Space Invaders

SE456 Design Document

Winter 2017-2018

Kaicheng Yan

Contents

Design Patterns

During the developing progress, there were many design patterns used throughout this game. Many of these design patterns will make mention of implementation that will be explained further below.

Singleton

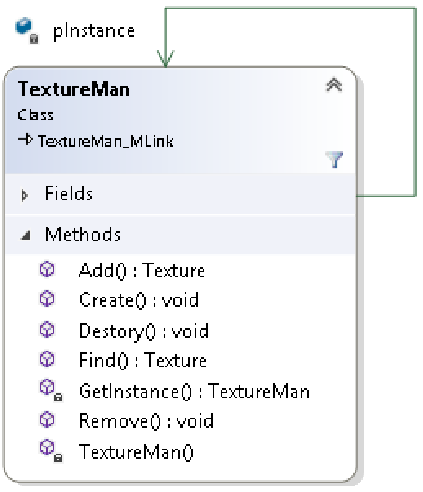
a. Problem

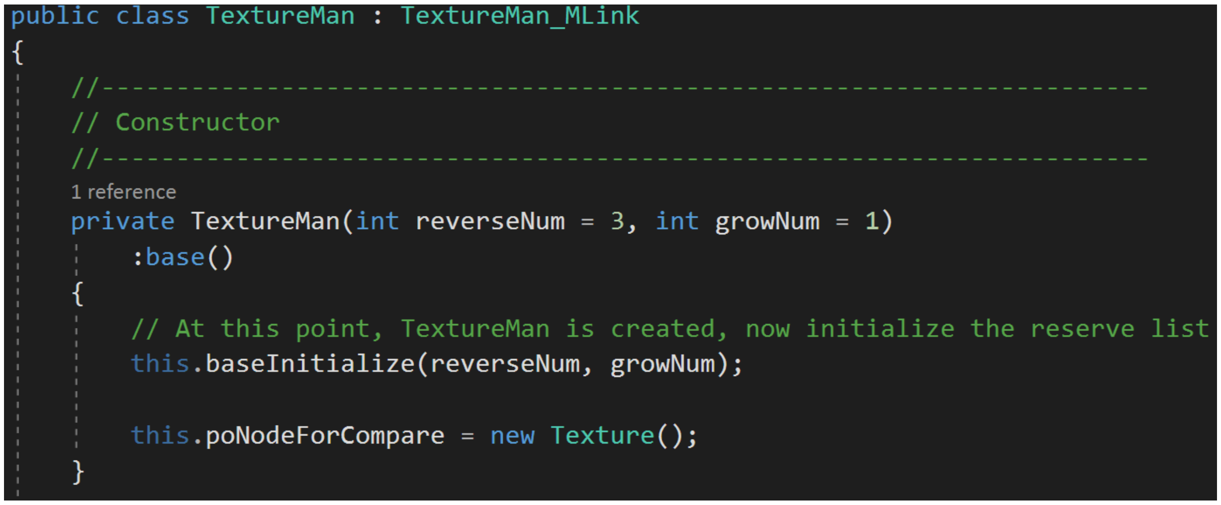
I have a problem of management of each component of the game, such as textures, images, game sprites and so on. I want to have an efficient and convenient way to refer to the instances of these mangers to let them interact with each other. To implement this efficient way, I created managers to control these components with Singleton pattern.

b. Pattern Intent

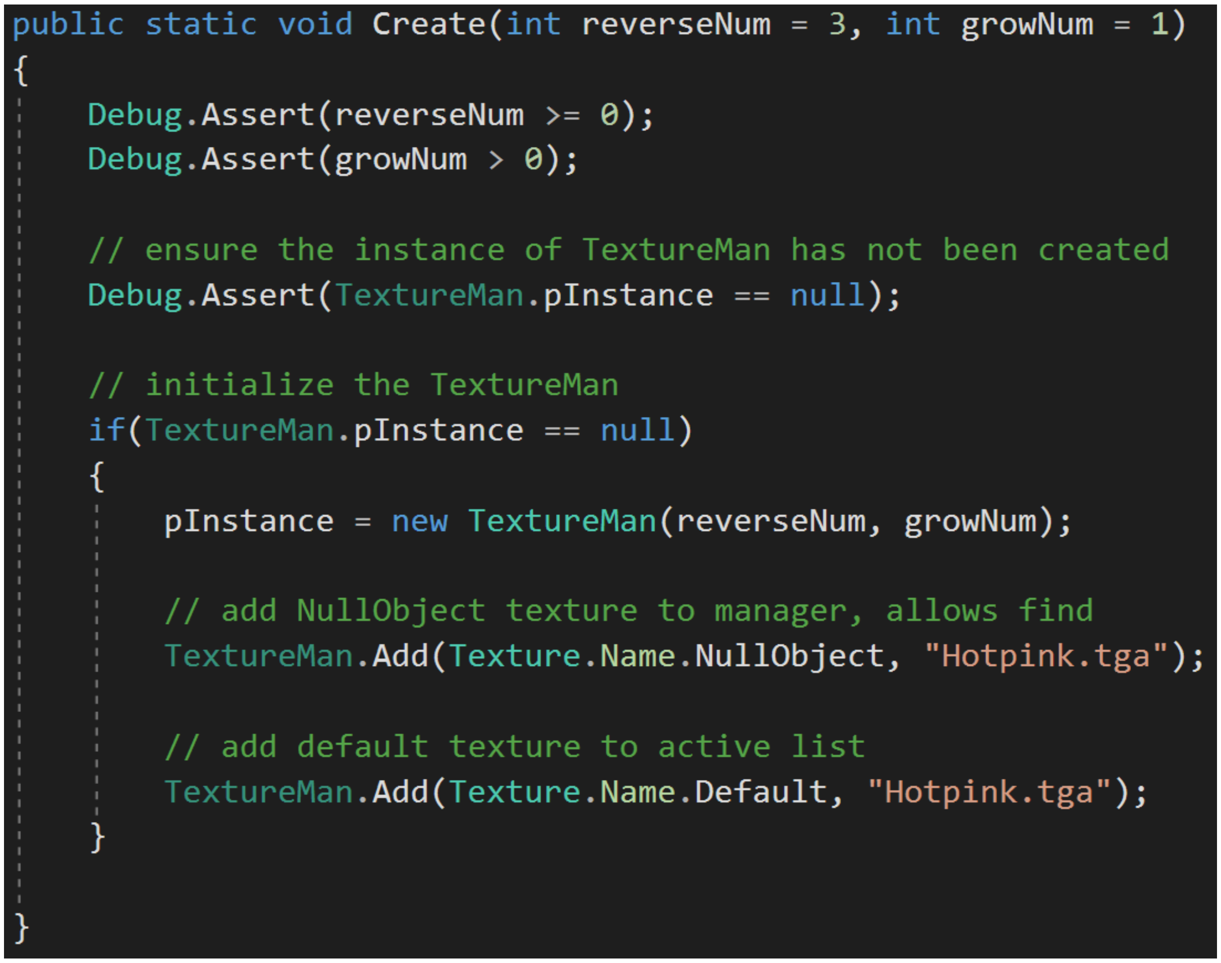
The Singleton pattern must ensure a class has only one instance and provide a global point of access to it. I used lazy instantiation to solve this problem, allowing the manager class to not to be initialized until the first time it will be used. Singleton pattern is when only one instance of the class is needed in the whole project. And it is a good choice whenever you need a global access.

c. UML and Code Snippet

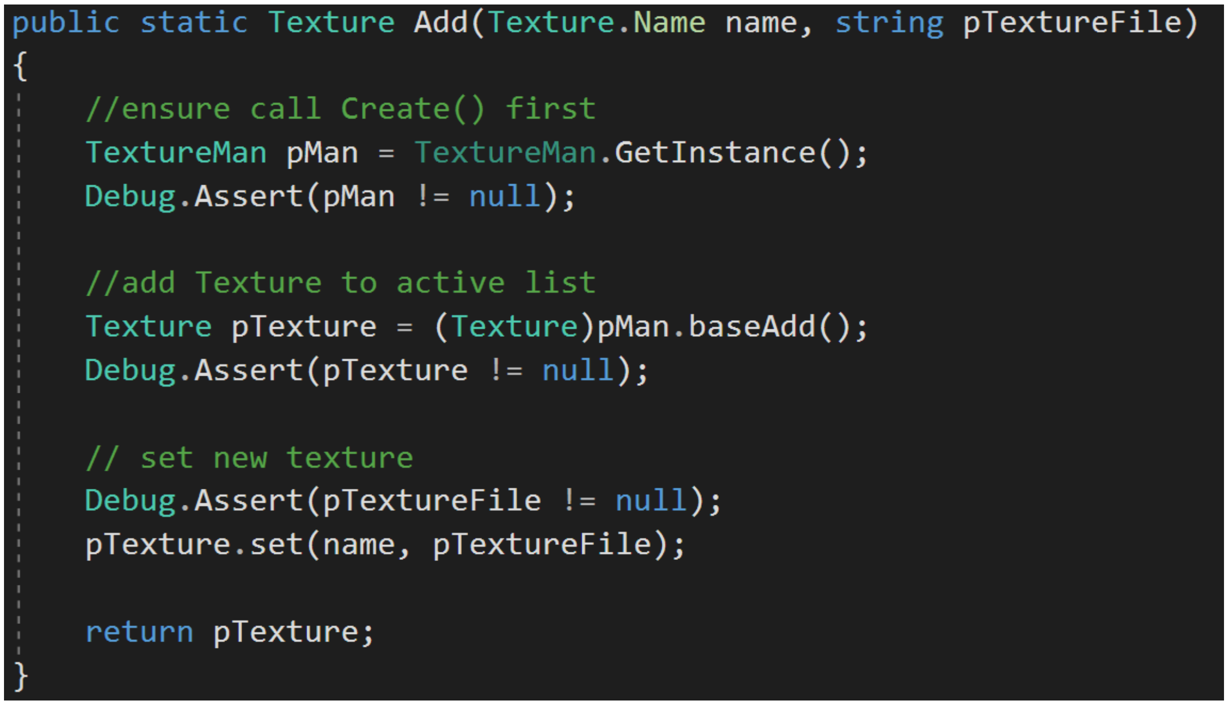




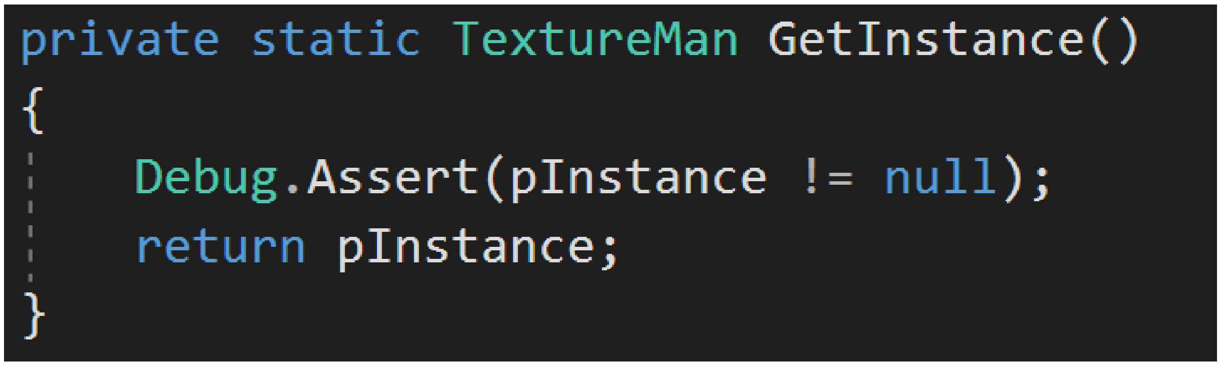
```
public class TextureMan : TextureMan_MLink
{
    //----------------------------------------------------------------------
    // Constructor
    //----------------------------------------------------------------------
    1 reference
    private TextureMan(int reverseNum = 3, int growNum = 1)
        :base()
    {
        // At this point, TextureMan is created, now initialize the reserve list
        this.baseInitialize(reverseNum, growNum);

        this.poNodeForCompare = new Texture();
    }
```



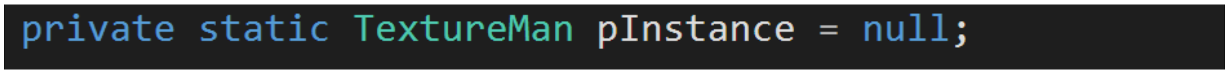
```
public static void Create(int reverseNum = 3, int growNum = 1)
{
    Debug.Assert(reverseNum >= 0);
    Debug.Assert(growNum > 0);

    // ensure the instance of TextureMan has not been created
    Debug.Assert(TextureMan.pInstance == null);

    // initialize the TextureMan
    if(TextureMan.pInstance == null)
    {
        pInstance = new TextureMan(reverseNum, growNum);

        // add NullObject texture to manager, allows find
        TextureMan.Add(Texture.Name.NullObject, "Hotpink.tga");

        // add default texture to active list
        TextureMan.Add(Texture.Name.Default, "Hotpink.tga");
    }

}
```



```
public static Texture Add(Texture.Name name, string pTextureFile)
{
    //ensure call Create() first
    TextureMan pMan = TextureMan.GetInstance();
    Debug.Assert(pMan != null);

    //add Texture to active list
    Texture pTexture = (Texture)pMan.baseAdd();
    Debug.Assert(pTexture != null);

    // set new texture
    Debug.Assert(pTextureFile != null);
    pTexture.set(name, pTextureFile);

    return pTexture;
}
```



```
private static TextureMan GetInstance()
{
    Debug.Assert(pInstance != null);
    return pInstance;
}
```



```
private static TextureMan pInstance = null;
```

d. Pattern Usage

The code snippet shows the texture manager with Singleton pattern, I made the constructor and GetInstance() methods private. It enforced client code to invoke the public static method Create() to create an instance of texture manager. Once the Create() method is called, when I use any public static methods of the texture manager class, I will call GetInstance() to get the instance of the manager inside them first. The Singleton pattern is used in the following management systems: collision pairs, font, glyph, game object, image, input, scene, ship, sound, game sprite, proxy sprite, sprite batch, texture and timer. These managers are used with singleton pattern to not allow multiple manager instances throughout the game. So, at the beginning of the LoadContent() in Game.cs I initialize these managers by calling the public static Create().

Iterator

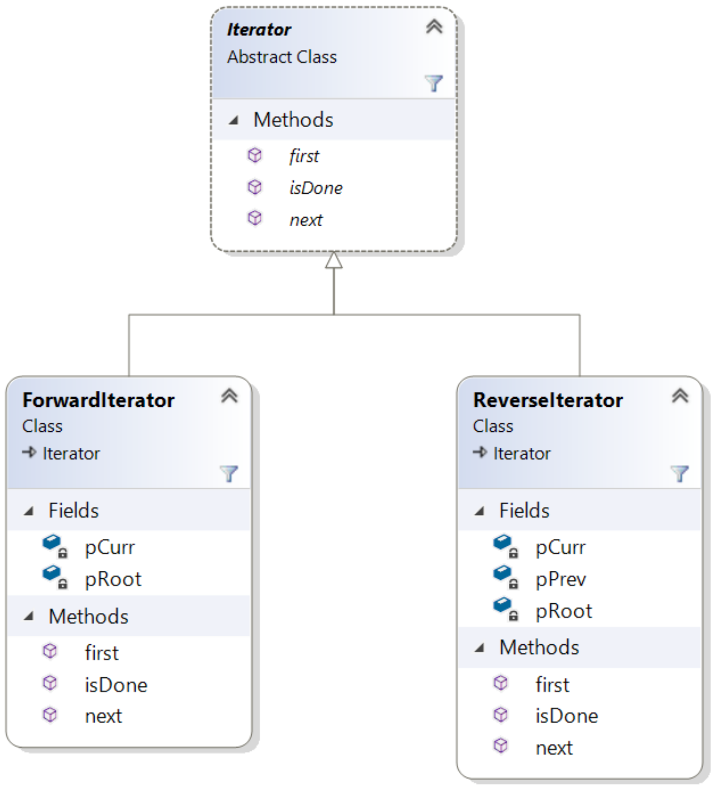
a. Problem

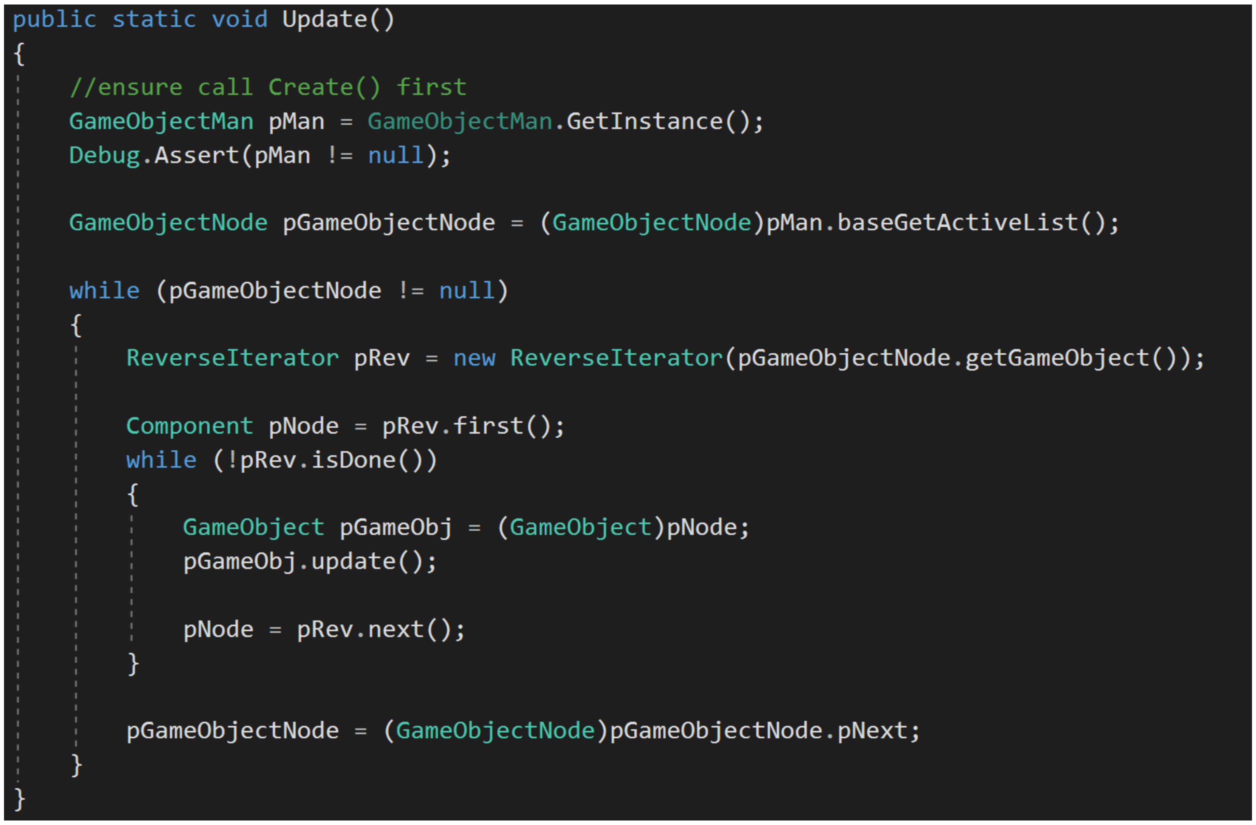
We did not use the built in C# collections like arrays or linked lists and created the data structures, like double linked list, tree, by points and design pattern. So, we have to implement own way to go through the structures. In order to iterate through these data structures, I used the Iterator pattern.

b. Pattern Intent

The Iterator pattern provides a way to access elements of the data structures that we created and encapsulates the underlying representation. It also can traverse on the iterator object, which can simplify the aggregate interface and have the same functionality. An abstract Iterator class will take the collection and create an iterator. In the iterator, we have first(), isDone(), next() to help to go through the collection. It provides an abstraction that makes it possible to decouple collection classes and algorithms

c. UML and Code Snippet





```
public static void Update()
{
    //ensure call Create() first
    GameObjectMan pMan = GameObjectMan.GetInstance();
    Debug.Assert(pMan != null);

    GameObjectNode pGameObjectNode = (GameObjectNode)pMan.baseGetActiveList();

    while (pGameObjectNode != null)
    {
        ReverseIterator pRev = new ReverseIterator(pGameObjectNode.getGameObject());

        Component pNode = pRev.first();
        while (!pRev.isDone())
        {
            GameObject pGameObj = (GameObject)pNode;
            pGameObj.update();

            pNode = pRev.next();
        }

        pGameObjectNode = (GameObjectNode)pGameObjectNode.pNext;
    }
}
```

d. Pattern Usage

In the project, I use the Iterator pattern when I go through the game object manager to update each game object. I create a reverse iterator for game object collections which is based on the forward iterator. In the reverse iterator, we use first() to get the last object in each game object collection, then use next() to go back to the root object. This way is more efficient to find and operate the objects, like update or compare. I use isDone() to check whether there still exists any object in the game object collection. The Iterator pattern promotes to “full object status” the traversal of a collection and provides polymorphic traversal.

Factory

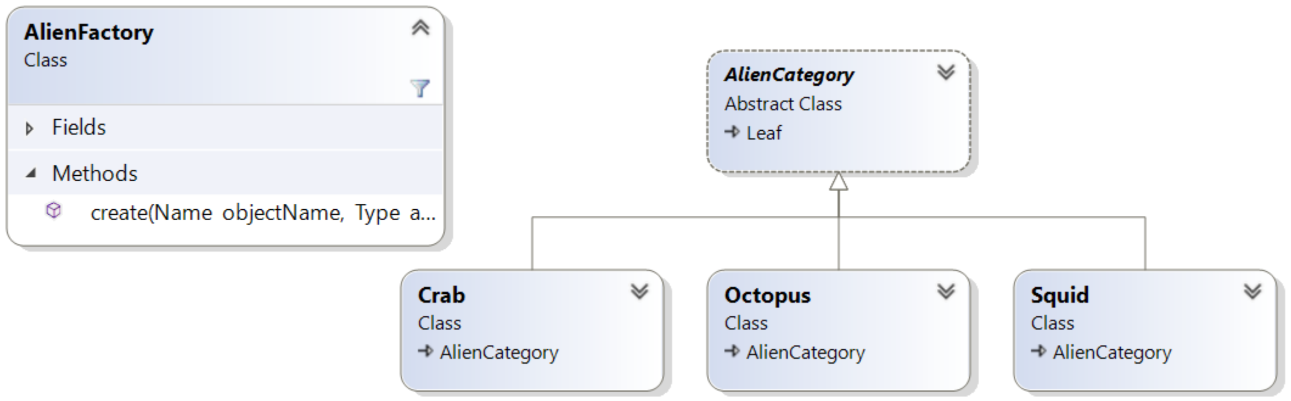
a. Problem

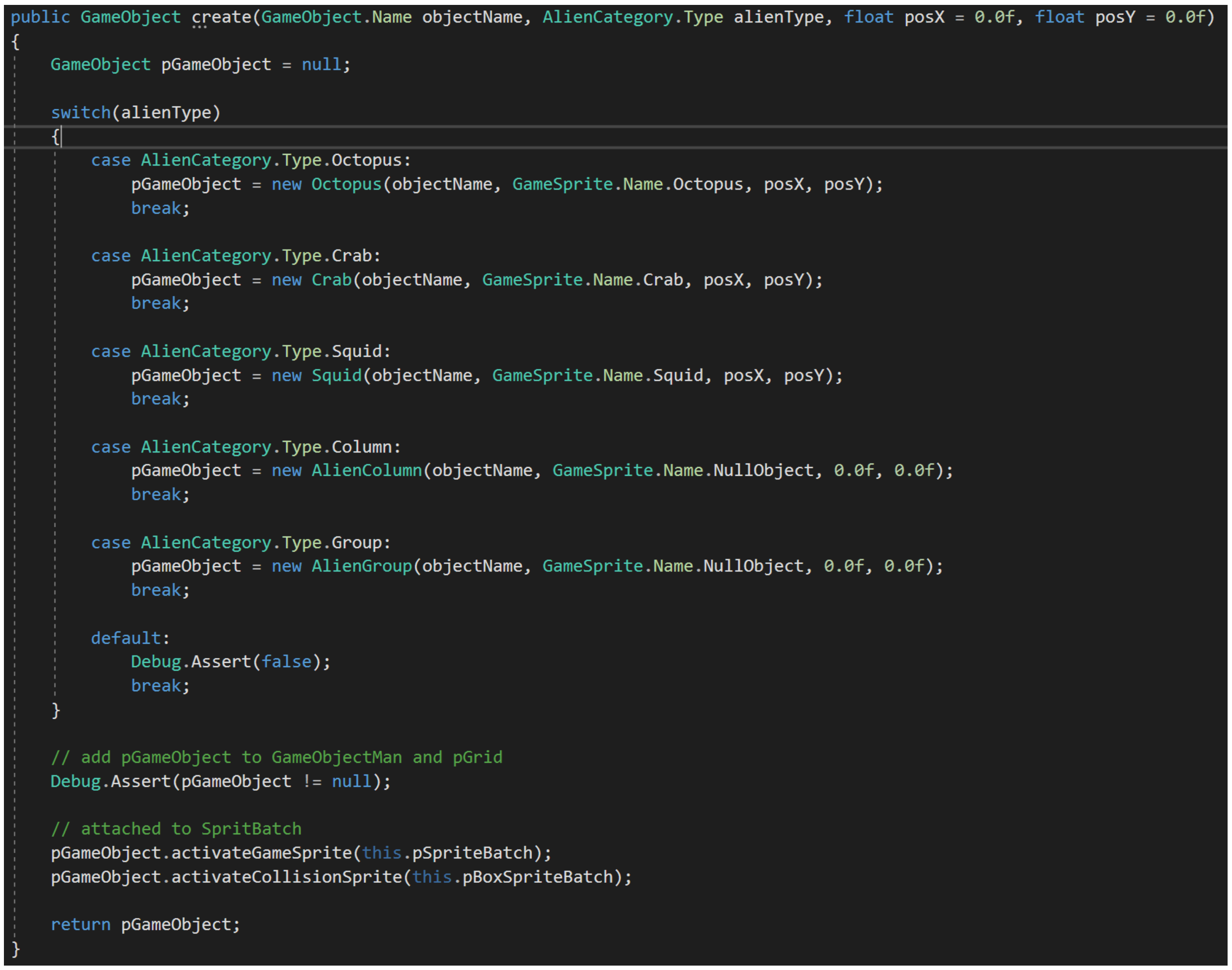
I want to create 55 alien sprites in a 5*11 grid in specific location and create 4 shields which consist of many bricks in a particular order. In order to create these objects more convenient and encapsulate the detail of them, I am going to use Factory Pattern to solve this problem.

b. Pattern Intent

The Factory pattern solves the design problem of creating object instantiation. The Factory method design pattern solves this problem by defining a separate method create() method to create objects. According to the parameters, the create() method will create specific instantiation without expose any details to the client.

c. UML and Code Snippet





```
public GameObject create(GameObject.Name objectName, AlienCategory.Type alienType, float posX = 0.0f, float posY = 0.0f)
{
    GameObject pGameObject = null;

    switch(alienType)
    {
        case AlienCategory.Type.Octopus:
            pGameObject = new Octopus(objectName, GameSprite.Name.Octopus, posX, posY);
            break;

        case AlienCategory.Type.Crab:
            pGameObject = new Crab(objectName, GameSprite.Name.Crab, posX, posY);
            break;

        case AlienCategory.Type.Squid:
            pGameObject = new Squid(objectName, GameSprite.Name.Squid, posX, posY);
            break;

        case AlienCategory.Type.Column:
            pGameObject = new AlienColumn(objectName, GameSprite.Name.NullObject, 0.0f, 0.0f);
            break;

        case AlienCategory.Type.Group:
            pGameObject = new AlienGroup(objectName, GameSprite.Name.NullObject, 0.0f, 0.0f);
            break;

        default:
            Debug.Assert(false);
            break;
    }

    // add pGameObject to GameObjectMan and pGrid
    Debug.Assert(pGameObject != null);

    // attached to SpritBatch
    pGameObject.activateGameSprite(this.pSpriteBatch);
    pGameObject.activateCollisionSprite(this.pBoxSpriteBatch);

    return pGameObject;
}
```

d. Pattern Usage

In the project, I pass an alien type to the create() method and determine the position of the aliens. Then I generate the specific alien in the create() method and return the alien instantiation to the client. The client does not know how the alien is created. As you can see, In the create() method of alien factory, when I create the alien instantiation. I add the instantiation to the alien sprite batch and box sprite batch, then I return the instantiation to the client. The client does not know I put the instantiation to these two sprite batches. So, I encapsulate the details to clients.

Composite

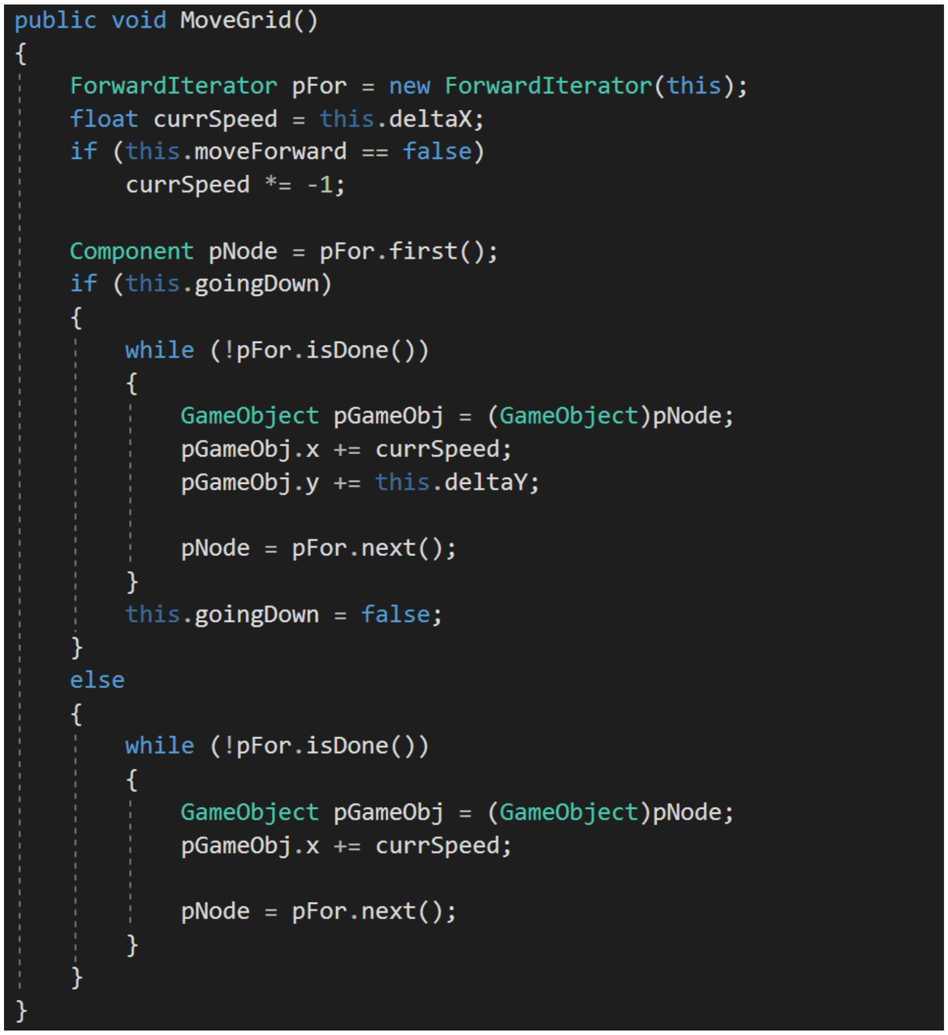
a. Problem

I have a problem of moving all the alien sprites together from right to left at the screen. And once the aliens hit the right or left boundaries, aliens will change their direction. I find it is difficult to control the movement of every alien. So, in order to solve this problem, the structure of all the aliens is very important. I decide to use Composite pattern to handle this problem and create the structure of alien to a tree.

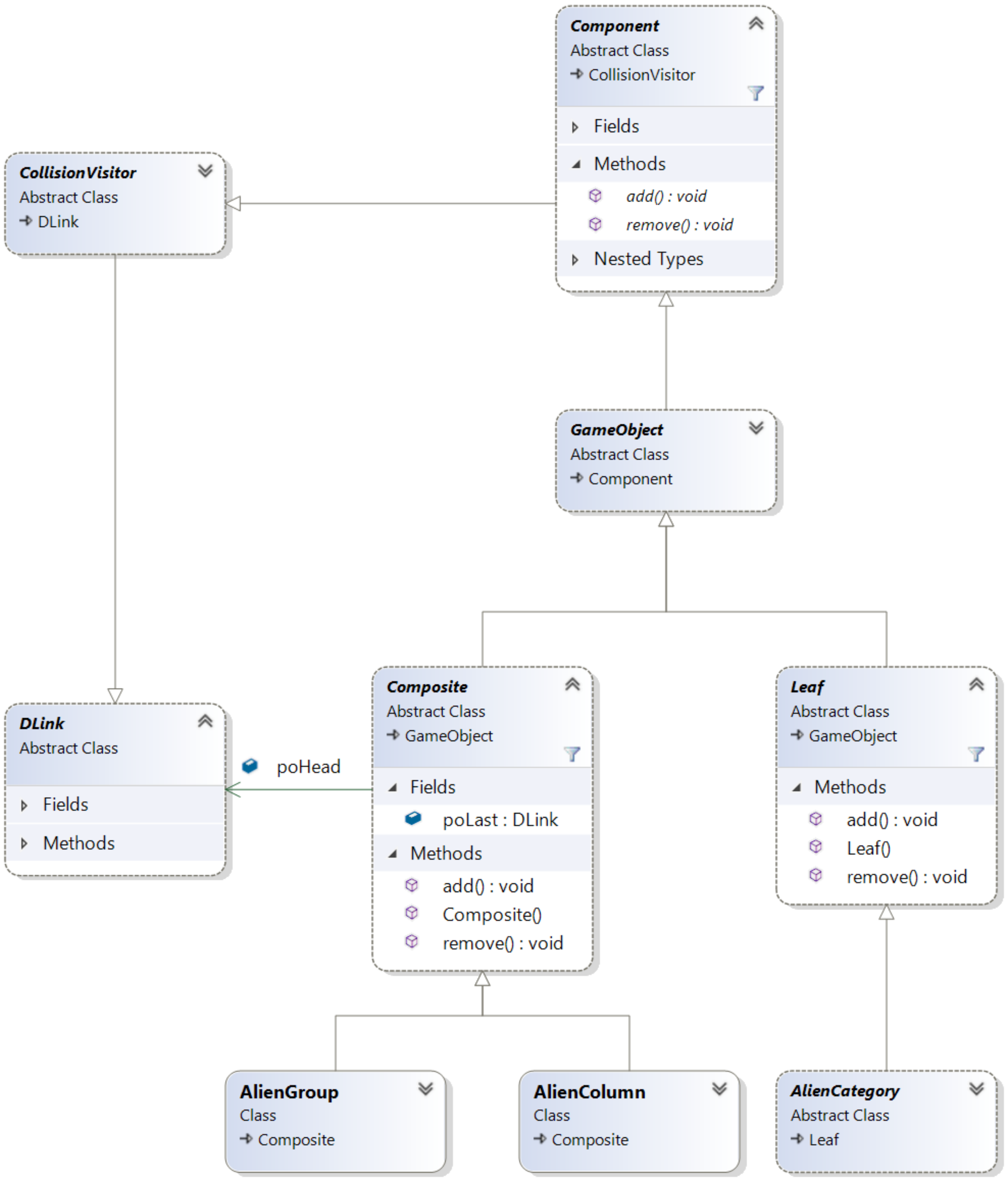
b. Pattern Intent

The Composite pattern is used to composite objects into tree structure to represent part-whole hierarchies. It lets clients treat individual objects(leaves) and composite objects as the same thing.

c. UML and Code Snippet



```
public void MoveGrid()
{
    ForwardIterator pFor = new ForwardIterator(this);
    float currSpeed = this.deltaX;
    if (this.moveForward == false)
        currSpeed *= -1;

    Component pNode = pFor.first();
    if (this.goingDown)
    {
        while (!pFor.isDone())
        {
            GameObject pGameObj = (GameObject)pNode;
            pGameObj.x += currSpeed;
            pGameObj.y += this.deltaY;

            pNode = pFor.next();
        }
        this.goingDown = false;
    }
    else
    {
        while (!pFor.isDone())
        {
            GameObject pGameObj = (GameObject)pNode;
            pGameObj.x += currSpeed;

            pNode = pFor.next();
        }
    }
}
```



d. Pattern Usage

We define a Component class which is an abstract class. The Component class is used to let Composite class and leaf class inheritance. The Composite object has a point to a list of Component object and has an add() method which is used to add composites and leaves to it. So that the structure of a Composite object is like a tree. The leaves hold the actual data and the composites are just placeholder. Because both composites and leaves are derived class of component, we treat them uniformly in high level. In the project, Component class is a derived class of DLink class and Composite class has a point to the DLink. Alien Group class and Alien Column class are derived class of composite class. They used to store the aliens as a tree. Alien Category class is the derived class of leaf class and it is also the base class of actual aliens, including Crab, Octopus and Squid class. I create an alien group object and 11 alien column objects via alien factory. Then I create 55 actual alien objects and add 5 of them to each alien column object in a specific order. Finally, I add these alien column objects to the alien group object. So far, the alien group object is a hierarchical tree and contain all the aliens. I only need to control how the alien group move. One more things to do is that I use Iterator pattern to create an iterator to go through the alien group and move all the aliens. So, when I call MoveGrid() method, all the aliens in the alien group will do the same operation. Another big usage of Composite pattern is in the collision system which I will discuss below at the Visitor pattern.

Proxy

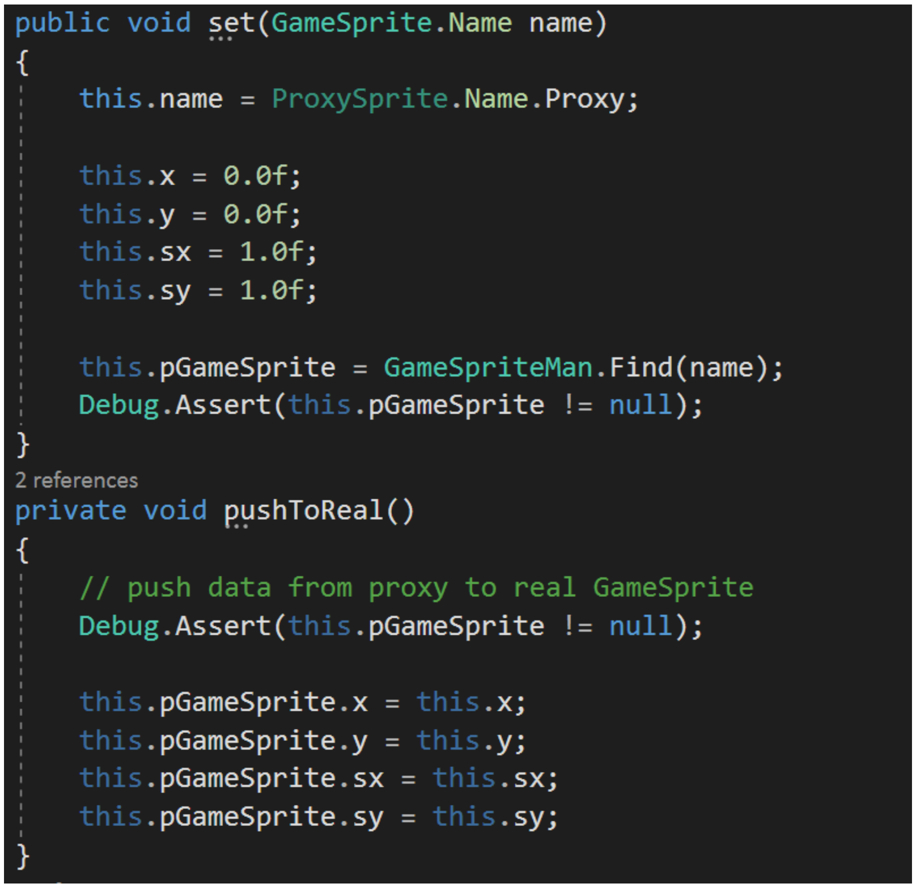
a. Problem

I need a group of 55 aliens consist of 11 squids, 22 crabs and 22 octopus and need 4 shields consist of several different types of bricks. Each squid, crab and octopus should share the same image and texture. So, I decide to use proxy pattern to solve this problem. For shields problem, it should be the same solution.

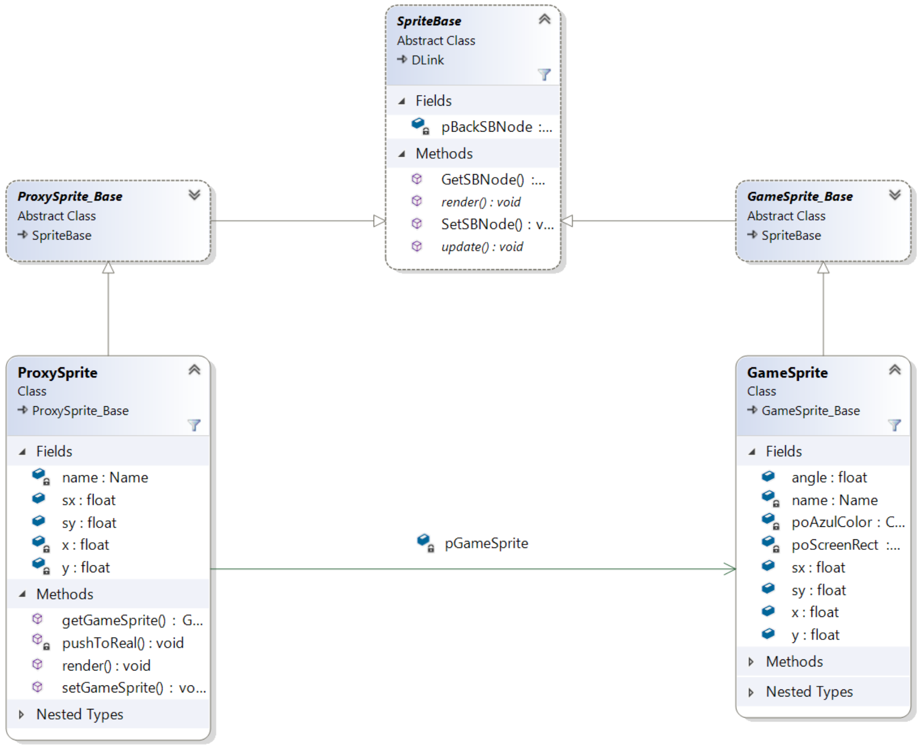
b. Pattern Intent

A proxy is a wrapper or agent that is being called to access the real serving object behind the scenes. The Proxy pattern will break an object into 2 parts, a commonly used part and a unique part. The commonly part will be used to a real class. The data in the unique part will be pushed to the commonly part. The proxy class and the real class are derived class of the same base class. Using the Proxy pattern can easily provide additional logic to the real object. The proxy object provides a placeholder for another object to control access to it. Proxy objects reuse the existing objects.

c. UML and Code Snippet



```
public void set(GameSprite.Name name)
{
    this.name = ProxySprite.Name.Proxy;

    this.x = 0.0f;
    this.y = 0.0f;
    this.sx = 1.0f;
    this.sy = 1.0f;

    this.pGameSprite = GameSpriteMan.Find(name);
    Debug.Assert(this.pGameSprite != null);
}
2 references
private void pushToReal()
{
    // push data from proxy to real GameSprite
    Debug.Assert(this.pGameSprite != null);

    this.pGameSprite.x = this.x;
    this.pGameSprite.y = this.y;
    this.pGameSprite.sx = this.sx;
    this.pGameSprite.sy = this.sy;
}
```



d. Pattern Usage

A proxy points to a full-service object and it will push part of the variables, overwriting the existing variables. Then it delegates the call on the full-service object. The proxy should have the same functionality as the full-service object because they extend from the same base class. In the project, a proxy sprite points to a game sprite which is a full-service object. In the proxy object, it has a pushToReal() which is used to push position information(x and y) to the game sprite object. In the render() of proxy sprite, it will call the render() of game sprite which does the real render work. I reuse these game sprites and render them with different position by the proxy pattern.

Command

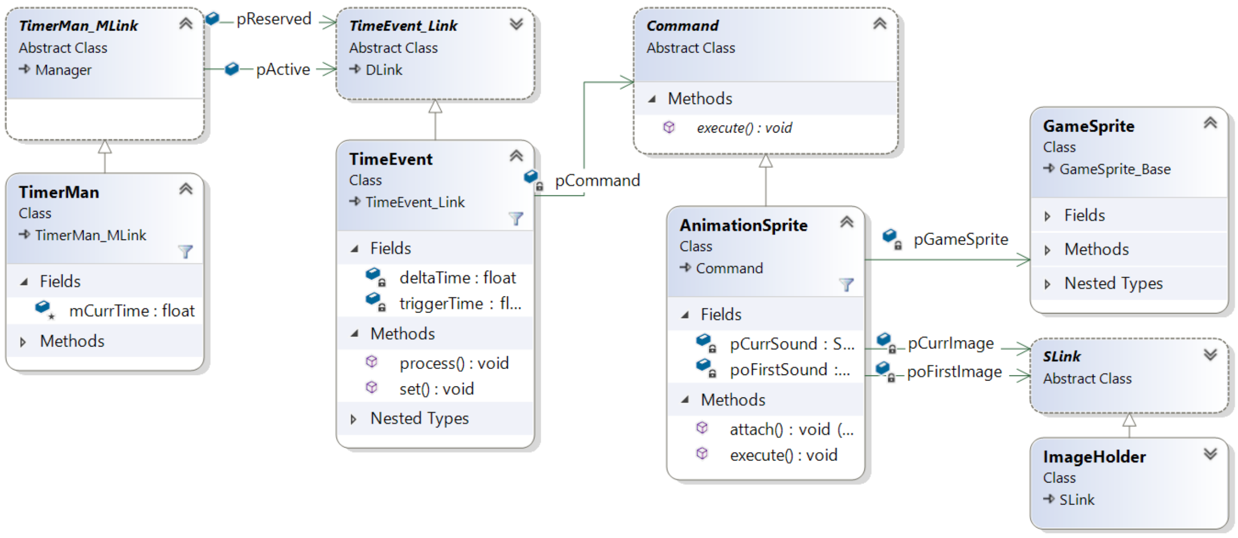
a. Problem

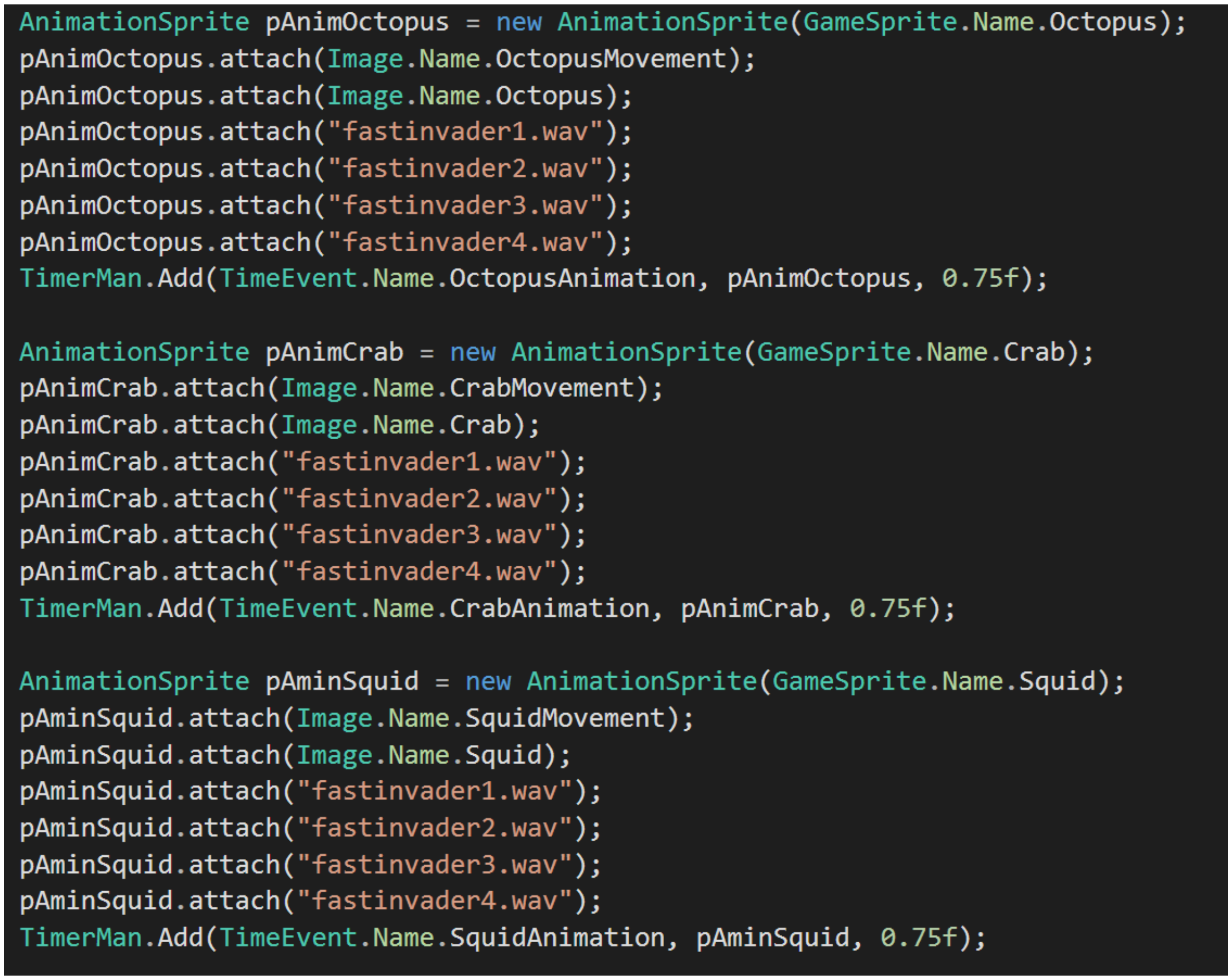
I have a problem about the animation of game sprites to switch between 2 images in a certain time gap to simulate the aliens marching. I need to add a sequence of events in a specific order based on time to do the animation. In order to solve this problem, I will use Command pattern.

b. Pattern Intent

The Command pattern provides an easy way to construct general components without knowing the class of the method and the parameters of the method. The Command pattern is used to encapsulate all information needed to perform an action or trigger an event at a later time. It allows the parameterization of clients with different requests and allows saving the requests in a queue.

c. UML and Code Snippet





```
AnimationSprite pAnimOctopus = new AnimationSprite(GameSprite.Name.Octopus);
pAnimOctopus.attach(Image.Name.OctopusMovement);
pAnimOctopus.attach(Image.Name.Octopus);
pAnimOctopus.attach("fastinvader1.wav");
pAnimOctopus.attach("fastinvader2.wav");
pAnimOctopus.attach("fastinvader3.wav");
pAnimOctopus.attach("fastinvader4.wav");
TimerMan.Add(TimeEvent.Name.OctopusAnimation, pAnimOctopus, 0.75f);

AnimationSprite pAnimCrab = new AnimationSprite(GameSprite.Name.Crab);
pAnimCrab.attach(Image.Name.CrabMovement);
pAnimCrab.attach(Image.Name.Crab);
pAnimCrab.attach("fastinvader1.wav");
pAnimCrab.attach("fastinvader2.wav");
pAnimCrab.attach("fastinvader3.wav");
pAnimCrab.attach("fastinvader4.wav");
TimerMan.Add(TimeEvent.Name.CrabAnimation, pAnimCrab, 0.75f);

AnimationSprite pAminSquid = new AnimationSprite(GameSprite.Name.Squid);
pAminSquid.attach(Image.Name.SquidMovement);
pAminSquid.attach(Image.Name.Squid);
pAminSquid.attach("fastinvader1.wav");
pAminSquid.attach("fastinvader2.wav");
pAminSquid.attach("fastinvader3.wav");
pAminSquid.attach("fastinvader4.wav");
TimerMan.Add(TimeEvent.Name.SquidAnimation, pAminSquid, 0.75f);
```

d. Pattern Usage

The Command class is an abstract class and it owns an abstract method execute(). Every concrete command class has to implement this function. The client will ask for a command to be executed. The invoker will take the command, encapsulate it and push it to the queue. When the concrete command class that is charged with the requested command, it will send the result to the receiver. In the project, time event has a point of command. Inside the time event, I use process() method to call the execute() in the command class. And in every execute() method, I add the time event itself back to the priority queue. The execute() method switch images according to an absolution time.

Visitor

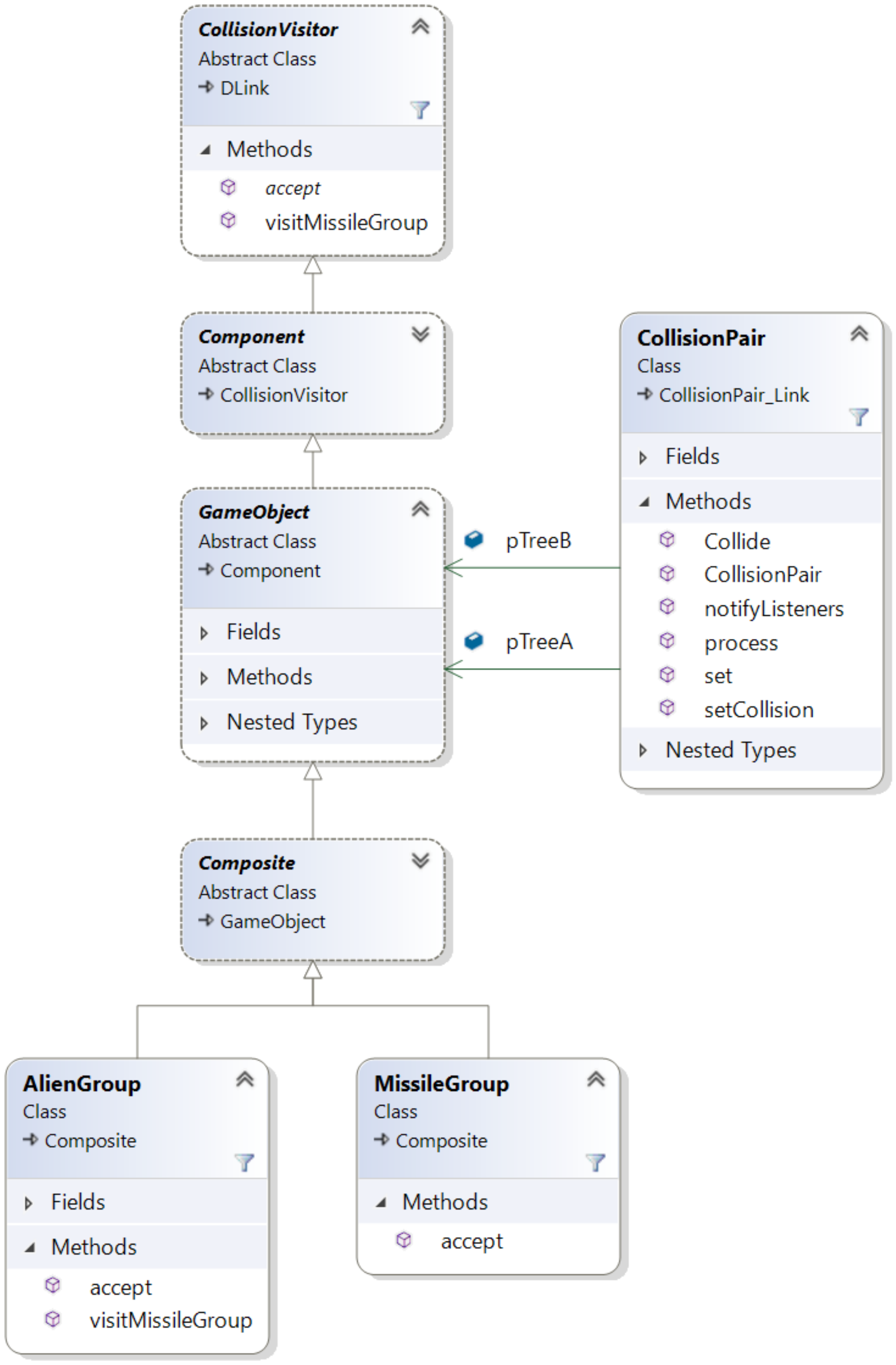
a. Problem

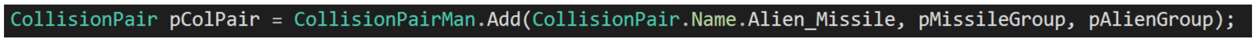
There are many different types of collisions between different objects in the project. For example, when a missile shoots an alien. I need to find an easy and efficient way to discriminate between objects that take part in a collision. To solve this problem, I decide to use Visitor pattern.

b. Pattern Intent

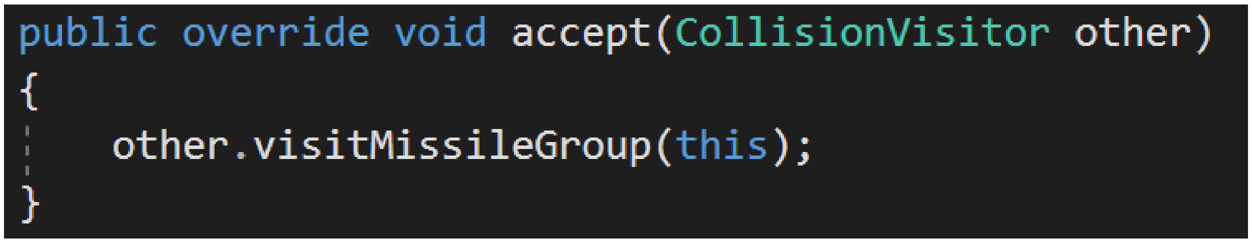
The Visitor pattern is a way to separate the algorithm from an object structure on which it operates. I add virtual functions to classes without modify the classes themselves. I create several visitor classes that implement the virtual functions that they need. The visitor takes the instance reference as input and implements the goal through double dispatch.

c. UML and Code Snippet

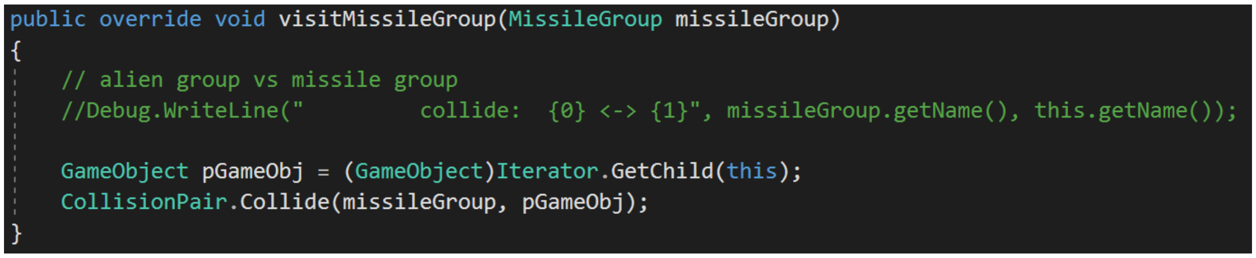




```
CollisionPair pColPair = CollisionPairMan.Add(CollisionPair.Name.Alien_Missile, pMissileGroup, pAlienGroup);
```



```
public override void accept(CollisionVisitor other)
{
    other.visitMissileGroup(this);
}
```



```
public override void visitMissileGroup(MissileGroup missileGroup)
{
    // alien group vs missile group
    //Debug.WriteLine("         collide:  {0} <-> {1}", missileGroup.getName(), this.getName());

    GameObject pGameObj = (GameObject)Iterator.GetChild(this);
    CollisionPair.Collide(missileGroup, pGameObj);
}
```

d. Pattern Usage

In the project, I create an abstract Collision Visitor class as a base class of the Game Object class. In the Collision Visitor class, I define an abstract method accept() and several virtual functions which are used to handle the collision between objects. As the UML and code snippet shows, I create a collision pair between the Alien Group object and Missile Group object. I establish the Visitor pattern A.accept(B). So, Missile Group is A and Alien Group is B. When they collide with each other, Missile Group calls its accept() method with passing the Alien Group object. Then, inside the accept() method, the Alien Group object will call visitMissileGroup() with the Missile Group object as a parameter. Then in visitMissileGroup() method, I will do further operation. The is the progress about how to discriminate the collided objects.

Observer

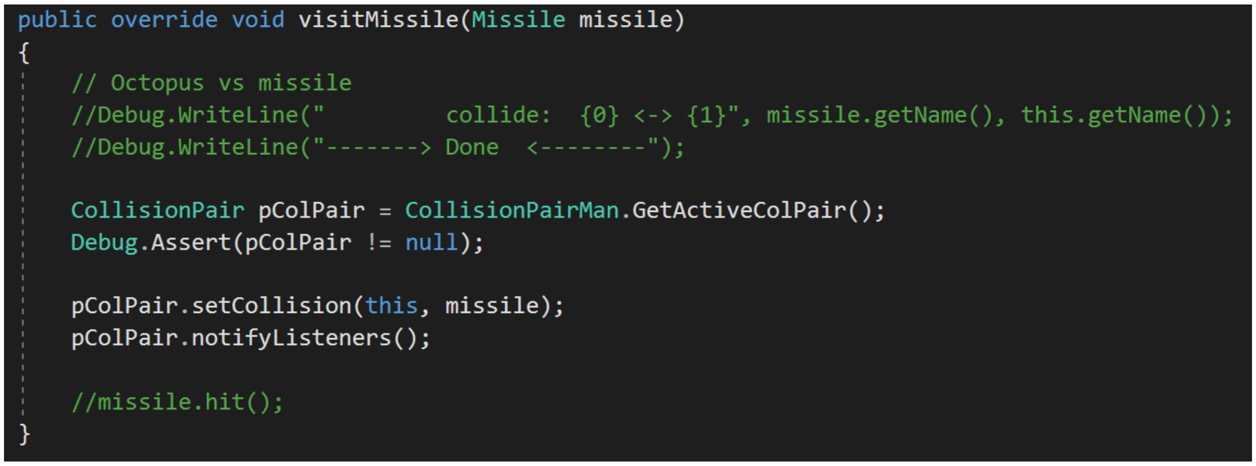
a. Problem

I use Visitor pattern to solve how to discriminate the objects between collision. Then, the next problem will be how to do operation on a specific collision. How to notify some objects to do some action. I will use Observer pattern to solve this problem.

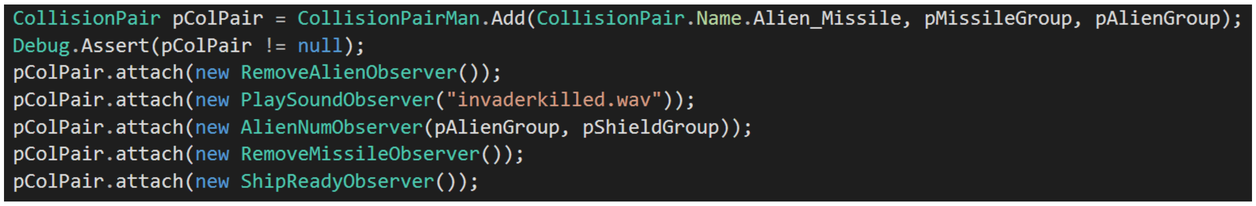
b. Pattern Intent

The Observer pattern defines a one-to-many dependency between objects. When one object changes the state, all of its dependents will be notified and do some action automatically. These observers are observing a specific subject. The actions will be taken upon state change to a list of observers

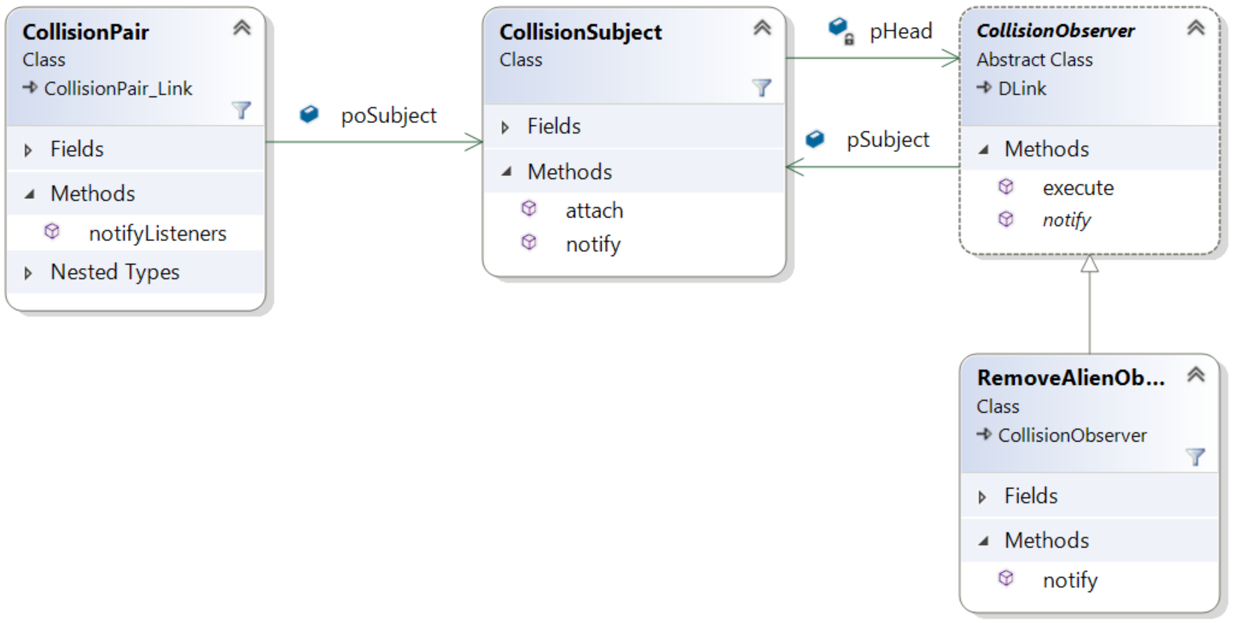
c. UML and Code Snippet



```
public override void visitMissile(Missile missile)
{
    // Octopus vs missile
    //Debug.WriteLine("         collide:  {0} <-> {1}", missile.getName(), this.getName());
    //Debug.WriteLine("-------> Done  <--------");

    CollisionPair pColPair = CollisionPairMan.GetActiveColPair();
    Debug.Assert(pColPair != null);

    pColPair.setCollision(this, missile);
    pColPair.notifyListeners();

    //missile.hit();
}
```



```
CollisionPair pColPair = CollisionPairMan.Add(CollisionPair.Name.Alien_Missile, pMissileGroup, pAlienGroup);
Debug.Assert(pColPair != null);
pColPair.attach(new RemoveAlienObserver());
pColPair.attach(new PlaySoundObserver("invaderkilled.wav"));
pColPair.attach(new AlienNumObserver(pAlienGroup, pShieldGroup));
pColPair.attach(new RemoveMissileObserver());
pColPair.attach(new ShipReadyObserver());
```



d. Pattern Usage

All the observers will inherit from an abstract Observer class and implement the notify() method which do the actual operation automatically. In the project, when we do further collision between alien and missile, we will find the missile has a collision with a specific alien object. When this collision happens, we invoke notifyListeners() method form the collision pair object. The method will notify all the observers that are attached with the collision pair's subject. Every Collision subject has a list of observers and each observer has a reference to the collision subject. So, when the missile hit the alien, all the observers that are attached will be called automatically and do the actual operations.

State

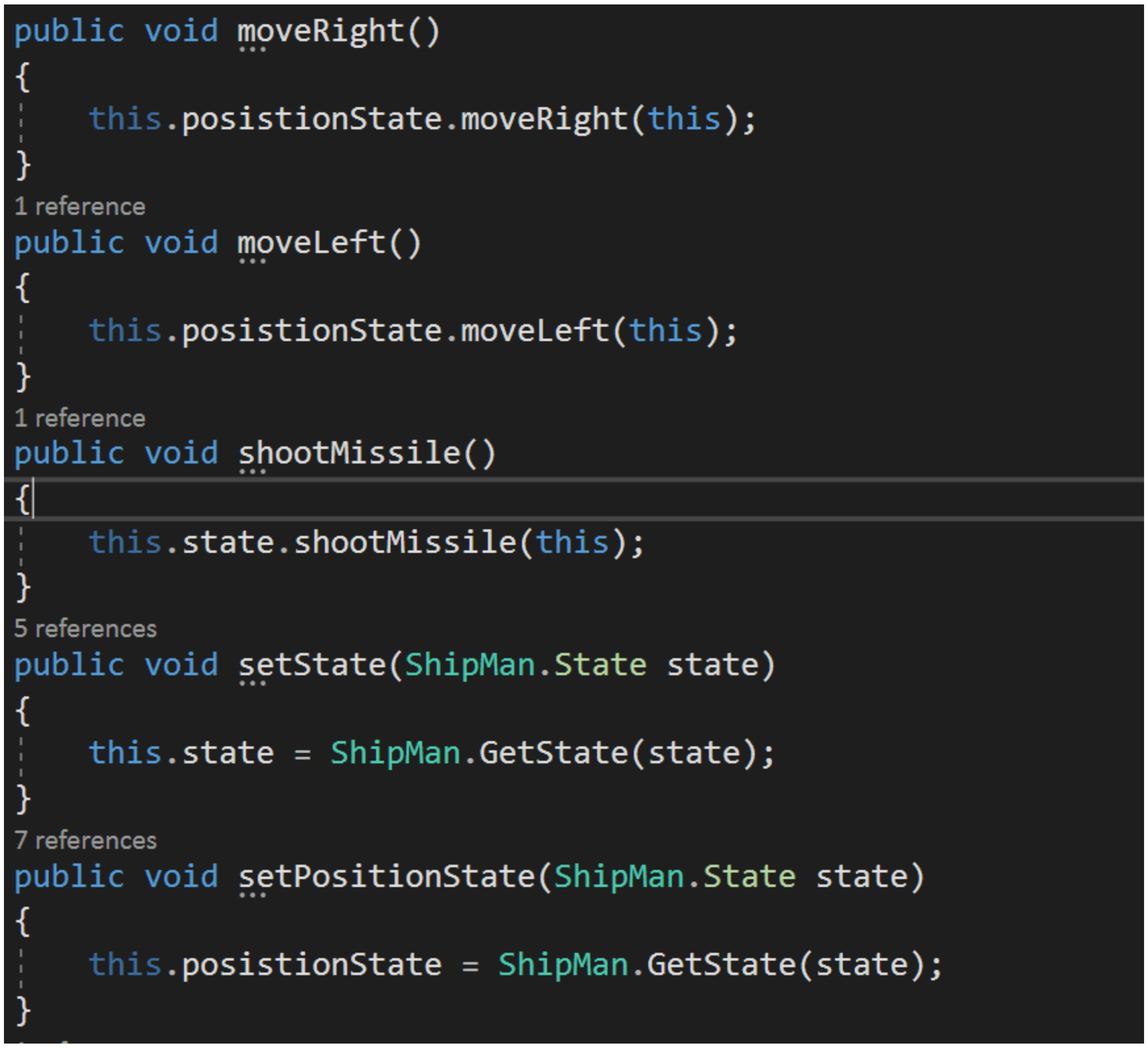
a. Problem

The ship can only one missile at a time and there should exist only one missile on the screen. So, ship cannot shoot missile until last missile is out of screen. I think ship should have different state to control whether it can shoot the missile. To solve this problem, I will use State pattern.

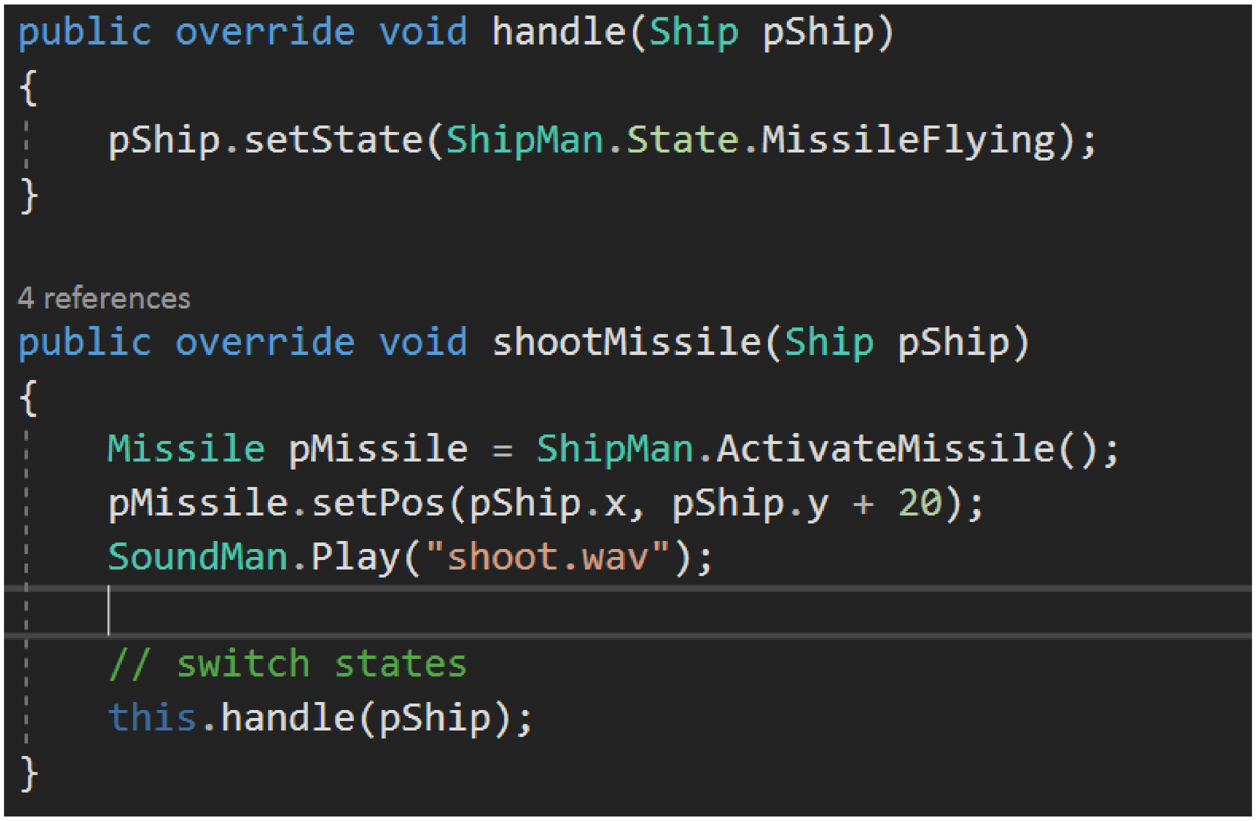
b. Pattern Intent

The state pattern allows the object to alter the action when its state changes. The State pattern represents the states as its own class. I create an abstract state class which contains the type of actions that the object can take between states. Each concrete state class is derived from the abstract state class and implement the abstract methods.

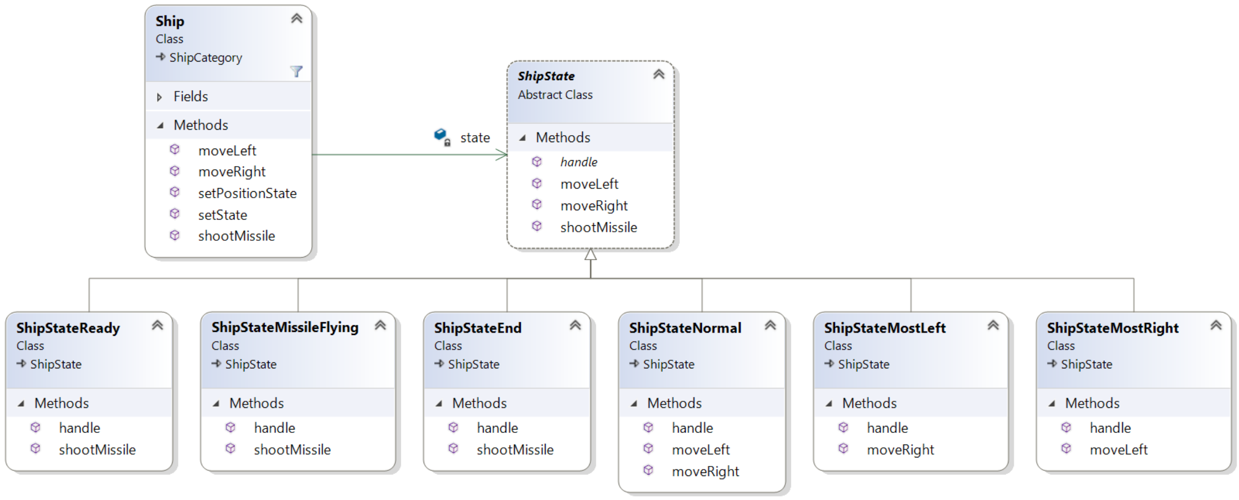
c. UML and Code Snippet



```
public void moveRight()
{
    this.posistionState.moveRight(this);
}
1 reference
public void moveLeft()
{
    this.posistionState.moveLeft(this);
}
1 reference
public void shootMissile()
{
    this.state.shootMissile(this);
}
5 references
public void setState(ShipMan.State state)
{
    this.state = ShipMan.GetState(state);
}
7 references
public void setPositionState(ShipMan.State state)
{
    this.posistionState = ShipMan.GetState(state);
}
```



```
public override void handle(Ship pShip)
{
    pShip.setState(ShipMan.State.MissileFlying);
}

4 references
public override void shootMissile(Ship pShip)
{
    Missile pMissile = ShipMan.ActivateMissile();
    pMissile.setPos(pShip.x, pShip.y + 20);
    SoundMan.Play("shoot.wav");
    
    // switch states
    this.handle(pShip);
}
```



d. Pattern Usage

In the project, Ship has 6 different states. Ready, Missile Flying and End are used to control whether ship can shoot the missile. Normal, Most Left and Most Right are used to control the limitation to move left or right of the ship. All these states are derived class from ShipState class which is an abstract class. The ship has a state which is point to the ShipState class. For a ship object, it can change the state via setState() method and change the position state via setPositonState() method. According to the different state of ship, moveLeft(), moveRight() and shootMissile() method will do different operations.

Strategy

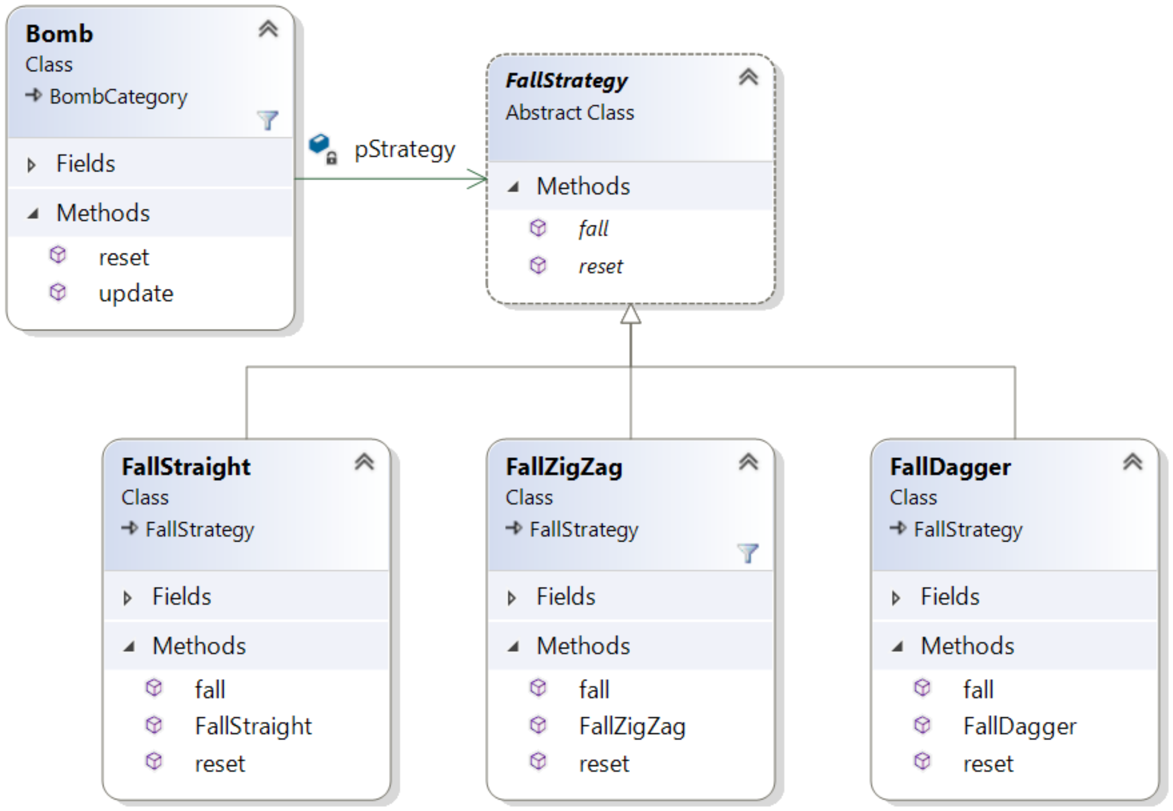
a. Problem

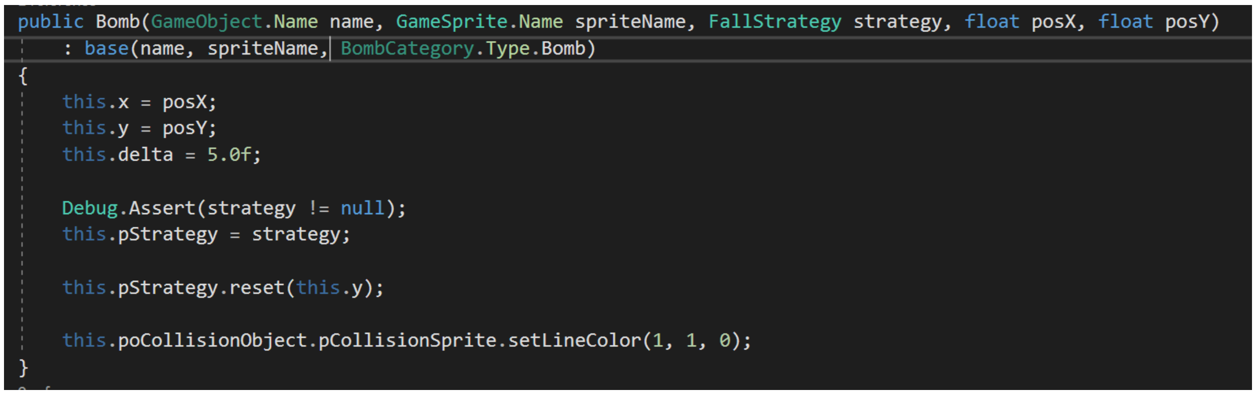
Aliens can drop different types of bombs, a zig zag bomb, a cross bomb and a straight bomb. All of these bombs are drop by aliens however the behavior has a slight different for each of them. I think the difference is the use distinct algorithms. To solve this problem, I will apply Strategy pattern.

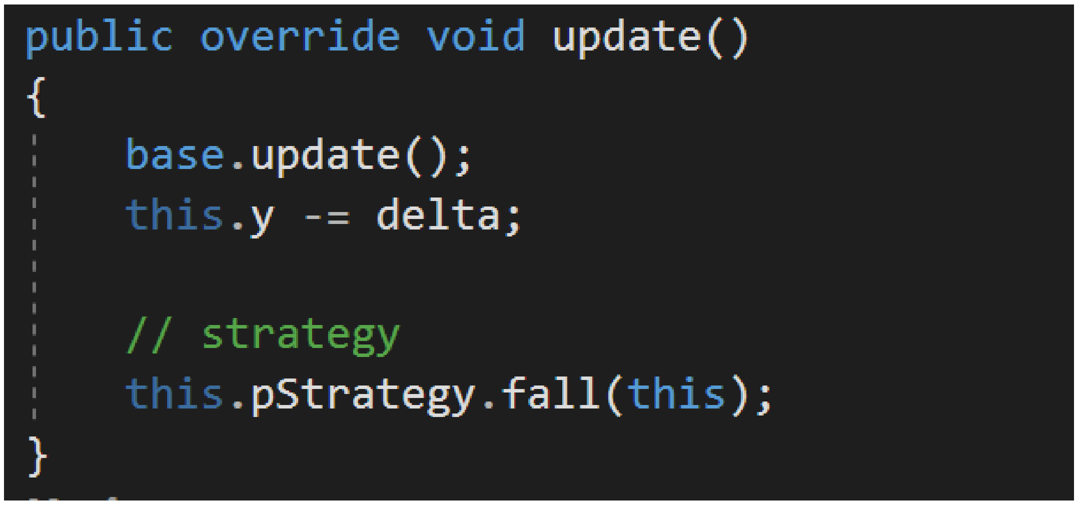
b. Pattern Intent

The Strategy pattern defines several different algorithms. The Strategy pattern encapsulates each algorithm and makes them interchangeable. The Strategy class is an abstract class and has an abstract method that will be implemented by concrete class with a specific behavior.

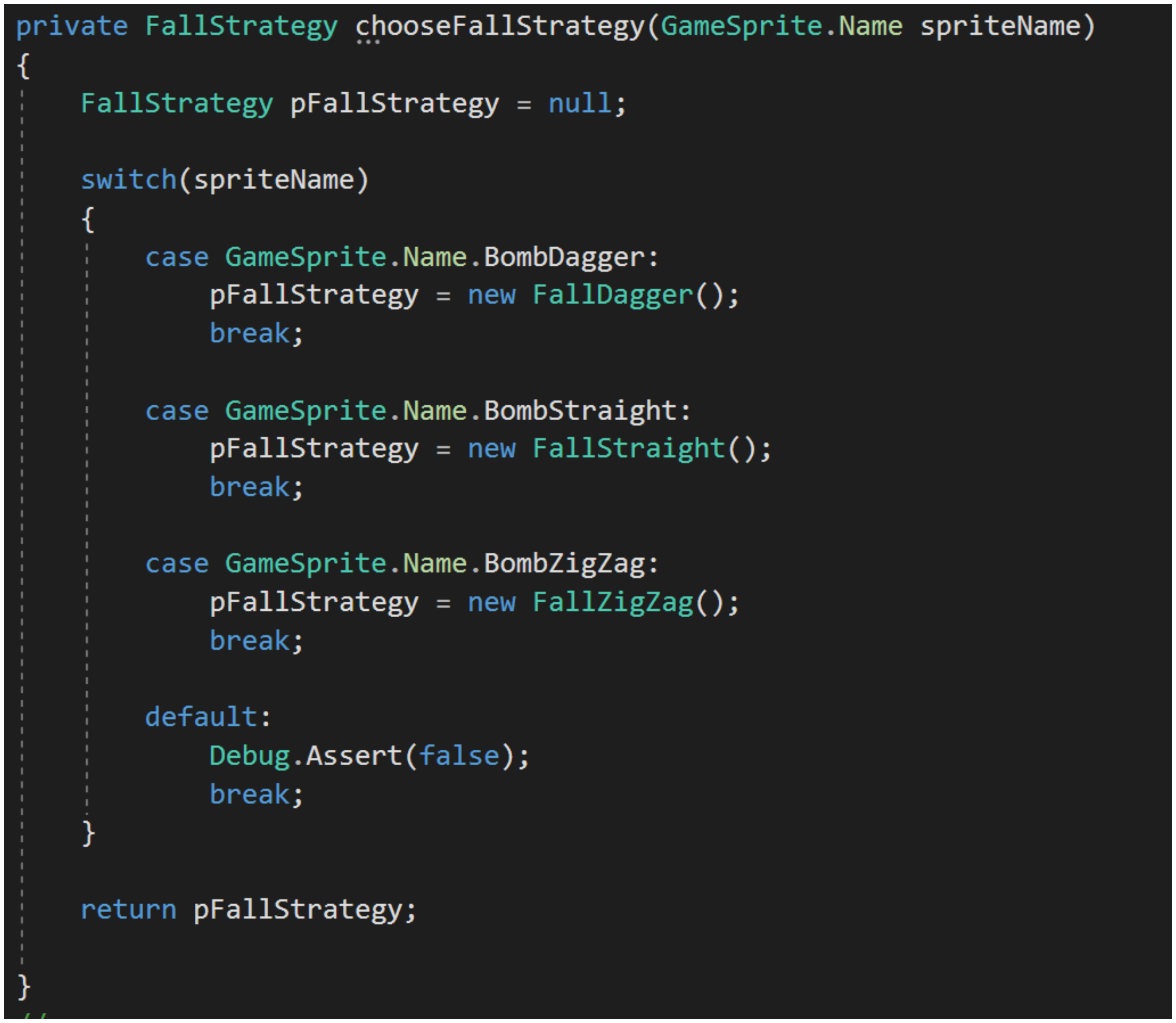
c. UML and Code Snippet




```
public Bomb(GameObject.Name name, GameSprite.Name spriteName, FallStrategy strategy, float posX, float posY)
    : base(name, spriteName, BombCategory.Type.Bomb)
{
    this.x = posX;
    this.y = posY;
    this.delta = 5.0f;

    Debug.Assert(strategy != null);
    this.pStrategy = strategy;

    this.pStrategy.reset(this.y);

    this.poCollisionObject.pCollisionSprite.setLineColor(1, 1, 0);
}
```



```
public override void update()
{
    base.update();
    this.y -= delta;

    // strategy
    this.pStrategy.fall(this);
}
```



```
private FallStrategy chooseFallStrategy(GameSprite.Name spriteName)
{
    FallStrategy pFallStrategy = null;

    switch(spriteName)
    {
        case GameSprite.Name.BombDagger:
            pFallStrategy = new FallDagger();
            break;

        case GameSprite.Name.BombStraight:
            pFallStrategy = new FallStraight();
            break;

        case GameSprite.Name.BombZigZag:
            pFallStrategy = new FallZigZag();
            break;

        default:
            Debug.Assert(false);
            break;
    }

    return pFallStrategy;

}
```

d. Pattern Usage

In the project, I create abstract FallStrategy class with abstract methods: fall() and reset(). FallZigzag, FallStraight and FallDagger are derived class from FallStrategy class. Each of the derived class has a different algorithm to drop the bomb. I pass the gamesprite name to create different type of strategy to drop the bomb.

Summary

This course teaches me a lot. I work on the space invader project almost the whole term. I start form some prototype, then applying design patterns to the project and refactoring the code again and again. Finally, I learn plenty of useful skills for real-time development. I learn not only the usage of design patterns but also the step to overcome the difficult problem form this course. Baby step is really useful at real-time work, sometimes it will give you some idea to figure the problem and make your ideas clearer. I have coded several years. I've formed some poor habits and haven't until recently started actually diving into design patterns. I really think this has helped me to think and find my design problems in a different way.