SWEN30006
Project 2 Report
26/05/2023

By Matthew Pham, Jason Hoang, Jason Yang.

Part 1: Design of Editor and Tester.

This section delves into the design of the editor and tester components, addressing the limitations of the original code and presenting the subsequent refactor. It explores the issues of low cohesion, unprotected variations, and lack of information experts in the original design, followed by the implementation of key enhancements such as the TileManager singleton and MapFileLoader static class, justifying the improvements achieved in terms of high cohesion, protected variability, information experts, and low coupling.

1.1 Limitations of the Editor and Tester:

The original code's editor Controller class violated several GRASP principles, including low cohesion, unprotected variables and lack of information experts. The editor's Controller class assumed multiple responsibilities such as managing file loading, integrating the map, handling the editor application, and coordinating the model and view, resulting in a lack of clear and focused responsibilities and a tightly coupled, poorly cohesive design.

- Low cohesion:
    - The original code exhibited low cohesion as the implementation of the map in the Controller class differed from that in the game, leading to inconsistency and confusion.
    - There was a lack of cohesion among multiple classes involved in representing the map, particularly between the editor and the game, as well as within the editor where various classes independently generated their own map representations.

- Unprotected Variations:
    - The original code suffered from the issue of unprotected variations, where if one class in the editor modified the representation of the map, it would render the map unreadable to other classes, resulting in potential errors and data inconsistencies.

- Lack of Information Experts:
    - The Controller class in the original code was too aware of every object in the map editor, indicating a lack of essential information experts.
    - Instead, specific information should have been delegated to necessary expert classes that specialise in holding, managing and manipulating that information.
    - Although this may lead to higher coupling, this would in turn ensure that consistency is maintained and would result in a more modular and maintainable design.

1.2 Refactor of the Editor and Tester:

In the process of fixing and refactoring the editor and tester components, several key enhancements were implemented. A TileManager singleton was created to ensure seamless conversion between different representations of grid models, while a MapFileLoader static class was introduced to handle file-to-model and model-to-file conversions, relieving the Controller of unnecessary responsibilities and enabling better focus on relaying changes between the View and the Model.

- Created TileManager singleton that manages the conversion between characters, tiles and files:
    - Although the use of a singleton can be risky, the editor must have a way to ensure that every representation of a gridModel, whether that'd be in an xml file or a char[ ][ ], can be converted to each other without having to worry about modifying the representation of tiles in future.
    - Our assumptions here are that every image file used to make a tile is named with the format of {letter}_{tilename}.png. Whilst this does limit the number of tiles one can add in future, it is fairly simple to refactor the TileManager and replace each letter with a number or some other simple representation.
    - This also has the added advantage of avoiding data duplication, since the representations of tiles and conversion hashmaps between tiles and characters are all managed by the TileManager, rather than being generated by individual classes (which was the case before the refactor).

- Created a MapFileLoader static class to manage the conversion of files to models and vice versa:
    - Takes away responsibility of knowing how to load and save maps from the Controller, so it can focus instead on relaying changes from the View to the Model.
    - This class takes advantage of the TileManager class to convert xml files to Grids and vice versa

- Additionally, the creation of GameMaps simply references a Grid to form a Space[ ][ ].
    - This means any model can be converted to a GameMap without having to have a Grid as a dependency, since Grid can be generated with or without the editor simply using a file.

1.3 Further Justification of the Refactored Design:

The refactored design of the editor justifies its improvements by addressing the issues of high cohesion, protected variability, lack of information experts, and low coupling. Generally, classes are now more free to manipulate and work with Grids and Tiles without having to understand the exact conversions between the two.

- High Cohesion and Protected Variability:
    - The implementation of TileManager and MapFileLoader means that any class can generate a consistent representation of a grid of tiles from a file without having to rely on any other class (and vice versa).
    - Additionally, since any changes to how tiles are represented are located in TileManager, it is fairly easy to modify the representation of tiles without affecting the whole codebase.

- More Information Experts and Low Coupling:
    - Generally, classes like Controller, TileManager, MapFileLoader and GameMap are now only focused on a few operations and have very little reliance on any other class.
    - As an extreme example, Controller isn't even linked to any classes that deal with GameMap, since GameMap doesn't rely on any model contained in Controller. This is because it can manage the conversion between .xml to the Space[ ][ ] it needs to function without referencing the model in Controller, but instead does this directly from file using MapFileLoader

Part 2: Design of Autoplayer.

The design of the Autoplayer used by PacActor addresses the limitations of the original implementation and presents the subsequent improvements. The original Autoplayer lacked extensibility and understanding of the game map, leading to errors and limitations. This prompted the implementation of a new approach utilising a more powerful abstraction of GameMap and pathfinding algorithms based on Spaces, allowing for flexible pathing. Extra adaptability is provided through PathObject adaptors.

2.1 Limitations of the AutoPlayer:

The original implementation of the "autoplayer" was simply a function in PacActor that searched for the closest item and travelled in its general direction, without using a concrete algorithm. Simply put, this was inextensible in almost every way, with the main reason being that it had no real understanding of the map the game was being played on. In reality, the limitations and errors of the AutoPlayer were due to the limitations of the representation of the map, which was the main thing that had to be fixed.

2.2 Implementation of the Autoplayer:

- GameMap contains all the information needed to navigate and interact with the game:
    - To be more precise, GameMap deals with Entities (which extends Actor) interacting with tiles like Portals, Paths and Walls without needing to know exactly where these tiles are located. Simply put, at any Space in the GameMap, it is easy to retrieve the list of adjacent spaces, as well as whether or not these spaces are walkable.
    - These spaces also make a distinction between neighbours available to spaces on entering them as opposed to exiting them, based on SpaceOccupiers. For example, Portals modify the space they're located in by replacing the neighbouring spaces with the neighbouring spaces of the Portal that they're linked to when entering them. However, exiting a Portal acts as if it were a normal tile.

- The AutoPlayer relies on the abstraction of GameMap into a grid of Spaces that each contain neighbour Spaces.
    - Since a Space is essentially a node linking to other spaces, the Autoplayer can calculate the next Space to move to by doing BFS between a start and end location. This also works with Portals because the Autoplayer will start by querying the neighbouring Spaces on exiting the Space its currently on, but will query consequent spaces by the neighbours

available when entering these Spaces. While the exact reason this works is fairly hard to explain in short, this allows the Autoplayer to pathfind without having to understand where every Wall, Path or Portal is.

- Additionally, modifying the pathing behaviour of AutoPlayer can be done using PathObject adaptors.
    - As of the current refactor, AutoPlayer pathing can be modified by giving the AutoPlayer a list of PathObjects to avoid. Currently, this is a simple adaptor for an object that can be queried to return a Location, regardless of the fact that the object being adapted has a Location.
    - This itself is fairly easy to modify to improve the pathfinding of Autoplayer: all that someone would have to do is modify the PathObject interface to provide more information than just a Location, and then subsequently alter how the Autoplayer deals with these objects. The Autoplayer would not need to know exactly what these objects are if they provide enough information.

Note: for the design class diagram, please refer to the DomainModel.pdf in Documentation.

Part 3: Extra Changes and Assumptions.

- Without the ability to manipulate the JGameGrid library, the functionality of the stop, pause, reset buttons that appear when playing/testing a game cannot be changed. Subsequently, to avoid dealing with the malfunction of these buttons, we have disabled every button except for the pause button when playing/testing.

- Although the specification details that there is test mode behaviour in the editor for running singular files, the behaviour for launching the game in test mode to test a singular map is also currently undefined by the specification. Thus, we have left this functionality out of the final refactor.

- Games are not responsible for generating the map that the game will be played on. Instead, a GameMap is passed through a Game on creation with all the information needed for the game to run. This means that a whole pacman “Game” is really just a shorter series of smaller games played on individually generated GameMaps (Think about it like Monopoly, the board is already set up, you only have to put the piece on the board and start playing).

- The current method of generating a GameMap is through GameMapValidator, which itself doesn’t follow any of the GoF design patterns for a variety of necessary reasons:
    - The goal of GameMap is to abstract the concept of moving and interacting with a map to moving to a Space, seeing the available movement options from that Space, and interacting with a Space, without having to know exactly if the space is a Wall, Path, Portal, or contains a consumable Item etc. This makes navigating and interacting with GameMap as simple as possible.
    - The specification details that a GameMap must follow certain criteria in order for it to be playable. Since the strength of GameMap is that it hides away its implementation very

well, a class that validates a GameMap must know everything about the GameMap and more, which is essentially why GameMapValidator is so well coupled with GameMap; because it essentially could be a GameMap out of necessity.

- A Space factory was considered in figuring out how to construct the Spaces in GameMapValidator without having to know exactly how a space was composed. While this works with most types of Spaces, this breaks down with Portals because Portals need to be instantiated in pairs, and a Space factory can only create one portal at a time. Since a class needs to know when a Space contains a Portal to be able to implement the Portal, this defeats the point of delegating that responsibility to an object that can't know that information.
- It was simply easier to make GameMapValidator an Information Expert on how to validate and create a GameMap, since creating a GameMap already necessitates having the required knowledge to validate that GameMap. It would be more confusing to abstract that away to another object.