Assignment 6i, SU

Aditya Fadhillah (hjg708)

Single responsibility principle

Single responsible principle states that a class should only have one reason to change. Every class should have responsibility over a single part of the functionality, and that responsibility should be entirely encapsulated by the class. Each class should focus on a a single task at a time. Everything in the class should be related to that single responsibility. With Single responsible principle, classes become smaller and cleaner, and prone to less error. An example to a Single responsibility principle is in **Example 1**. The class enemy is responsible for one function, and that function is the attributes of an enemy. All other elements in enemy class is related to the task.

Open-closed principle

Open-closed principle states that entities such as classes, should be open for extension, but closed for modification. Any new functionality should be implemented by adding new classes and methods, instead of changing the current ones or existing ones. An example to an Open-closed principle is in **Example 2**. The class down moves a group of enemy downward. It fulfilled the open-closed principle because this move functionality is implemented as a new class and it therefore does not change the existing enemy class.

Liskov substitution principle

Liskov substitution principle states that if A is a Subtype of B, then objects of type B can be replaced with objects of type A. It means that the inheritor types should be replaceable with their inherited types. An example to something that does not fulfill Liskov substitution principle is in **Example 2**. The class Enemy is an subtype of the class Entity, but it also have a new method that does not exist in Entity class. Enemy class is not replaceable with Entity class, and it therefore does not fulfill Liskov substitution principle.

Interface segregation principle

Interface segregation principle states that clients should not be forced to depend on methods that they do not use. A class should not receive requirements from an interface if it does not have the ability to fulfill that requirement. A class should only perform actions that are needed to fulfill its role. All other requirement from that interface should be removed or moved somewhere else. An example to an Interface segregation principle is in **Example 2**. The class down uses all the requirements for MovementStrategy.

Dependency inversion principle

Dependency inversion principle states that high-level modules should not depend on low-level modules. Both should depend on abstractions. And Abstractions should not depend on details. Details should depend on abstractions. It means that interaction between high level and low level modules should be thought of as an abstract interaction between them. Can be pictured as two different classes that can abstractly interact with each other by utilizing interface.

Example 1

```
namespace Galaga {
    public class Enemy : Entity {
        private int hitpoints;
        public Enemy(DynamicShape shape, IBaseImage image) : base(shape, image) {
            hitpoints = 3;
        }

        public int Hitpoints {
            get{return hitpoints;}
            set{Hitpoints = value;}
        }

        /// <summary>
        /// Reduces the enemy's hitpoints by 1
        /// </summary>
        public void ReduceHP () {
            hitpoints -= 1;
        }
    }
}
```

Example 2

```
namespace Galaga {
    public class Down : IMovementStrategy {
        private float s;
        public Down() {
            s = 0.0003f;
        }
        public float Speed {
            get {return s;}
            set {s = value;}
        }

        public void MoveEnemy (Enemy enemy) {
            enemy.Shape.MoveY(-s);
        }

        public void MoveEnemies (EntityContainer<Enemy> enemies) {
            foreach(Enemy enemy in enemies) {
                MoveEnemy(enemy);
            }
        }

        public void IncreaseSpeed (float inc) {
            Speed =+ inc;
        }
    }
}
```

Appendix

Source 1:

https://medium.com/
backticks-tildes/the-s-o-l-i-d-principles-in-pictures-b34ce2f1e898

Source 2:

https://www.c-sharpcorner.com/
UploadFile/damubetha/solid-principles-in-C-Sharp/
?fbclid=IwAR2NJHiIsOxxH59NcjiBV6CB1tXnDNTGGaq53mW4muRETRmKoJw3kwxGBKI

Source 3:

Hall G. M. , 2022, Adaptive Code: Agile coding with design patterns and SOLID principles 2. edition, Microsoft