SWEN30006 Report – Project 2 PacMan in the TorusVerse

Part 1: Design of editor

In the process of developing PacMan in the TorusVerse, we added several new functionalities and enhanced the design of the existing editor code. These major changes are listed below:

1.1 Addition of Level and Game Checker classes

In accordance with the project specification, we added Game checking and Level checking logic to our game. Level checking occurs prior to testing a specific level, and Game checking occurs prior to testing a folder.
In the original version of the code, the Controller class implements all file processing code. Hence, one option was to implement Level checking and Game checking as two separate functions in the Controller class. However, such an approach would lead to poor cohesion of Controller, which would have to handle checking, on top of file processing and action listening.
Hence, instead of implementing both Game and Level checks in the *Controller*, we decided to delegate these checking implementations to new classes, which handle Game and Level checks respectively. Doing so will decouple this logic from *Controller*, ensuring that future modifications/additions to the checking logic will not require any changes to *Controller* (or any other class, for that matter), thereby enhancing the maintainability and extensibility of our code. Moreover, creating new checker classes will also reduce code bloat in *Controller*, thereby supporting code readability and cohesion. It is also worth noting that based on the requirements documented in the project specification, we have also logically documented Game/Level checking as their own domain classes (see *Fig. 1*).

1.2 Implementing Level Checker following the Singleton and Composite pattern

To preface, Level checking consists of checking a level against a set of rules (i.e., checks). Moreover, the project specification establishes that our design should allow checks to be easily modified or added. Hence, we chose to use the Composite Pattern to fulfil this requirement (see *Fig. 2*).
In our Level checking implementation, we included: an abstract *LevelCheckComponent* class, a *LevelCheckerComposite* class extending from *LevelCheckComponent*, and several *check* classes all extending from *LevelCheckComponent*, namely: *CheckA* (validates the number of PacMan), *CheckB* (validates the number of Portals), *CheckC* (validates the number of Gold and Pills combined), and *CheckD* (validates the accessibility of Gold and Pills). Furthermore, we chose to make *LevelCheckerComposite* a Singleton class, such that it contains a static instance of itself in addition to an ArrayList of *checks*.
To use our level-checking implementation: in the *Controller* class, we first obtain the globally accessible instance of *LevelCheckerComposite* through the *getInstance* method. At this stage, if the static object is not yet instantiated, the constructor will be called, which instantiates and adds all level checks to the ArrayList of *checks* stored in *LevelCheckerComposite* (see diagram note in *Fig. 2*). Then, we pass the file (*.xml*) and model (of type *Grid*) of the current level into the *checkLevel* method of the static *LevelCheckerComposite* object, which traverses all checks and returns an empty string if the level is valid, and a non-empty string if invalid (containing the specifics of errors, to be printed in the log file).

In addition to decoupling the level-checking logic from the Controller, this design provides several additional benefits.
Firstly, the use of the Singleton pattern in our Level Checker provides a single entry point to access the functionality of level checking, ensuring that the validation process is consistent for all game components. This is achieved by permitting only one instance of the Level Checker, and by adding all checks during the instantiation of the static object. Consequently, all game components requiring level checking will utilise the same set of level checks, thereby allowing for centralised control of the level checking process. Moreover, the use of Singleton here also allows the *Controller* class to delegate the responsibility of instantiating *LevelCheckerComposite*, since it is already completed within the *LevelCheckerComposite* class.
Secondly, the use of Composite for the Level checking implementation allows developers to easily modify and add checks. To modify checks, the developer only has to alter the class that implements the specific check (*CheckA*, *CheckB*, etc). As our design ensures that the logic of each check is implemented in its own class, not only are we able to create a more modular design and support cohesion, but we are also able to modify existing checks in an isolated manner, leaving other classes unaffected by our changes. To add checks, the developer may simply create a new *check* class extending from *LevelCheckComponent*, and add it to the ArrayList of checks during the instantiation of *LevelCheckerComposite* (i.e., within the *LevelCheckerComposite* constructor), thereby minimally altering the existing code.
It is also worth noting that this design supports GRASP Creator since we are delegating the creation and management of *checks* to *LevelCheckerComposite* - which is the class that uses *checks* most closely. Since other classes do not need to create, maintain, or interact with *checks*, this reduces unnecessary code dependencies across other classes, therefore also supporting low coupling.
Hence, our current implementation of Level checking allows our code to be more maintainable and extensible.

1.3 Implementing Game Checker following the Singleton pattern

For Game checking, we created a new class called *GameChecker*, which follows the Singleton pattern. Since game checking purely validates the input argument, we decided that we only needed one globally accessible instance of *GameChecker*. In addition to enhancing code cohesion (due to checking logic being delegated to an independent class), this design incorporating Singleton provides a convenient way to access the functionality of game checking, and also allows the *Controller* class to delegate the responsibility of instantiating *GameChecker*, since it is already completed by *GameChecker* itself.
To use our game checking implementation: whenever a folder is passed into the editor, pass the folder (of type *File*) to the *checkGame* method, which is found in the globally accessible instance of *GameChecker*. If the folder passes Game checking, the *checkGame* method will return an ArrayList of *.xml* files extracted from the folder; otherwise, the *checkGame* method will simply return *null*.
It should also be added that, since it is not mentioned in the project specification, we assume that all future versions of this game will use the same two game checks to assess a folder (i.e., checking a folder is non-empty and checking a folder contains correctly-named *.xml* files). Hence, we chose to implement all game checks together in the *checkGame* method, rather than follow the Composite pattern (as seen with Level Checker).

1.4 Façade for *File* processing

For the file processing responsibilities including file selection, loading and saving of XML files, and the additional I/O responsibilities including error logging, selecting files based on file names and processing of folders, we have created a class *FileHandler* (*see Fig. 3*). The *FileHandler* class simplified the interface to these complicated file operations from the *Controller* class and the other parts of the game system, which resonates with the Facade pattern. The Facade pattern that is being applied here also eases possible future extensions by localising the file operations, for example, to deal with other file formats. Therefore, the *FileHandler* makes the game design more maintainable and scalable, and creates a clean and simplified interaction for the rest of the game system.

1.5 Updated design of Pacman Game

While unrelated to the design of the *editor/tester*, we deemed it necessary to also outline some changes we made to the provided Pacman game code (i.e., code found in the *pacman > src* directory), as these changes form a crucial component of the new Pacman in the TorusVerse game (*see Fig. 4 for design model of updated Pacman game*).

From a design perspective, we made the following changes to the provided code:

1.5.1 Replacement of provided code with Project 1 code

We chose to replace all of the provided Pacman game code (i.e., Pacman simple version) with our finished code from Project 1. This included the addition of an Item class, an ItemType enum, Troll and TX5 subclasses (both extending from the Monster superclass, which we made abstract), an abstract MoveableActor class (which acts as the parent class for both Monster and PacActor), and separation of the bloated *Game* class into *GameEngine* and *Game* classes. As previously mentioned in the Project 1 report, these alterations will significantly enhance our game design and improve code extensibility.

1.5.2 Addition of *Monster* and *Item* Singleton Factory

In our old design, the responsibility of creating all game objects was designated to the same class that runs the game (i.e., *GameEngine*). We acknowledge that this approach may be problematic, as not only does this bloat this class with unrelated responsibilities in addition to game management, it also makes it hard to introduce new characters or modify the character instantiation processes without directly modifying *GameEngine*, thereby reducing the extensibility and flexibility of our code.
To further enhance the scalability and cohesion of the Pacman game design, we chose to create two Singleton Factories, which creates different monster subtypes and different item subtypes, respectively (see *Fig. 5* and *Fig. 6*). The former is called *MonsterFactory*, which is called by *GameEngine*. Specifically, the specific subtype of *Monster* to be created is passed into the *createMonster* method (belonging to *MonsterFactory*). We are then returned with a newly created instance of that specific *Monster*. The latter Singleton Factory is called *ItemFactory*, which is also called by *GameEngine*. The specific type of *Item* (including pills, gold pieces, ice cubes, white portals, yellow portals, dark gold portals and dark gray portals) to be created is passed into the *createItem* method (belonging to *ItemFactory*), which returns a newly created instance of that specific Item.

By using Singleton factories to instantiate these objects, we are able to guarantee effective use of resources (since multiple instances of a factory is unnecessary). In addition, by utilising the Factory pattern, we are able to delegate the responsibility of instantiation to a more suitable class and decouple GameEngine from specific object implementations, thereby enhancing code modularity and maintainability.

Furthermore, from a functionality perspective, we included several new components to our Game as according to the project specifications. They are the following:

1.5.3 Merging editor with PacMan game

To merge the editor with the PacMan game, we decided to instantiate a *Controller* object in *GameGrid*. The *Controller* object will be used by *GameGrid* to check for the existence of queued maps (if the player wins the current level), and to load the *Grid* of the current game level. In addition, the *Controller* object is also used by *GameEngine* to edit the current level *Grid*.
Note that under new Pacman game specifications, we no longer need the *PacManGameGrid* class to draw the levels, since map details (such as walls and entity locations) are read in from an *.xml* file by our editor, instead of hardcoded as a string constant. Hence, *PacManGameGrid* was removed from our new design.

1.5.4 Addition of Portals

We also added portals through the addition of the *PortalPair* class (see *Fig. 4*). Each *PortalPair* object represents an instance of a specific portal type (i.e., white, yellow, dark gray, and dark gold portal types), and contains an ArrayList of *Item*s. If a valid map is passed into the editor, each PortalPair object should consist of exactly two or zero portal *Item*s of the same colour.
While integrating portals into our current design, we had the option of either using the existing *Item* class to represent each portal tile or creating a new class that contains all information regarding a set of portals (i.e., a *PortalPair* class). We concluded that the former option would lead to many design problems. This is because valid portals usually come in pairs (or do not exist at all), therefore instantiating each portal tile as an *Item* in our game code will force the *Game* class to become responsible for implementing portal transportation logic. Hence, changing the portal transportation logic will also require changes to the *Game* class, leading to high coupling between the Game and Portal components. Therefore, we chose to create a *PortalPair* class that encapsulates information about all portals of a specific type and implements portal transportation logic. This reduces code coupling and creates more maintainable and reusable code.

1.5.5 Addition of a smarter autoplayer

Last but not least, we also added newer and smarter abilities for the autoplayer, which is employed when the isAuto property is set to true in the properties file. This feature is described in greater detail in *Part 2* of this report.

Part 2: Design of autoplayer

2.1 Implementation of current requirements

To preface, we will first introduce our autoplayer design, which takes into account portals, and eats pills and gold pieces in a directed way.
Since the project specification requires that the implemented autoplayer should have potential for further improvement, we chose to employ the Strategy pattern to support the autoplayer's movement logic (*see Fig. 7*). Specifically, our current design of the autoplayer features an interface called *MoveStrategy*, featuring only one function declaration: *move*. Implementing this interface is the *DirectedApproach* concrete class, which contains the current implementation for PacMan to move in a specific manner (i.e., deals with gold pieces and pills, but not Monsters and ice cubes). *PacActor* will contain an attribute of type MoveStrategy, which acts as a reference to a *DirectedApproach* object.
It should also be added that this particular design supports several GRASP principles. Firstly, it supports high cohesion, since it encapsulates the Pacman automove implementation in an isolated class, and does not need to be implemented or maintained by PacActor. This allows PacActor to maintain its well-defined responsibilities. Secondly, it supports low coupling, since PacActor is able to utilise the autoplayer without having to be concerned with specific algorithms and implementations, which promotes the interchangeability of autoplayer move strategies. It is also worthy to note that this design also supports the GRASP Polymorphism principle, since PacActor perceives different *MoveStrategy* implementations through the same interface, and allows developers to switch PacActor move strategies without having to alter its existing codebase.
Hence, the current design has the potential to further improve the autoplayer, in a clean and extensible manner, without affecting other classes.

2.2 Modification to work with monsters and ice cubes

To modify the autoplayer to have further improvement, future developers of this game may choose to create a (smarter) concrete class that implements *MoveStrategy*. Specifically, developers can add relevant code to take into account monsters and ice cubes, and may also choose to implement a smarter path-finding algorithm. With our current design of the autoplayer components, this can be achieved whilst minimally affecting pre-existing components.
With our current function declaration in *MoveStrategy* (i.e., *move(Pacman pacman, Grid grid), see Fig. 7*), we can ensure that any concrete classes implementing *MoveStrategy* will have the capacity to take into ice cubes, since the locations of ice cubes - which are static - are all readily accessible from the *Grid* argument.
Contrastingly, the location of monsters will continuously update throughout the game. Hence, to take into account the location of *Monster*s, future developers may simply pass in the *Game* object as an argument into the constructor of the concrete class implementing *MoveStrategy* (e.g., the constructor of our current concrete implementation, *DirectedApproach*, has the method signature *public DirectedApproach(Game game), see Fig. 7*). From here, developers can simply call appropriate getter methods in the *Game* object to obtain up-to-date monster locations.
Assuming that eating an ice cube freezes monsters for a specific length of time, it may be beneficial for future implementations of *MoveStrategy* to prioritise eating ice cubes, since it gives Pacman some time to safely access gold pills and ice cubes. Additionally, future implementations may also continuously measure the distance between Pacman and Monsters, changing Pacman's path if in close proximity with Monsters. However, the details of this feature are wholly up to future developers.
Hence, the current design of the autoplayer will allow it to seamlessly obtain greater functionality in future versions.

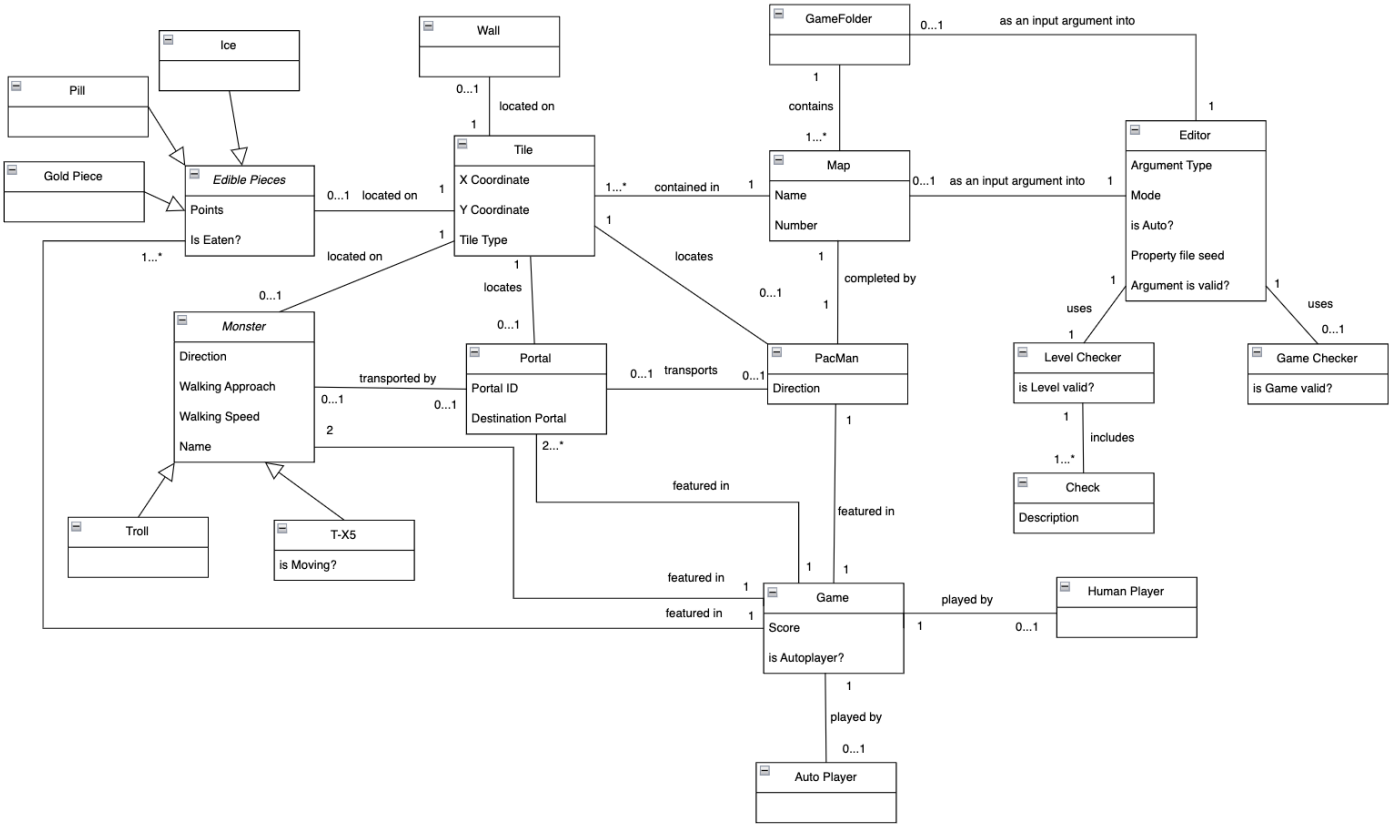

Figure 1: Domain Model describing PacMan in the TorusVerse Game based on provided application requirements

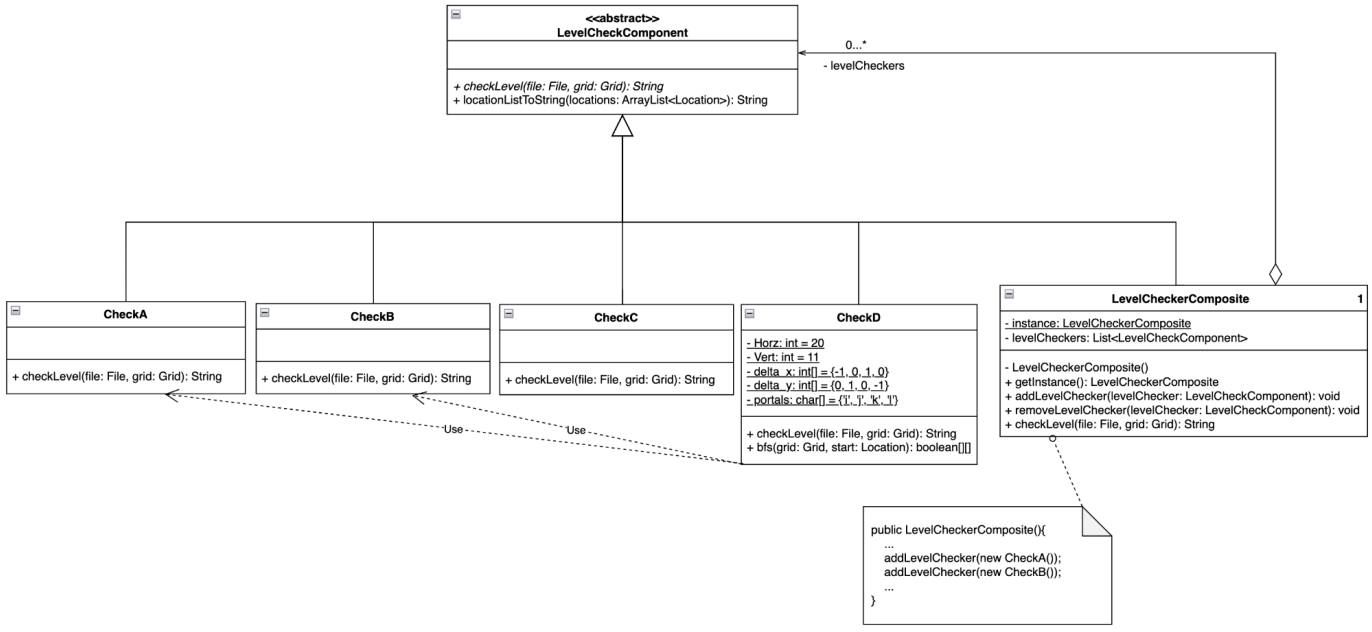

Figure 2: Partial Design Model demonstrating use of Composite Pattern for Level Checking

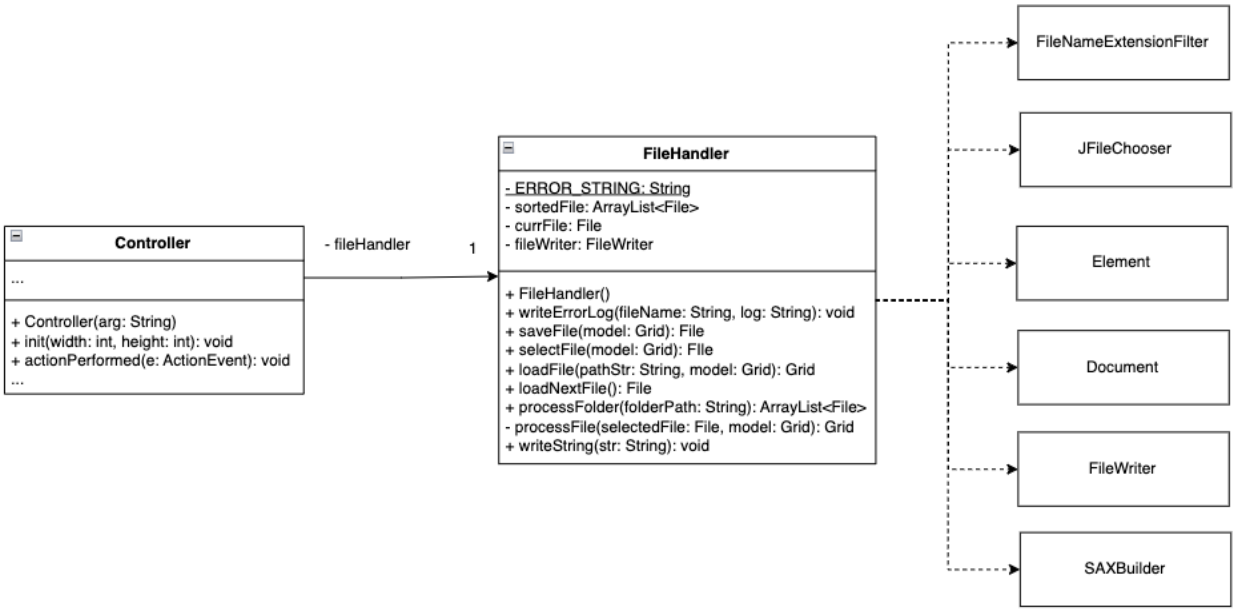

Figure 3: Partial Design Model demonstrating use of Facade pattern for File handling

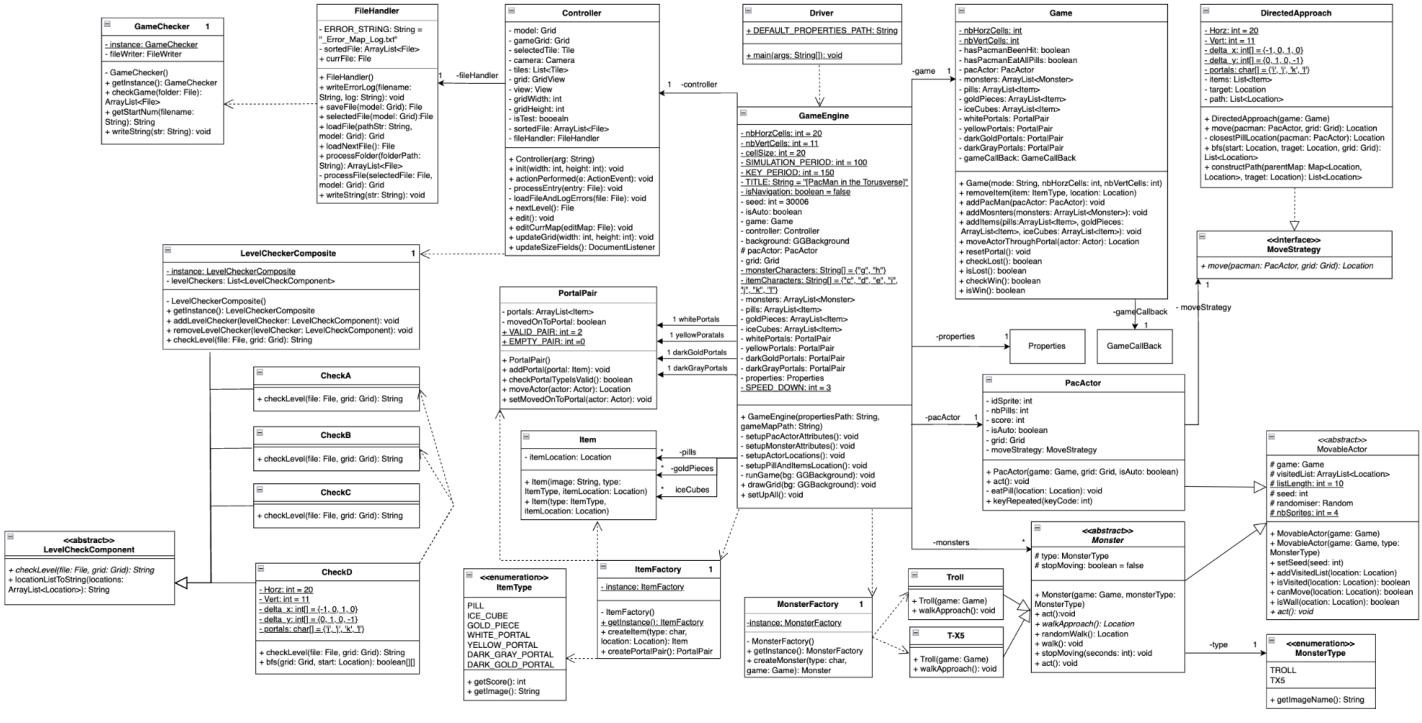

Figure 4: Partial Design Model of PacMan in the TorusVerse, which showcases newly added components such as the autoplayer, portals, editor, and its interactions with our pre-existing design

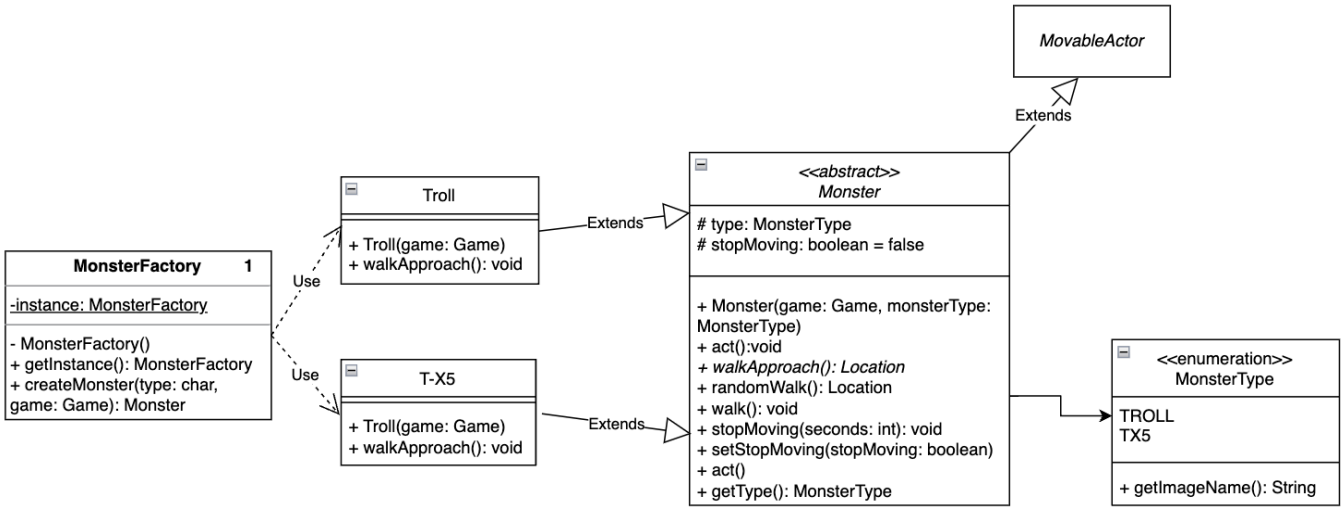

Figure 5: Partial Design Model demonstrating use of Singleton Factory for Monster Creation

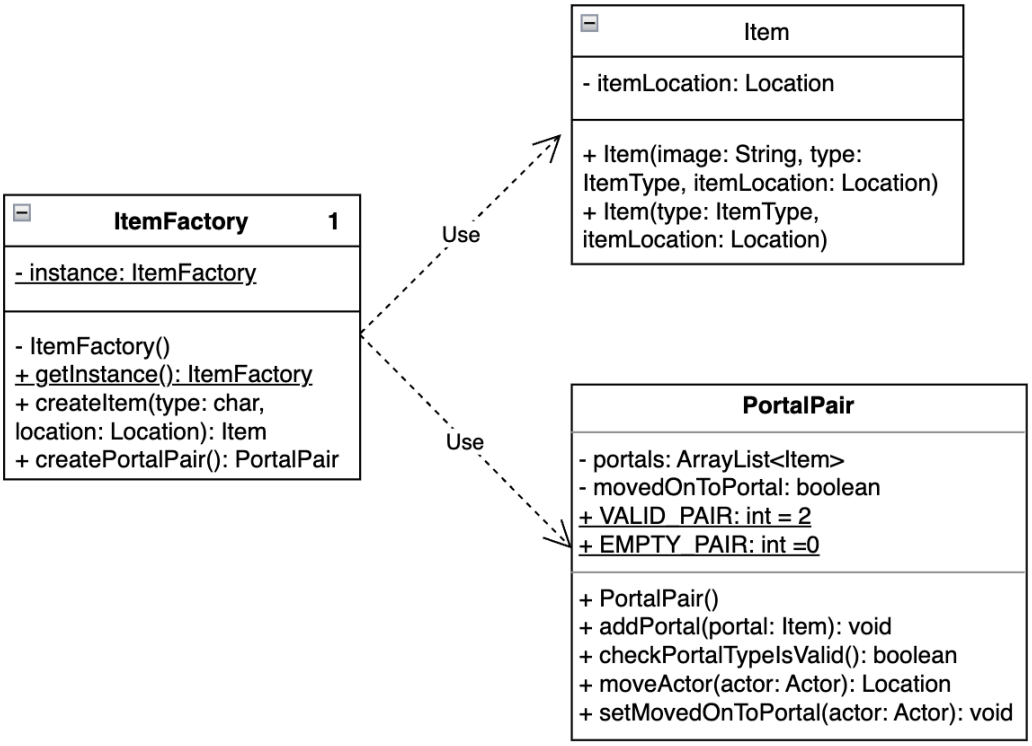

Figure 6: Partial Design Model demonstrating use of Singleton Factory for Item Creation

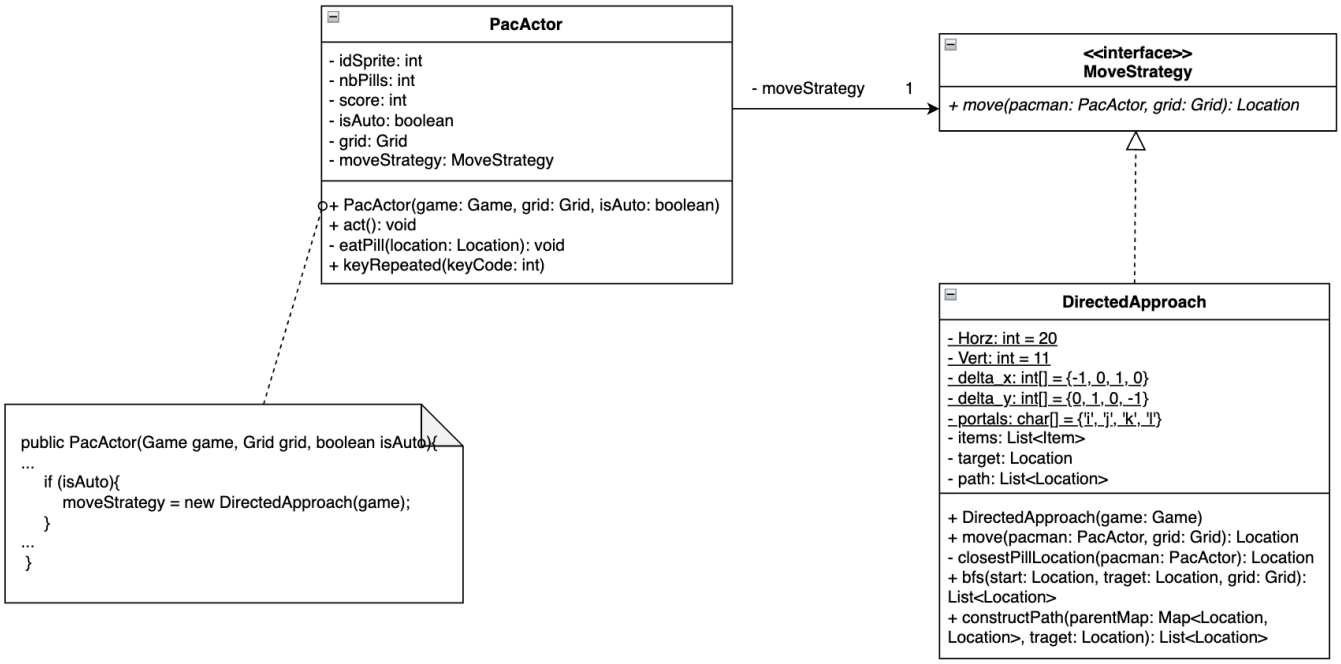

Figure 7: Partial Design Model demonstrating use of Strategy Pattern for PacMan AutoMove