SWEN30006 - Software Modelling and Design

Project 2 Design Analyse Report

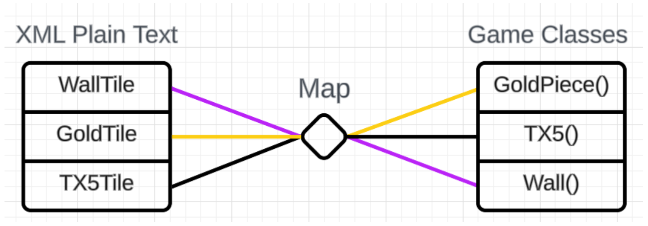
TEAM (Wed 11:00 #8): Yanhai Zhang [1254937] | Wenfei Zhang [1255000]

Introduction

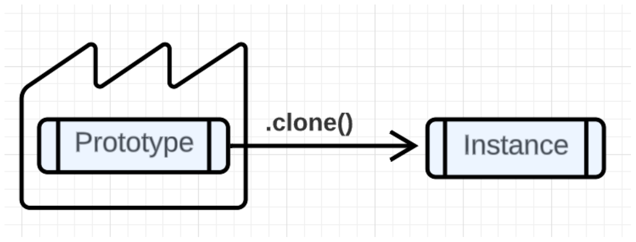
Since our code for project 1 retained all the functionality of the original version, and our implementation is highly object-oriented and flexible, we started our project with our own work from project 1, which had been refactored back to a simple version. In order to better understand the new version of the game, we firstly constructed a domain class mode (as in DomainModel.pdf) by extracting nouns from the application requirement from the specification document given. After analysing and discussing about the new system, a static design model (as in StaticDesignModel.pdf) is constructed based on the team's idea of improving the system so that it is more extensible and flexible. While we were converting the diagram to codes, the diagram itself was constantly evolving, not only in order to better fit one of the GRASP principles "low coupling, high cohesion", but also for solving specific problems with design patterns. These two diagrams are all partial diagrams since there are many classes that had already been introduced in our previous project and keeping them may substantially lower the readability of the diagram. Lastly, a state machine diagram for the Test-Editor switching is constructed (StateMachineDiagram.pdf) to make the system clearer.

Part 1 – Design of *PacMan In TorusVerse* — The Editor

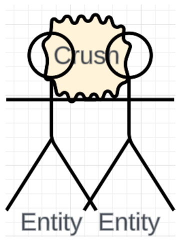
Speaking of the GoF design patterns, it's designed for solving kinds of problems for software modelling and design. The first problem we encountered in the project is to load a game map from a given XML file. XML files provided text telling specific kind of tiles in game and its position, to achieve a higher flexibility and lower coupling, we decided to use an **Adapter** pattern with some modification. We create a map that stores the exact text for fields in the XML as keys, and corresponding class as value. This adapts the XML type of game map to the game grid. By implementing this pattern, we separated conversion codes from the game logic itself (a.k.a the Single Responsibility Principle). Also we can still preserve any other adapters like loading maps from properties files.



However, at the same time, another problem was found: if we use a map to store the plain text and its corresponding class, we have to find some way to have new instances when loading multiple entities of the same kind. This is where we decided to use another design pattern from GoF, called the **Prototype** pattern that allows classes to make an exact copy from one instance — that is, the prototype. By applying a Cloneable interface, we can store an instance in the map and whenever a new instance is needed, we can just call the clone() method and get a copy of the prototype of its class and be able to add it to our game map without any problem. The Prototype pattern helped us to get rid of repeating if-else statements in order to find which class to instance. This also provides the potential to add more classes (Entity) without needing to add if statements.



Then we start implementing the new feature of *PacMan In The TorusVerse*, the portal. Portal as a class, it does not keep all the portals as an attribute and does not know where the destination is. Therefore, according to the Expert (Information Expert from GRASP), this class should not be responsible for checking where the destination is. Hence a PortalManager is introduced to manage all the pairs of portals, and at the same time, check if the portals are valid (one portal cannot appear in multiple pairs, in one map, there must be exactly 2 portals of the same colour, etc.). In this way, both the portal and the entity does not need to keep in memory which is the other end of which portal, PortalManager will tell the information. And for detecting if an entity is being teleported (stepping on the portal), we made use of a class from the last project: CollisionManager, which implements **Observer** pattern. Collidable classes will be registered to observe onCollision events by implementing the Collidable interface, then the event will be triggered on collision between two Collidable entities. This restricts the game to only teleport entities on collision (since collisions are detected once every move) instead of hard-coded to make



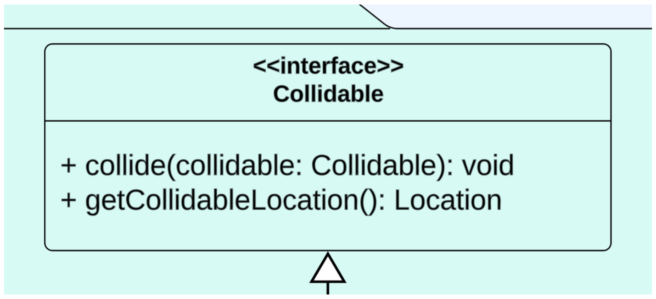
the portal continuously teleport entities back and forth. Because of this, the implementation of portal teleportation and detection is highly lightweight.

After we implement this pattern, we are able to decouple some over-complicated classes (like Game), and assign responsibilities more precisely. When we proceed to do the level check, we notice that our application should allow developers to modify or add checks easily (as in spec sheet), we choose to apply a **Command** pattern (also from GoF) on it. In the context of the game, we separate different level checking logic into different classes, but they all implement our new interface called LevelCheck, which regulates the checker class method signature that we can call it from command. By having this pattern of classes, we can have a list of LevelCheckers and call their checking function one by one. With the command pattern, it becomes easier to modify any of the components from an overall complicated operation, providing the structure with a low coupling and high flexibility.

For the ability to switch between different levels, we need only one single instance of Game. If we initialise a new instance of Game, it will create the whole game window. To make sure the whole application only has one Game instance running, the **Singleton** pattern is used. Every time we launch the game (i.e. launch the game for the first time, switching between editor and test mode, etc.), a new instance is created (and the old instance is disposed of if there is one), and return it to whoever is asking for it. However when switching to a new level, the Game instance will not be recreated. Instead, the initialise function will remove all actors on the scene and replace with the entities from the new level and re-register InputManager and PortalManager. In short, Singleton pattern makes sure that we have a global access to the one instance of game class, but also provides the potential to destroy an old instance and create a new instance, solving the problem in a convenient and robust way.

Part 2 – AutoPlayer

Since we implemented collision detection in a mutual manner (collision detection is implemented using Observer pattern, in which both the entity colliding on the other and the entity being collided by the former are subscribers of the collide event), PacMan object is aware of what Consumable (will also be Collidable of course) object it is going to consume. And since it can get the singleton instance of the game board, it also knows the exact location of monsters and pills/golds. The speed of an entity is also a field that it is aware of. Using the above information, PacMan will be able to calculate an optimal route that eats as many pills/golds as it can during the defined seconds of freeze upon consuming an IceCube.



<<interface>>
Collidable
+ collide(collidable: Collidable): void
+ getCollidableLocation(): Location

[Collidable interface that allows PacMan know who is colliding with it]

Part 3 – Further improves

With our object-oriented design, content can be easily extended. Monsters can have their own unique walking approach by overriding the default behaviour.

Level checks can be very easily modified due to the use of Command pattern. All the level checks implement the LevelCheck interface, and LevelChecker simply holds a list of checks that will be checked before loading the map into the game.

Game XML format can also be changed very easily since we chose to deserialise the XML into a schema class instead of manually deciding the structure. And then the schema will be adapted to an object readable by the game. As long as the XML file matches the schema, there is no need to modify huge amounts of codes.

As mentioned above, the Prototype pattern of instantiating entity classes also provides flexibility of future implementations of more monsters and entities.