DePaul University 2018-2019 Winter

ARCHITECTURE OF REAL-TIME SYSTEM

SPACE INVADERS GAME

Xuefan Wang

Introduction

This paper shows the detailed design patterns of Space Invaders Game.

The space invaders game engine is provided by professor, meanwhile, no modern features are allowed in the program, e.g. no containers/arrays, no template or generic parameters, so the whole program is implemented using only basic data types and basic language features.

Implementation of Space Invaders Game.

Problem 1:

Before we start building the whole project, we knew that we don't want to create a new object every time when we want to use it. Instead, after we create one object, we want to reuse it as much as possible. Now, how could we save these objects?

Design Pattern: Object Pool Pattern

Performance can be sometimes the key issue during the object creation (class instantiation). Especially when cost of initializing a class instance is high, and the rate of instantiation of a class is high. The Object Pool pattern offer a mechanism to reuse objects that are expensive to create. For the best performance, the number of instantiations should be finite.

Pattern in the game: Manager and DoubleLinkedList.

In this game, I create DLinkedNode as a "Mixin" class. Every real object that is going to be used is derived from it. So that, every object is "connected" to its neighbors, now, we can use a manager class to whole a list of objects in the "active list", when it is not used, it will be put to the "reserved list".

UML:

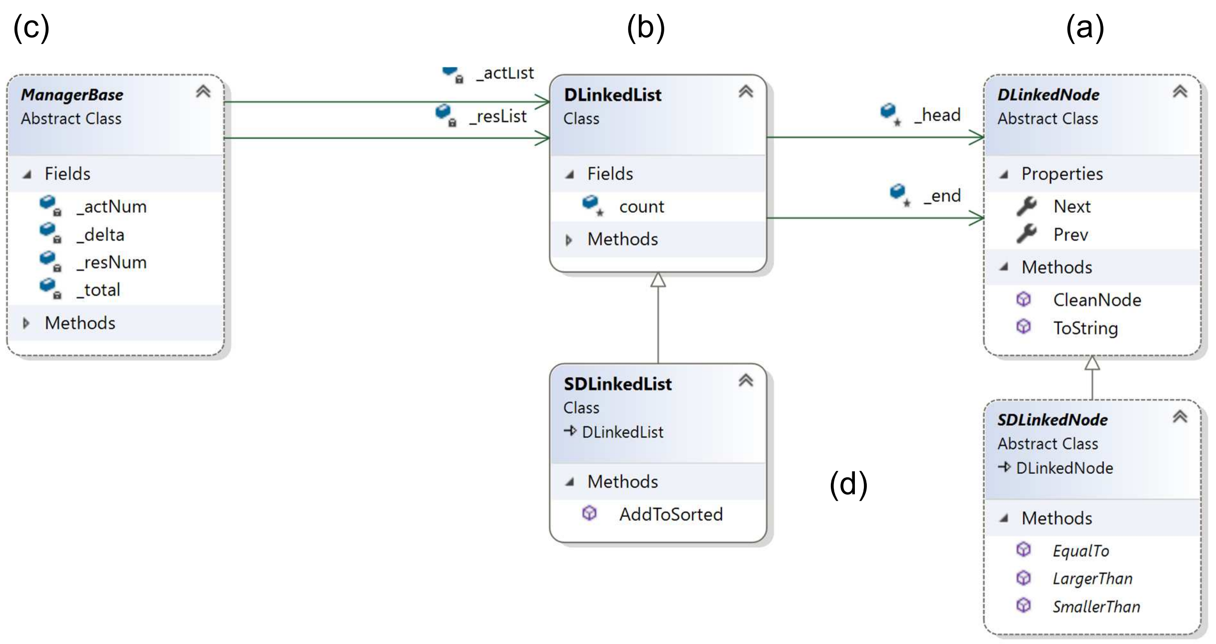

Figure 1: UML of Object Pool Pattern

Details of Implementation:

a. The DLinkedNode is an abstract class, for any object that would like to be in a list, should derive from the DLinkedNode. It has only Prev and Next fields which also are DLinkedNode.

b. The DLinkedList use DLinkedNode and have functions on it, e.g. AddToFront(), AddToEnd(), Remove(), and many more.

c. The ManagerBase class use DLinkedList, for any objects will be added to the manager, it will be added to the active list, when you remove some object it will go directly to the reserved list.

d. Since we sometimes need sorted list, so I also added Sorted Double Linked List (SDLinkedList) and SDLinkedNode, which are derived from their base class.

Problem 2:

Since the Game Engine is provided by the professor, so we don't want the clients to know the detailed implementation of it, and also, we require different objects works together as our need, how should we do?

Design Pattern: Adaptor Pattern

Adapter lets classes work together, that could not otherwise because of incompatible interfaces. Wrap an existing class with a new interface and convert the interface of a class into another interface clients expect. Impedance match an old component to a new system

Pattern in the game: Texture, Image, Sprite, and CollisionRect.

In this game, we will use textures, images, sprites, sounds engine and so on. We do not want to use the original game engine code since those classes cannot work together. Now we create our own adapter class and they have pointers to the original game engine code. As the Figure 2 (c) shows, Sprite and BoxSprite are different classes in the engine, but after putting them into an adapter, they now can both work as SpriteBase class. What's more, Texture, Image, and sprite are separate classes in the engine, now they can work together. In this UML, we can clearly see the associations of different classes.

Details of Implementation:

a. Left-up shows the Texture class and its manager. Our own texture class has a pointer (in the blue rectangle) to the game engine Azul.Texture and have functions working on it. So that client would never have to reach Azul.Texture to create or change features of a texture, and client only have to use the new Texture class, which can be manipulated by its manager.

b. Left lower shows Image class, which has the same feature as Texture class.

c. On the right hand, we have Sprite classes. Since we have 2 different kinds of sprites

(GameSprites and BoxSprites), we need a base to wrap them. (which will be used in near future)

UML:

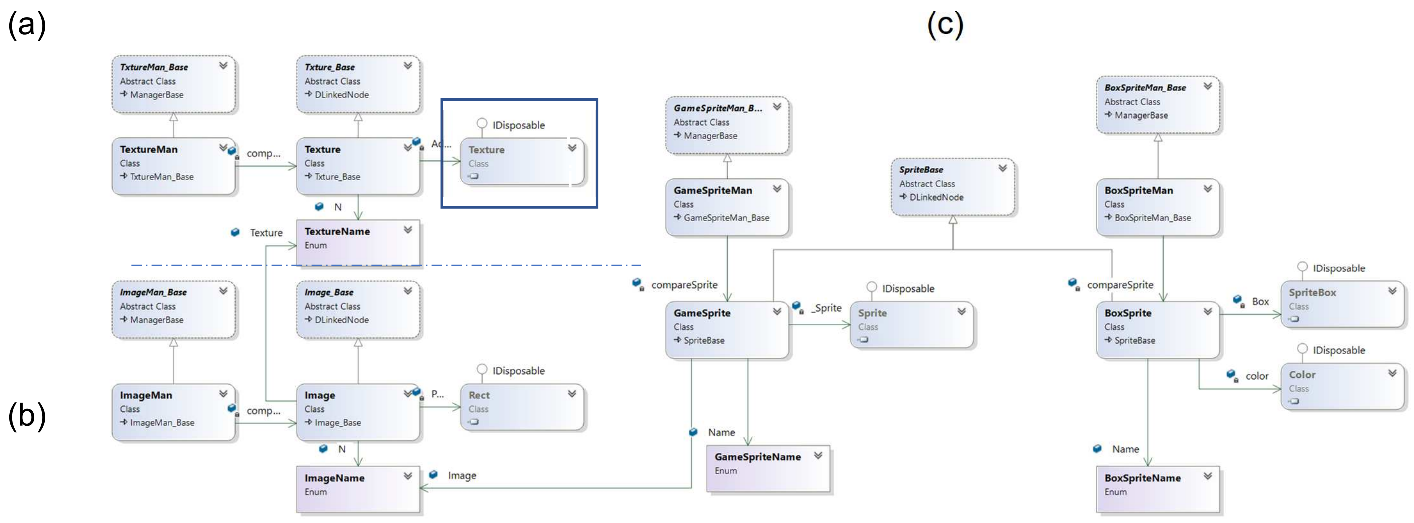

Figure 2: UML of Adaptor Pattern

Problem 3:

Since one texture have many font images, so when we create a font image, we don't want to create a Texture at the same time, instead, we want to have only one texture, and every font image, can share it. What should we do next?

Design Pattern: Flyweight Pattern

Some programs require many objects that have some shared fields among them. Creating many soldier objects is a necessity however it would incur a huge memory cost. So that we can share the common part where the other part can vary. Each "flyweight" object is divided into two pieces: state-dependent (extrinsic) part, and the state-independent (intrinsic) part. Intrinsic state is stored (shared) in the Flyweight object. Extrinsic state is stored or computed by client objects, and passed to the Flyweight when its operations are invoked.

Pattern in the game: Texture, Image, Font

In this game, different images can share the same Texture, also, all the Font are sharing one texture, now we can take the sharing part out as extrinsic piece, which is Texture class. And the intrinsic part has a pointer to it.

UML:

(a) (b)

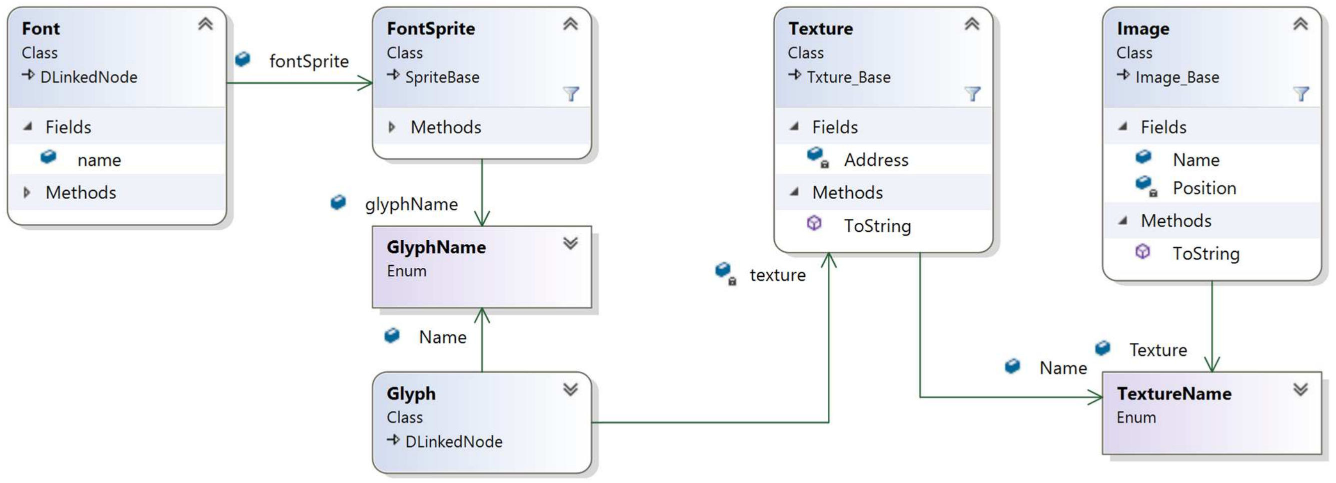

Figure 3: UML of Flyweight Pattern

Details of Implementation:

a. Font Class has a pointer to FontSprite, which in turn points to Glyph Class, which points to the Texture Class. In consequence, all font points to only one texture, so the texture is the shared part and font are the different part.

b. Similarly, all images has a pointer to the texture class, even though it is represented by TextureName, it can easily find the real texture. I used only 3 textures in the whole program, but they contain 20-30 images and 50-60 fonts.

Problem 4:

Sprite is an expensive object to create, in the game, we need to add, move, delete them all the time. However, we do not to operate on such expensive object, instead, we use smaller object to control the behavior of it. What should we do next?

Design Pattern: Proxy Pattern

Sometimes we need the ability to control the access to an object. For example, if we need to use only a few methods of some costly objects we'll initialize those objects when we need them entirely. Until that point, we can use some light objects exposing the same interface as the heavy objects. These light objects are called proxies and they will instantiate those heavy objects when they are really need and by then we'll use some light objects instead. So that proxy pattern can provide a placeholder for another object to control access to it, and only keep an instance of smaller object using a full feature object

Pattern in the game: Proxy Sprite

In this game, we have 55 aliens moving all the time, and hundreds of shield objects on the screen, instead of keeping all the features of an object, we actually only need (x ,y) coordination of it. Every time we add, move, update, remove an alien or a shield, we only need to update the (x, y) position of it. The only time we need the whole object is when we draw it on the screen, so at that time, we can push the proxy sprite to real.

Details of Implementation:

Each game object uses a proxy sprite, every time we update a state of a game object, e.g. Aliens are moving, we only need to update its proxy's (x, y) position. Then, when the proxy is going to be drawn, it pushes the position to the real object.

The major difference between Flyweight and Proxy, is that, flyweight pattern shares the common part, but they cannot mutate it, so the common part is immutable. However, the proxy pattern may also use a common part, but the common part is mutable, and each proxy has fully control of it.

UML:

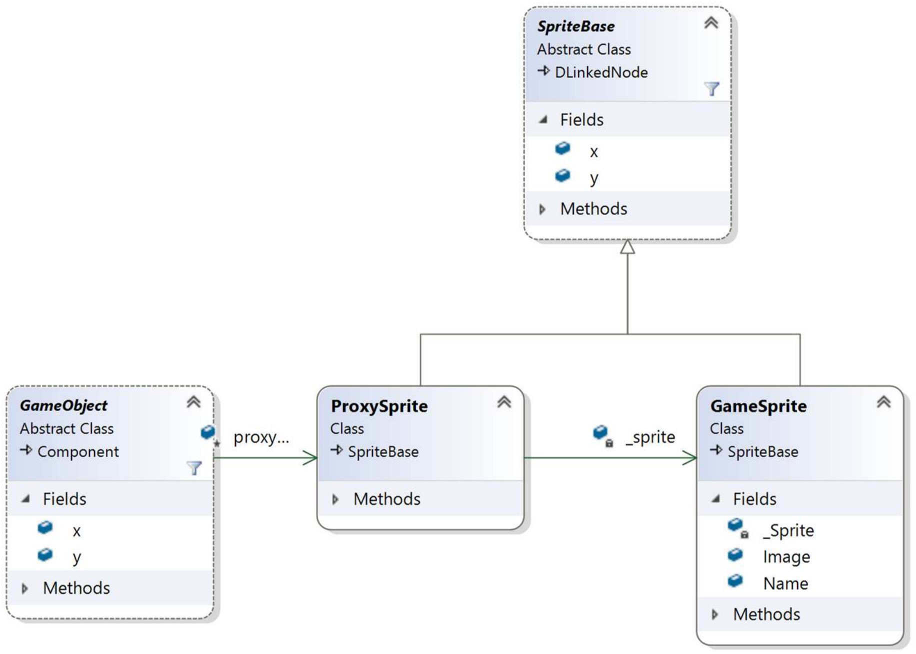

Figure 4: UML of Proxy Pattern

Problem 5:

Different objects have been but in their object pool, now we have other classes to use them. We want to make sure that every time we want to refer an object, we are using the same and only one pool. What should we do?

Design Pattern: Singleton Pattern

Sometimes it's important to have only one instance for a class. Singleton Pattern involves only one class which is responsible to instantiate itself, to make sure it creates not more than one instance; at the same time, it provides a global point of access to that instance. In this case the same instance can be used from everywhere, being impossible to invoke directly the constructor each time. Singletons also ensure that "just-in-time initialization" or "initialization on first use". If we don't use it, it will not be initialized. Singleton is unlike static classes or fields, which always sit in the memory stack.

Pattern in the game: Managers

Form previous discussion, we have put all the objects in their object pool, now when we want to get an object, we always want to get it from the same pool. In this case, we create all managers using Singleton Pattern. Now, when the first time we want to use a manager, we instantiate it, and keep a global pointer to it. Next time, we always get to the same object manager without create it again.

UML:

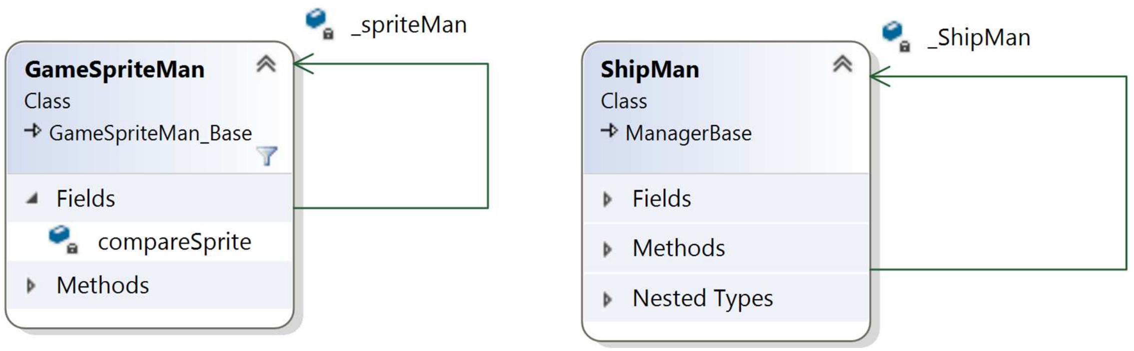

Figure 5: UML of Singleton Pattern

Details of Implementation:

First, we have to have a private constructor to make sure no other classes could initialize the Singleton. Second, we need to have a private member of an instance of singleton itself. Third, we need a public GetInstance() function to get that private instance.

Now, the first time we call GetInstance(), it create a new manager instance and store it in the private member. After that, each time we want to use it again, and call GetInstance(), it will return us the created private instance, and no more new objects is created.

Problem 6:

Since we are using proxy to move our sprite, but at the same time, we also want to change the image of the sprite to form an animation. However, the proxy has no knowledge of the image, what should we do?

Design Pattern: Command Pattern

Command design pattern encapsulates commands (method calls) in objects allowing us to issue requests without knowing the requested operation or the requesting object. Command pattern provides the options to queue commands, undo/redo actions and other manipulations.
Command pattern promote "invocation of a method on an object" to full object status, is used to model an action you need more to do with than just execute it now.

Pattern in the game: Animation Command, Remove command.

In this game, I use lots of command pattern, animation command to change image of a sprite, aliens shoot command to shoot a bomb, UFO moving command to let a UFO move. Also, when a game object gets shot and is going to be removed, I mark it for death at first, and then, execute remove command of that object. Since animation command have to change image contiguously, so we add a timer to control it, and an image holder to hold all the images.

Details of Implementation:

a. Animation command is derived from command base. Each time we want the proxy sprite to draw a different, the proxy sprite does not know the details how to change the image of a sprite, what it does is only to draw whatever in the sprite. Instead, we use animation command to change the images of the sprite.

b. In order to form an animation, we need to change the image repeatedly, so we need a timer to control it, after certain time, the command will run.

c. When the animation command run, it changes the images in its image item list, and all the image items are in a Double linked List. Every time it reaches to the end, it goes back to the front again.

Other similar commands are added under the same desire. After a certain time, aliens will shoot a bomb, and UFO start moving. However, ship is not shoot regularly, which shoot only after we hit space bar, so it is not added here.

UML:

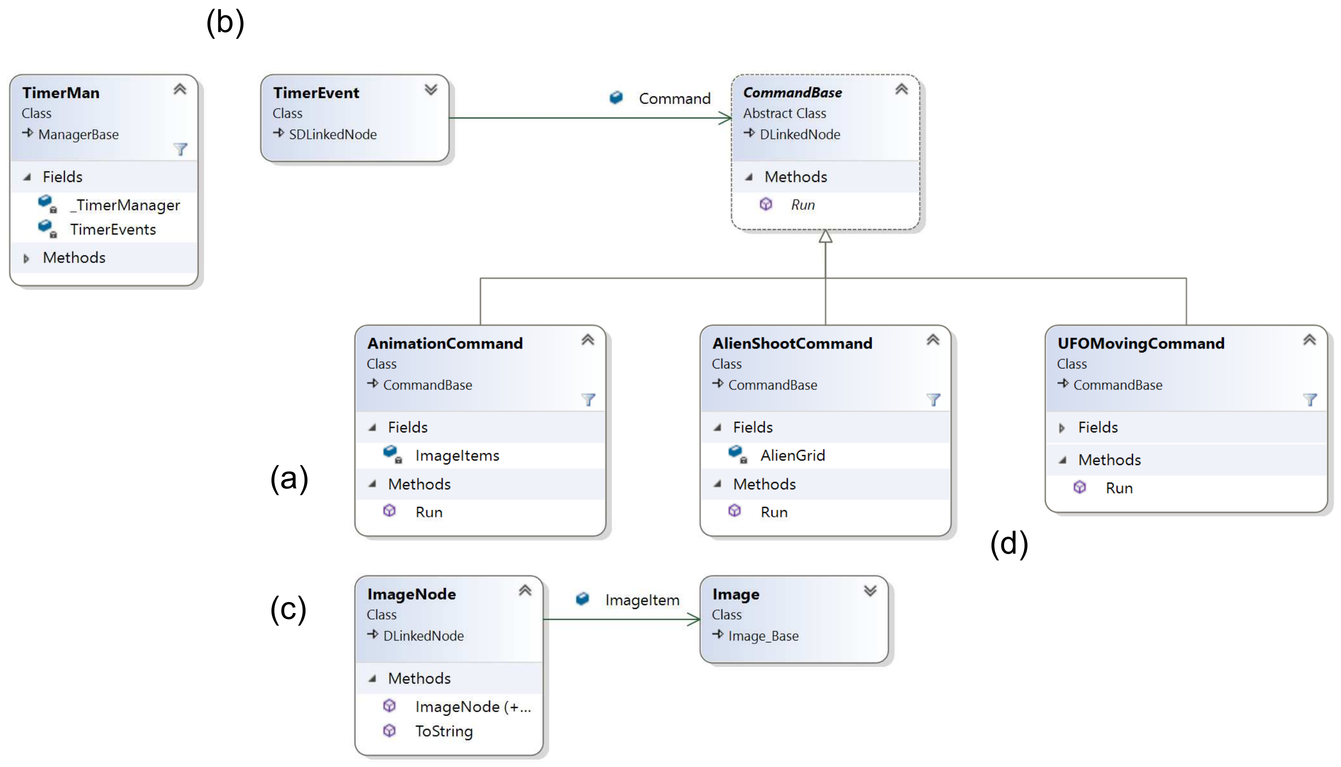

Figure 6: UML of Command Pattern

Problem 7:

Every time, when we want to create a texture or image object, we need to know the exact parameters of it, but we want to hide the details of all the parameters from client, what should we do?

Design Pattern: Factory Pattern & Factory Method

Both factory patterns are used to create objects without exposing the instantiation logic to the client. And refers to the newly created object through a common interface.

Factory Pattern: When client needs a product, but instead of using “new”, it asks the factory object for a new product, providing the information about the type of object it needs. The factory instantiates a new concrete product and then returns to the client the newly created product.

Factory Method Pattern: Defines an interface for creating objects but let subclasses to decide which class to instantiate. Concrete Creator overrides the generating method for creating objects Since they look similar, they can be changed into each other very easily. The major difference is that, factory pattern has different create functions to create different object, whereas factory method is an interface which has only create method, and let its subclass decide which it is going to create.

Pattern in the game:

Some expensive objects, such as Shields, Images, and so on, I use factory pattern to create them. Instead of writing some detailed process of creating such expensive objects, just wrap all the creating semantics into a factory. Now when we need an object, we just call the factory to create it.

Details of Implementation:

a. When Texture Manager want to create a texture, it just calls the Texture Factory’s corresponding methods to create the needed object without knowing the details of how to create it. Similarly, when creating a shield object, we only call the shield factory to create a shield for us, without knowing all the different relative locations of all the bricks. Since the UML of a shield factory is less exciting, I won’t put it here.

b. When Image Factory want to create a new Image, it calls the sub-factories directly to create corresponding one. This is a Factory Method pattern. In this game project, these two patterns are interchangeable, so I put them together here. However, Factory method pattern can also have other functions working on the object, which can be used in the real life.

UML:

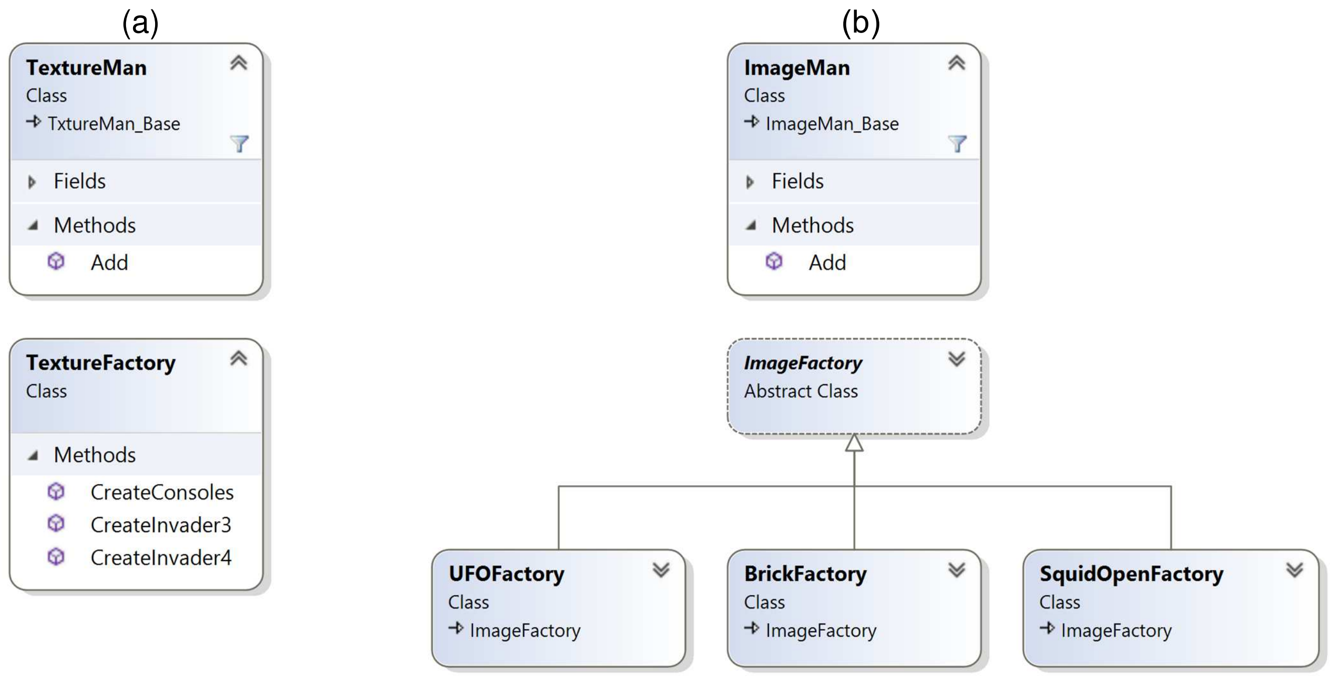

Figure 7: UML of Texture Factory and Image Factory

Problem 8:

Now, we want to build the real game, we notice that all the aliens can be grouped by column, and all the column can be put in a grid. How can we keep the objects ordered as a tree?

Design Pattern: Composite Pattern

There are times when a program needs to manipulate a tree data structure and it is necessary to treat both Branches as well as Leaf Nodes uniformly. The intent of this pattern is to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. Every time we call a function, it is recursively called both in the branch and the leaf.

Pattern in the game: All Game Objects (Aliens, Shields, Bombs)

In this game, all the real game objects are put into a tree structure because we have to detect collision all the time, if we always check all the movable parts, it will waste lots of time and resource. After we put every objects in a tree structure, we just have to check if the two roots have collision or not, if they did not collide, we can skip all the branches and leaves. When the two roots collide, we can check which branch and leaf is collided, this would save us a lot of time and effort when run the program.

UML:

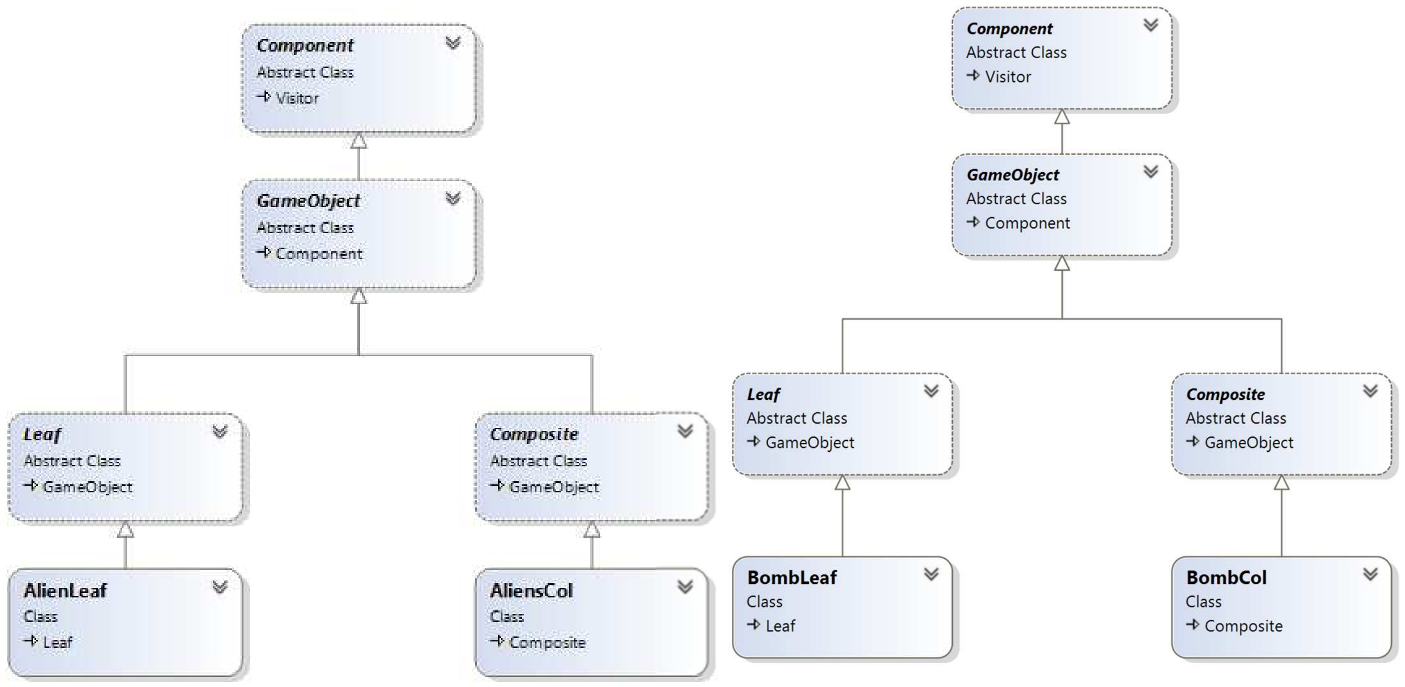

Figure 8: UML of Composite Pattern

Details of Implementation:

At the beginning, we have a component abstract class, both branch (composite) and leaf are derived from component, and both have the same function that is defined in the component. The only difference is the implementation, we a function is called from leaf, it will do something, but when it is called by composite, is will call its child the same function. So that, with only one call to the root, we can get a recursive call to all its descendants.

Problem 9:

After we put everything in a composite pattern, it is in a clear structure, now when we want to get to the descendants, we do not want to know the detailed implementation of the composite, if we could treat every thing like a tree, when we need destination object, we just call next to get to it, wat should we do?

Design Pattern: Iterator Pattern

A collection should provide a way to access its elements without exposing its internal structure. We should have a mechanism to traverse in the same way a list or an array. It doesn't matter how they are internally represented. The idea of the iterator pattern is to take the responsibility of accessing and passing through the objects of the collection and put it in the iterator object. The iterator object will maintain the state of the iteration, keeping track of the current item and having a way of identifying what elements are next to be iterated. Iterator pattern provides an abstraction that makes it possible to decouple collection classes and algorithms.

Pattern in the game: Game objects, Collision pairs

In this game, if two objects get collided, we want to know which two objects are collided. Without knowing all the details of how to traverse the tree structure, we just call Next() function to get to the next object. When we update a composite's position and its collision box, we have to update all its children first, because the position of the composite is determined by its children, now we need a traverse iterator to execute function calls from the very bottom.

UML:

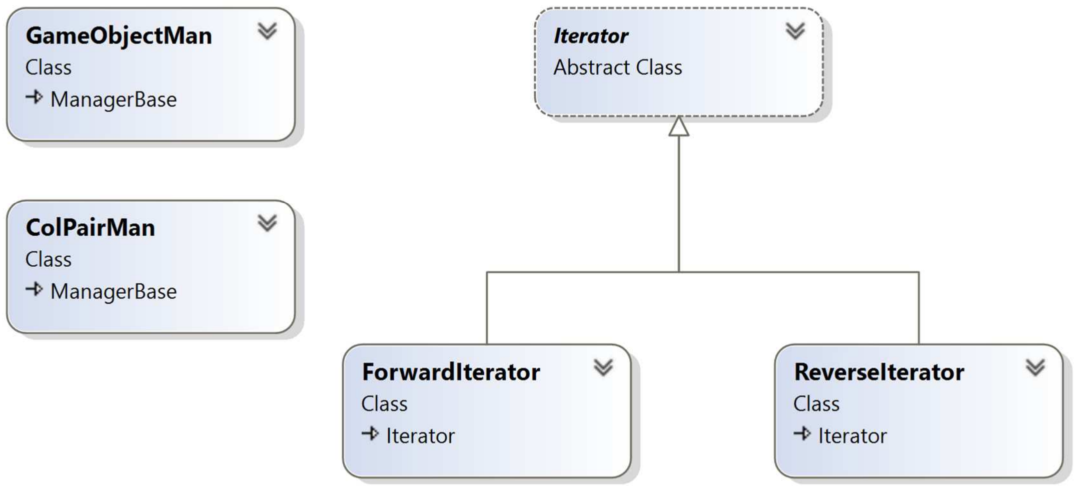

Figure 9: UML of Iterator pattern

Details of Implementation:

In the Collision pair manager, when we detect two object roots get collided, we want to know which descendant is really get collided. So that, we traverse from all the children of a tree and compare with the other tree.

When we update the game objects, we have to calculate its (x, y) position based on its children, so this time, we have to iterator form the bottom of the tree, so we have a Reverse iterator class, and the implementation is just to copy the forward iterator from end to head.

Problem 10:

As we have talked about collisions in the previous problems, now we have another problem, when 2 objects are collided, how could one object to visit a function of the other object, and they are not connected before collision happen?

Design Pattern: Visitor Pattern

Visitor Pattern lets you define a new operation without changing the classes of the elements on which it operates. It represents an operation to be performed on the elements of an object structure. The classic technique for recovering lost type information. Do the right thing based on the type of two objects.

Pattern in the game: Game objects

In this game, when two objects are collided, as we discussed before, we need iterator to traverse both of those trees. However, each step, we just travers one level deeper, and to determine which real child get collided. When detected the collided child, we will let the child to visit the other tree, and so forth. In each step, we let one object to accept the other object, then the other object could visit the calling object, until we find the leaf of both trees.

Details of Implementation:

First, we create a abstract visitor class, which has Accept() and Visit() functions. For all the game objects that is going to be visited, is derived from the Visitor class. Since different game objects has to visit each other, all of them must implement the Accept() method. And for the Visit() method, we just implement those that is going to be used. When new collision pairs are added, the corresponding Visit() method should also be implemented.

At the final stage of developing the game project, the main work is to figure out all the collision pairs and to implement the Visit() methods and their observers which we will discuss later.

UML:

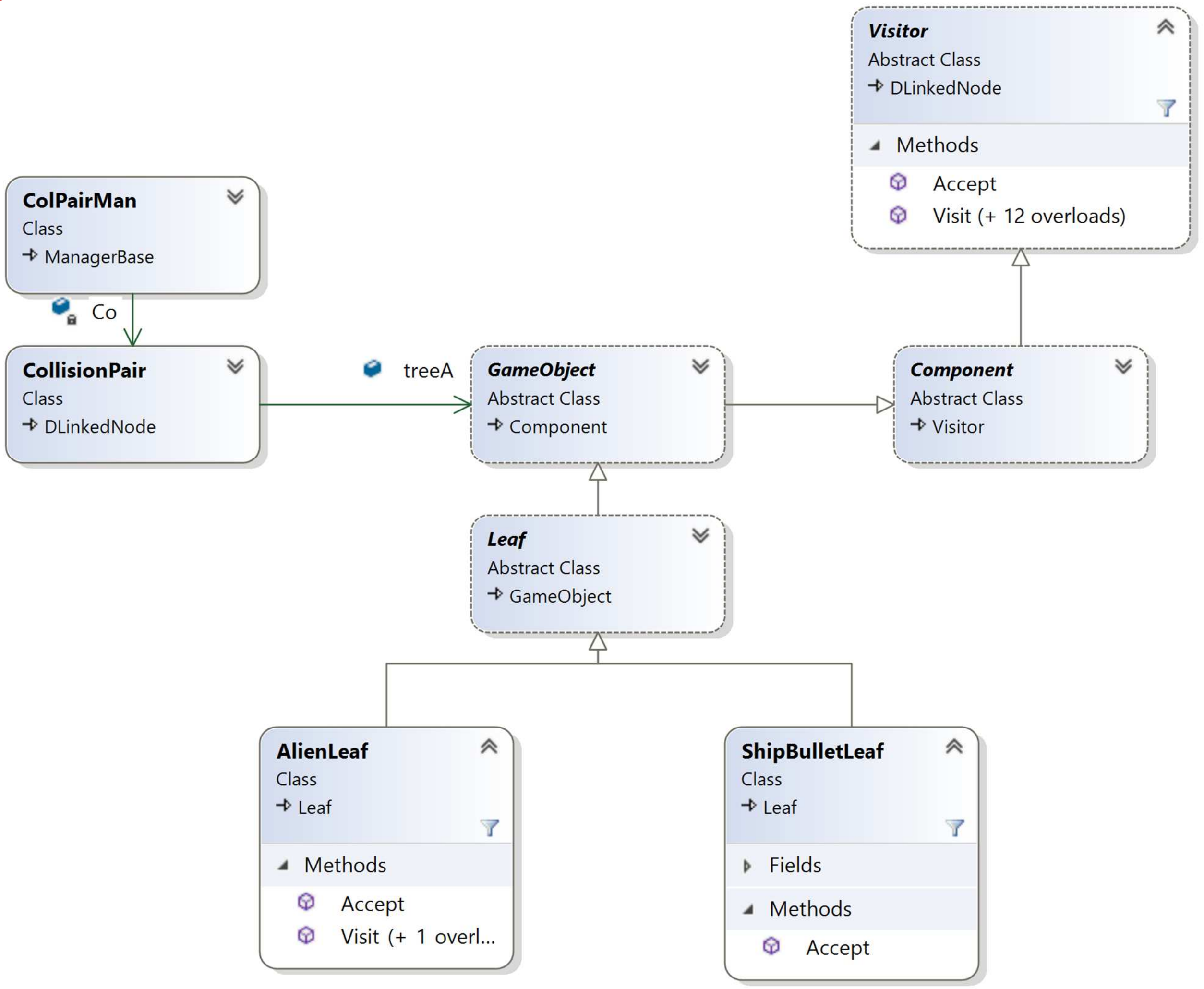

Figure 10: UML of Visitor Pattern

Problem 11:

After we detect which two leaves are collided, we have to do something on these nodes. For example, delete the node or reset it to original position or something else. Also, we want to do some other things not related to the 2 nodes, for example, play crash sound, change explosion image, and so on. What should we do to notify all those objects to do their desired work?

Design Pattern: Observer Pattern

All object-oriented programming is about objects and their interaction. The cases when certain objects need to be informed about the changes occurred in other objects are frequent. To have a good design means to decouple as much as possible and to reduce the dependencies. The Observer Pattern can be used whenever a subject has to be observed by one or more observers. Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. Observer is used when the Observable needs to notify other objects about certain events, but doesn't know “which” objects to notify.

Pattern in the game: Collision pairs

In this game, when ship missile hit an alien, it has to delete the alien, delete the missile, reset the ship state, play “Boom” sound, and so on. After such a collision happens, we just notify all the listeners connected to them.

UML:

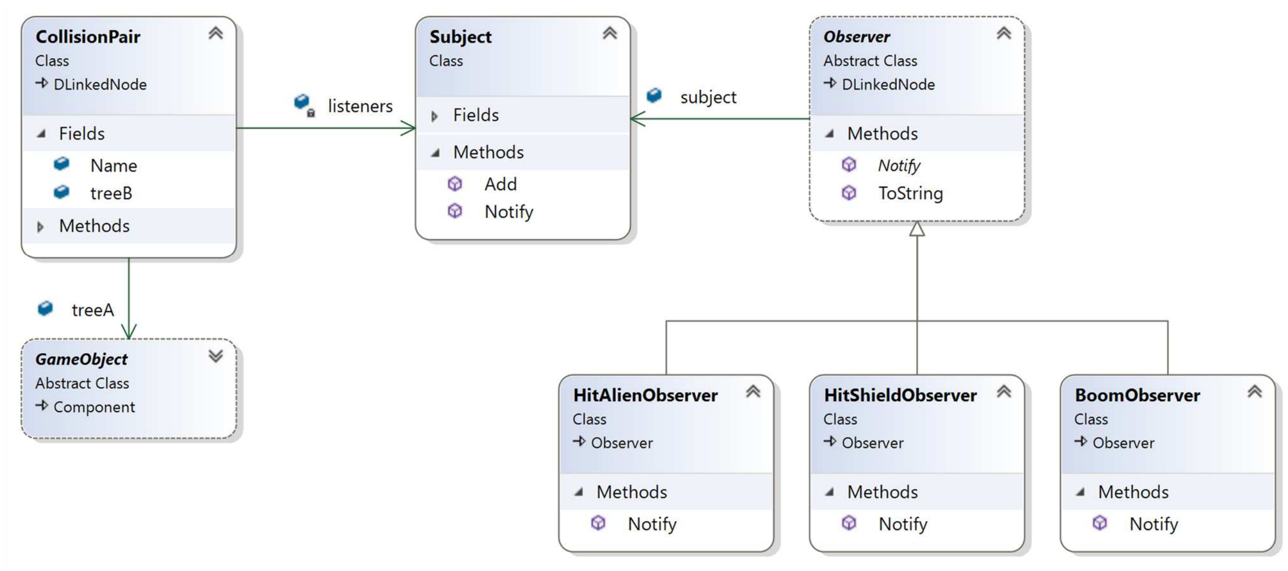

Figure 11: UML of Observer Pattern

Details of Implementation:

First, We create an Observer abstract class which has only one function Notify(), and all the real observers are derived from it.

Then, in the collision pair, we have a pointer to a subject class, which in turn holds all the observers, when the two Game Objects in the collision pair hit each other, they will notify the all the listeners stored in the subject.

When we create a collision pair, we also add all need listeners to it, so that almost every observer can be used in different situation. For example, the “Boom” Observer can be used in both missile hit alien and bomb hit ship.

Problem 12:

A ship can have different states: ready state, which means ship can move and shoot; Missile flying state, which means ship can move but not shoot; And death state, it can neither move nor shoot. How can we change the state of a ship instead of change to different object?

Design Pattern: State Pattern

State Pattern allows an object to alter its behavior when its internal state changes. But it does not specify where the state transition will happen. State pattern allow the same object changes its pointer to different state classes, which is different from strategy pattern.

Pattern in the game: Ship State, Game State

In this game, as we shown in the problem, ship may have different states. Every state, the same ship can have different behavior. Since the state pattern does not know when to change the state of a ship, we always let use the observers to notify the change of state.

Also, the game have different state, Select State, Play State, and Game Over State, in different state, the game looks quite different, but as long as we understand the implementation is similar to the Ship state, and know changing from one state to another state is based on the observer, it is not difficult to implement.

UML:

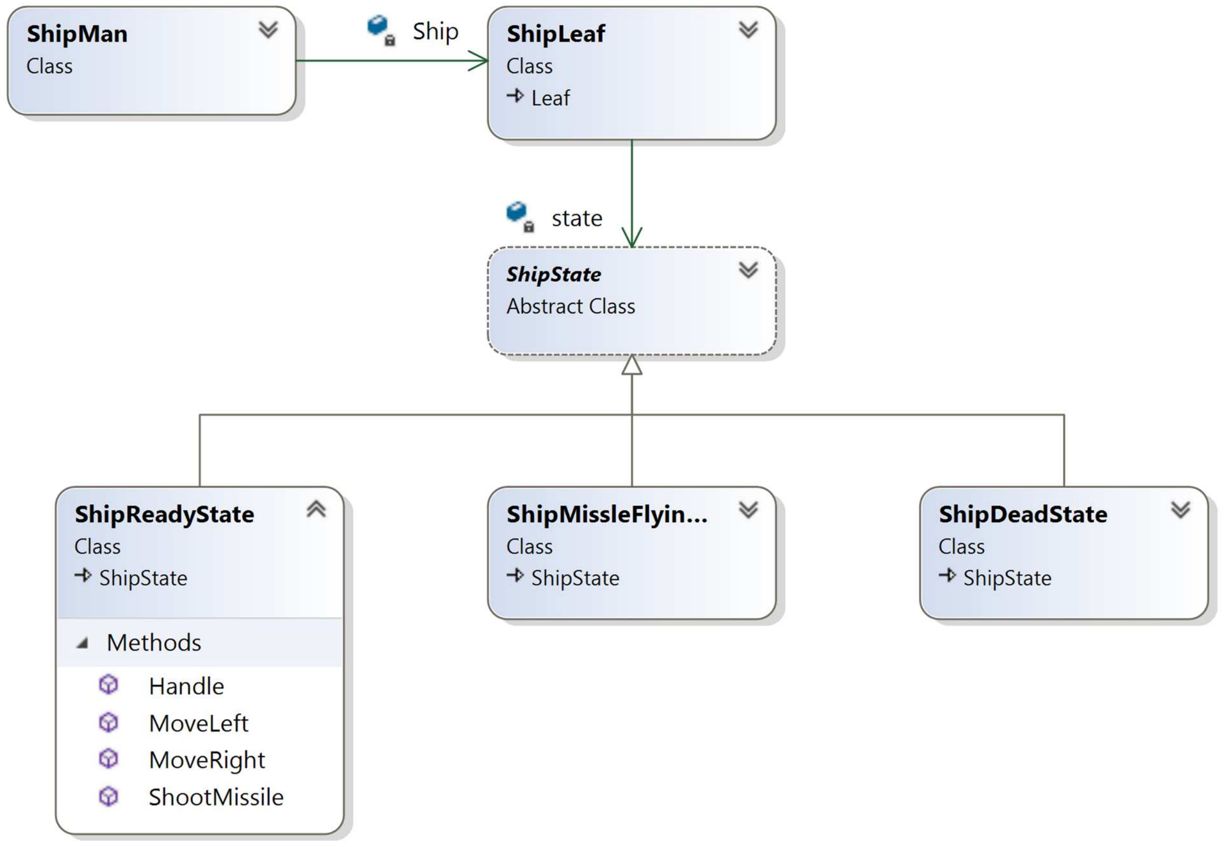

Figure 12: UML of State Pattern

Details of Implementation:

We create a Ship State abstract class, and different ship state have to implement differently on their inherited functions. When a ship shoots a missile, the state is changed from Ship Ready State to Missile Flying State. So, the pointer saved in the ship leaf class is changed to different concreate state.

Similarly, when a game scene is changed its state, it should call change its state pointer to different scene.

Problem 13:

When aliens drop a bomb, different bomb should have different falling style, and we do not want to swap images of the bomb, then what should we do?

Design Pattern: Strategy Pattern

There are common situations when classes differ only in their behavior. Isolating the algorithms in separate classes is a great choice in order to have the ability to select different algorithms at runtime. Strategy pattern can define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Pattern in the game: Bomb fall strategy

In this game, the bombs are required to fall differently, some are falling zigzag, some are falling flip over and some are falling straight. In this case, we create different strategies to achieve this goal. Before a Bomb is create, it does not know what it is going to fall, but it knows it will follow some Fall Strategy. After it is created, it will always fall as that concreate strategy, and the strategy cannot be changed.

Details of Implementation:

We Create an abstract Fall Strategy class, and all the different concreated fall strategies have different implementation of it. We a bomb leaf is created, we also pass the desired fall pattern as a parameter to get the bomb. Even though the bomb may be recycled later, it always keeps the same strategy, and never will change it. This is the main difference between strategy pattern and state pattern.

UML:

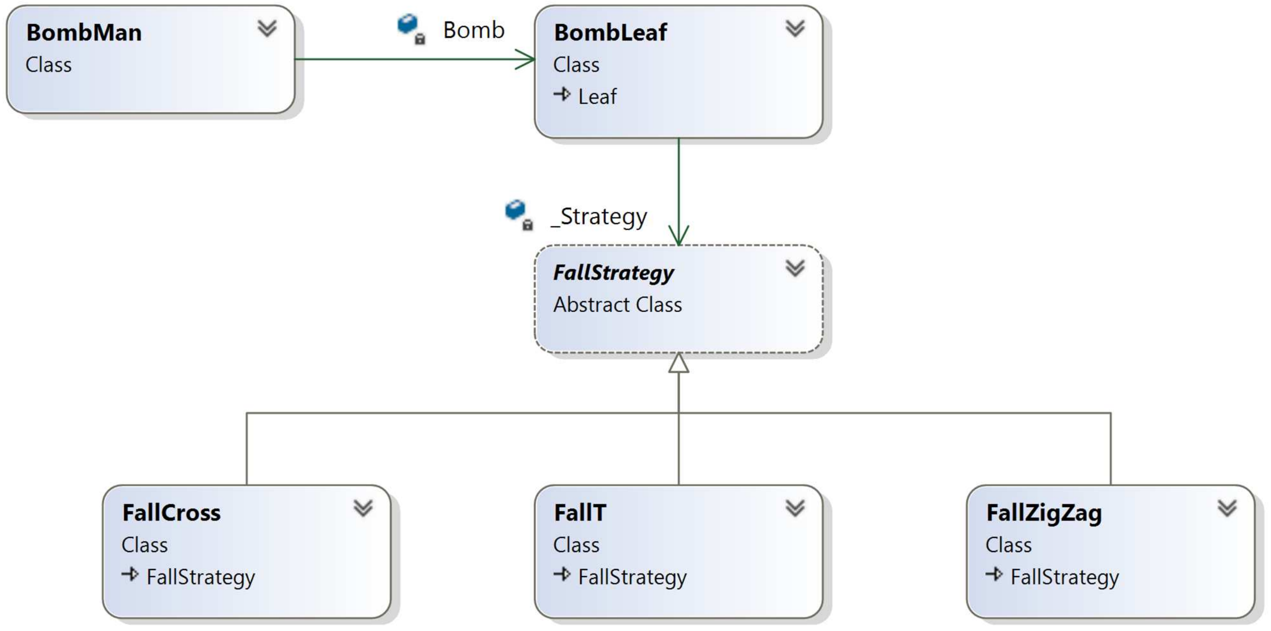

Figure 13: UML of Strategy Pattern

Problem 14:

When developing the whole program, we use a lot of classes derived from their base abstract class, and in the base class, only abstract operations are defined. Each derived class has to define their own concreate operation to override the base abstract operation. Is there a pattern related to it?

Design Pattern: Template Pattern

Template Pattern defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. It lets subclasses redefine certain steps of an algorithm without letting them to change the algorithm's structure. Base class declares algorithm “placeholders”, and derived classes implement the placeholders.

Pattern in the game: All class name end with base, and Game Object

This is a natural way to write object-oriented programing, every class we want to inherit from an abstract class or an interface, both may define some functions that have to be implemented, but without too much details, whereas all the subclasses derived from the base class have to override those functions with concreate implementations.

UML:

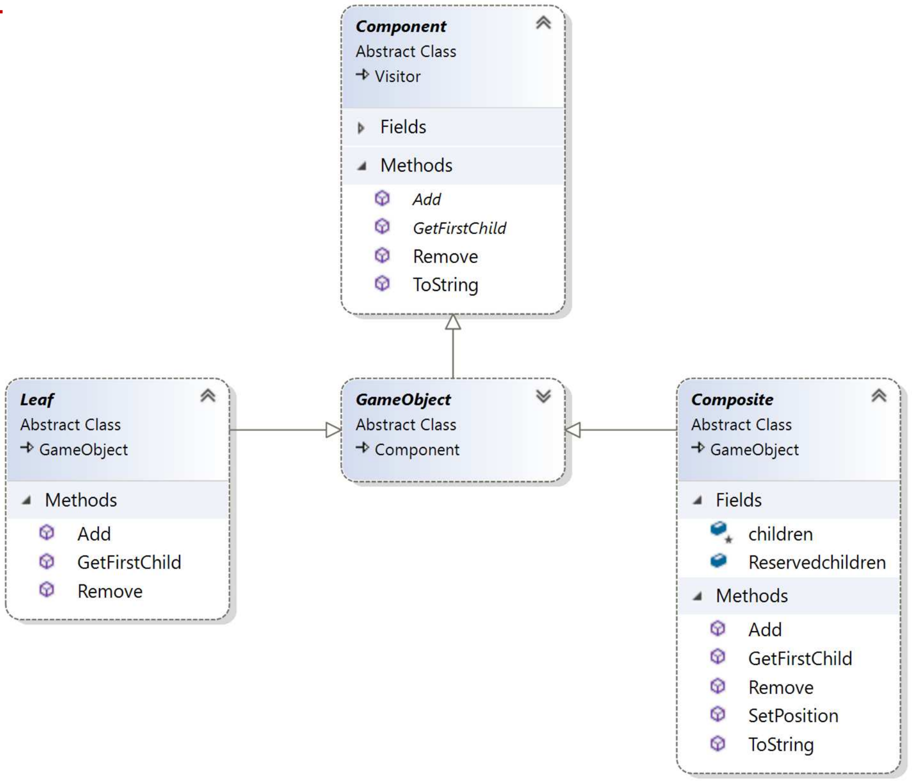

Figure 14: UML of Template Pattern

Details of Implementation:

This UML is from the Composite pattern, but this time we focus on the methods of them. The base abstract class has 3 abstract methods, Add, Remove, and Get First Child, both of the derived classes, composite and leaf, has to implement all of them. However, it does not make sense to Get Child of a leaf, (actually all of them does not make sense). So the leaf class just do nothing, which means the functions just do nothing or return null, but it has to implement them.

Problem 15:

As we discussed in the previous problem, the leaf does nothing when try to get its child. Sometimes, we need an object to do nothing, what should we do?

Design Pattern: Null Object Pattern

Null object pattern provides an object as a surrogate for the lack of an object of a given type. The Null Object Pattern provides intelligent do-nothing behavior, hiding the details from its collaborators.

Pattern in the game: Null Game object

In this game, the null game object, is just as it defined, all the methods are returning null, or doing nothing.

Details of Implementation:

Implementation of null objects is so straight forward. As Figure 15 shows, all the methods which are defined in the abstract class as abstract method should get a concrete implementation in the null object. However, all the concrete implementation is just doing nothing.

UML:

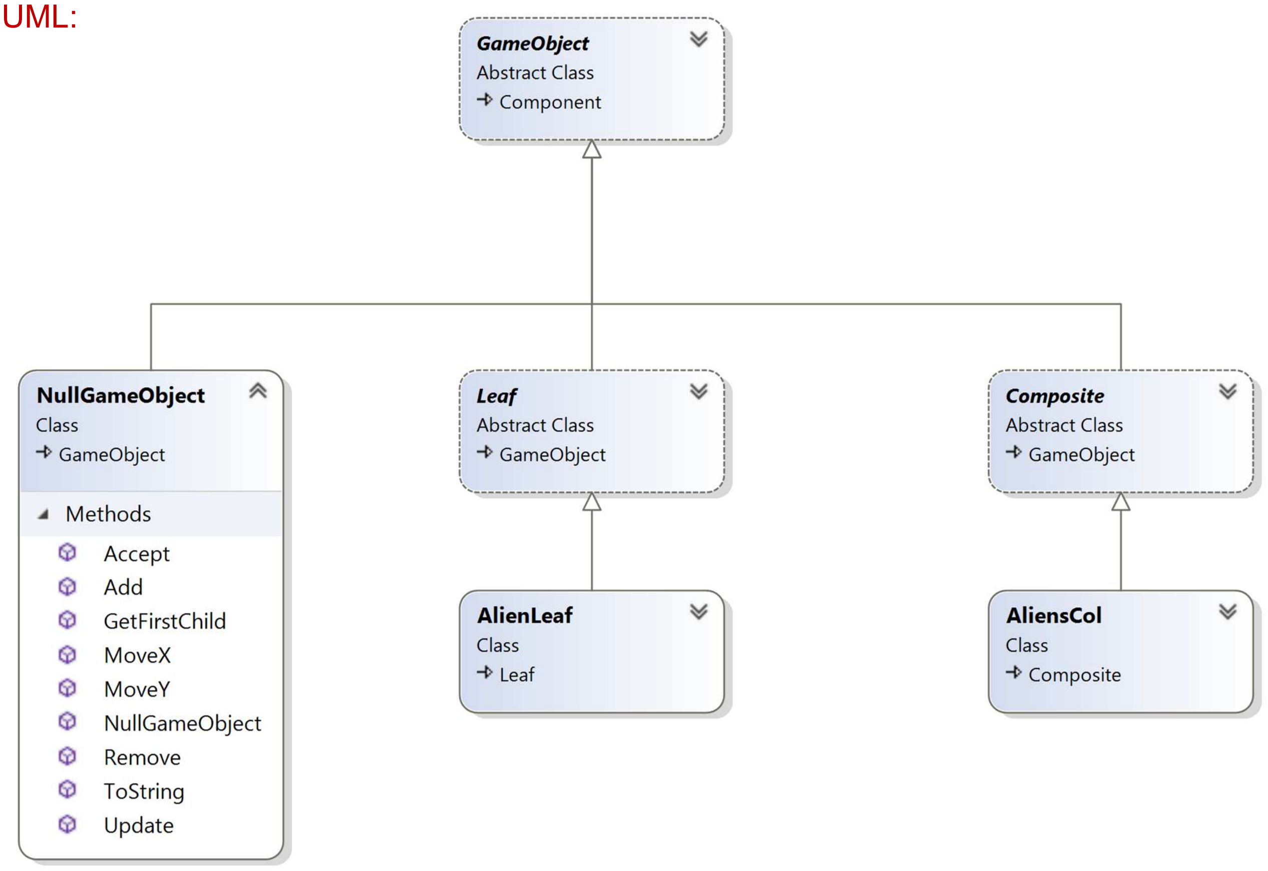

Figure 15: UML of Null Object Pattern

Conclusion

In the development of the whole space invaders game, as discussed in the previous section, at least 15 design patterns are used. And during the developing process, I tried my best to follow SOLID principle.