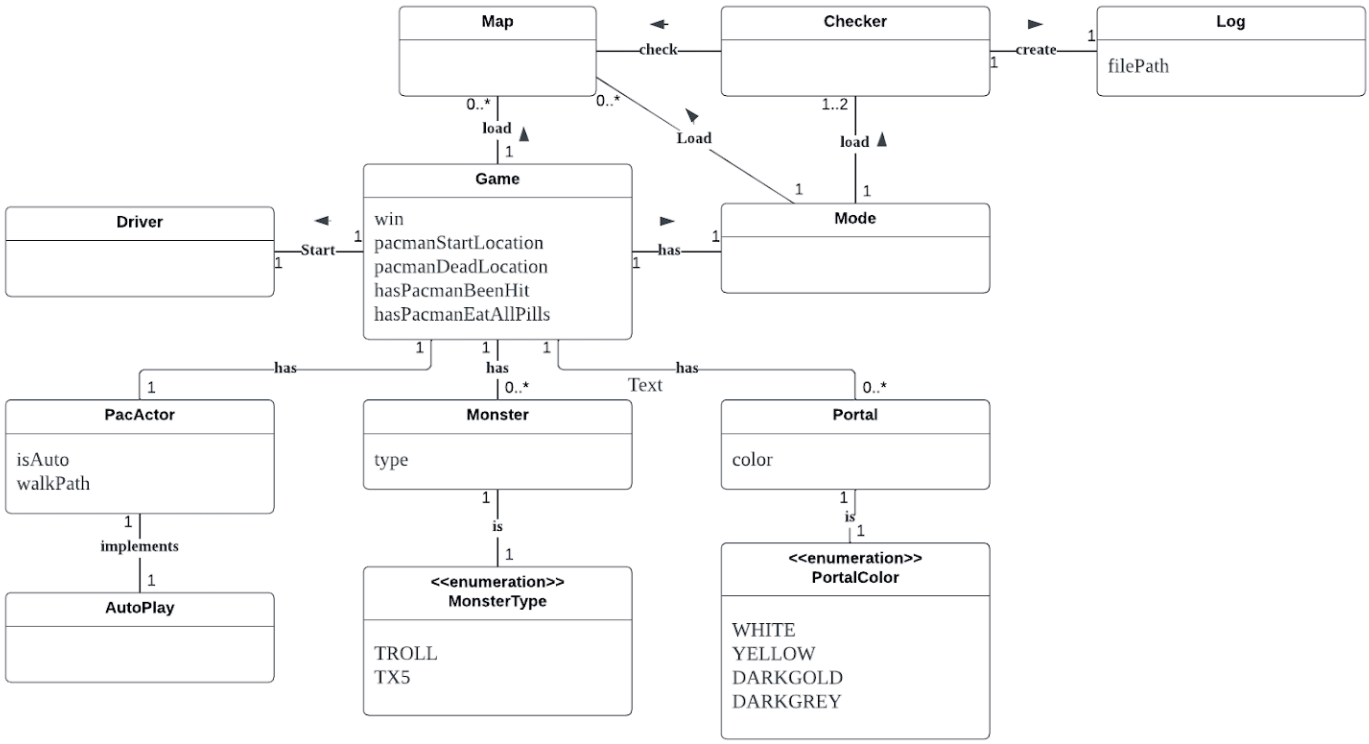

Figure 3. Domain class diagram for capturing the covering of the domain concepts relating to the autoplayer and game levels/maps for PacMan in the TorusVerse.

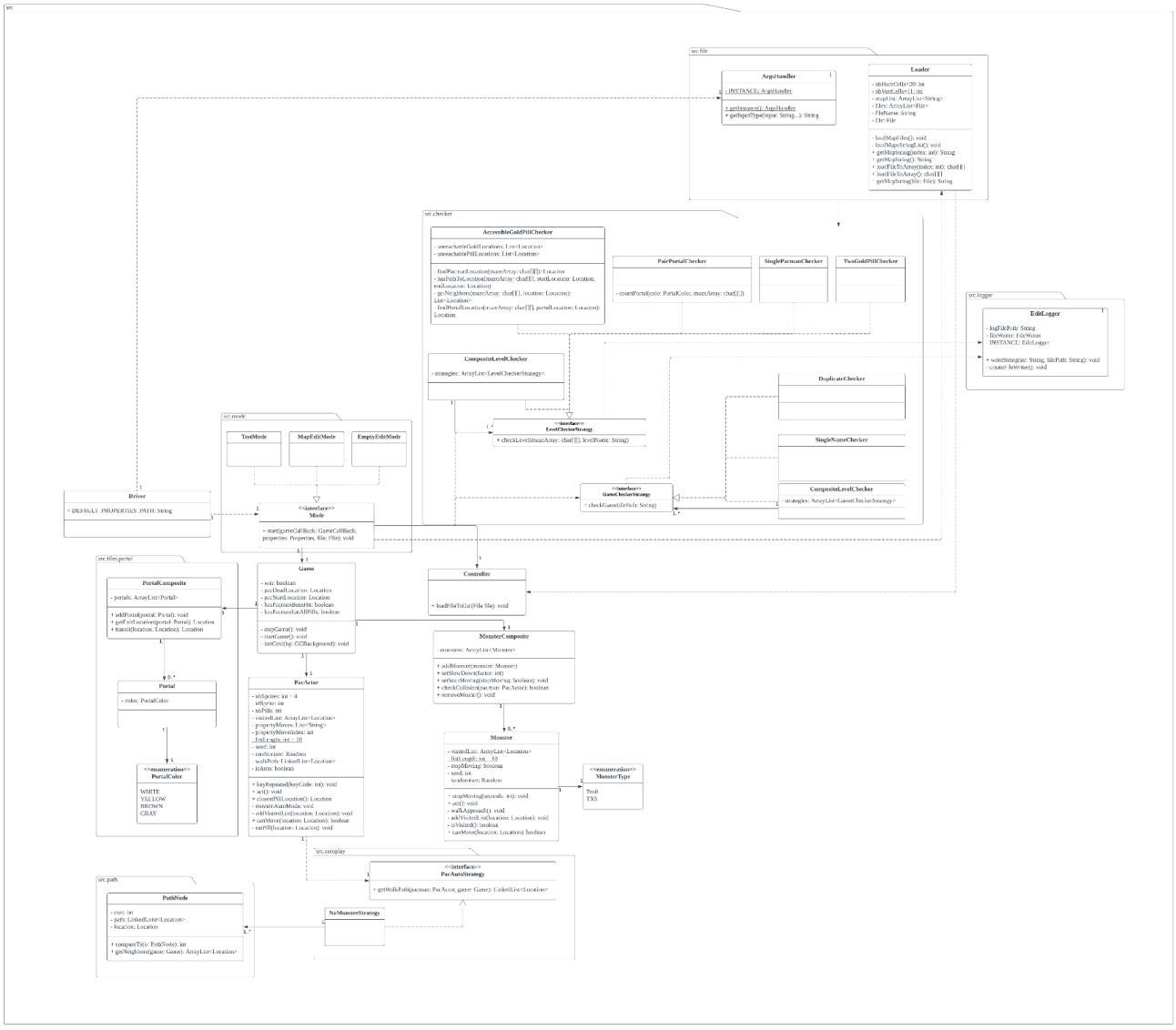

Figure 4. Design class diagram for documenting your new design for PacMan in the Multiverse, including the extensions (some details of the old design are hidden).