Group 2

The Duy Nguyen - 1100548
Ramon Javier L. Felipe VI - 1233281
Jonathan Chen Jie Kong - 1263651

SWEN30006 - Project 2 Report

Design of Editor

The original code of the editor remains for the most part the same in our implementation However, modifications were made most notably to Controller and View. Other than that, the original editor was preserved as it serves the intended purpose and fulfils the requirements of this project. Furthermore, the design is fairly reasonable to not consider refactoring. While the design is reasonable, there are still areas for improvements.

Firstly, while the Controller class somewhat achieves an individual coupling reduction for others in the editor package, its dependency with the grid package can be significantly improved. Specifically, the grid package only requires information for a grid to update itself. So instead of using a GUIInformation as an interface for Controller within the editor package to send information, simply passing the controller’s grid to GridController and GridView would suffice. With this, the grid package will not be dependent upon the editor package, and instead the editor package will only have to use objects from the latter. Sending the tile information this way will thus reduce the dependency and achieve indirection. Secondly, the class's cohesion can be improved. Notably, Controller is dealing with converting tiles from its encoded character to the XML string format name. To improve this, this task can be delegated to Tile instead.

Firstly, some classes in the editor such as Controller have rather high coupling, which can be reduced by evaluating it with the use of GRASP principles. For example, the use of the *information expert* principle is able to inform us of where to reassign responsibilities to increase the overall cohesion of the design or even indirection to reduce direct coupling between classes. The problem with high coupling with the *Controller* is that changes in one class might have a trickle-down effect on all the classes that are coupled with it. This is especially problematic when the editor depends on the *Controller* to function, it also impedes the extensibility of the editor, making it harder to maintain and add new functionality. Furthermore, the use of *indirection* could also reduce the direct coupling between classes which reduces the overall dependencies of a class, given the original design contains classes that are highly coupled. The original editor contains classes whose constructor does not focus solely on instantiation of objects. This is highly unrecommended and any operations that do not serve that purpose should be extracted to ensure that functions are concise and perform what is expected.

Despite the issues mentioned above, most classes in the original editor showed good *cohesion*, where the functionalities within each class are closely related. The classes are designed in a way that solves a clear and focused problem. This allows for easier maintenance and extensibility as related responsibilities can simply be added or removed. Furthermore, the original editor displays the use of *polymorphism* through the use of interface. It allows classes to implement the functions independent of other classes that also implement it. This supports code reuse and the ability to handle other classes with varying implementation. The *TileManager* in the original editor demonstrates the use of the *creator* principle where it creates new instances of *Tile*s which are then stored and returned as a list. Moreover, *TileManager* also shows the use of *information expert* principle. Given that *TileManager* knows the information about the tile's image, it was given the responsibility to create it. The original editor also shows the uses of *pure fabrication* where classes such as *CameraInformationLabel*, *GridMouseInformationLabel*, and *Controller* are created to ensure that other classes can have more focused responsibilities, while retaining their cohesiveness. The points mentioned above work towards protected variation, ensuring that a stable interface is created for the objects that use the classes. It also allows the game and editor to be more extensible, as responsibilities are clearly defined and modularised, new implementations can more easily extend it.

A design pattern that was used in the original editor is the *observer* pattern, notably through the implementation of listeners in classes such as *GridController*, *GridView*, and *Controller*. The use of this pattern reduces the overall coupling between subscribers and publishers, and allows subscribers to automatically update objects without being specified by the user. This is a particularly important pattern that was implemented as the editor itself contains multiple components that need to be updated frequently due to user input, which effectively reduces the overall coupling and dependencies between classes.

Diagrams Discussion

Referencing DomainModel.pdf, the domain class model illustrates the workings of the editor on an abstract level, as such it is unsuitable to describe the intricacies of the autoplayer and game levels as these are discussions more suited for the solutions space. On a conceptual level, the *PacActor* would be the one responsible for using the autoplayer, which involves scanning *Grid* for objects that it is actively moving towards and actively avoiding. Furthermore, the process of switching levels can be shown by the relationship between *Editor*, *Game* and *Grid*. This involves the *Game* to go through the playable *Grids* that have been edited and started from the *Editor*.

In the design class diagram in StaticDesignModel.pdf, we have largely retained our design for the game from project 1 with some additional classes. The reason we have chosen to retain our design was because our previous design was extensible to the addition of new objects, moveable or non-moveable. Furthermore, our previous design showed high cohesion where the responsibilities of each class are clear and well encapsulated, it also attempts to keep coupling minimal. Our previous design also heavily promotes code reusability, evident through the relationship between superclasses and subclasses. On top of that, the previous design also uses template method (most evidently with monster's moveApproach in act method algorithm) and strategy pattern (similarly with each differing implementations of act method for LiveActor subclasses), all working towards protected variation. The additional classes in this design are *LevelChecker*, *GameChecker*, *PathFinder*, *GridFileManager*, *Portal*, *PortalFactory* and *XMLParser* which we will discuss in more detail below.

Design of Autoplayer

Our implementation is a simple pure fabrication pathfinding class which does not separate the rules (using Iterative Deepening Search algorithm). The most immediate approach to extend our current implementation to monsters is to apply pathfinding for every individual move for pacActor and consider monsters as unobstructive objects. We refer to this as periodic pathfinding (where each period would be a single move). Nevertheless, for better extensibility, given that autoplayer is a structural issue, the ideal design pattern for this problem would be a *composite* pattern. The way this would work would be that the individual leaf would represent a given rule of a pathfinder, which inherits from the component that is PathFinderRule which is aggregated by PathFinderComposite. These particulars made up the *composite* pattern, which allows for extensions such as ice and monsters to be extended by defining their own rules (AdditionalModels.pdf, Figure 3). For example, if ice and monsters were implemented in pathfinder, the PathFilterComposite will aggregate rules of *PillFinderRule*, *GoldFinderRule*, *IceFinderRule*, and *MonsterFinderRule* into a single rule without *PathFinder* knowing if it is working with an individual object or a composite. Then *PathFinder* would implement the new rule and set out to find its new path according to the rule specified. The choice of design pattern allows the pathfinder logic to be easily extended.

Our current implementation of the pathfinder works as illustrated in AdditionalModels.pdf, Figure 4. Firstly, PacActor will find a path to the closest pill or gold. After finding the path, it will initiate the move along the predetermined path. Upon completion of the move, if the PacActor collides with a monster then the game terminates and the player loses, otherwise it determines what are the possible moves. If PacMan has eaten all pills and golds then the game ends and the player wins, otherwise the PacActor repeats the process of determining the next path to the closest pills or gold.

Modifications made to the editor and game

Aside from the editor and pathfinder, we have implemented design patterns with the use of GRASP principles to guide our decisions. Firstly, the use of the *template method* and *strategy* design patterns is evident, as discussed. The implementation also adheres to *polymorphism* and *protected variation* principle, which allows different *LiveActor* and *Item* to extend it if required. Secondly, object instantiation in the game is complex and requires the interaction of many different objects which could lead to unnecessarily high coupling. Namely, successful object creation in the game requires classes such as *PacActor*, *Monsters*, *XMLParser*, *Items*, *Portals* and *GameCallback*. With evaluative principles such as low coupling and high cohesion in mind, the use of *Facade* as a design pattern was intentional and implemented through *ObjectManager*. Using the *Facade* pattern, we are able to effectively reduce the coupling between objects and create a unified interface where objects that require information about other objects are able to do so through the *ObjectManager*. The *ObjectManager* itself is an example of pure fabrication. Furthermore, it also displays the principle of *indirection* as it acts as a mediator to other classes and avoids direct coupling between different classes. This not only decouples objects but also supports code reusability as it became a central class for holding objects' information. Our implementation of PacActor utilises the *Observer* pattern to react to the program in an event-driven fashion.

In addition, due to the complex logic in creating *Portal*, where each portal must always be paired and there exists only one pair of a given colour in a game, we use a *Singleton Factory* to assist in dealing with this logic - *SingletonPortalFactory* (AdditionalModels.pdf, Figure 5). This ensures that there will only be a single access point for *Portal* creation, resolving the possibility of multiple *Portal* creation as well as the rules in creating them. The *PortalFactory* itself is an example of *pure fabrication*. Since the *ObjectManager* is responsible for object instantiation, naturally the *PortalFactory* would be called by it. In implementing additional capabilities, we have created *LevelChecker* and *GameChecker*, as the checking logic is complex. Thus, the creation of these classes encapsulates a set of responsibilities that helps increase the cohesion of *Game* and *Controller*. The creation of *LevelChecker* and *GameChecker* are also examples of *pure fabrication* as it does not represent a problem domain concept. Furthermore, this allows the editor to be more extensible to future checking conditions that can be implemented by defining new conditions within their classes. In LevelChecker, we used a PathFinder's algorithm (we name it Greedy DFS) to check whether there are any unreachable mandatory items. This is because we design PathFinder to be a collection of move approaches for PacActor, and finding unreachable items is ultimately part of PathFinder's responsibilities. As such PathFinder achieves high cohesion.

Similarly, *GridFileManager* is an example of *pure fabrication* as well. Likewise, it has specialised responsibilities, which improves the cohesion of the editor package (specifically the *Controller*), as the logic of loading and saving grid can be extracted to serve more specified purposes. It is also in line with the principle of *information expert* as it contains information about the different files and state of the game to perform the necessary actions.

Future modifications to the editor and game

One of the design choices that we recognised could be good but was not implemented was pulling the logic for handling UI and UI interaction out from the *Game* class (namely with PacActor's listener). This would increase the overall cohesion of the *Game* class, allowing it to strictly handle the game logic. Furthermore, the act of separating the logic for handling UI and UI interaction would be an example of the *controller* principle. The reason we did not implement this design was because it currently fulfils the requirement of the specification, and the logic for handling UI is not sufficiently complicated to warrant a new class, which could lead to increased coupling. Next, the *ObjectManager* could be more cohesive by extracting the process of reading properties files into a separate class. While it currently does so with XML, it does not do it with the properties files. The reason we did not implement the design was similar to the point above, where the complexity of reading the properties file is not sufficiently complicated to warrant a new class to be created. Other modifications that should be made include the changes to editor and grid packages mentioned above to reduce the dependencies, as well as the composite pathfinder.

Assumptions

Some of the assumptions we made in our implementation are:

1. Square brackets are included in the error log lines. For example, it will be in the form *[Game foldername - no maps found]* instead of *Game foldername - no maps found*
2. Error logs for unreachable are printed on separate lines. The error log for pill and gold are printed on separate lines
3. Error logs for portal names are not space separated. For example, it will be reported as *darkgold* instead of *dark gold*
4. The behaviour “exiting from left and reappearing on the right” implies that if the path is open, then moving outside the left boundary of the game will spawn the actor on the right boundary of the game. Eg. Move to (3, -1) will spawn the actor to (3, 19), likewise for moving up and down.

Software Models

Software models are submitted as separate documents in *documentation*.