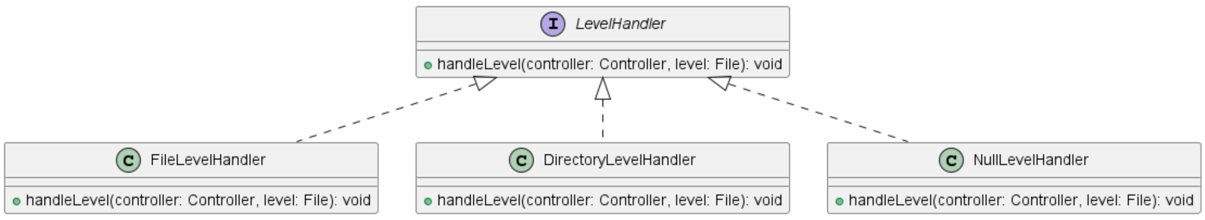
SWEN30006 Project 2: PacMan in the TorusVerse

Amritesh Dasgupta, Ruitong Wang, Vincent Luu
fri11-00-team-07

1 Design of Editor

The editor follows closely to what was provided by *Daniel "MaTachi" Jonsson* in the skeleton 2D Map Editor. The changes in design were implemented primarily in the *Controller* class, as it was the main point of contact from the *Driver*. The Driver creates a Controller object depending on the arguments passed in, as per the systems requirement. As there are 2 types of *File* objects that can be passed in to the argument, a folder or a file, a Handler interface was implemented for each case.


LevelHandler
handleLevel(controller: Controller, level: File): void
FileLevelHandler
handleLevel(controller: Controller, level: File): void
DirectoryLevelHandler
handleLevel(controller: Controller, level: File): void
NullLevelHandler
handleLevel(controller: Controller, level: File): void

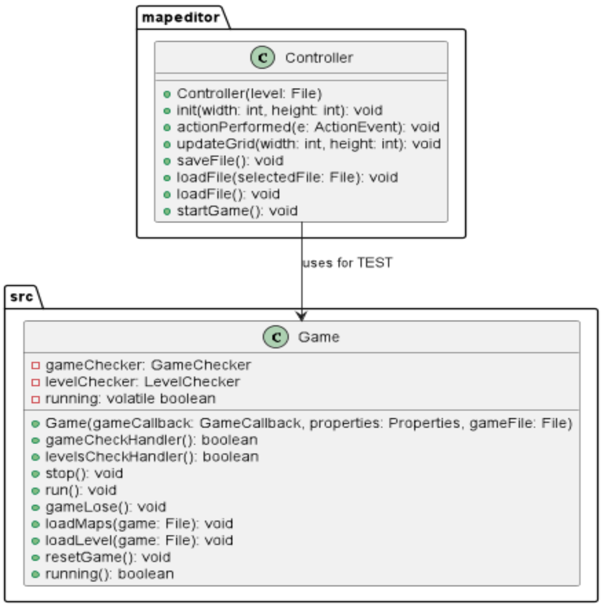
Creating the interface depicted above improves the extensibility and maintainability of the system. The Single Responsiblity Principle is used as the responsibility of handling different file types is delegated to different classes (FolderHandler and FileHandler), promoting low-coupling. Open-Closed Principle is also followed since new argument types can be introduced without needing modifying the existing codebase, ensuring extensibility.

In order to connect the controller to the *Game* class, while maintaining both the editor View and the game view, threading was required. The main thread (EDIT) needs to continue running once the pacgame thread (TEST) has finished running, regardless of whether the program started in TEST mode or EDIT mode. The threading is incorporated with *LevelHandler* as a directory argument passed in initializes the program into TEST mode, without needing to create the editor. Lazy creation has been used here in order to reduce overhead time, particularly the startup of the program by leaving the unnecessary creation of *View* object till later, when it is actually required.

The *View* class has also been slightly changed to allow the hiding and reopening of the same view object. The reason behind this is to ensure while other threads (testing game) take priority, the editor view can be hidden and reopened, without having to create another object. The two classes, *Game* and *Controller* also have their own variables indicating the current file being used by them. By having these variables, the program is able to share information between the two classes to ensure the user can start a game with any file and go back to editor from any game level. The relationship between them is unidirectional, however, as the Game class is not dependent on the editor but the editor uses attributes and behaviours provided by the Game class. This design increases the cohesion of both these classes as well as reduces the coupling between the two.

2 Design of Tester

The tester view is similar to that created for SWEN30006 Project 1, with the majority of the code remaining untouched. The main changes were the refactoring of the Game class, which was the point of interaction for the Editor application. The relationship between the editor and tested is illustrated in the diagram below:


mapeditor
Controller
Controller(level: File)
init(width: int, height: int): void
actionPerformed(e: ActionEvent): void
updateGrid(width: int, height: int): void
saveFile(): void
loadFile(selectedFile: File): void
loadFile(): void
startGame(): void
uses for TEST
src
Game
gameChecker: GameChecker
levelChecker: LevelChecker
running: volatile boolean
Game(gameCallback: GameCallback, properties: Properties, gameFile: File)
gameCheckHandler(): boolean
levelsCheckHandler(): boolean
stop(): void
run(): void
gameLose(): void
loadMaps(game: File): void
loadLevel(game: File): void
resetGame(): void
running(): boolean

The figure above is an abstracted relationship between the Tester and Editor. The Game instantiation requires 3 parameter inputs: GameCallback (for logging), Properties and File. Emphasizing on the File input, since the Editor only calls it with a folder or level the Game will always require a map directory. The level/s are added to an ArrayList of Files, which contain levels. Having this ArrayList ensures the *run()* function is abstracted from the File structure passed in for the maps itself. This abstraction is provided by *loadLevel(File)*, which adds to the ArrayList accordingly.

Run() is used to and calls *gameCheckHandler()* and *levelCheckHandler()*, which are used for Game checks and level checks respectively. Depending on the outputs of the checks, the variable *curFile* variable is updated and used in Editor as mentioned before. The GameCheck and LevelCheck implementations are described further in 4.

As previously stated, In order for both the editor and tester to work simultaneously, threading was required. A volatile variable, *running*, was used to perform this, as it ensures all threads are up-to-date with the status of this variable. The *stop()* method indicates that the game is over (either by victory or loss), and correspondingly changes the value of the variable. This allows for the editor to take over and continue the program as it reads *running* to be false, indicating the program is in EDIT mode.

3 Design of Portals

Following the creator principle, the application assigns the responsibility of creating portals to the PacManGameGrid (PMGG) class. This is because PMGG contain information about the portals and is the only class that uses them directly. Likewise, the information expert principle is used when dealing with teleportation. Since PMGG has the relevant information; the locations of actors and the locations of portal pairs, the responsibility of teleportation is assigned to PMGG. While the application of these two GRASP principles in this way slightly lowers the cohesion of PMGG, it greatly reduces the overall coupling of the application and is therefore a fair trade-off.

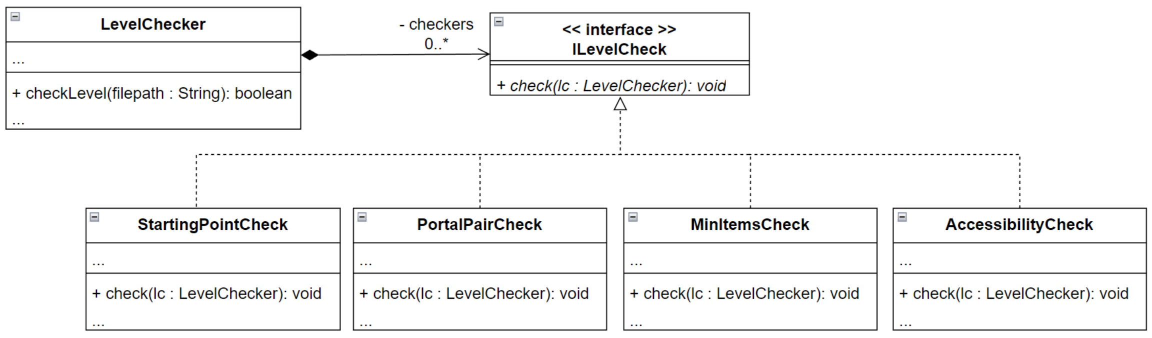
4 Design of Checkers

The application requires the capability to check the validity of levels as well as the validity of a game folder according to defined sets of rules. In order to do this in a cohesive and extensible way, the composite design pattern was used for both the level checking and game checking. The level and game checks each use a single composite object.
The current level checks require the fulfilment of the following rules:

- Exactly one starting point for PacMan.
- Exactly two tiles for each portal appearing on the map.
- At least two Gold and Pill in total.
- Each Gold and Pill is accessible to PacMan from the starting point, ignoring monsters but accounting for valid portals.

Since these rules are clearly distinct and the specifications requires easy modifications and/or extension, the composite design pattern is a perfect fit for this situation. The composite pattern allows the application to easily perform a defined sequence of checks. Additionally, its checks can be easily modified due to the modularization of the rules into individual classes.

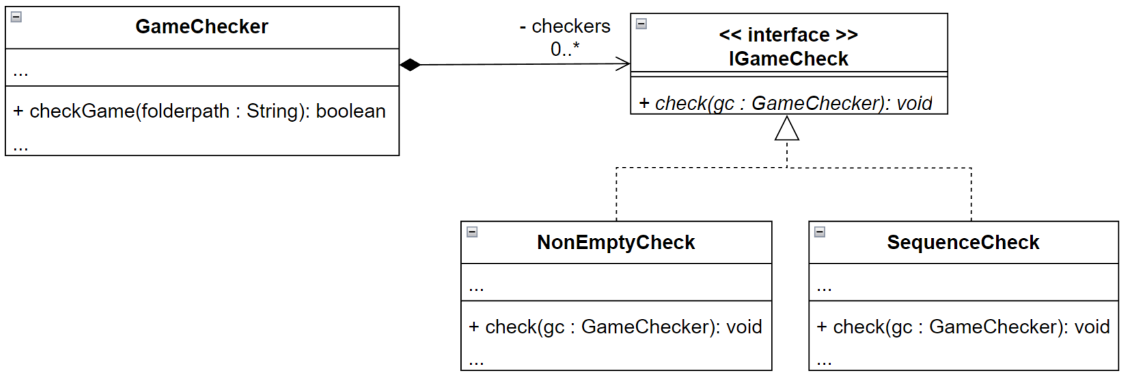
Furthermore, the specific checks done (i.e. what rules are applied) can easily be changed due to the lack of coupling afforded by the design pattern and the polymorphism in the form of a shared interface. In other words, existing checks can be modified or removed and new checks can be implemented in a simple and cohesive way. Doing so, also follows the open/closed principle. Below is a partial static design model of the composite pattern of levelCheck:



The current game checks require the fulfilment of the following rules:

- At least one correctly named map file in the folder:
  - Each map file has the extension ".xml" and is stored in xml format.
  - Each map file name starts with a number.
- The sequence of map files well-defined:
  - No two maps can have the same number.
  - The numbers do not need to be contiguous.

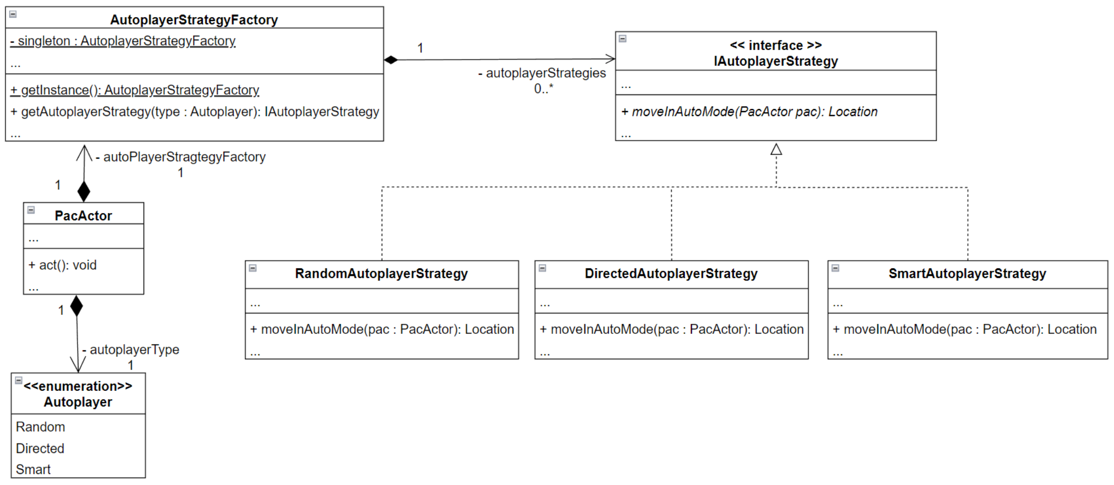
The game checking code also uses the composite design pattern for the same reasons as for the level checking. Below is a partial static design model of the composite pattern of gameCheck:



GameChecker
...
+ checkGame(folderpath : String): boolean
...
- checkers
0..*
<< interface >>
IGameCheck
+ check(gc : GameChecker): void
NonEmptyCheck
...
+ check(gc : GameChecker): void
...
SequenceCheck
...
+ check(gc : GameChecker): void
...

5 Design of Auto Player

The implementation of the auto player involves the strategy and factory pattern as a cohesive way to define the behaviour of the auto player. Below is a partial static design model of auto player:



AutoplayerStrategyFactory
- singleton : AutoplayerStrategyFactory
...
+ getInstance(): AutoplayerStrategyFactory
+ getAutoplayerStrategy(type : Autoplayer): IAutoplayerStrategy
...
1
- autoplayerStrategies
0..*
<< interface >>
IAutoplayerStrategy
...
+ moveInAutoMode(PacActor pac): Location
...
- autoPlayerStragtegyFactory
1
1
PacActor
...
+ act(): void
...
1
- autoplayerType
1
<<enumeration>>
Autoplayer
Random
Directed
Smart
RandomAutoplayerStrategy
...
+ moveInAutoMode(pac : PacActor): Location
...
DirectedAutoplayerStrategy
...
+ moveInAutoMode(pac : PacActor): Location
...
SmartAutoplayerStrategy
...
+ moveInAutoMode(pac : PacActor): Location
...

The implementation uses pure fabrication, fabricating the AutoplayerStrategyFactory class and IAutoplayerStrategy interface which were not in the domain representation. This allows for higher cohesion within PacActor. In addition, this factory class follows the principle of indirection, reducing coupling between PacActor and any auto player logic. Furthermore, the use of polymorphism to create a stable interface; the auto player strategies are protected from variation and can easily be modified.

Currently, the code only utilises the directedAutoplayerStrategy and uses it at all times. However, due to the use of the factory and strategy design patterns, the code could easily be extended to create a smarter auto player strategy. This new strategy would need to polymorphically implement the interface, IAutoplayerStrategy. Doing so would allow the strategy pattern to dynamically use the smarter algorithm.

Due to the interface and polymorphism, auto player strategies can obtain information about the game state through PacActor such as pill, gold and ice locations as well as monster locations. Using this information, a smarter auto player can come to an informed decision on its next best move using any of the various path finding algorithms and game playing agents.

Furthermore, due to the use of the strategy design pattern, it is possible for PacActor to dynamically switch between different strategies during runtime according to some game-play logic. For example, a more aggressive and risky strategy can be used when monsters are frozen, but a more defensive and passive strategy can be used when monsters are furious. The auto player is able to be designed to deliberately eat or not eat special items. This would result in a better and smarter auto player.

The factory itself implements the singleton design pattern to provide a single point of reference for the PacActor. Furthermore, the auto player strategies inside the factory are also references to singletons. Implementing the auto player in this way improves efficiency and reduces coupling.

As such, the strategy, factory and singleton patterns all combine to facilitate easy modification and extension, allowing for further improvement.