Design patterns in space invader game

Jing Li

To build the classic game Space Invader using C#, I used different kinds of design patterns to keep my code maintainable, reusable and to make the codes more structured. Here I'm going to explain how I used these patterns to solve problems and why it's a better solution.

In this design paper, I've covered the following design patterns I've come across in the process of implementation:

1, Singleton

2, Factory

3, Proxy

4, Object Pool

5, Composite

6, Strategy

7, State

8, Visitor

9, Observer

10, Command

11, Iterator

12, Null Object

13, Adapter

14, Template

1, Singleton

Singleton pattern is used to make sure there exists only one instance of a certain class. Therefore, we can't create multiple instances of the class, only one can get instantiated. It gives us global access to the instance, which means when we need the instance, we don't need to pass the object we created all around to get the access. Instead, we can use the static methods inside the class for us to make changes or get the reference.

In my space invader game, I used singleton pattern for most of the manager classes. Because I use different managers to keep track of the available items which are on the active linked list. One manager is like the storage place we have, it saves all of the items we get. So for each manager class, we only need one instance to save all of our choices. If we can create more instances, we wouldn't know which instance to go to and may need to pass the reference around to get access to it. Then in our whole project, it will need to pass objects of different managers and need to manually make sure there's only one instance which would make our project messy and problematic.

When we use the singleton pattern in our project, the first thing we do to initialize is to create the instance of each manager.

The following code is how I implemented the Create method:

```
public static void Create(int reserveNum = 3, int reserveGrow = 1)
{
            if (pInstance == null)
            {
                pInstance = new ImageManager(reserveNum, reserveGrow);
            }
}
```

pInstance is the only instance of the class, which is a static type. As the example shows, if someone accidentally use the create method more than once, and trying to make multiple instances, the second time or even third wouldn't do anything.

We're checking whether pInstance is null, if it's null, that happened when we call it for the first time, we're going to create the instance; if it's not, which means the instance has already been created, it's not going to create another instance anymore. In this way, it protects there exists at most one instance of the class.

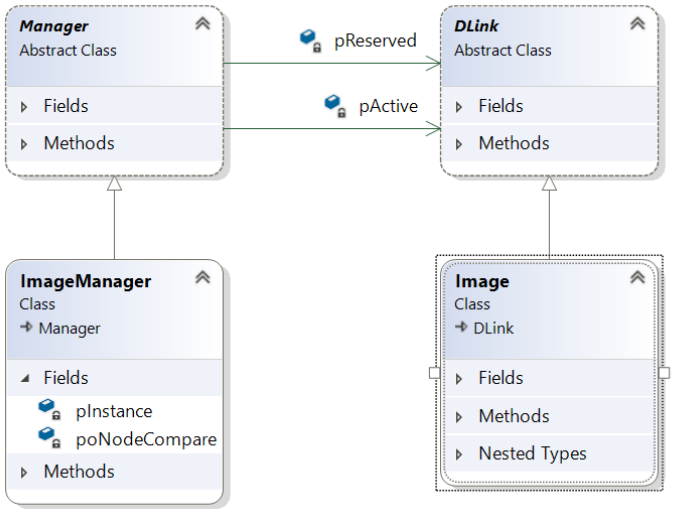
```
private ImageManager(int reserveNum = 3, int reserveGrow = 1)
        : base(reserveNum, reserveGrow)
{
        this.baseRefill();
        this.poNodeCompare = new Image();
}
```

As above, the constructor is private, which means we cannot use this constructor to build the instance of the class, the only way user can create the object is to go through Create method, where it makes sure only one instance as well.

```
private static SpriteManager PrivGetInstance()
{
        return pInstance;
}

public static Sprite Add(Parameter one, parameter two…)
{
       SpriteManager pMan = SpriteManager.PrivGetInstance();
       //do something
}
```

In my version of the game, and when I make a class to be singleton, I make a method called PrivGetInstance, this method is private, which means outside this class, we can't get access it, it protects the data from bad use too. When we have a functionality to do, the static method inside the class comes along. Like above, the Add method, it adds another item to the manager list. When we call the add method, it would add the item for us to the instance. To do that, inside the method Add, it calls the private method PrivGetInstance to modify it.

In every functionality we need, we can make it a static method and then get access to the instance through PrivGetInstance.



In the UML above, ImageManager is of Singleton pattern, we have a private property of that class called pInstance, and that would be the only one instance the class as well.

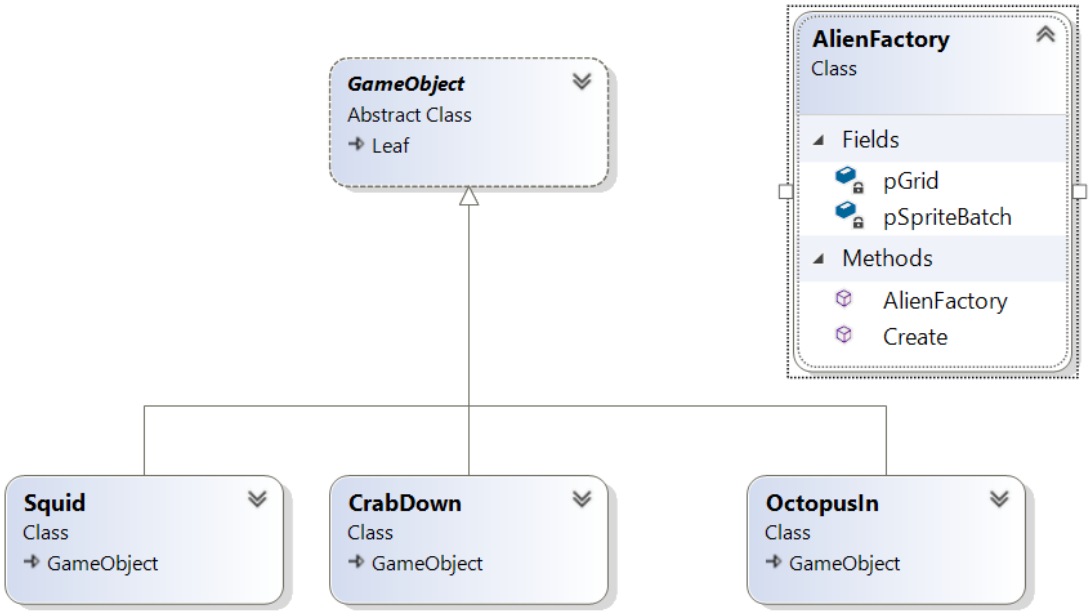
2, Factory

Factory design pattern is like a factory to build up an object depends on its type. How to create the object is separated from the user, so when we are trying to make an object, we don't know how it's created, only the factory class knows the way how to create a class.

The factory pattern can create objects without exposing instantiation logic to its user, and the user can create different types of objects through its common interface.

For example, in my space invader project, we need to create different objects, some of them are of class Squid, some of them are Crab and the others are Octopus class. If we didn't use factory pattern, we need to create different objects by using separate constructors of three different classes. However, if we use the Factory pattern, no matter we want to create Squid, or Crab or Octopus, we can just do create and let the Factory class decide which objects we really want to instantiate.

The UML for the Factory class is as followed:



As the UML above shows, we have a GameObject which is an abstract class, it has three child classes as well. The Alien Factory is used to decide which constructor of the three classes would be used to create the object. So in our main method, we only need to use an AlienFactory object together with its Create method, and the rest of the class instantiation is left to Create method where a switch statement is used based on the name of GameObject.

The Create method inside of AlienFactory is as below:

```
public void Create(GameObject.Name type, float posX, float posY)
{
       GameObject pGameObj = null;

        switch (type)
        {
            case GameObject.Name.SquidOut:
```

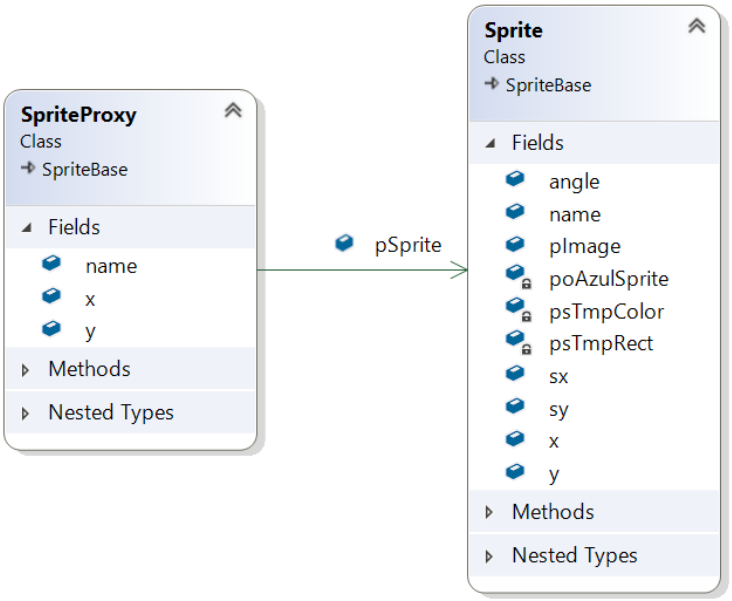
```
                pGameObj = new Squid(GameObject.Name.SquidOut, Sprite.Name.SquidOut,
posX, posY);
                break;

            case GameObject.Name.CrabUp:
                pGameObj = new CrabUp(GameObject.Name.CrabUp, Sprite.Name.CrabUp, posX,
posY);
                break;
              //some other codes
       }
       //some other codes
}
```

From the code above, we can see that in the implementation inside the class, we are using a switch case to decide which type of object to create. If the type of the GameObject is SquidOut, we are going to create a Squid object, if the type of the GameObject is CrabUp, we are going to create a CrabUp object. And the type of the object we are creating is of class GameObject which is the abstract class of all the child classes. Hence polymorphism is used a lot in factory pattern.

3, Proxy

Proxy pattern is very useful when we need to use a lot of costly objects. When the object is really large, and the space is taken is kind of a lot, and also to create hundreds of such objects by using new would definitely make the whole program run much slower. That where proxy pattern come to use. The proxy is the light weight objects whole part of the property of the costly object and holds a reference to it so that we could get refer to the real objects when we need to. It would save a lot of space and time.

In our space invader project, we need to create 55 aliens on the screen. And the sprite class is big, it contains Image, Azul.Rect and lots of other attributes as well. But if we need to create 55 aliens, that means we need to create 55 such objects which is slow and takes much space. Proxy pattern is useful here because we are actually creating 11 squid objects, 22 crab objects and 22 octopus objects. Instead of creating a lot of sprite objects, we can use proxies.



As the UML shows, the sprite proxy has x and y and it keep the reference to sprite class. When we create objects, we're actually creating 55 proxies, and each proxy points to a sprite object. Each proxy object has its own x and y and when we access to the sprite object, we're reusing the same object and update the x and y value in sprite by the x and y in proxy. This way, we could reuse the costly objects and make the program run faster and take less space.

```
AlienFactory AF = new AlienFactory(SpriteBatch.Name.Aliens, pGrid);
for (int i = 0; i < 11; i++)
{
      AF.Create(GameObject.Name.SquidOut, 300.0f + 25 * i, 600.0f);
      AF.Create(GameObject.Name.CrabUp, 300.0f + 25 * i, 575.0f);
      AF.Create(GameObject.Name.CrabUp, 300.0f + 25 * i, 550.0f);
      AF.Create(GameObject.Name.OctopusOut, 300.0f + 25 * i, 525.0f);
      AF.Create(GameObject.Name.OctopusOut, 300.0f + 25 * i, 500.0f);
}

public void Create(GameObject.Name type, float posX, float posY)
{
       GameObject pGameObj = null;
       switch (type)
       {
              //case for creating new objects
       }
       //some other code
       this.pSpriteBatch.Attach(pGameObj.pProxySprite);
}
```

GameObject holds a reference to the proxy sprite. We have already seen before the Create method for AlienFactory class. Inside the Create method, the last line is important to us. We're attaching the proxy objects to the spriteBatch, so when we update and render the spritebatch, the 55 proxy sprites get drawn on the screen, each of which holds a reference to sprite object we created.

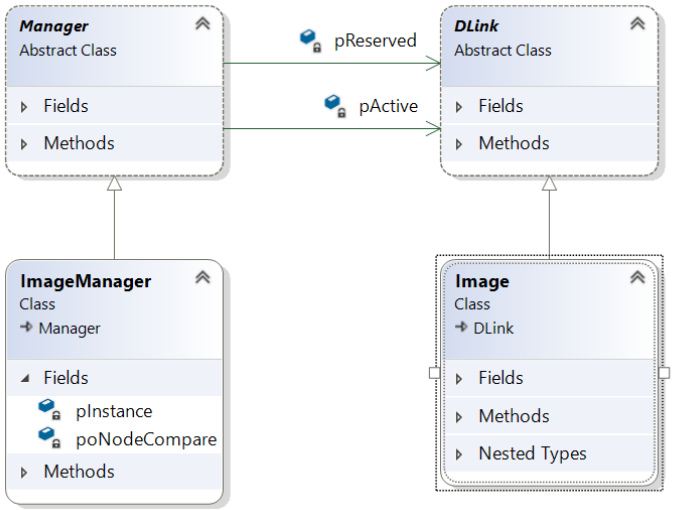
For each proxy object, we used a for loop like above, the x value and y value get assigned differently for those proxy objects, that's why those aliens are on different location, since the x value and y value of proxy would be assigned to the sprite object it points to.

4, Object Pool

It's always costly to create the objects and delete them and recreate them whenever needed. Not to mention if it's in a real time system and in the game design. For the performance perspective, Object pool pattern is really helpful when we need to recreate a large number of expensive objects. It creates a way for us to reuse the objects when it's expensive to create.

For this pattern to work, we will have a pool of objects. When we need an object, we would reach out to the pool and see whether it has some available reusable objects: if it does, we'll reuse that object, while if it doesn't, the pool would then generate one or several more objects and return the newly created one. And how many of new objects to create, it depends on different situations. If the objects are too expensive and we randomly need more of them, we would tend to create fewer objects or probably just one of them. Instead, if we tend to need more of them and they are not that expensive to create, it makes sense to create more at once then keeping calling the make new objects function.

In our Space Invader project, every Manager is like an object pool, to use our previous UML for example:



Here, ImageManager is of singleton pattern as well. We use the Image Manager like a pool to manage those Image objects. We have two pointers, one is pReserved and the other one is pActive. The pReserved pointer points to a doubly linked list, which holds the objects haven't been used, or available reusable objects; while pActive points to the current using objects. So when we want to create a new image object, we can find the one in pReserved list, remove it from the reserved list and then add the object to the active list. Then set the attributed as we want to that object. When we don't want to use one Image object anymore, we can find the object from the active list, remove it from the active list and add it back to the reserved list. So later on, if we want to create another Image object, we can grab the object from reserved list again.

```
protected DLink baseAdd()
{
	if (this.mnumReserved == 0)
	{
		refill(this.growthSize);
	}
	DLink removed = DLink.removeFromFront(ref this.pReserved);
	DLink.addToFront(ref this.pActive, removed);

	this.mNumActive += 1;
	this.mnumReserved -= 1;
	return removed;
}
```

For example, if we look at this baseAdd method in our Manager class, the first thing it did is not just create a specific type of object, instead, it checks its reserve list. If the reserved list doesn't contain any node, it creates some nodes in reserve list by growthSize, and then it removes the first node from the front, and add that node to our active node, then return that node.

The baseRemove method does the similar thing:

```
protected void baseRemove(DLink k)
{
	DLink.removeByReference(ref this.pActive, k);

	k.clear();
	DLink.addToFront(ref this.pReserved, k);
	this.mNumActive -= 1;
	this.mnumReserved += 1;
}
```

When we want to remove an Image object, we wouldn't just delete it completely. Instead, I removed it from the Active list and add it to our reserve list so that we can get access to and use that same object later.

In our Space Invader project, we used this pattern a lot, since we have different Managers, like ImageMan, TextureMan, GameSpriteMan, BoxSpriteMan, ProxySpriteMan, GameObjectMan and so on. Each of these managers are like an object pool that we can reuse the same objects so that we don't need to depend on garbage collection to destroy each object and spend lot of time on creating new ones.

5, Composite

Composite is used when we are using a tree data structure and want that the leaf node and the branch node could perform uniformly. For example, if we have a collection of cars to move, then we can just do move method on the collection, then the collection would go down their tree to invoke the move method of each of its member, it could be another collection or a leaf node.

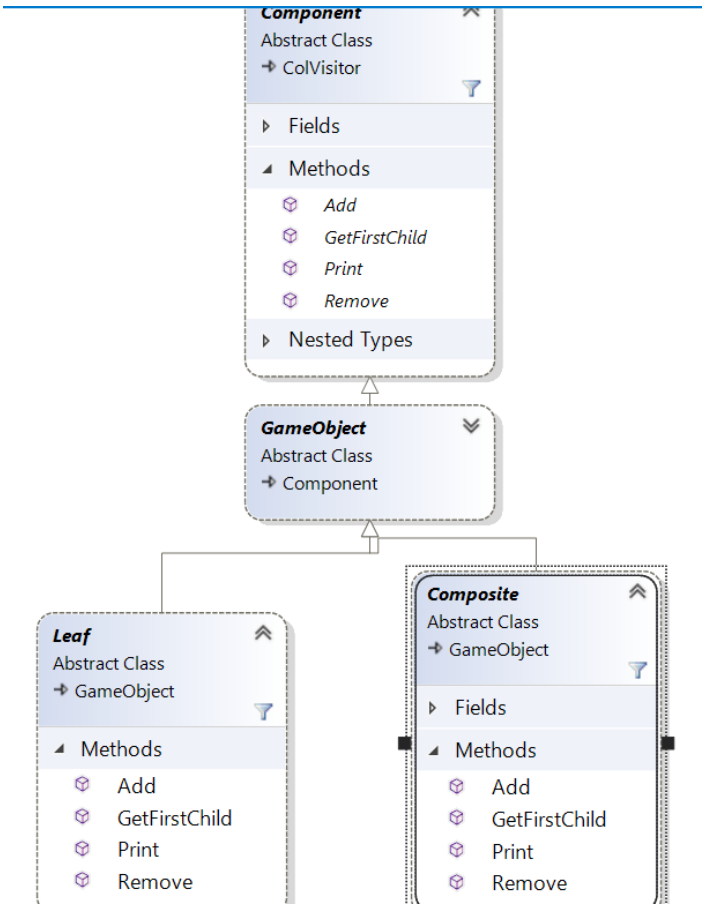
To implement this pattern, we'll have an abstract class which is the component, and make the leaf class and composite class inherit from the component class. In this way, we'll need to implement the same methods for both leaf and composite class, and they'll have the same behaviors to the outside world and the same interface. For the composite class, what we do is often to iterate through its child and invoke the same methods on them.

In our Space invader project, we have to show 55 aliens on screen, there are 11 columns of them and 5 aliens for each column. We have this tree pattern that each alien grid holds 11 alien columns and each column holds 5 aliens. And what's more, we want the grid, column and alien would have similar behaviors. For example, we want the whole grid can move as well as a particular alien and in the same way.

If we look at our code in Space Invader on Component pattern, here's an example:

```
public override void Print()
{
	DLink pNode = this.poHead;

	 while (pNode != null)
	 {
		Component pComponent = (Component)pNode;
		pComponent.Print();

		pNode = pNode.pNext;
	 }
}
```

This is the Print method in Composite class, which is a collection of Leaf. In this class, we actually go into each component of the collection and call Print method on each of them. So when we use them, we don't have to worry about whether it's a collection of objects or a single object, because they have the same interface and behave similarly.



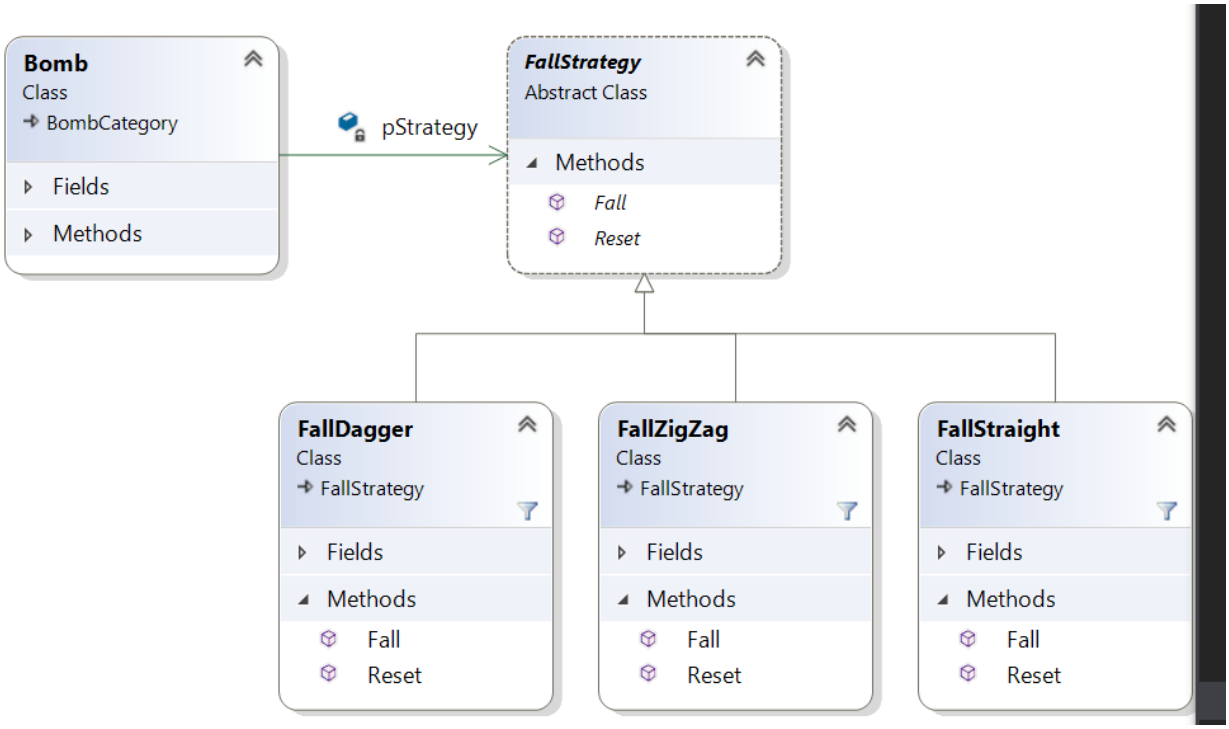
In the UML diagram above, we can see that the Component is an abstract class, and it has methods like Add, GetFirstChild, Print and Remove, and in both of its child classes, we have those 4 methods as well. And we'll have a class uses the reference of the Component class, no matter it's a Leaf node, or it contains a collection of objects, we can use those 4 methods no matter what. In this way, we are decoupled from the detailed implementation of those classes and can make them behave similarly.

6, Strategy

Strategy pattern is very useful when we have different implementations for one method for different objects. We would define a family of algorithms to implement to the same method. According to the client that calls this method, it would behave quite differently. And we don't want to know which method or which behavior it's going to call. In this way, it encapsulates the implementation from the user and it isolates algorithms in separate classes and have the ability to select different algorithms at run time.

The strategy pattern always has an abstract class which contains an abstract method which we want to define differently. All of its sub classes have to implement this method in a different way. For example, if we have different vehicles which are moving at different speeds, we might want to update the location of them in a different way. For this situation, we can have a move strategy which is basically just an abstract class where defines a method called move and it has several classes inherit from it, for example, we could have SlowMove, MedianMove, or FastMove classes where they update their locations in the same interface Move method, but have different implementation. For example, the move method in SlowMove might just increase their x by 5, the MedianMove calss by 10 and FastMove class by 15. Then we can have different vehicles holding a reference to move strategy class. In this way, according to which classes the reference really is, it would choose the right move method by itself at runtime.

In our Space Invader, we use this pattern when we are making 3 different bombs dropping in a different way.



Here, bomb is our client, it holds a reference (let's say r) to the abstract class FallStrategy, which has 3 sub classes with its own implementation of Fall method. The Fall method in the FallDagger class would fall down with flipping up and down, the one in FallStraight class is going to just move down, and the one in FallZigzag is going to fall down with switching its left and right side. Because of Liskov's substitution principle, derived class can be completely substitutable for its base class, r is of type FallStrategy, it would belong to one of the 3 derived classes, and which Fall method to use will be decided at runtime.

Here, we only have 2 abstract method defined in Fall Strategy class which hasn't been implemented, and in its derived class, only we can see its implementation and they're different for all those 3 derived class.

This is the interface of class FallStrategy:

```
abstract public class FallStrategy
{
    abstract public void Fall(Bomb pBomb);
    abstract public void Reset(float posY);

}
```

And this is its own Fall method defined in FallDagger class. In our bomb class, we have a reference to this FallStrategy class:

```
public class Bomb : BombCategory

{    // Data
     public float delta;
     private FallStrategy pStrategy;

     //more method
}
```

When we actually want to use the Fall method, we used this delegate:

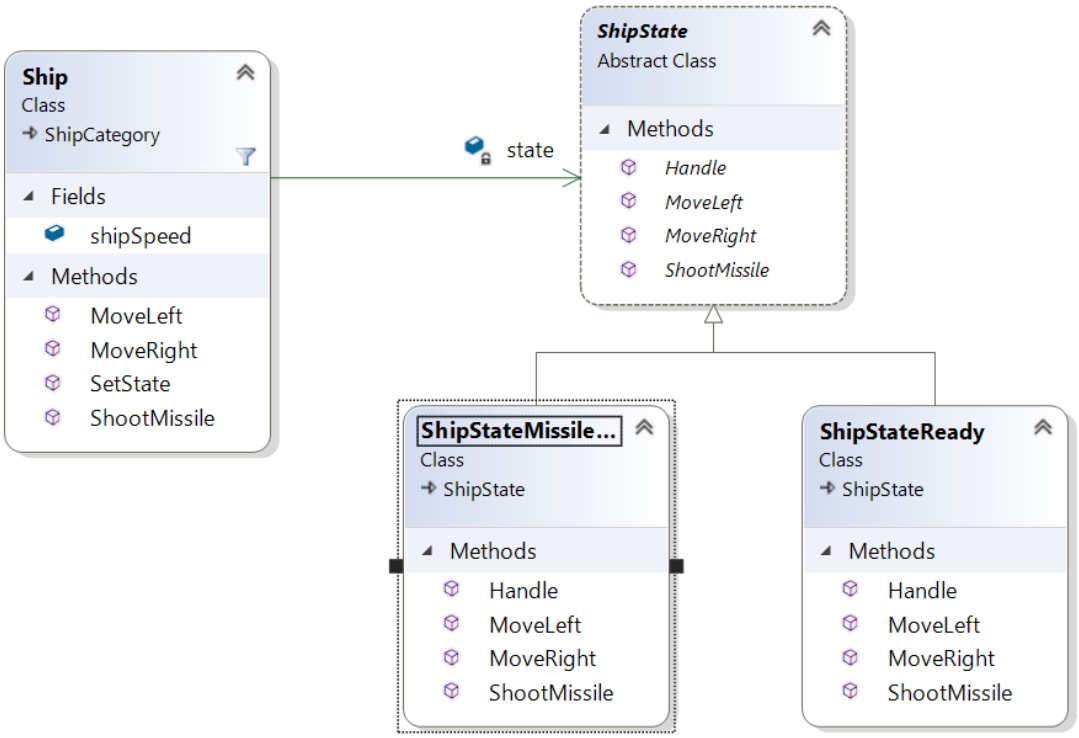
```
public override void Update()
{
    base.Update();
    this.y -= delta;

    // Strategy
    this.pStrategy.Fall(this);
}
```

We would get access to its pStrategy first, and see which class it is actually of, and then would be directed to that Fall method depending on what specific class pTrategy is.

7, State

The State pattern is a little similar to Strategy pattern, since both of the patterns have different implementation for the same method. But the different lies on the Strategy pattern, one client would only get one implementation according to which type the object is. While for the State pattern, which implementation to use depends on which state the client is on, and the class also has a method called Handle, which is meant for switching the state to one another. So when the client holds a reference to a Strategy pattern, the reference doesn't change any more; while for the state pattern, the reference gets switched back and forth between different state objects.

In our Space Invader game, there are several places using this pattern. The first place is when we are dealing with the ship. There should only be 1 missile showing on the screen. That means when the ship fires off one missile, it can't fire off the other one until the missile hits the top of the screen. In other words, the ship would have different states, Ready state and MissileFlyingState, and EndState. And they'll act differently for the ShootMissile method.



In ReadyState, our ShootMissile method looks like this:

```
public override void ShootMissile(Ship pShip)
{
    Missile pMissile = ShipMan.ActivateMissile();

    pMissile.SetPos(pShip.x, pShip.y + 5);
    // switch states
    this.Handle(pShip);
}
```

In MissilFlyingState, our ShootMissile method would look like this:

```
public override void ShootMissile(Ship pShip)
{
    //Nothing's here because the ship at this state cannot shoot a missile
}
```

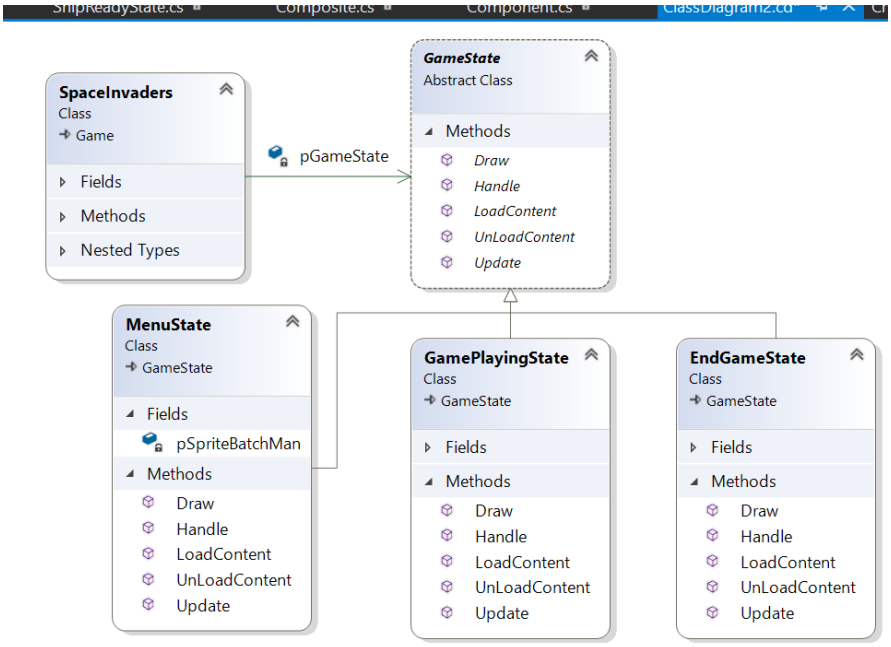
So, in different states, they are implementing different algorithms too. This looks similar to Strategy pattern, the difference is in the Handle method. In each state, the Handle class could switch the state of the same object to another class, then it could do as the other implementation asks to do. In the example above, we can see after shoot the missile in Ready state, it calls the this.Handle(pShip), and the Handle method inside the Ready state looks like this:

```
public override void Handle(Ship pShip)
{
    pShip.SetState(ShipMan.State.MissileFlying);
}
```

Right after the ship in ready state fires off its missile, it calls the Handle class, where it basically just set the state to MissileFlyingState, because in this state, the ShootMissile method doesn't do anything which means it cannot fire off any missile at that state. Only when the collision happens between the missile and the top screen, the state of this ship would be set back to Ready state.

The other place where we use the State pattern is our whole game cycle part. The whole game would be divided to 3 states: the menu state, the game playing state, and the game over state.

The UML looks similar to what we have for the ship:

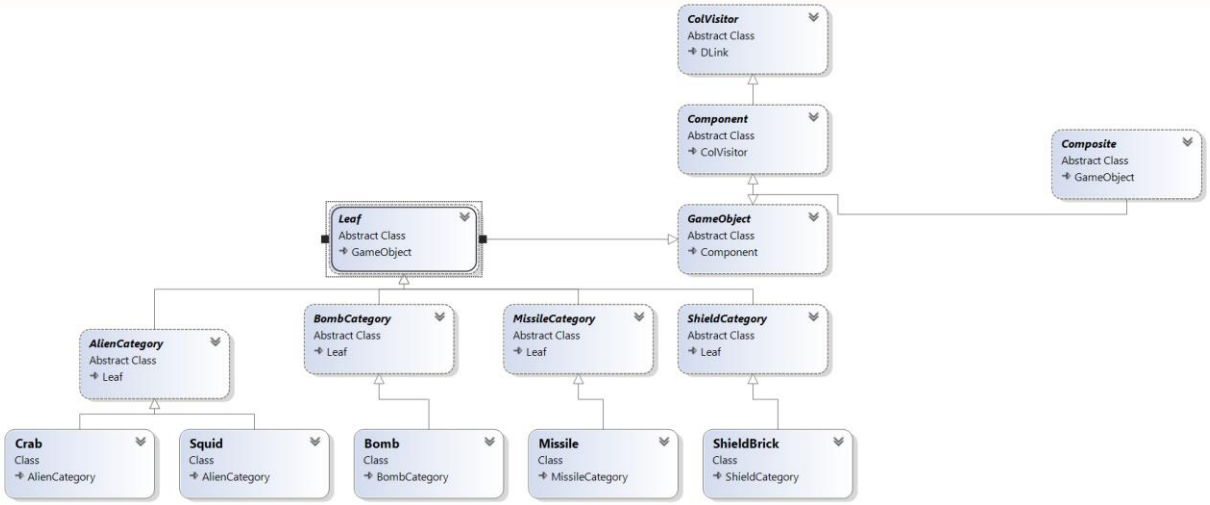


We have SpaceInvaders class holds a reference to the GameState which can be either one of the three derived classes, and the Handle method can help to switch among these states. And when the reference gets switched, the pGameState would refer to a different object then. And the implementation of those methods executed would be different as well.

8, Visitor

When we have many different classes defined and when they have different behaviors for each two classes. For example, when instance of class A interacts with instance of class B, we might set some value to 10; if instance of class A interacts with instance of class C, we might need to change the value to 10. If we don't use Visitor pattern, it's a little tricky to find out which class is interacting with which other one. We could use switch statements. The problem is that if we have a lot of different classes and each two of them could be interacting with each other, we will end up seeing ourselves writing hundreds of different cases, which might not be ideal. Hence the visitor pattern comes for rescue.

The way how visitor pattern works is that we would make an abstract class, and make all of the classes we are interested in finding them interact with each other inherit from the abstract class. And each one of them should implement its Accept method as well. When let's say we are at class A, and in the Accept class, we would switch to the class it's interacting with, and for the method name we could use something that shows which class did we come from. Then all of a sudden, we'll see we have both class information to decide what should happen if instances of these two classes visit the other.



In out ColVisitor class, we've defined one abstract method

```
abstract public void Accept(ColVisitor other)
```

So for each object, it contains the method called Accept, and then it woud generate different method like vistisB, visitC, visitD where all of classes A, B C D should inherit from the abstract class:

The following Accept method is from bomb class, so when we dive into the other class other, we know the object it interact with is bomb.

```
public override void Accept(ColVisitor other)
{
   other.VisitBomb(this);
}
```

Here the Accept method works for all of the classes because we know that all of the classes inherit from ColVisitor class, and using Liskov's substitution principle, we know that the class ColVisiter could be substituted by any class we are interested in.

```
public override void VisitMissile(Missile m)
{
    ColPair pColPair = ColPairMan.GetActiveColPair();
    pColPair.SetCollision(m, this);
    pColPair.NotifyListeners();
}
```

So when we know we are at the bomb class, and look into the method of VisitMissile, then we'll know that we are writing for the interaction for missile with bomb. It gave us a really quick way to figure out which two objects are interacting with each other without using switch statements.

What's more, if we are looking at missile interacting with bomb or bomb interacting with missile, they should behave in the same way. Using Visitor pattern, we can write it once with order or we can create another function to do the same behaviors. But in wither way, it follows our DRY principle, which is don't repeat yourself. In this way, it's easier to maintain or to make modifications on it.

If we are at some other objects where we are not interested in what's going to happen when these two are interact, we could just not write a specific method for it or we can direct it in a way to the ones we're interested.

9, Observer

This design pattern is used a lot in everyday life. It described an one to many dependency relationship. If one object changes its state, it would go to tell all of the objects that subscribe this changing object to update accordingly. The subject would hold a container of all of the objects in a certain way, it could be an array, a linked list, a tree. When the subject changes the state, it would iterate though all of the observer objects, and to invoke their own Notify method.

In this way, the observer can be attached to or detached from a certain subject freely. If they are interested in getting notified when a certain state changes for the subject, then they would stay in the list of observers. If not, they can choose to detach from the observer group and they wouldn't get notified when state changes in the specific subject.

In our Space Invader game, we used observer pattern a lot. The main part of this design is used towards those collisions. We have tons of different game objects which could interact with each other. For example, a missile can hit an alien, can hit a bomb, a UFO, and the top of the screen. Even when the missile hits aliens, they are acting differently. Because the player would get different scores when they hit different type of aliens. In my game, when we hit an octopus, we get 20 points, a crab, 40 points, and a squid 60. And the alien gird can hit the left or right wall. If that happens, we will make the whole grid move down and move towards the other direction. So if two objects collide, how could we make a series of actions accordingly? We can use the Observer pattern.

For example, we the missile hits the alien, my code looks like this.

```
pColPair = ColPairMan.Add(ColPair.Name.Bombs_Player, pBombRoot, pShipRoot);
pColPair.Attach(new CheckPlayerStatusObserver(pSpriteBatchMan));
pColPair.Attach(new SndObserver(SpaceInvaders.GetInstance().sndEngine, "explosion.wav"));
pColPair.Attach(new SwitchToSplashObserver(GameSprite.Name.PlayerEnd, pSpriteBatchMan));
pColPair.Attach(new RemoveBombObserver(pSpriteBatchMan));
pColPair.Attach(new ShipEndObserver());
```

our pColPair is an instance for ColPair class. For each collision pair, there's a collision subject associated with it. For example, the collision pair of aliens vs missiles have its own collision subject, the pair of missiles vs UFOs have its own collision subject. When we do pColPair.Attach, we are actually attach the collision observers to those collision subject the collision pair holds.

```
public void Attach(ColObserver observer)
{
    this.poSubject.Attach(observer);
}
```

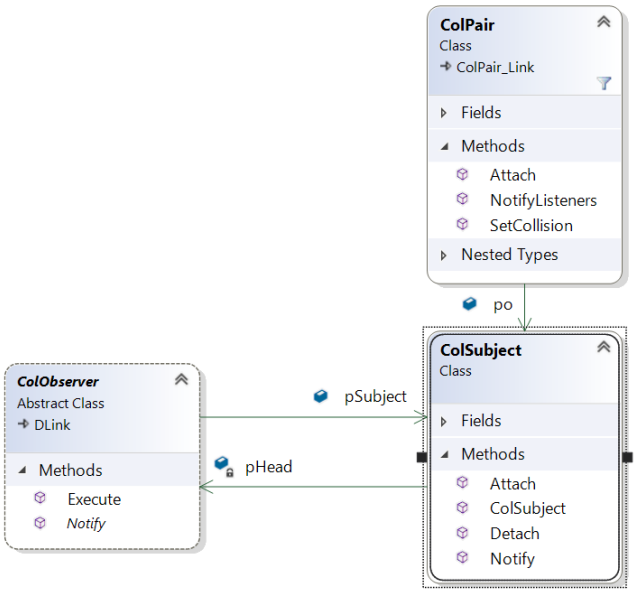
For this collision, we care attaching five observers to the collision subject. So when the collision actually happens, it's going to walk through the observers this particular subject has and goes to invoke the Notify method of each observer.

My CheckPlayerStatusObserver's Notify method looks like this:

```
public override void Notify()
{
    if (GamePlayingState.playLives > 1)
    {
        GamePlayingState.playLives -= 1;
        ShipMan.Attach(pSpriteBatchMan);
    }
    else
```

```
    {

        CheckPlayerStatusObserver pCheck = new CheckPlayerStatusObserver(this);
        DelayedObjectMan.Attach(pCheck);

    }
}
```

Each of those five observers is a derived class of ColObserver, so each of the objects has a Notify method to implement, and to call. When the status of the subject changes, we don't need to know which type of observers are there, instead we can just invoke the Notify methods of them since they all have a Notify method to call.



The UML diagram looks like above, each collision pair holds its own collision subject, and when each type of the collision happens, it will go to notifyListeners, and that method is going to walk though the observer list, and to invoke Notify method for each of the observers.

```
public void NotifyListeners()
{
    this.poSubject.Notify();
}
```

The `NotifyListeners invokes the Notify method of ColSubject, and then the ColSubject's Notify invokes the Notify method for each of the observers.`

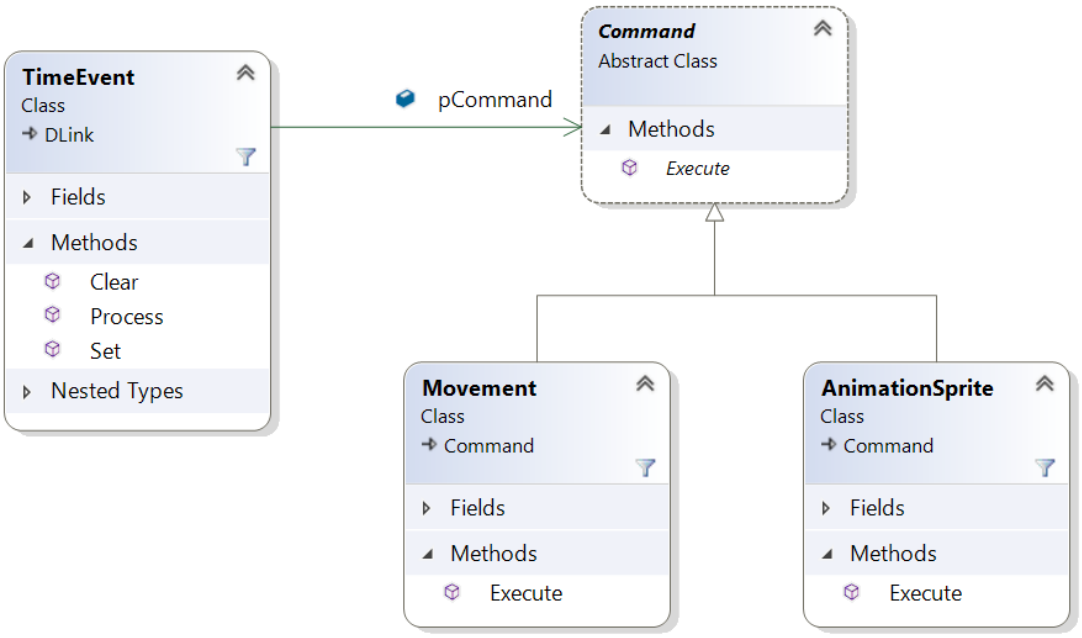
```
public void Notify()
{
    ColObserver pNode = this.pHead;

    while (pNode != null)
    {
        pNode.Notify();

        pNode = (ColObserver)pNode.pNext;
    }
}
```

10, Command

For the Command pattern, we are able to encapsulate different requests into objects and put those requests in a queue. When we are actually processing those requests, we don't need to know what type of requests they are, and what request objects we're dealing with. They all implement the same method interface for how to execute its request.

For this pattern to work, we'll need an abstract class called Command, and it will declare the contract method called Execute. Then we'll need a concrete Command class where implements the Execute method. We can add this object to a queue and the Execute method would get invoked by invoking some corresponding operations on the receiver. The main difference between observer and command pattern is that for observer, if it's being attached to a subject, it's there forever until it's detached. So every time some state changes on the subject, it would notify its observers; While for Command pattern, it tends to only be executed once. When the operation gets executed, it'll be taken out of the queue, and is going to be executed again until we add invoke the same operation again and put it back onto the queue.

For our space invader project, we used the Command pattern to do execute some time events. And it's times and sorted on the queue. The events which has closer executing time would be at the front of the queue, and get executed first. Our AnimationSprite is one of the concrete Command class, and Movement is another one. As shown below in the UML diagram, inherits from the base class Command, and both of them implements the method Execute. And we have a TimeEvent class which holds a reference to the Command object and it uses the process method to invoke the Execute method of Command object. Here, it uses Adapter pattern as well, where basically it delegates to another object where we actually need it to be executed.



So when we have a TimeEvent object, to call its Process method, it would delegate the work to Execute method of a particular concrete command object, the code would look like as below.

```
public void Process()
{
     this.pCommand.Execute(deltaTime, this.name);
}
```

For our aliens grid to move in a synchronous way as the aliens swapping images. I did make two concrete Command classes, AnimationSprite and Movement. For example in Movement:

```
public override void Execute(float deltaTime, TimeEvent.Name name)
{
    grid.MoveGrid();

    TimerMan.Add(name, this, deltaTime);

}
```

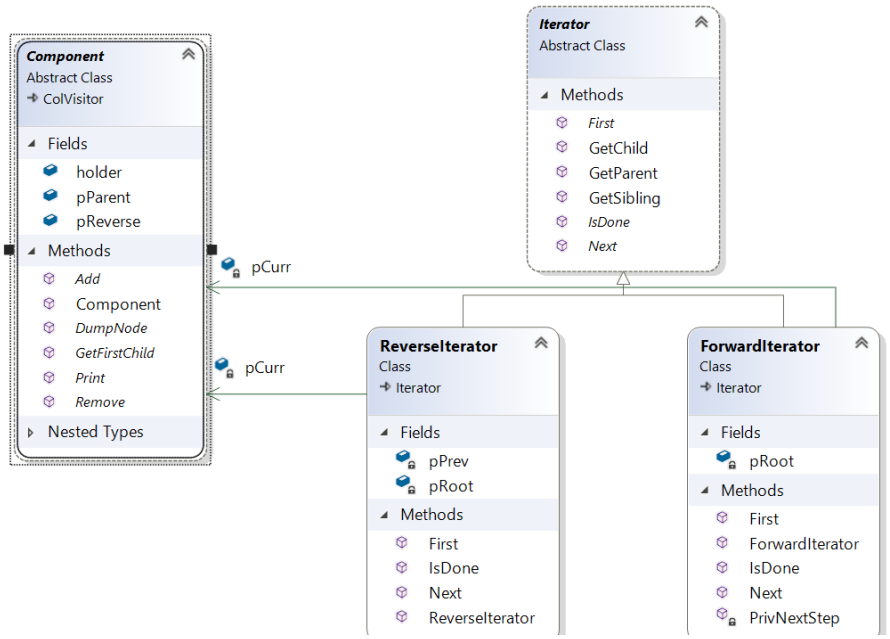
I added this back to the TimerManager, because as I just mentioned, the Command execution is only good for once. If we need to make the aliens moving constantly, we need to add the event back to the TimerManager and get executed continuously. The way to do this is to regularly add this TimerEvent which holds a reference to the command back to the TimerManager. If we want it to be executed at the same time interval, we can keep the delta time; otherwise, we can update the delta time to what we need when we add it to the manager again.

11, Iterator

Sometimes we use different data structures to implement a collection of objects. Sometimes, we use Linked list, sometimes tree or stack or queue. And the interface for all of the data structures are similar, but quite different. With iterator design pattern, we can treat the collection of objects in the same way no matter what data structure or detail implementation we are using under the hood.

For example, in our space invader project, we are using the tree data structure when we're using the Composite pattern. That way, clients need to know how we implement the tree or they need to see the interface to figure out how to use it. With iterator, we can treat every data structure in the same way, so the client wouldn't need to know what's the data structure under the hood. It gets encapsulated as well.

And also, we can have two iterators to iterate through in a different order.



As the UML diagram shown above, The ReverseIterator and the ForwardIterator behave similarly and we can always get the first node and iterate through to the last node by using Next, or we can get the last node first, and then iterate from the last back to the first by using Next. But no matter which way is it, the clients could iterate though the whole collection without knowing how the data gets implemented and how it's arranged. In this situation, our nodes are of Component type, because it's either of type Leaf or Composite, where both of them inherits from Component.

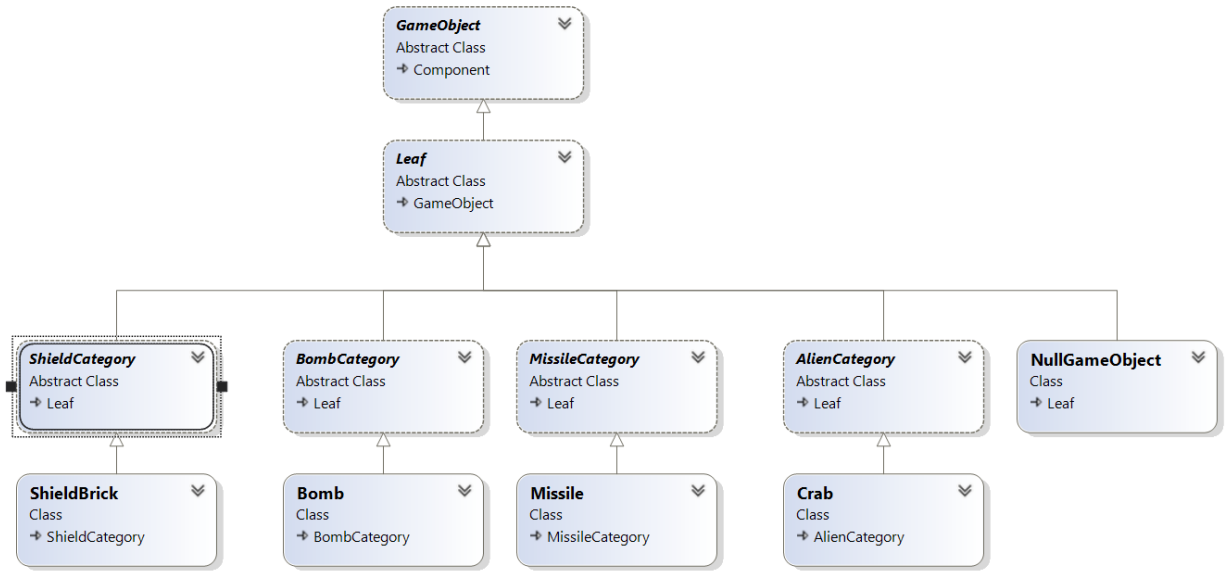
12, Null Object

Null object is just a surrogate for a lack of an object of a certain type. It contains and holds all of the interface, attributes or behaviors f a certain type object, however it holds no data, and actually it doesn't do anything. In this way, it hides the details or the information from the clients, so that we don't have to deal with lots of edge cases, instead we can just use the Null Object to make it behave similar to other regular objects but do nothing.

In our Space Invader, we have a Null Game Object which does nothing.

```
public NullGameObject()
: base(GameObject.Name.Null_Object, GameSprite.Name.NullObject)
{

}
```

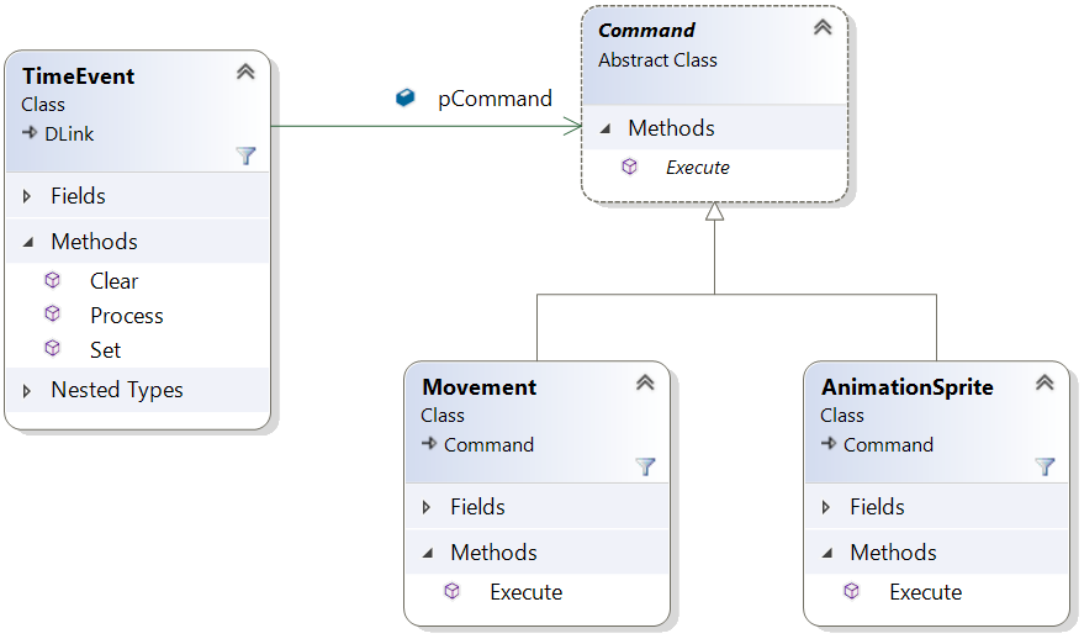
It has no data and behaves like a placeholder. But it has the same interface like other game objects.



13, Adapter

The adapter pattern behaves like a delegate and a bridge between 2 different classes. If two classes have different interfaces and couldn't work together and we have a class that have the interface we'd like to use, then we can just make a reference of the adaptee class and make that object do the work instead.

Like what we have seen in the previous example for Command pattern:



Here, the TimeEvent class has the Process method, however, the process method doesn't actually do the work by TimeEvent class. Instead, it holds a reference to the Command class, and call the Execute method of the Command class and make the concrete command class do the work for it.

In this way, we can connect two different classes together and we don't need to implement the same functionality multiple of times. Instead, we can make advantage of another class which has the functionality and delegate to that class and make it do the work for the client.

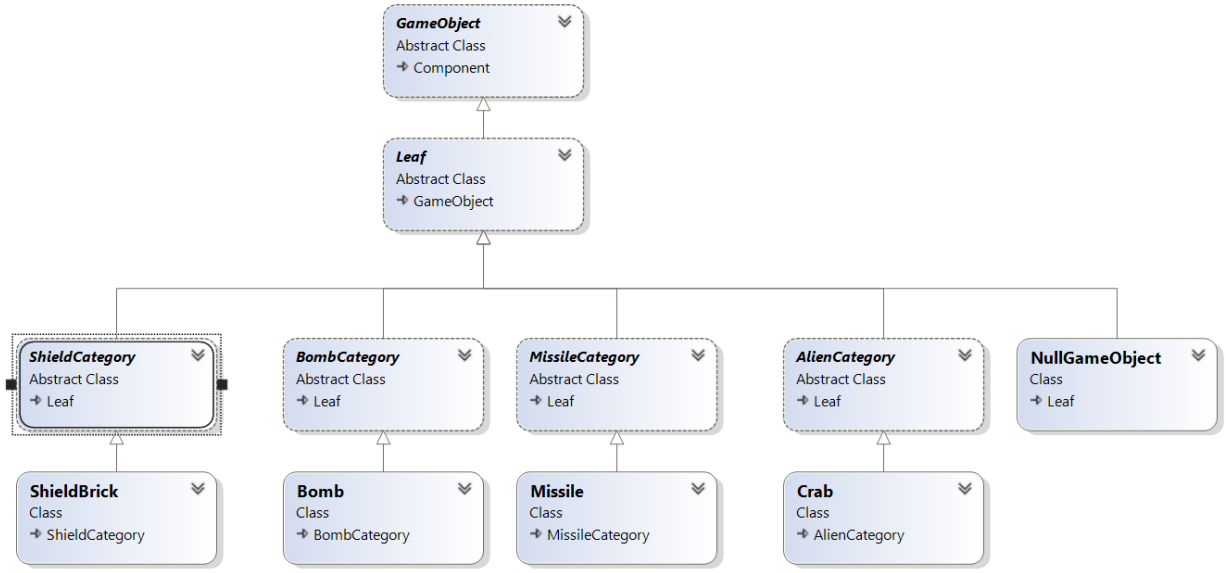
14, Template

Template is another simple design pattern we're using almost everywhere. We have an abstract class or only an abstract method in base class, we will defer the implementation of this method later to its derived class. In this way, the subclasses get to redefine the methods of its base class without changing the structure of its base class.

We have come across this pattern multiple of times.

If we look back at the same example of the previous pattern, the Command class defers its implementation of Execute method to its subclasses.

If we look back at the example for our game objects:



The Leaf, ShieldCategory, ShieldBrick, BombCategory, Bomb, MissileCategory, Missile, AlienCategory, Crab, and NullGameObject are all GameObjects. We have defined some method in GameObject, for example, Remove, we can redefine Remove method in all of the subclasses as well. We could also call base.Remove to use the same Remove method from its own base class which wouldn't change the hierarchy or structure of algorithm of the whole system.