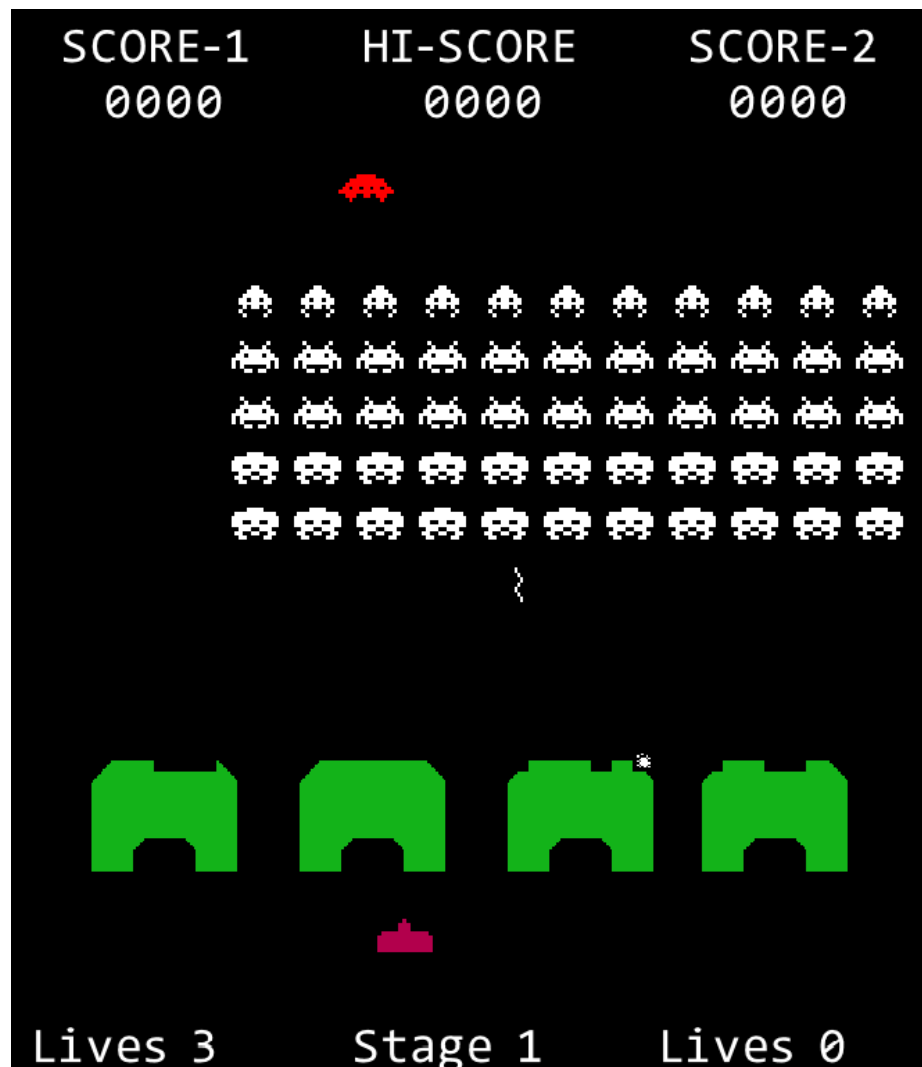


# DESIGN REPORT FOR SPACE INVADERS

YOUNGJO KIM



# 1. ORGANIZATION

## WHAT IS PROBLEM?

In space invader project, there exist tons of item instances with various data types to keep track of. One problem is how to deal with various data types. Another problem is how to manage the many instances?

## HOW TO SOLVE?

If they are only a controller which control all data types and it will be called manager from now on, this manager will be too much big and complex. One of solutions is creating managers for each data types and each manager controls its corresponding data type instances. Each manager must have a way to deal with its items well. So, the managers will carry out creating, holding, adding, removing and some unique behaviors based on data type. One manager shouldn't be more than one. For example, many images will be held by image manager. The image manager will create, find and remove the image instances so that only one image manager instance is required.

## TEMPLATE PATTERN

Template pattern defines the template for its subclasses. By providing general blueprint such as general functions and abstract methods, subclasses can use its base functions and implement abstract method which are contracted by super class.

The respective managers share similar functions so that those similar functions are grouped in an abstract generalized manager for the derived managers to derive from. All the subclasses will share Base methods and each subclass will implement derived methods which corresponding to its item type.

In the project, the template method defines common methods such as create, remove, find and destroy and they will be used in all subclass managers. The subclass managers which inherited from Manager override the abstract functions so that the methods in Manager can depend on the methods in subclasses managers.

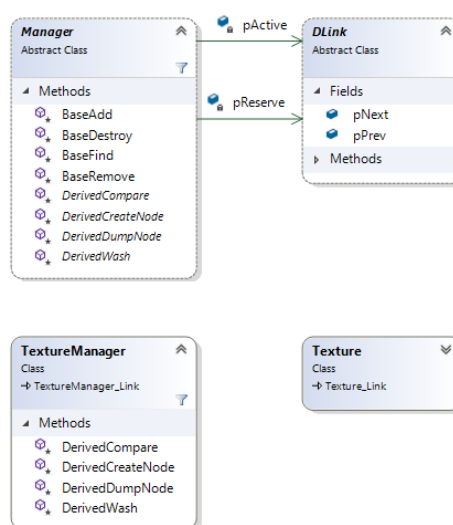


Figure 1. Generalized abstract Manager and one of Subclass of Manager

For example, Figure 1 shows that TextureManager is derived class from Manager. Since, Base methods in Manager are already implemented, TextureManager can use them. By overriding Derived methods in TextureManager, TextureManager can have unique implementation to deal with Texture objects and Manager class knows the Derived methods in TextureManager because the derived methods exist in Manager as abstract that should be implemented in its subclass.

## OBJECT POOL PATTERN

Object pool pattern helps to use “new” much less.

Using “new” allocating object into memory is very expensive in runtime. To avoid dynamic memory allocation in runtime, prepare many uninitialized items in reserve pool in its manager and don’t deallocate it. By doing so, whenever a node is needed by manager, it will move a node from reserve list to active list instead of creating new node. Only new node will be created when there is no node in reserve pool. Once a node in active list is used, the manager moves the node back to reserve list from active list rather than deallocating the node.

The Figure 2 shows how this pattern is used. TextureManager create many uninitialized texture data and keep them in pReserve. When TextureManager get a meaningful data, it takes a texture object in pReserve out and move it to the pActive and set data into the texture object. If the texture in pActive is no longer needed, it will be moved to pReserve from pActive rather than deallocate it. In this way, creating new object while program running can be very minimal and is needed when pReserve hold no data.

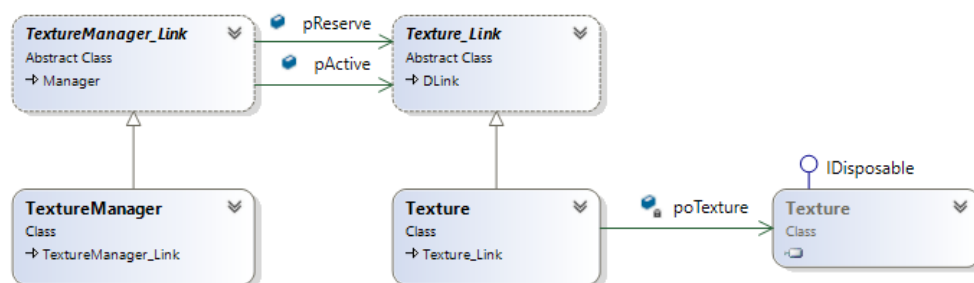
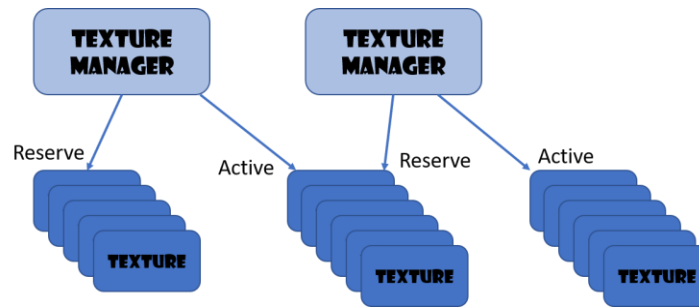


Figure 2. Texture Manager System

This pattern dramatically decreases time of creation and destruction of nodes.

## SINGLETON PATTERN

The singleton pattern ensures that only one instance of the class will exist, and it can be accessed at anywhere in program. In this project, one manager should hold only one type of item. For example, Texture Manager hold many texture object pool in reserve and active list. If there is two Texture Manager and they hold same data type, we cannot guarantee which texture manager hold a texture which I want to use. And textures are linked as double linked list, so one small mistake in Texture Manager can cause serious issue in the other Texture Manager.



*Figure 3. More than one Texture Manager*

To ensure that only one instance exist, Texture manager hold its instance and is private static. Its constructor is private as well so that the users cannot access or create manager instance directly. Also, manager instance is static, we can assure that the instance will be only one. To create its instance, static Create() should be called. Since the method is static, initialization is not required to execute function and the function can be called at anywhere.

The singleton pattern is used in most of managers as well such as Image, SpriteBatch, Timer, Sound and so on.

## 2. REPEATED CREATION

### WHAT IS PROBLEM?

In space invader, there are 3 different types of aliens: crab, octopus and squid. The aliens are 55 in total and arranged in 5 X 11 grid. Creating an alien and assign its location 55 times is very inefficient and code look so dirty. How about shield bricks? One shield block composed of about 55 bricks and there are 4 shield blocks. Need a way to avoid dirty code creating those objects.

### HOW TO SOLVE?

By hiding the process of creating aliens into factory, code looks cleaner. The client doesn't have to pass specific information such as width, height or location into create function to create aliens. And, by providing very simple information to the object creator such as location data x and y, code look much better than before.

### FACTORY PATTERN

Squid, Octopus and Crab aliens are derived from GameObject class. This factory defines a function which creating GameObject and this object could be Squid, Octopus or Crab. The complex implementation which creating aliens is inside of Alien Factory so that the client doesn't have to know the complex process. Just pass simple information such as object name and its location x and y. Create() method in Alien Factory will decide which subclass alien should be created using switch statement. No matter what alien is created, it will be returned as GameObject type by polymorphism.

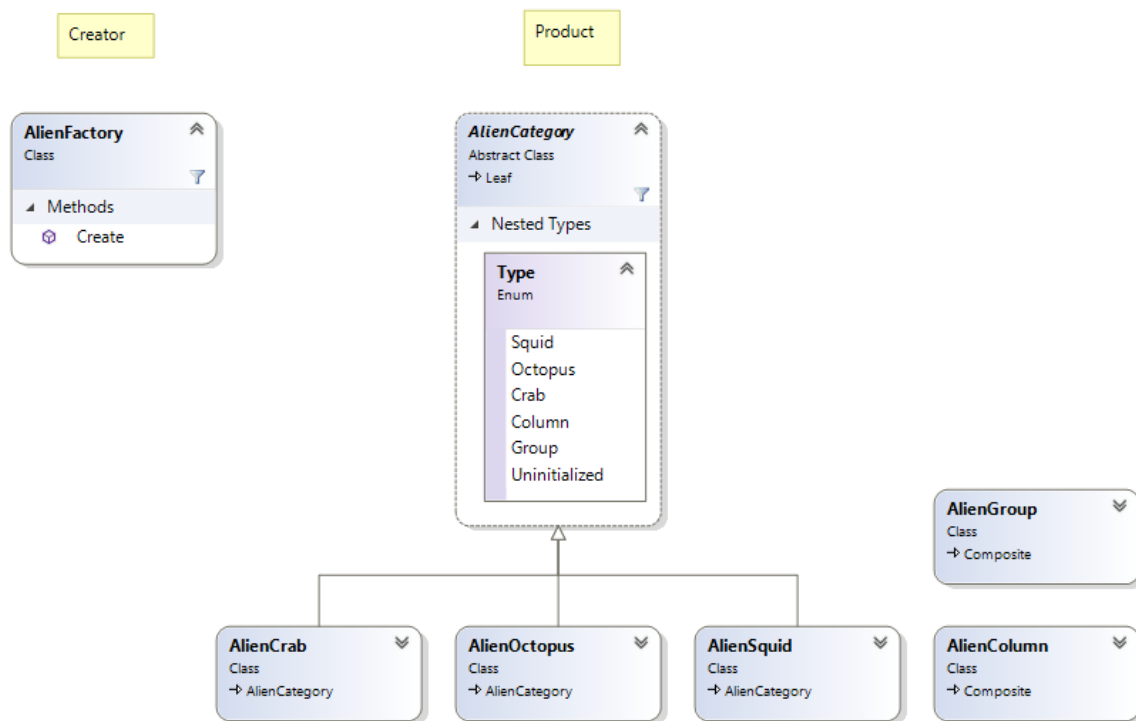
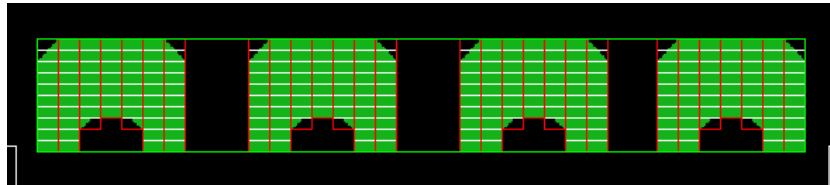


Figure 4. Alien Factory and Aliens

Figure 4 shows that how AlienFactory.Create() returns diverse type of alien objects. AlienFactory.Create() functions need arguments and one of them is AlienCategory.Type. By passing one of enum value to AlienFactory, users can create AlienCrab, AlienOctopus, AlienSquid, AlienGroup and AlienColumn. Users don't have to indicate what game sprite, spritebatch are needed. Attaching to SpriteBatch and choosing GameSprite are done in the AlienFactory. Since all alien class are subclass of GameObject, AlienFactory.Create() returns those alien objects as type of GameObject and it works by polymorphism.



*Figure 5. Shield Blocks*

The factory pattern is used to create shield blocks. Figure 5 shows that there are 4 blocks of big shields and each one composed of 63 shield bricks. By using shield factory, creating those shield blocks become much easier and the code looks much clean.

### 3. TIME EVENT

#### WHAT IS PROBLEM?

To mimic space invader arcade game, there are alien animation by changing images. Also, alien group have to move together to downward or to side. These movement and animation must occur at the same time. Also, whenever game object collision occurs, its corresponding explosion effect must happen and disappear after a few seconds. How about UFO? UFO appear after several seconds since the game start. There are many events which should occur depend on its trigger time.

#### HOW TO SOLVE?

To solve this problem, Timer manage system is required, and it holds many time-events nodes including changing position of aliens for movement, swapping images for animation and to trigger events after a few seconds. What if timer manager holds 1000 time-events? Do we have to iterate all of them? Some of them “yes”, and the others “no”. What if it holds 1000000 time-event? If so, the game will be horribly slow.

Let timer manager hold time-events node in sorted order by its trigger time. If so, timer manager iterates time-event nodes from beginning in the active list and it will stop when it meets nodes whose trigger time is later than current time. By doing so, timer manager will not iterate all time events node and this program can be faster.

#### PRIORITY QUEUE TIMER

By adding InsertionAdd() method in Timer Manager, only Timer Manager can use Priority Queue. The timer manager will add new time-event node in sorted order by the node’s trigger time. During Update() in Timer Manager, it will iterate the event nodes in active list and decide either stop or execute next event node by comparing node’s trigger time and current time. Only event nodes whose trigger time is earlier than current time will be executed.

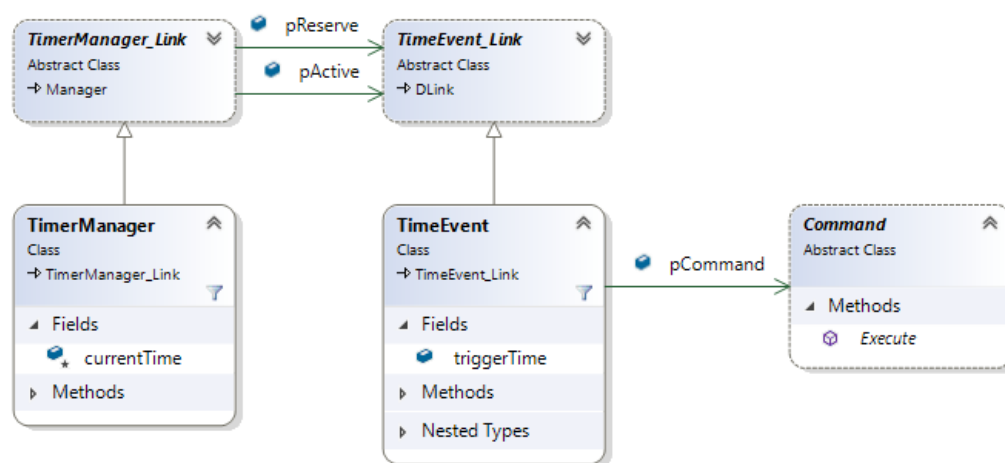


Figure 6. Timer System

## COMMAND PATTERN

To create time events, this pattern encapsulates a request in an object and save it to the priority queue by packaging the receiver and action into subclass of Command. An invoker, which is TimeEvent, calls the Command's Execute method to invoke the action.

In the figure below, the Time Event can invoke a command in any of subclass of Command. The AnimationSprite has all information to swap image so that its sprite seems animated. The SpawnBomb has all information to create bomb in AlienGroup. Like this, the Timer Manager holds Time Event that are invoked at certain time. When the moment come, the timer asks the command to execute.

Timer Manager insert time events in sorted order by its trigger time so that those command does not need to take place immediately.

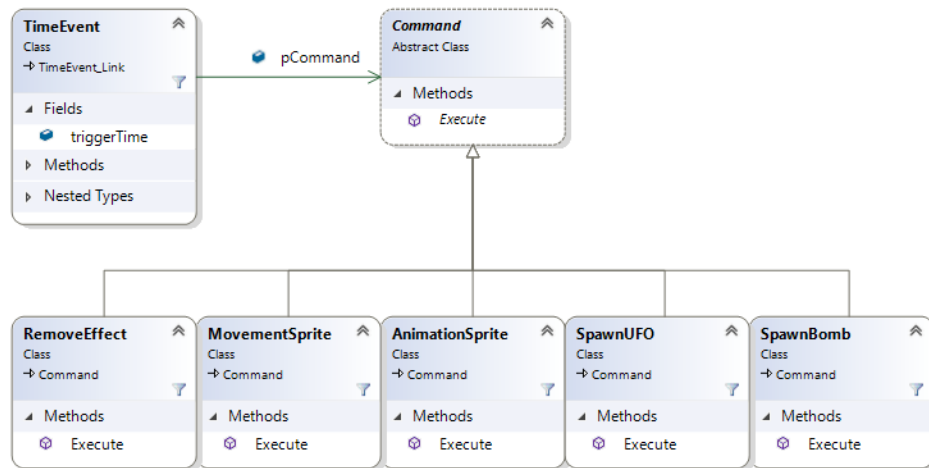


Figure 7. Timer system and Command Pattern



## 4. NULL CASES

---

### WHAT IS PROBLEM?

When creating game objects, sometimes there is a case where the game object has nothing to do with some part of it. In this project, the walls have only collision object and they don't have any wall sprites. Also, explosion effect objects have only explosion sprites, not collision object.

The problem is if those unnecessary part of object are declared as null, there will be too many if statement to check whether they null or not. Those repetitive if querying for special cases in the code will make the program slower and too many if statement cause unexpected bugs.

### HOW TO SOLVE?

Use Null Object pattern. Any game object will be treated as same game objects. They will be processed in the same way, but its null part will do nothing. For example, the wall will have sprite object and collision object. But the sprite object is null object so that it will do nothing. Let's see explosion effect objects. The object has explosion sprite and null collision sprite. So, null collision sprite will do nothing and if statement is not required.

### NULL OBJECT

Null Object is to encapsulate the absence of an object by providing an empty behavior such as do nothing. By defining empty method, null object will do nothing. This pattern is used for game objects in the project.

```
8 references  
public override void Update()  
{  
    // It is Null Object. It does not have any behavior.  
}
```

*Figure 8. Update method in Null Game Object*

In this project, the null object pattern is used for proxySprite in GameObject class rather than just Null Game Object. GameObject class have pointer to collision object and proxy sprite. Proxy sprite has reference to GameSprite so that it shows an image on screen such as alien, missile, bomb, UFO and player. These game object have a reference to gameSprite corresponding to its name.

However, there are some invisible game object as well such as wall which shows sort of boundary of game screen and bounding box which restrict players location. These objects shouldn't be visible to users on screen so that they don't have reference to gameSprite.

If they are set as null in game object, it needs if statement to check if some data is null and it will make the program slower.

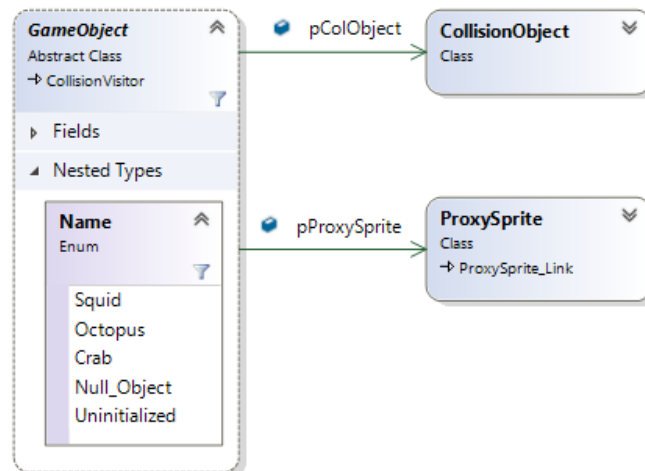


Figure 9. GameObject class

By let pProxySprite point to ProxySprite whose name is Null Object, it doesn't need if statement to check null of pProxySprite and it guarantee it does nothing.

## 5. HIERARCHY

---

### WHAT IS PROBLEM?

There are opportunities to optimize the project. One of them is organize game objects in same category into a hierarchy. By doing so, system can avoid unnecessary operations and program can be faster. For example, aliens collide with missiles, shields and walls. The collision checking in each alien can be done by comparing position of each side of them at every frame. But it need a lot of operations and it will make the program slower.

For example, let's see collision between aliens and missile. In game, there are 55 aliens and one missile. Each object has 4 side so that total number of comparing position of side will be more than 200. This is only aliens and missile. If we think of all collisions, it needs to check more 10,000 side comparison per frame and that make the program dramatically slower.

If the aliens are organized into columns, which are organized into a big group, checking the big group collision is good enough. Checking collision of aliens needs to be done only when a big group have a collision. If the big group don't collide with other objects, it doesn't have to check alien collisions.

### HOW TO SOLVE?

Organize aliens into columns, the columns will be organized into a grid. Instead of checking collisions of all aliens, just detect whether the big grid is in collision. If so, check what column is in collision and find which alien is in collision in the column. The time of computation will dramatically decrease.

### COMPOSITE PATTERN

The Composite pattern compose objects into tree structures and let us treat them in similar way as a single object. This pattern allows checking collision of aliens more optimized.

This pattern defines an abstract base class which is Component in Figure 10. This Component class have abstract methods as contract that needs to be implemented in its subclasses. Composite is sort of container of Component. It can hold a group of Component and it means composite can hold leaf and composite. That is how the composite pattern build tree structure. Leaf cannot hold component.

In this project, AlienGroup is subclass of Composite so that it can hold Component and it can work as container of aliens. AlienColumn is subclass of Composite and Aliens are subclass of Leaf. When a column is created, it adds 5 aliens in it because each column should hold 5 aliens. Since AlienGroup and AlienColumn are subclass of Composite so that they can have reference to Component. This Component can be treated as leaf or composite so that composite can hold leaf and composite. The columns are inserted into the Alien Group. Repeat this job 10 times since there are 11 columns. To check the collision of aliens, only the AlienGroup needs to be check. If it doesn't have any collision, we don't have to check collisions of all aliens. 55 collision checks become only 1 collision check. Only each alien's collision will be checked when the AlienGroup have a collision.

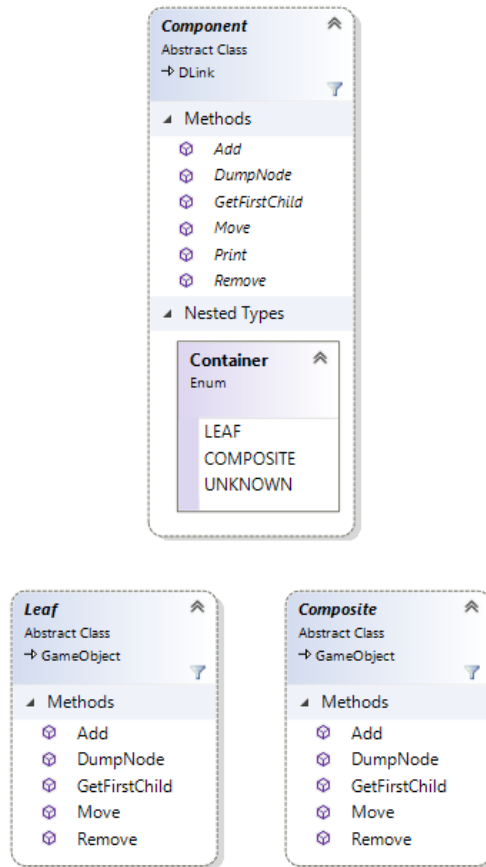


Figure 10. Composite pattern

## 6. ACCESSING DATA IN HIERARCHY

---

### WHAT IS PROBLEM?

Since composite pattern is applied in our project for optimizing collision system, aliens compose tree structure. Most of game object are grouped by similar categories and each of group compose tree structure to optimize collision system. All game object must be updated and rendered to be shown on screen. However, it is not easy to iterate all game object in hierarchy in each frame.

### HOW TO SOLVE?

Need to find a way to iterate all game object in the hierarchy

### ITERATOR PATTERN

Iterator pattern provide a way to access the elements of object hierarchy sequentially without showing its underlying complex data structure.

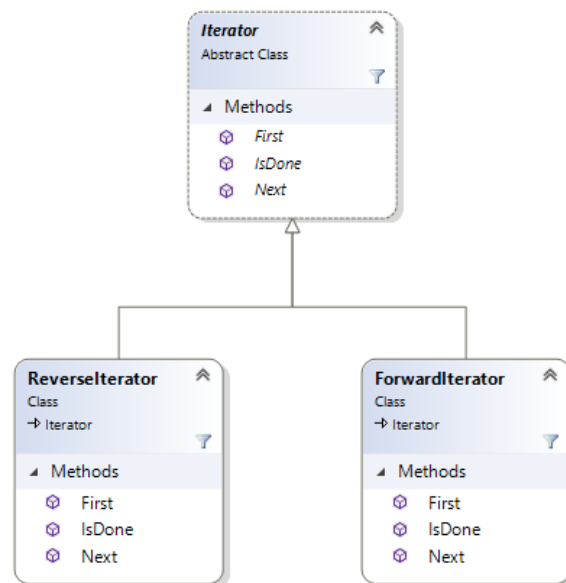


Figure 11. Iterator

The iterators pattern works by traveling through the data until it accesses all the data in the structure.

In this project, there are two iterator class, Reverseliterator and Forwardliterator. Forwardliterator start to travel from the root of the hierarchy. By using Next(), it can access to the next component. If it walks through all data, its IsDone() returns true so that user can know iterating is done without knowing the complex data structure.

ReverseIterator is very important. Alien objects compose Alien Column and the columns compose Alien Grid. Those column and grid exist for detecting collision in optimized way. Since the size and location of columns and grid depend on aliens, aliens must be updated first and then the columns and grid. If the columns and grid are updated earlier than the aliens, the column collision box and its grid collision box will be dislocated from aliens. It will trigger wrong collision.

ReverseIterator start to travel from the last object in the hierarchy. By using Next(), it can access to the previous component. If it walks through all data, its IsDone() return true so that user can know iterating is done without knowing the complex data structure.

Iterator is not limited to aliens. Since all object compose its data structure, the iterator is used at everywhere to update game objects.

## 7. COLLISION

---

### WHAT IS PROBLEM?

Now we have alien group, shield group, bomb group and many other object groups. Each group has its own collision boxes and it can be used to detect collision with other object group. However, how a group know the other groups? Does each group need references to the others? There are many collision pairs. Player VS Bomb, Wall VS Bomb, Shield VS Bomb, Alien VS Shield and so on. How to keep track of them?

### HOW TO SOLVE?

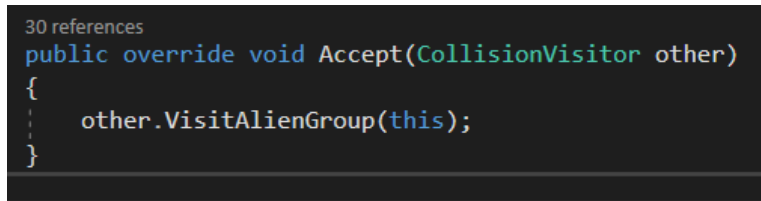
Built collision system. This collision system will hold collision pair node and managed by Collision Pair Manager. By setting two collision groups into one collision pair node, we can set pairs easily without checking all game object groups. Since each object group doesn't have reference to the other group in a pair, there must be a way to reach the other group. Visitor pattern can solve this.

### VISITOR PATTERN

Game Object is subclass of CollisionVisitor class. CollisionVisitor class is abstract and it has one abstract method Accept() and lots of virtual VisitBy() method. Since Accept() is abstract, its subclass must implement it so that all concrete object inherited from CollisionVisitor have Accept() method.

Collision system knows what objects are in collision and let one of them knows it. Then the object will execute the Accept() method and it will call VisitBy() method.

For example, if AlienGroup collide with MissileGroup, AlienGroup.Accept(MissileGroup) will be executed. The Accept() in AlienGroup knows that it is in AlienGroup so that it will execute MissileGroup.VisitByAlienGroup(this). Figure 12 shows the code.

A screenshot of a code editor showing the implementation of the Accept method in the AlienGroup class. The code is as follows:

```
30 references
public override void Accept(CollisionVisitor other)
{
    other.VisitAlienGroup(this);
}
```

*Figure 12. Accept() in AlienGroup class*

If VisitByAlienGroup() is not overridden in MissileGroup class, it will call default method which is VisitByAlienGroup() in CollisionVisitor class. Calling default method in CollisionVisitor class shouldn't happen. Programmer need pick appropriate method in CollisionVisitor class and override it in one of subclasses based on collision pair.

We are still on the group level. Now we know what groups are collided and we can go deeper what column in in collision. And, what concrete object such as Crab Alien or Missile actually in collision.

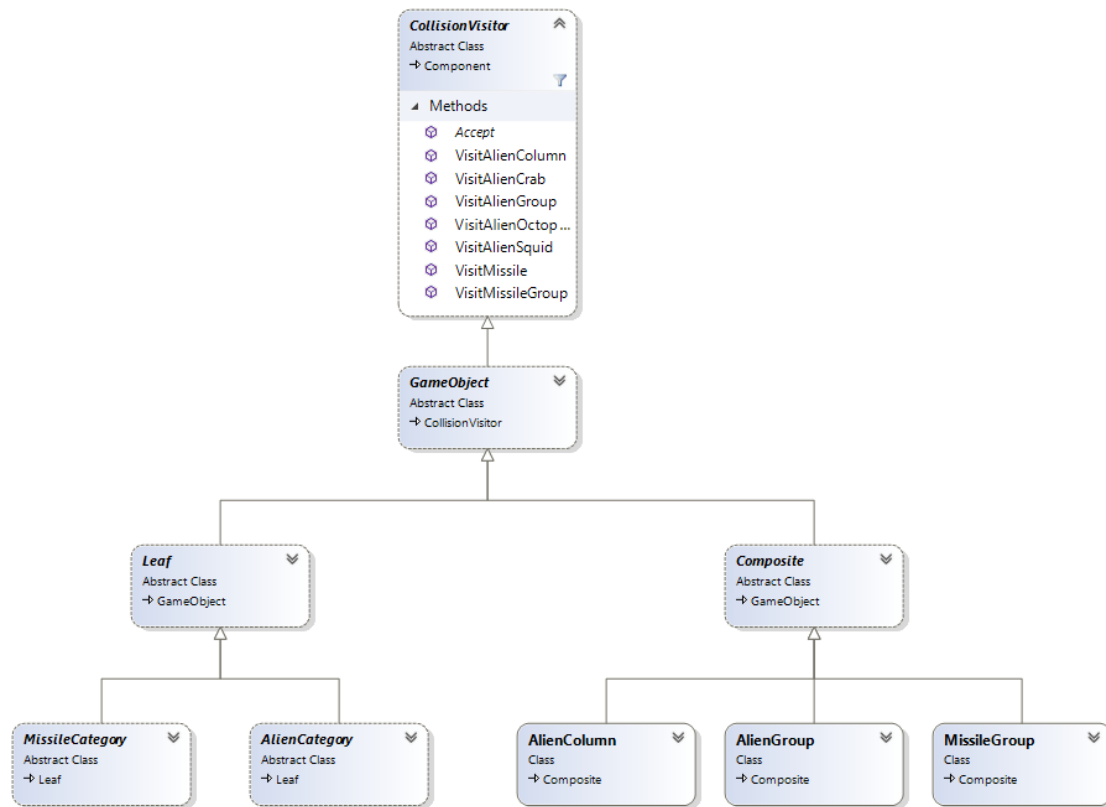


Figure 13. Visitor Pattern

Using the visitor pattern, the combination of the `Accept()` and `VisitBy()` methods find which objects have a collision and it provide efficient way without holding reference to other objects. Same process is applied for all collision pairs in this project. CollisionVisitor in Figure 13 shows part of abstract visit methods.



## 8. NOTIFICATION

### WHAT IS PROBLEM?

To make collision system works, composite and visitor pattern are applied. Composite provide efficient way to check collision status. Visitor pattern helps to find what exact object collide in a group. Also need the system to react to collisions. After we found the alien object collide with missile object, those objects must know what their job is. In this case, alien will be removed and leave explosion effect. Missile will be disappeared as well. It could be done in Visitor pattern code. However, giving this job to visitor pattern would be excessive. Let's try decoupling this job from visitor pattern.

### HOW TO SOLVE?

When the collision is detected and found what concrete objects are in collision, a notification is sent to the objects and they will know how to react to the collision. When one of alien got hit from missile, a notification will be sent to them. Alien will be removed from screen, show explosion effect, make explosion sound, update player score.

### OBSERVER PATTERN

Observer pattern is used when there is relationship between objects. The relationship is one-to-many. That means if one object is modified, its dependent objects get notification automatically. Observer pattern uses Subject and Observer. Subject has methods Attach() which can attach many observers in it. Also it can execute the observers by using method Notify(). Observer class is abstract and concrete observer classes are inherited from it.

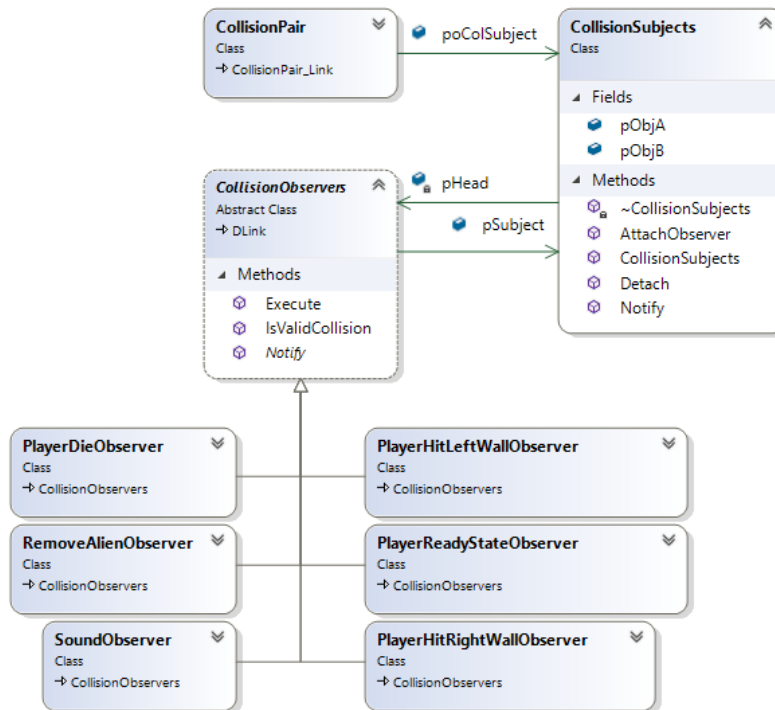


Figure 14. Observer pattern in Collision

In the collision system, the Observer is CollisionObserver and the Subject is CollisionSubjectObserver. Since CollisionObserver have a pointer to CollisionSubject, it knows where to attach its concrete observers. CollisionSubject have a pointer to CollisionObserver so that it knows concrete observers have Execute() and Notify() methods.

Each collision pair have a pointer to a collision subject. Since a group of observers are attached to each subject, when the collision pair catch collision between two objects in collision pair, it can trigger notification to all dependent objects.

The used observer is not removed after executed. It keeps living in collision subject and waiting until get trigger signal.

```
public override void VisitShieldBrick(ShieldBrick sb)
{
    CollisionPair pColPair = CollisionPairManager.GetCurrentCollisionPair();
    pColPair.SetObserverSubject(sb, this);
    pColPair.NotifyListeners();

    // Shield will be deactivated by removeObserver
}
```

*Figure 15. Execute observers*

Figure 15 shows that how collision pair execute all observers in the subject.

## 9. SPECIAL BEHAVIORS

### WHAT IS PROBLEM?

There are 3 kinds of bomb type: Cross, Zigzag and straight. Each bomb has its unique way of falling to the bottom. Only different is gameSprite and the algorithm the way they fall. Is there a way to use them efficiently?

### HOW TO SOLVE?

All kinds of bombs are Bomb class object. I'd like to set them have different GameSprite and falling algorithm.

### STRATEGY PATTERN

The strategy pattern allows a class behavior, or its algorithm can be changed at runtime.

In this project, abstract class FallStrategy is created. Bomb object have a pointer to this FallStrategy object. There are 3 concrete class of FallStrategy depending on how they fall. The implementation of Fall() in each concrete strategy class are different.

```
62 references
public override void Update()
{
    base.Update();
    this.y -= this.dropSpeed;
    this.pFallStrategy.BombFall(this);
}

4 references
public override void BombFall(Bomb pBomb)
{
    Debug.Assert(pBomb != null);

    float distance = this.prevY - pBomb.GetY();
    if(distance > 50)
    {
        pBomb.SetScaleX(-1.0f);
        this.prevY = pBomb.GetY();
    }
}
```

Figure 16. Update() in Bomb Class and implementation in one of concrete strategy.

Figure 16 shows that the implementation of Update() in Bomb is very simple. Just update position data in bomb and pass the bomb object to its strategy object. Then the strategy object knows how to let bomb fall.

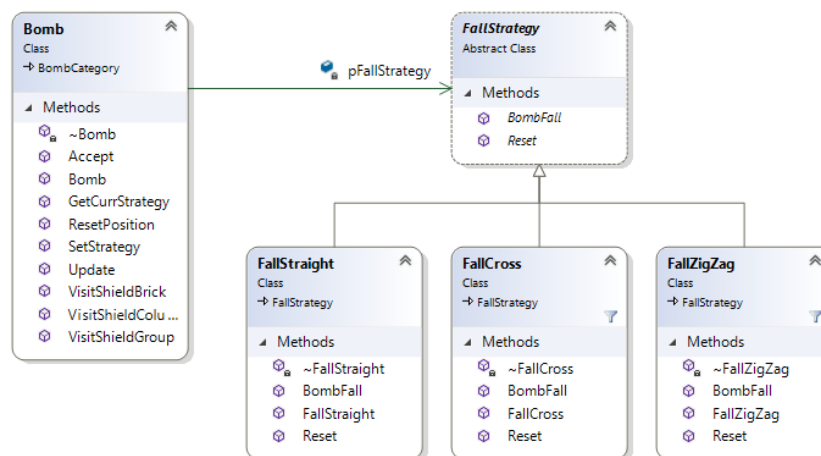


Figure 17. Strategy pattern

## 10. BEHAVIOR CHANGE DYNAMICALLY

### WHAT IS PROBLEM?

In space invader game, the spaceship can shoot only a missile at a time. That means while a missile is flying in the screen, the spaceship should not fire another missile. Also, the player ship cannot move out of the screen. It sounds like the whole behavior of space ship needs to change depend on its current state.

### HOW TO SOLVE?

To make sure that the player cannot fire more than one missile at a time, we need a tool to control its shooting state. Also, need to control its moving state together since it should not move beyond the boundary of game screen.

### STATE PATTERN

The state pattern allows an object to change its behavior when its state change.

This pattern very similar to strategy pattern. However, they have very different intention. Strategy pattern can specialize only one or two methods depending the number of algorithm. And they cannot change their strategy until they are deallocated. State pattern allows the object can change its own behavior. Object can modify its behavior when it's state changes using the Handle() method.

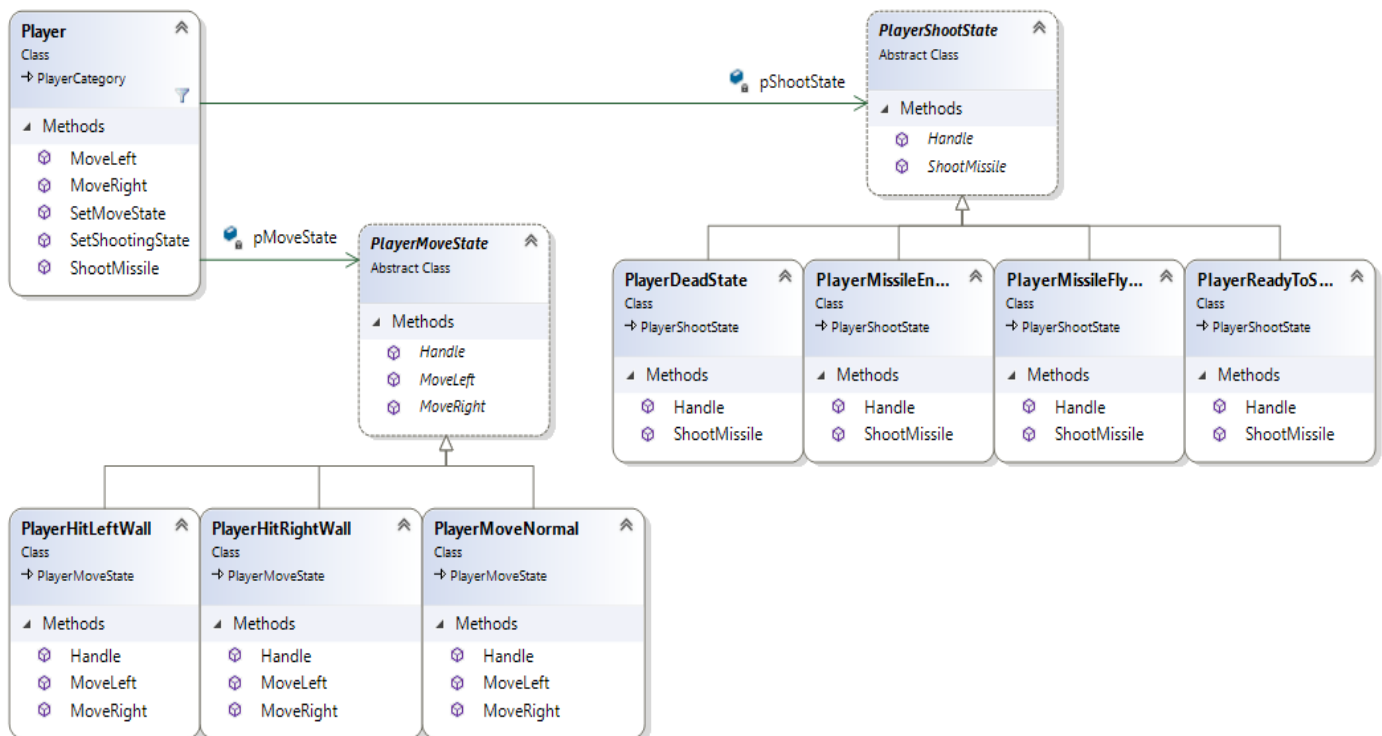


Figure 18. State Pattern for Player Class

For example, Figure 18 shows the state pattern for player class. If the playership fire a missile, then the Handle() in PlayerShootState class change the shooting state to MissileFlyingState so that player cannot fire missile any more while the shooting state is MissileFlyingState. Only when the missile hit something, observer will be triggered so that player's shoot state will be back to ReadyToShootState. The way how player use state object in the class is same as strategy pattern.

The move state is like shoot state. Since it shouldn't move out of the screen, I set 2 bumpers at the edge of inner screen. So that when player move and hit the right bumper, the observer will change player move state to PlayerHitRightWall and the implementation of MoveRight() is empty. The left bumper works in same way.

This state pattern is not limited to only Player class. Game scene also use state pattern to change its select mode, playing mode and game over mode. Alien Group class use state pattern to change its moving direction.

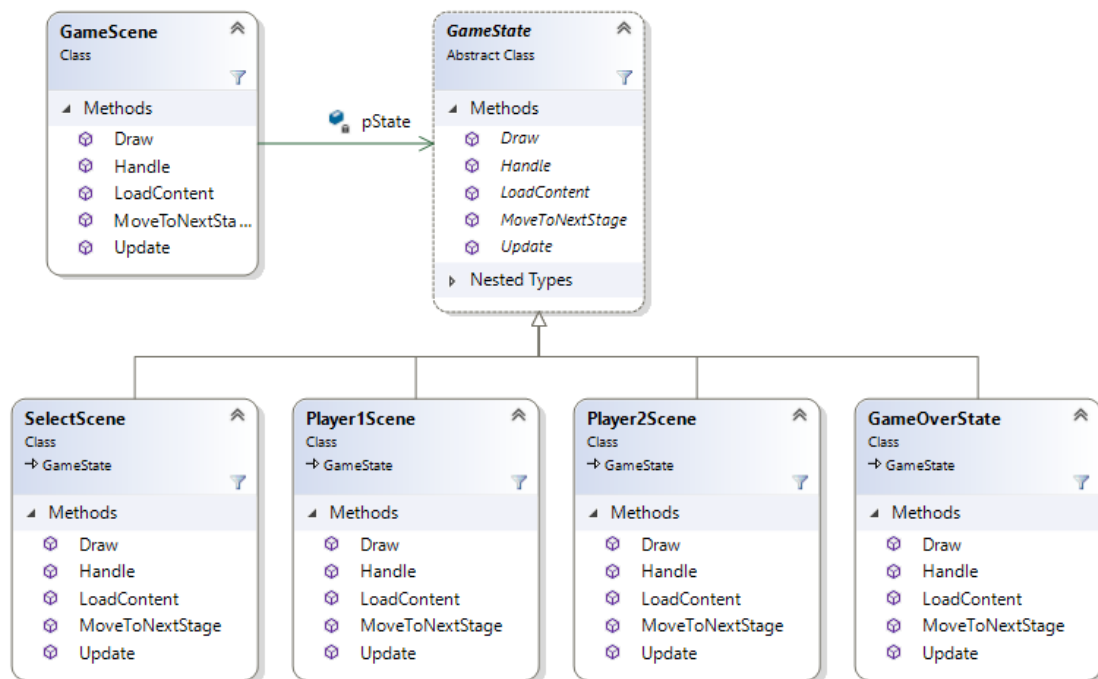


Figure 19. State pattern for Game Scene class

## 11. REUSE

### WHAT IS PROBLEM?

There are only 3 types of aliens: Crab, Squid and Octopus. They move around across the screen and animate at the same time together. In grid, 11 squid aliens at the first row, 22 crab aliens at the second and third rows and 22 octopus aliens at the fourth and fifth rows are arranged as 5 X 11 formation. Then, we can think an idea why not reuse the animated sprite to each alien? Only difference of each type of aliens will be position and we can come up with an idea how to use it.

### HOW TO SOLVE?

Since each type of aliens share same animation sprite and only difference is its position in grid. Create a class which have a reference to animation sprite and let it have data about its location in grid.

### PROXY PATTERN

Proxy pattern let us use some lighter objects instead heavier objects. That means, if we need to use only a part of method or data of some relatively big object, we can use some lighter object having same interface as the big object. The lighter objects are called proxies and we will use them until the heavier objects are needed. Proxy objects should be very little different from the big object.

Let's see squid aliens in this project. There are 11 squids and the difference of each squid is its position. Rather than creating 11 squid sprites, the proxy holds x and y and have a reference to the squid sprite. When the Game Object update x and y, proxy will get the updated x and y. The proxy pushes the x and y data to the real squid sprites.

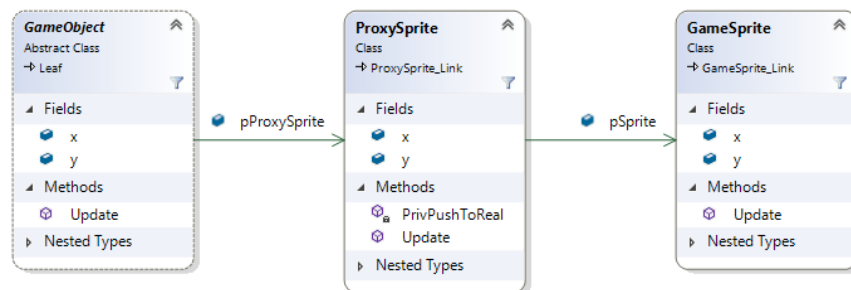


Figure 20. Proxy Pattern

## 12. COMPATIBILITY

---

### WHAT IS PROBLEM?

This project use Azul rendering game engine to draw sprites and boxes on the screen. To draw sprite, `Azul.Sprite.Update()` and `Azul.Sprite.Render()` function must be called. To draw boxes, `Azul.SpriteBox.Update()` and `Azul.SpriteBox.Render()` methods should be executed. Using those Azul members by directly accessing them is inconvenient because some class or methods I need are not compatible with them.

### HOW TO SOLVE?

Wrap the Azul objects with some new methods and data members to let them compatible with my class and methods.

### ADAPTOR PATTERN

The adaptor pattern allows incompatible classes to work together by converting the interface of one class into an interface. Since I added methods and data into wrapping class, the class can have compatible methods which will be widely used in the project.

By using this pattern, Texture class wraps `Azul.Texture`, Image class wraps `Azul.Rect` and it let Image class to take a rectangular area in Texture so that it can hold one of image in Textutre.

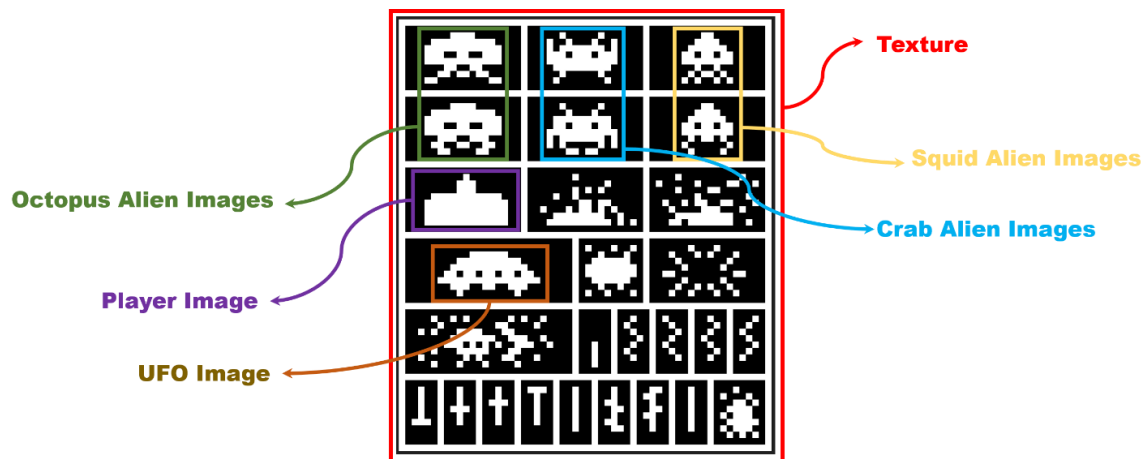


Figure 21. Texture and Image

My `GameSprite` class wraps each `Azul.Sprite` object. This class can use all `Azul.Sprite` method by using its own functions. This pattern makes it easy to update the sprite and render it.

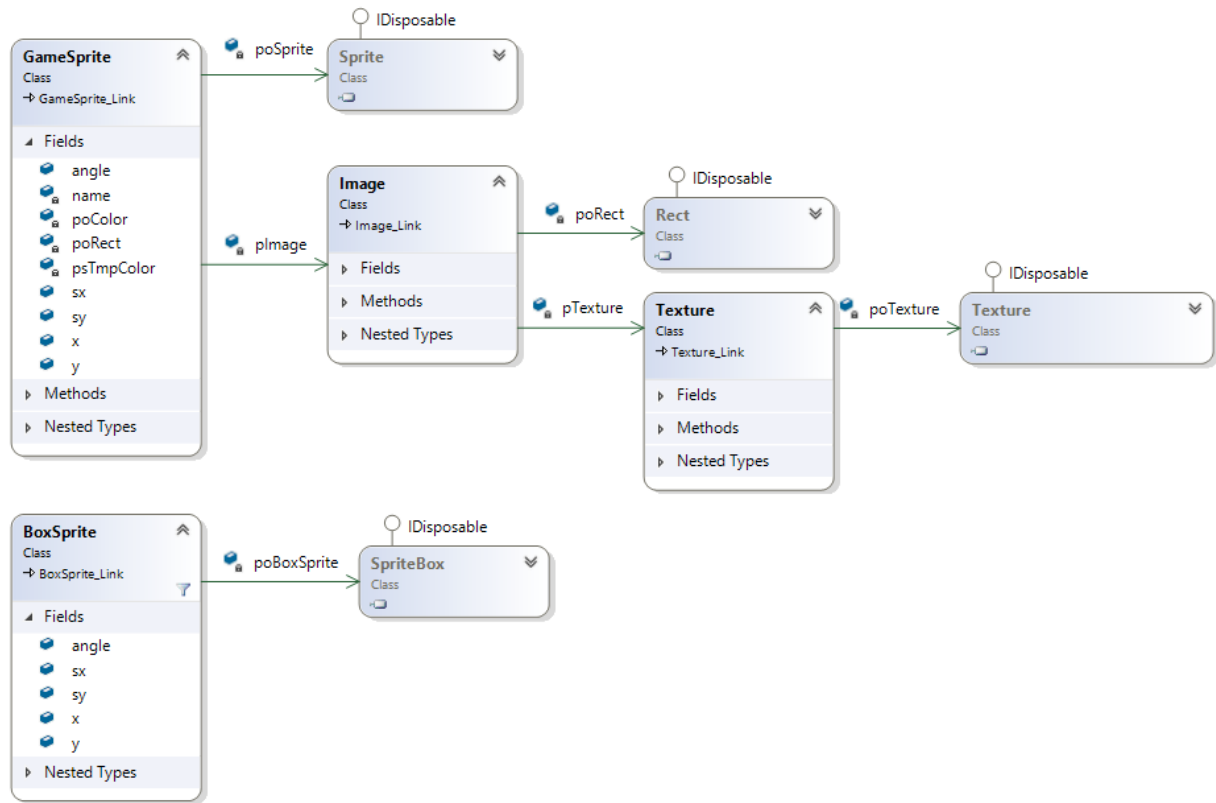


Figure 22. Adapter Pattern



## POST-MORTEM

---

### Alien grid movement:

The state pattern is applied to alien grid so that when it collides with wall, it changes its move state to move down. After it moves down, state pattern changes its moving state to move in opposite direction by itself. That is the mechanism of its move behavior. However, they don't move down and just move in opposite direction. The move state in side direction cannot change to the state in opposite direction directly. It must change state moving down and the moving down state change to move state in side direction. It randomly happens so that it was very hard to find which part trigger this issue and have remained in the program now.

### 2 Players mode:

This project has a serious problem and that is lack of the 2 players mode. Users can play only single player mode. At this moment, if the developer has a chance to improve this project, making 2 players mode will be the priority. I thought that time was enough to make 2 players mode. However, the alien grid moved beyond the boundary of the game screen once in a while, and program crashed. Spent very long to find the reason why it happened and just deleted bool variables which controls the alien grid movement. That was the reason why the state pattern was used for alien grid movement at the end. Now the program doesn't crash anymore, and the developer have an idea how to do it.

### Reuse game objects:

The dead aliens and exploded shield are removed from its group hierarchy and sprite batches so that they cannot exist on the screen. Those dead game objects are saved in Alien Manager and Shield Manager. These managers hold dead game objects and they are reused for games. Program create them only in the beginning of the program. However, many observers, explosion objects and bombs are recreated to be used. They also can be crated only in the beginning and reused while the program runs.

### Clean Code:

Nearing the due date, I was in rush to finish project and couldn't think about optimization and clean code. It also needs to be done in the future.

### Experience:

This is the biggest project ever I have done so far since I started coding last year. My project is not completed though, I am so happy and proud about project. One thing this class change me is that I started to think about architecture of program such as design patterns and how I need to manage and control objects.