SWEN30006
Project 2 Report

[Tue 09:00] TEAM 03

Erick Wong (1173104)

Dillon Han Ren Wong (1236449)

Jonathan Linardi (1272545)

Introduction

The aim of this project was to design and develop an editor and test app for PacMan in the TorusVerse. This new app required the addition of new capabilities, namely the ability for the editor to check or test levels and games, and a smart autoplayer which is able to find and consume each **Gold**/**Pill** in turn. Previously, work had been done to implement an earlier version of the game: PacMan in the Multiverse. This involved refactoring a base PacMan game whilst upholding GRASP principles. Given that the same base PacMan game was provided as part of the starting code base for PacMan in the TorusVerse, we decided to reuse our previous refactored version of PacMan. This was the best course of action because the refactored PacMan code would be easily extensible for the purposes of meeting the editor app specifications, compared to the base PacMan implementation.

To recapitulate, the most notable change from the base implementation was the abstraction of **Monsters** and **Items** into their own classes, extended by subclasses to define various concrete classes such as **TX5** and **Gold**. Other significant changes were the introduction of a **GameLoaderHandler** to handle the loading of **Games** by Pure Fabrication, as well as the delegation of responsibilities to classes that have task-relevant information by the Information Expert principle.

This report begins by describing and justifying the new design for the editor, followed by an exposition of the methods which may be used to extend the new autoplayer to work in the presence of **Monsters** and **Pills**. All changes are discussed with reference to the Static Design Model in Figure 2.

(P1) Design of Editor

Initially, the skeleton code for the editor had only the ability to load, edit, and save files. A brief overview of the requirements to be met are listed below. We will delve deeper into the specifics of each change in turn.

1. Changes on startup: the editor had to have the ability to start in different modes depending on command-line inputs (a file, a folder, or neither).
2. Level checking: the editor had to have the ability to check the validity of maps that were saved and loaded.
3. Game checking: the editor had to have the ability to perform checks on the validity of a given game folder.
4. Error Logging: whenever a check fails, the failure should be reported to a log file.

(P1.1) Changes on Startup

This requirement stipulated that starting the editor with a folder as an argument should cause it to start playing game levels within the folder (Test Mode), starting the editor with a map file should cause it to enter Edit mode on that map (Map Edit Mode), and starting the editor without any arguments should make the editor enter Edit mode with no map (Blank Edit Mode). In order to accomplish this task, we decided to employ the Strategy and Factory design patterns. Going in order, the Strategy Pattern is a behavioural pattern that allows developers to design a suite of

algorithms which could be used interchangeably at runtime depending on context. This pattern was chosen as the different inputs posed the problem of using different algorithms interchangeably depending on the runtime input.

We utilised a **StartingStrategy** interface which contains a *startEditor* method that takes in the command-line arguments upon the editor starting up. Said interface would be the backbone for the family of algorithms pertaining to dealing with different cases of arguments. From this **StartingStrategy** spawned 3 implementing classes, **EditStartingStrategy**, **FileStartingStrategy,** and **FolderStartingStrategy**, these classes were designed to deal with the cases where there are no inputs, a singular XML file input, and a folder input respectively.

With the use of this pattern, we allow for easy and cohesive future extensions relating to the behaviour of the editor app upon startup with new forms of command-line arguments. For instance, if the editor was ever required to automatically generate valid maps with the command-line input of a keyword “generate”, one would only need to implement a new **StartingStrategy** to achieve this. The Strategy Pattern also allows us to adhere to the Open-Closed Principle, as new strategies can be introduced without having to change any context.

The Strategy Pattern ties in with the next pattern we employed, the Concrete Factory Pattern, which is a creational design pattern that enables the production of a set of related objects without specifying their concrete classes. This pattern was combined with the Strategy Pattern in order to manage the creation of the aforementioned strategy objects, thereby forming the Strategy Factory Pattern. By encapsulating the logic of strategy selection, this allows for not only easier addition of future strategies, but also the instantiation of strategies whilst avoiding coupling with the **Driver** class.

In order to facilitate the Concrete Factory Pattern, the class **EditorStartingStrategyFactory** was employed. This class contains a method *getStartingStrategy,* which takes in an array of strings (the command line arguments), and returns the appropriate **StartingStrategy** depending on the cases described above. This is in line with the GRASP principles of Pure Fabrication and Indirection, as this class is not representative of a real-world concept, but is instead solely responsible for the creation of **StartingStrategies.**

The startup cases of no input and file input were simple, but the case of folder input had to be handled with more attention due to the complex additional responsibility of a game test which entails playing all levels in sequence. To avoid bloating the **FolderStartingStrategy** with incohesive responsibilities, we decided to integrate a Facade under the moniker of **GameTesterFacade** to manage the game testing. This simplified the **FolderStartingStrategy** by isolating it from the complexity of the subsystem used for running a game test.

As such, **GameTesterFacade** is another great example of applying the GRASP principles of Pure Fabrication and Indirection to allocate a particular set of responsibilities (game testing in this case) to a new class, to achieve higher cohesion and lower coupling amongst classes. Without the use of the Facade Pattern, the **FolderStartingStrategy** would have been highly coupled with other classes such as **Game**, as well as the classes required for game and level checking, which are described in the following section.

(P1.2) Level Checking

The next substantial application capability that we incorporated was level checking. These are a series of checks that are performed whenever a map is saved or loaded. Should a map fail a level check, the app should prohibit the map from being tested. For reference, Figure 3 displays a Dynamic Design Model detailing the various states and trigger events of the proposed editor system, including some assumptions of behaviour. Listed below are the criteria that a map must meet to be deemed valid for testing purposes.

a. There is only one starting point for Pacman.
b. Exactly two tiles exist for each **Portal** colour.
c. There are at least two **Gold** pieces and **Pills** in total.
d. Each **Gold** and **Pill** is accessible to Pacman from the starting point, ignoring **Monsters** but accounting for **Portals**.

To tackle this feature, we decided to apply the Composite design pattern. This pattern was applicable in this scenario as it enabled us to define class hierarchies that comprised simple and more complex objects, and we could operate on these objects using the same interface, regardless of their position in the hierarchy. To elaborate, we could have a few leaf classes which implement the basic checks, and a container class which aggregated them, whilst performing operations on these objects via the same interface.

A **LevelCheck** interface was added, containing the abstract *checkLevel* method which returns a boolean value indicating if the map passed or failed a check. Leaf classes of **LevelCheckA, LevelCheckB, LevelCheckC, LevelCheckD,** and the container class **CompositeLevelCheck** all implement the **LevelCheck.** The leaf classes' *checkLevel* methods implement the checks for each individual criterion, whereas the container class' *checkLevel* method iterates through its ArrayList of **LevelChecks**, calling the *levelCheck* method of each one in succession.

The Composite pattern promotes high cohesion and Protected Variation by encapsulating the responsibility of each unique check into its own simple class. Protected Variation is achieved because the variations in these sub-elements would not have an undesirable impact on other sub-elements. For example, our current implementation of **LevelCheckD** performs Depth-First Search to verify for **Gold** and **Pill** accessibility. If a future version of the editor expanded the dimensions of maps so that a more optimised search algorithm should be chosen instead, then the necessary changes would only be localised to **LevelCheckD,** making it easy to update**.** Thus, our design would allow future developers to easily add or modify checks.

By the GRASP principle of Polymorphism, we declared **CompositeLevelCheck** as an abstract class, so that if a different order or a new combination of level checks were to be required in the future (i.e., only checks A-C), it would be trivial to define a new concrete subclass containing only the desired checks. We currently define a **CompositeABCDLevelCheck** to aggregate all of the **LevelChecks**, but a new **CompositeABCLevelCheck** could easily be added. This also leads to greater extensibility and maintainability. It should however be noted that if too many different combinations of **LevelChecks** may be required in the future, then alternative design patterns should be considered to avoid having a single class for each possible combination.

(P1.3) Game Checking

When a folder is passed in as a command-line argument, game checks should be applied before the editor goes into Test Mode with the game. As with the level checks, should the game folder fail any check, the game should not be tested and the editor should return to its edit mode with an empty canvas. Game checks to be applied are as follows:

a. There is at least one correctly named map in the folder.
b. The sequence of maps are well-defined; there is only one map file with a particular number.

The implementation of game checking follows closely with the implementation of level checking given their similar nature. Hence, we used the same principles of polymorphism in combination with the Composite Pattern to implement game checks. In essence, we handled game checking with the aggregation of individual checks by a container class **CompositeGameCheck**, which implemented the same **GameCheck** interface as the leaf classes, and defined *checkGame* methods with which leaf classes would perform their individual checks, while the **CompositeGameCheck** calls the *checkGame* of the leaves in turn. This gives the same benefits of easy addition of new **GameChecks** and modular code.

Implementing game checking and level checking with the Composite Pattern is also in alignment with the GRASP principle of Information Expert. In the Composite Pattern, the responsibility of manipulating a hierarchy of **GameChecks** is delegated to the **CompositeGameCheck**, which can contain both leaf objects and even other **CompositeGameChecks**. Therefore, **CompositeGameCheck** acts as an Information Expert by managing the collection of **GameChecks** contained within itself.

(P1.4) Error Logging

The last notable feature to add to the editor was the ability for the application to log failures whenever a game or level check failed. An **ErrorLogger** class was introduced to help meet this requirement. However, a problem immediately presented is that a class such as **ErrorLogger** is to be used multiple times by disparate parts of the system (**GameChecks** and **LevelChecks**). Furthermore, there should ever only be one instance of it at all times since it violates the Creator GRASP pattern to call its constructor before using it every time, as that would assign creational responsibility of the **ErrorLogger** to classes which should contain or aggregate it.

To overcome these issues, the Singleton Pattern can be used. The Singleton is a creational pattern which ensures that only a single instance of a class can exist, whilst providing global access to this instance. This design pattern is especially important in scenarios where only one instance of a class should be instantiated at any time. Thus, the **ErrorLogger** class was implemented as a Singleton.

In a similar vein, the **Controller** class starts the editor GUI and houses the primary editor logic, so it was essential to guarantee that there was only one **Controller** active at any given time. Thus, the **Controller** was refactored to become a Singleton.

(P2) Design of Autoplayer

When designing the autoplayer, we had to ensure that it was capable of completing all levels of a game in the case where no **Monsters** were present and that it would be straightforward to extend into being an improved autoplayer in the future. In order to achieve these goals, we used a combination of the Strategy, Decorator, and Builder Patterns.

The Strategy Pattern was used as it would allow the autoplayer to adapt and change its approach dynamically depending on the game state. For instance, if eating an **Ice** Cube freezes **Monsters**, a future autoplayer might take this opportunity to play more aggressively. Currently, our autoplayer makes use of a **BaseAutoPlayerStrategy** which does not take Monsters into account, implementing an **AutoPlayerStrategy** interface. This interface has the *moveInAutoMode* method that returns a **Location** that **PacActor** should move to, as determined by the concrete strategy.

Next, the Decorator Pattern was used to assign extra functionalities to the autoplayer at runtime, depending on the presence/absence of certain tiles (paths, portals, gold, etc.). This is achieved by treating the concrete implementations of **AutoPlayerStrategies** as components to be wrapped by the concrete decorators. An **AutoPlayer** interface was defined, with the abstract method *computeWeights*. The interface also contains a HashMap, mapping Characters (representing various tiles) to weight values. These values are determined by *computeWeights*, first checking for the presence of the tiles within the 4 neighbouring tiles of **PacActor**, then incrementing it by a weight value defined in the concrete decorator of each tile if present. Concrete **AutoPlayerStrategies** should implement *computeWeights* to check for **Pills** and **Gold**, since these are both considered basic requirements to win a game. Our current implementation only additionally considers the decorator **AutoPlayerPortalDecorator** to as an example, but more tiles such as **Ice** can be easily added as decorators wrapping the **AutoPlayerStrategy**, each defining their own weight values.

Armed with these weight values indicating the presence of different tiles surrounding **PacActor**, an **AutoPlayerStrategy** could use this additional information to devise a better move. For example, if the **AutoPlayer** knows that an **Ice** cube, a **Gold** piece, and a **Monster** are all one tile away, this information could help the **AutoPlayerStrategy** choose to eat the Ice cube to freeze the Monster, whereas a greedier approach which did not use **AutoPlayerDecorators** to check its surroundings may choose to eat the **Gold** instead, and subsequently lose the game after colliding with the **Monster**. Again, these decisions are infinitely customizable using different **AutoPlayerStrategies** which make use of the information in different ways.

Lastly, since the construction of the **AutoPlayer** becomes relatively complex with these patterns, we introduced an **AutoPlayerBuilder**, by the Builder design pattern. For a smarter autoplayer, the **AutoPlayerBuilder** first decides the **AutoPlayerStrategy** to use (currently defaults to **BaseAutoPlayerStrategy**), then wraps the **AutoPlayerStrategy** in **AutoPlayerDecorators** based on the neighbouring tiles, and finally calls *computeWeights* to fill the values in the HashMap. **PacActor** can then access the concrete strategy via the **AutoPlayerBuilder**, and call *moveInAutoMode* to move automatically. This allows the creational responsibility to be encapsulated by Pure Fabrication and Indirection once again, contributing to greater cohesion and less coupling as **PacActor** does not need to contain the complex AutoPlayer logic, nor does it need to possess dependencies on **AutoPlayer** and its implementing classes.

Appendix

Figure 1: Domain Class Diagram for PacMan in the TorusVerse

SWEN30006 SMD Project 2 Domain Class Diagram
Erick Wong (1173104)
Dillon Han Ren Wong (1236449)
Jonathan Linardi (1272545)

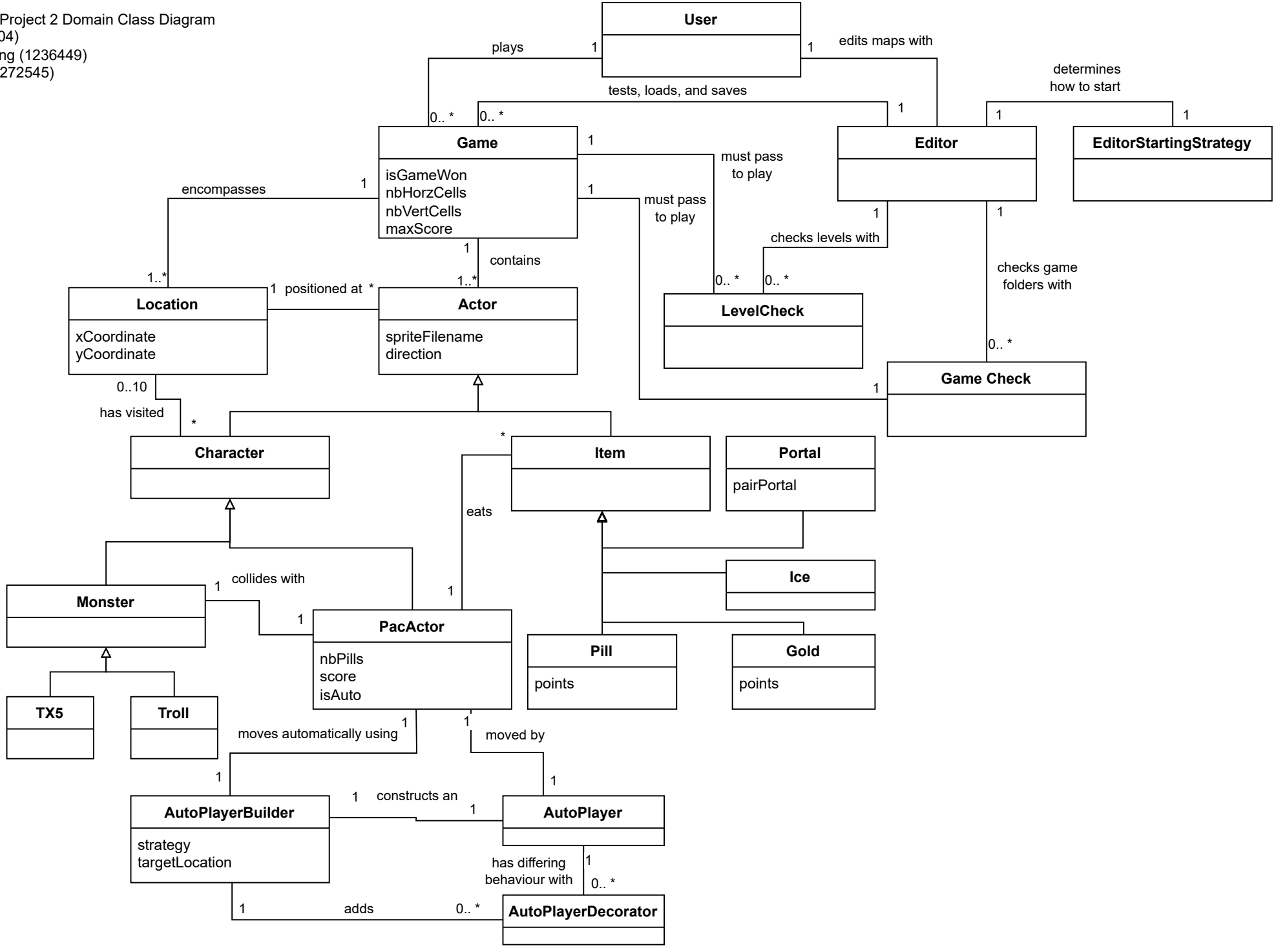

Figure 2: Static Design Model for PacMan in the TorusVerse

SWEN30006 SMD Project 2 Static Design Model
Erick Wong (1173104)
Dillon Han Ren Wong (1236449)
Jonathan Linardi (1272545)

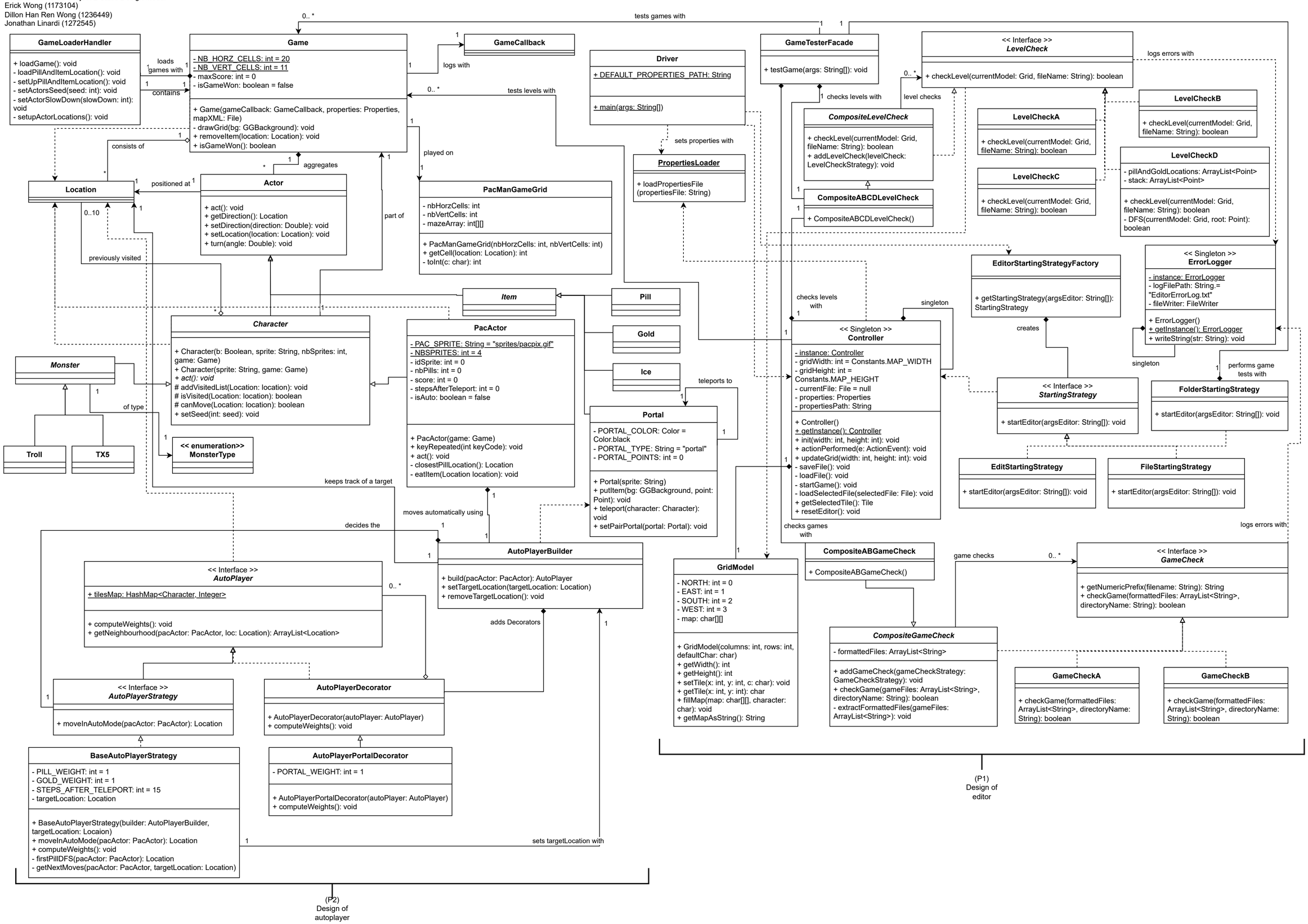

Figure 3: Dynamic Design Model of Editor States

SWEN30006 SMD Project 2 Dynamic Design Model
Erick Wong (1173104)
Dillon Han Ren Wong (1236449)
Jonathan Linardi (1272545)