Assignment 2 Report

Assignment 1 Discussion

The UML diagram from assignment help me know what kind of class will be used in assignment 2 beforehand as it hindered some design decision. For example, GameApp work similarly to GameEngine, There's Player, Enemy, Bullet, Bunker which is similar to what is in for assignment 2. Also, I had predicted that it will need a collider detection and it did in assignment 2.

There are features from assignment 1 diagram that I've used in assignment 2 such as Object (which is GameObject) that is an abstract class for every object in the game. Bullet class that contains owner variable and Living interface had been changed to Shootable. This is because in assignment 2 there are many more interface which had different responsibility hence I need to adapt to the circumstance. However, since every object in the game behave quite similar so I just implement most of them in GameObject abstract class.

The big different is when I need to apply design pattern for assignment 2, which made UML class more complex than what we had in assignment 1 since when initialise any Enemy and Bunker that will show on the screen we used Builder pattern and when we create a bullet we used Factory pattern instead of just initialise it in main App class like I did in assignment 1. The enemy shoot pattern also controlled by Strategy pattern and Bunker colour change also controlled by State pattern which doesn't initially plan when I draw UML diagram class for assignment 1.

Factory Design pattern

The Factory pattern had been used to produce the bullet whether it's from Player or alien Enemy. Bullet is product of this pattern where BulletPlayer and BulletEnemy are concrete product that's extend Bullet abstract class. BulletFactory acts as a creator where BulletPlayerFactory and BulletEnemyFactory are concrete creator that's implement BulletFactory create method.

This pattern centralised the process of creating bullet within runtime in single place (S). Client relies on a single create method from BulletFactory on high level without need to know anything from other concrete creator on low level (O, D). With the used of abstract class in Bullet and interface in BulletFactory (L, I). The Factory pattern support creator pattern by initialise the Bullet within Factory class and polymorphism on Bullet and BulletFactory class (GRASP).

The notable benefit is that it's super easy to create a bullet within GameEngine class by just calling shoot from Shootable object and by categorised them as Bullet we able to easily update its logic. However, this method used so many class to do such a simple, having many class can be confusing if we not organise the package properly.

State Design pattern

The State pattern had been used to control the colour change of a Bunker when receive damage. Where the context is Bunker which had BunkerState as a state variable where BunkerStateGreen, BunkerStateYellow, BunkerStateRed as a concrete state which implements BunkerState interface.

Each state had its own implementation when calling the same interface method and had only one responsibility is to control the state change of Bunker (S, I, L). The client can call change state on high level without modifying anything, even there had been more state add we are open to implement it in new low level state class (O, D). Each state class often possesses expertise in handling different behaviour for its specific state aligning with the Information Expert and able to control the state of Bunker like a controller (GRASP).

The notable benefits is that we can easily add another state for Bunker without needing to modify Bunker class and within Bunker class we can just call the same method to change the color of Bunker without using if statement. The downside is that to modify the Bunker we need to pass down the Bunker itself to the BunkerState class, where modifying image within Bunker class is easier. Also, if we want to add more state then we need to create a class for every new state.

Builder Design pattern

The Builder pattern had been used to help instantiate the Enemy and Bunker at the start of the GameEngine by taking information from JSON object and processed it within Builder class. Director had been used to construct list of Enemy or Bunker object using Builder to build each entity, the client using different concrete builder such as BunkerBuilder and EnemyBuilder pass it down to different director to build list of product such as Bunker and Enemy.

The Builder had one job is to build a product and each concrete builder, build different product, the Director had job to control the builder to construct a product in a list (S). Client can use the same Director and Builder class even the new builder concrete or entity product is introduced which made client to rely solely on abstraction method (O, D). The concrete builder also had the same method is to build a product which implemented Builder class (L, I). Instantiate entity product by creator pattern, the director control the construction of a list of product, and low coupling when director used builder to build an object in similar manner (GRASP)

There is a benefit to used this method as it group building process in one responsibility for client in GameEngine. However, notable downside that I've seen in code is that I had to do explicit type down casting so that Bunker and Enemy stays within its own type because Director and Builder abstract will output only GameObject type otherwise I need a create new Director and Builder class every time I need to build different type of product which is bad design practice.

Strategy Design pattern

The Strategy pattern had been used to control different kind of movement of BulletEnemy. Where the context is BulletEnemy and had the move strategy BulletMove which had different concrete strategy such as BulletMoveFast and BulletMoveSlow implemented it.

Each strategy behave differently accorded to its role that had been initialise with the EnemyBullet class and only use to control the behaviour of enemy bullet (S) We can add new move strategy or modify its behaviour without modifying BulletEnemy where the BulletEnemy context can just call move from strategy (O, D) All concrete strategy implements the same method as BulletMove interface (I) hence it can categorise as BulletMove object (L). This pattern encourage high cohesion pattern by grouping related algorithms into separate strategy classes, each focused on a single responsibility (GRASP).

The notable benefit is that we can easily develop a new enemy bullet type by just creating a new concrete strategy class and let it acts differently. However, the downside is that we need to initialise different strategy every time when we create BulletEnemy but BulletEnemy had been create using factory pattern that mean I need to change BulletFactory to be able to receive strategy pattern as a parameter to build BulletEnemy which impact BulletPlayer that doesn't need strategy pattern which made the situation more complex, also to move the bullet we need to modify the BulletEnemy by pass it down as a parameter.

Spcae Invader UML Class diagram

pcha3724 | September 30, 2023