SE 456

SPACE INAVDERS

Design Document

By Shashank Indukuri

[Demo Link]

Table of Contents

1. Introduction: ---------- 3
2. Design Patterns: ---------- 3
2.1 State Design Pattern ---------- 3
2.2 Iterator Design Pattern ---------- 5
2.3 Strategy Design Pattern ---------- 6
2.4 Factory Design Pattern ---------- 8
2.5 Proxy Design Pattern ---------- 9
2.6 Visitor Design Pattern ---------- 11
2.7 Observer Design Pattern ---------- 13
2.8 Command Design Pattern ---------- 14
2.9 Composite Design Pattern ---------- 16
2.10 Object Pooling Design Pattern ---------- 17
2.11 Singleton Design Pattern ---------- 19
2.12 Flyweight Design Pattern ---------- 20
2.13 Adaptor Design Pattern ---------- 22
2.14 Null Object Design Pattern ---------- 23
2.15 Template Design Pattern ---------- 24
3. Post-Mortem: ---------- 25

1. Introduction:

The design document describes the process of creation of a Space Invaders game using efficient techniques from various design patterns. The document lists the features and design patterns that were used and implemented during the game's development. This game was created primarily with the Azul framework as a foundation and using the simple C# and Visual Studio.

The goal of this project is to create a 5 x 11 alien grid that marches from left to right and top to bottom over time in the Space Invaders game. The player, as a ship at the bottom, must shoot the aliens by moving horizontally. There are three screens in the game: Home, Play, and Game Over. A player can begin the game by pressing a key and has three lives.

The scores and the number of lives is displayed using font system. Points are added to the scoreboard for each type of alien's death. Aliens can drop bombs at random, and a special type of alien UFO can cross in different directions with specialized points. The game is also implemented for two player mode.

2. Design Patterns:

2.1 State Design Pattern

Problem:

One of the challenges in the Space Invaders game is regulating the missile's state. The ship can only launch one missile at a time and must manage the ship's inputs.

Solution:

I utilized a state pattern to keep the missile's current state in the ship class and managed missile shooting depending on the ship by simply altering the ship's current state and implementing the appropriate methods.

Pattern Description:

The State design pattern manages the change of behavior of an object based on its different states. It modifies the behavior without changing the class. To prevent several conditional statements that are dependent on the object's state. The State pattern helps to avoid the large number of conditional statements in a single file by splitting into various classes.

It is very beneficial for reducing the number of objects created for different behaviors. It is similar to the strategy pattern. The main difference between strategy and state is that the state context has a replaceable connection and may simply change the state behavior by modifying the context's present state.

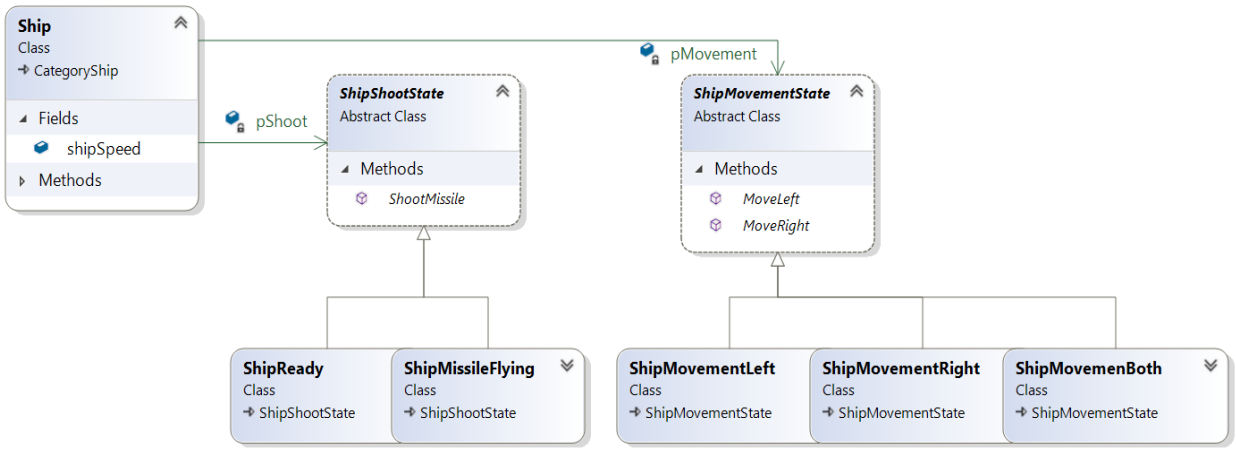
Key OO Mechanics:

The initial step is to create an abstract class and create subclasses for different behaviors by inheriting the abstract state class. In the context of the state class, that manages the current state and triggers the respective state methods.

It is quite easy to change the current state of the context from the state class method. It is achieved by replacing the link with the new link by changing the object reference. In this way, I can easily use the state and manage different states efficiently. The classes needed for this state pattern are the Context class

that holds the current state, an abstract class, and the subclasses that are implemented from the abstract class.

- Context: It acts as an interface for the clients to perform some actions and hides the internal behavior changes. It holds an instance of the current state.
- State: It is an abstract class or interface that defines the methods that needs to be overriden by base classes.
- Concrete States: These are the subclasses for the abstract class State and implement their own behavior logic.



The above UML diagram is an example for State pattern for Ship game object.

Uses:

I used the state pattern to restrict the missile from the ship. As per the requirements, the ship can only shoot the missile once until the existing missile dies. To overcome this problem, I implemented a state pattern for maintaining the state of the missile in the Ship class. I utilized two separate states in this case: Ready, and Missile Flying.

First, I developed an abstract class, ShipShootState, which has abstract methods for dealing with various behaviors. Second, I inherited the parent class and created two new ones: ShipReady, and ShipMissileFlying. These classes have their own behavior and override the abstract classes.

When the missile launches, the ship's state changes to missile flying, and the shooting method in the state has no effect. As a result, the ship cannot fire another missile until the status of the ship returns to "Ready."

Furthermore, I utilized it to regulate the ship's movement. The ship is controlled by input keys and uses the move methods on both sides to move. Suppose if the ship reaches the left wall, the MoveLeft () method should not move further to the left and only the MoveRight () method should work. Similarly, if the ship reaches the rightmost wall, MoveRight() should not do nothing, and MoveLeft() works. The third state is the normal state where the ship can move in both directions. In this way, the state pattern is used to manage both missile and ship movement in the Space Invaders.

2.2 Iterator Design Pattern

Problem:

We used various data structures in Space Invaders, such as LinkedLists and Tress, to store various nodes such as sprites, gameobjects, and many others. The methods for navigating to each node in the data structure are difficult to remember.

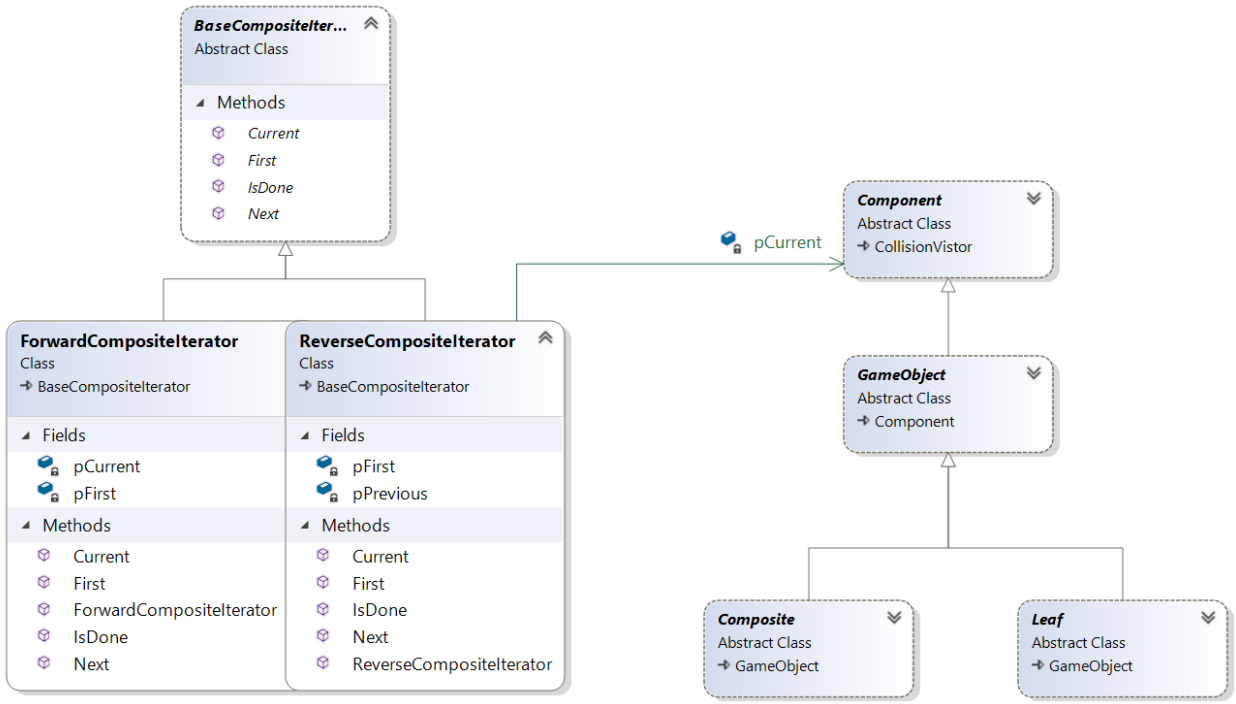
Solution:

I used an iterator pattern to navigate and walk through the nodes of various data structures sequentially by hiding the inner structure. Some of them include iterating over the alien tree, a sprite batch manager for drawing sprite batches sequentially, and many others.

Pattern Description:

The iterator pattern is very useful for sequentially iterating aggregated objects without knowing the structure of the object.

The primary application is to hide the complexity of aggregate objects and to aid in the reducing of decoupling on that object.



The above UML diagram is an example for Iterator pattern to iterator for Composite.

Key OO Mechanics:

The first step in implementing the iterator is to create an abstract iterator with abstract methods like First(), Current(), Next(), and IsDone() to access the elements. To iterate the aggregate, we extend the abstract class and implement the abstract methods.

- Base Iterator: It serves as an interface for clients to access the aggregate object elements sequentially.
- Concrete Iterator: It is a concrete class that extends the base iterator and provides the logic for the contract methods.
- Aggregate: This is a class that holds various data structures such as Linked Lists and Trees.

Uses:

In many places in Space Invaders, I used the iterator pattern. One of the most important places is to iterate over the composite trees that I've built for Aliens. To accomplish this, I created two Iterators: Forward Iterators and Reverse Iterators. These iterators are subclasses of the BaseCompositeIterator class. As we can see in the UML, pCurrent has an aggregation to Component. As a result, in the game, we use Forward Iterator to display and draw the aliens from the Composite, which iterates the entire structure from group, grid, columns, and game objects (Aliens).

Similarly, Reverse Iterator is used to remove the elements one by one from the bottom of the tree. Because this is maintained as a Tree, it is suggested that they be deleted from the bottom of the tree.

Furthermore, I created DoubleLinkItertor to iterate over a double linked list, and SingleLinkItertor to iterate over a single linked list. In this case, a Double Linked List is used in managers, and thus it is simple to access the elements in managers by retrieving the iterator and accessing them with ease.

2.3 Strategy Design Pattern

Problem:

There is a requirement in Space Invaders that different bombs be deployed at random by the aliens.

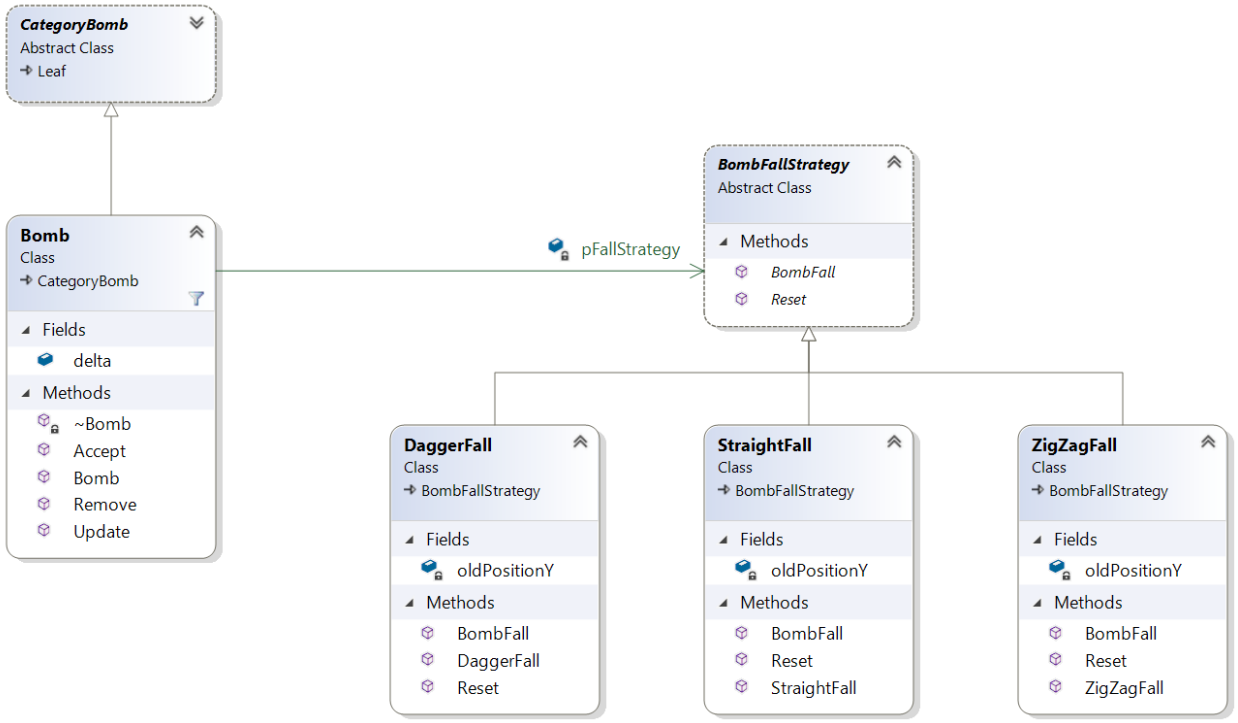
Solution:

I used the Strategy pattern to define three distinct bomb strategies with distinct behaviors. As a result, it is simple to define the behavior of each bomb without changing the structure.

Pattern Description:

The Strategy pattern contains multiple classes, each of which defines its own behavior, and the client can access the object with a different behavior depending on the usage.

It is simple to change the behavior here without affecting the client's usage. As a result, it conceals the implementation details while providing abstraction.



The above UML diagram is an example for Strategy pattern to implement the Bomb Strategy

Key OO Mechanics:

It is simple to design and implement a strategy that resembles the state pattern. Initially, an abstract class with an abstract method is created. Now, different subclasses must be created and the abstract method must be overridden based on their usage. As a result, clients have access to the same abstract method but can behave differently depending on the link in context.

- Base Strategy: It serves as an interface, providing the abstract methods that must be implemented.
- Concrete Strategy: A concrete class that extends the base strategy and adds their own behavior.
- Context: It has access to the methods and has an object reference to the Concrete Strategy.

Uses:

I used the strategy pattern in Space Invaders to implement three different bombs that fall from the aliens. Normally, it's difficult to implement three distinct classes for three distinct behaviors, and I also need to create distinct Bomb classes. However, by utilizing strategy, I was able to create a bomb class that contains an instance of bomb strategy.

As a result, by removing any edge conditions and complexity, we can easily create three different instances of bombs with different links and achieve different behaviors with the same abstraction.

Furthermore, it is very simple to extend Bomb's behaviors by simply adding a new subclass for Base Strategy and implementing the BombFall and Reset methods.

2.4 Factory Design Pattern

Problem:

It is necessary to create many game objects, and the same objects in different locations, in Space Invaders, such as 55 aliens and 4 different shields.

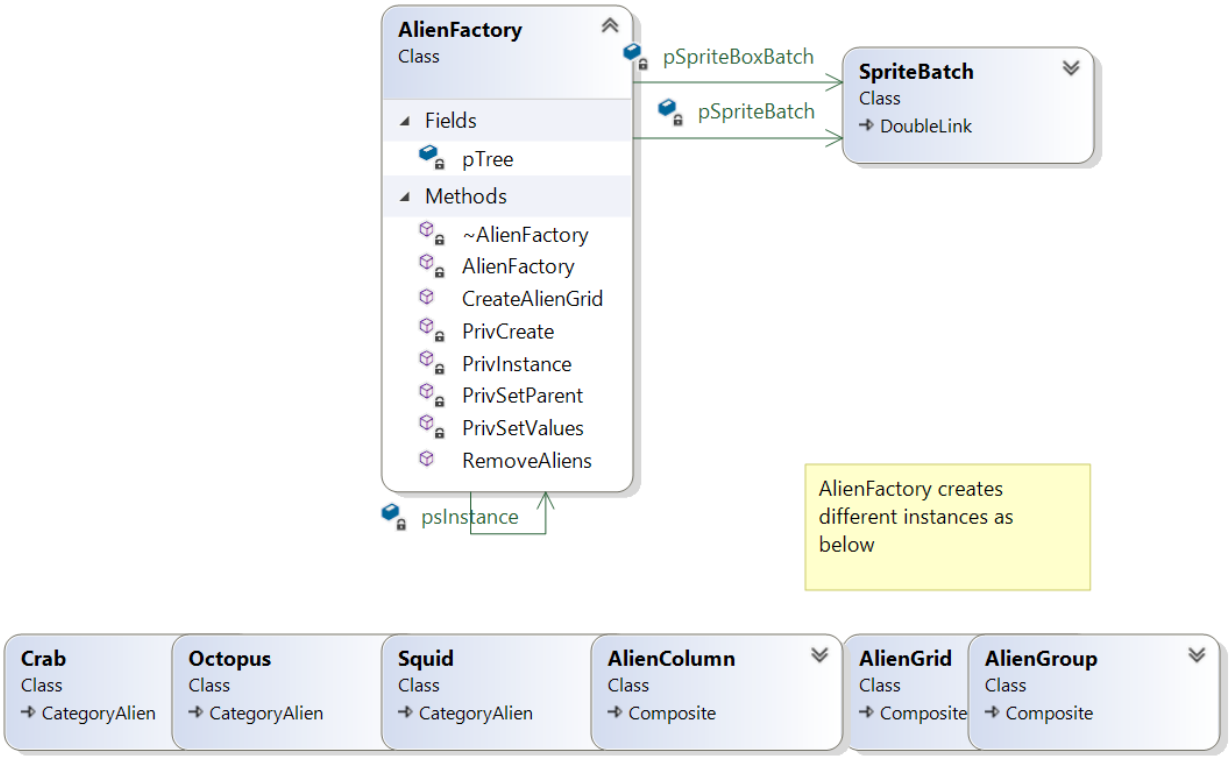
Solution:

Since we know what type of object we need to create, I've created factories that generate different instances based on the type of arguments. I set up two separate factories for aliens and shields here.

Pattern Description:

The Factory pattern is a creational pattern that aids in the creation of instances by hiding the creational logic from users.

It provides a common interface for creating different instances of related classes without exposing the concrete classes.



The above UML diagram is an example for Factory pattern to create composite of Aliens Grid

Key OO Mechanics:

The main mechanism is to implement the abstract factory that the Concrete Factory extends. This factory can produce a variety of objects from similar related products. As a result, by utilizing this factory, one can easily create instances without having to worry about the internal representation of the products.

- Base Factory: It acts as an interface that provide the abstract methods to create different products.
- Concrete Factory: It is a concrete class that extends the base factory and creates the different products based on input.
- Client: By using the Concrete Factory, one can easily create the instances by calling the interface method with different parameters.

Uses:

I used the factory pattern in two places in Space Invaders, AlienFactory and ShieldFactory. Because we're using the composite pattern to keep the 55 aliens together, it's difficult for the user to create different types by using separate classes. As a result, AlienFactory assists in the creation of various instances such as Crab, Squid, Octopus, and so on, based on the arguments.

This is accomplished by including switch statements with multiple cases, adding the instances to the GameObjects, and linking to the Composite via a tree.

ShieldFactory is another factory that is used. It works similarly to AlienFactory, but it creates shields with Gird, Column, and Bricks composite trees.

I implemented the Create method in both factories, which creates the AlienGroup and ShieldGroup. As a result, whenever I need to, I can simply refer to the method for creating the same as simple as it is.

2.5 Proxy Design Pattern

Problem:

In Space Invaders, you must create 55 Aliens from 11 Squids, 22 Crabs, and 22 Octopuses. Similarly, the same issue with shields that required the same sprite to be used multiple times was resolved.

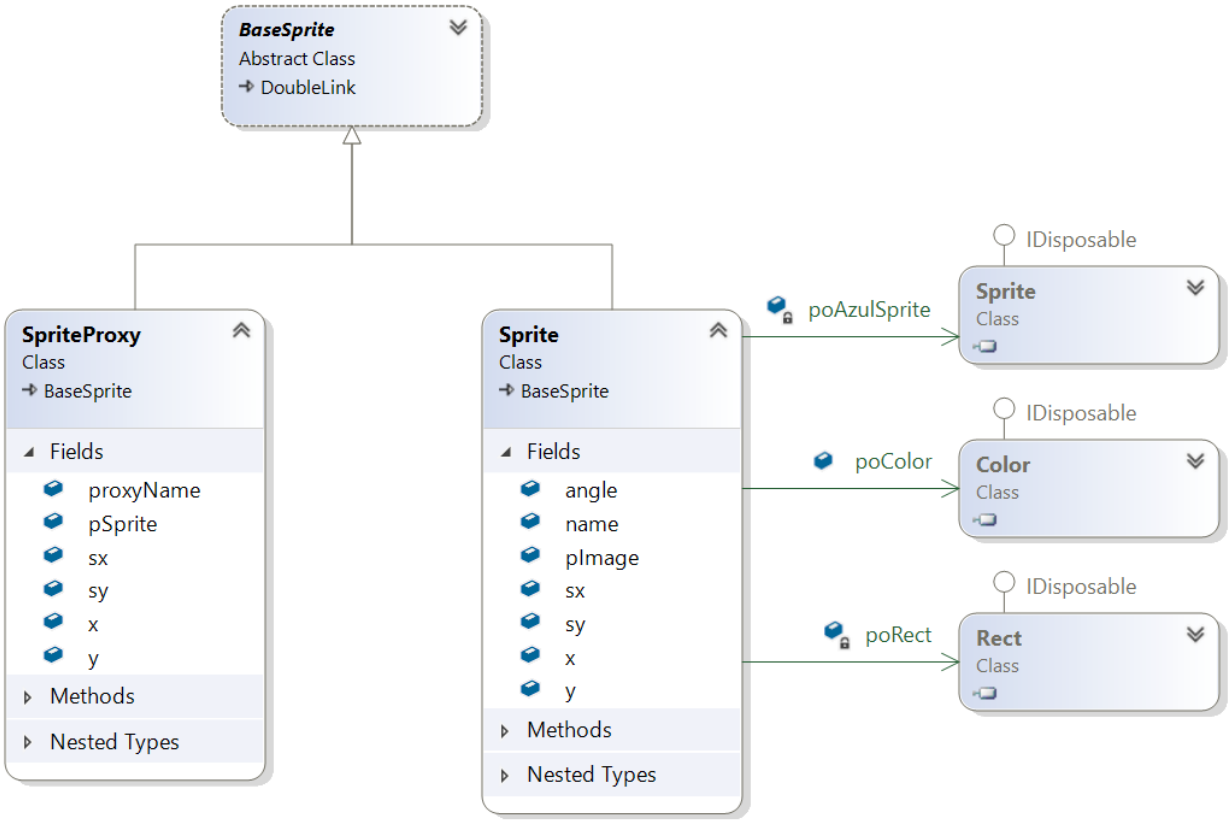
Solution:

To avoid the above problem, instead of creating multiple sprites that hold large amounts of data, we can reduce the unnecessary overhead by using a proxy pattern that points to the real sprite but holds very small amounts of information.

Pattern Description:

The Proxy pattern serves as a placeholder for the real object, acting as if it were real but containing less data.

The proxy has a reference to the real object and can access its information when necessary.



The above UML diagram is an example for Proxy pattern that points to the Original Sprite

Key OO Mechanics:

The proxy pattern is simple to implement because it is similar to the real subject but holds less data than is required. Assume there is an abstract class with some abstract methods and a subclass, Subject, that extends from that abstract class and holds more information.

In this case, if the client requires the creation of similar objects with slight differences in information, we need a proxy subject class that points to the original subject class with only required information because it has access to the original subject.

- Base Subject: It serves as an interface for the abstract methods.
- Proxy Subject: It is a concrete class that extends the base subject and has a reference to the real subject.
- Real Subject: It is a concrete class that contains the original data.

Uses:

I used the Proxy pattern in Space Invaders to create similar types of sprite objects using the SpriteProxy class. Because we needed to make aliens with similar sprites but different positions. Instead of creating

multiple sprite objects, we create a sprite proxy that links to the original sprite. As a result, the overhead of the original sprite class is reduced.

In this case, the original sprite points to the Azul Sprite, Color, and Rect, all of which require a lot of memory to hold the data. SpriteProxy, on the other hand, only has position information x, y, angle sx and sy, and a pointer to the real Sprite. By doing so, I am removing unnecessary data and overwriting it if necessary to draw the sprites on the screen.

In addition, I created a SpriteBoxProxy that points to the real SpriteBox and does the same thing as above but uses less memory.

2.6 Visitor Design Pattern

Problem:

I created the collision system between the game objects in Space Invaders. However, the main issue is that when two different objects collide, we must decide what to do and which object collided with whom.

Solution:

To solve this tricky issue, I used the Visitor pattern on GameObjects to perform a double dispatch on both objects that collided with each other.

Pattern Description:

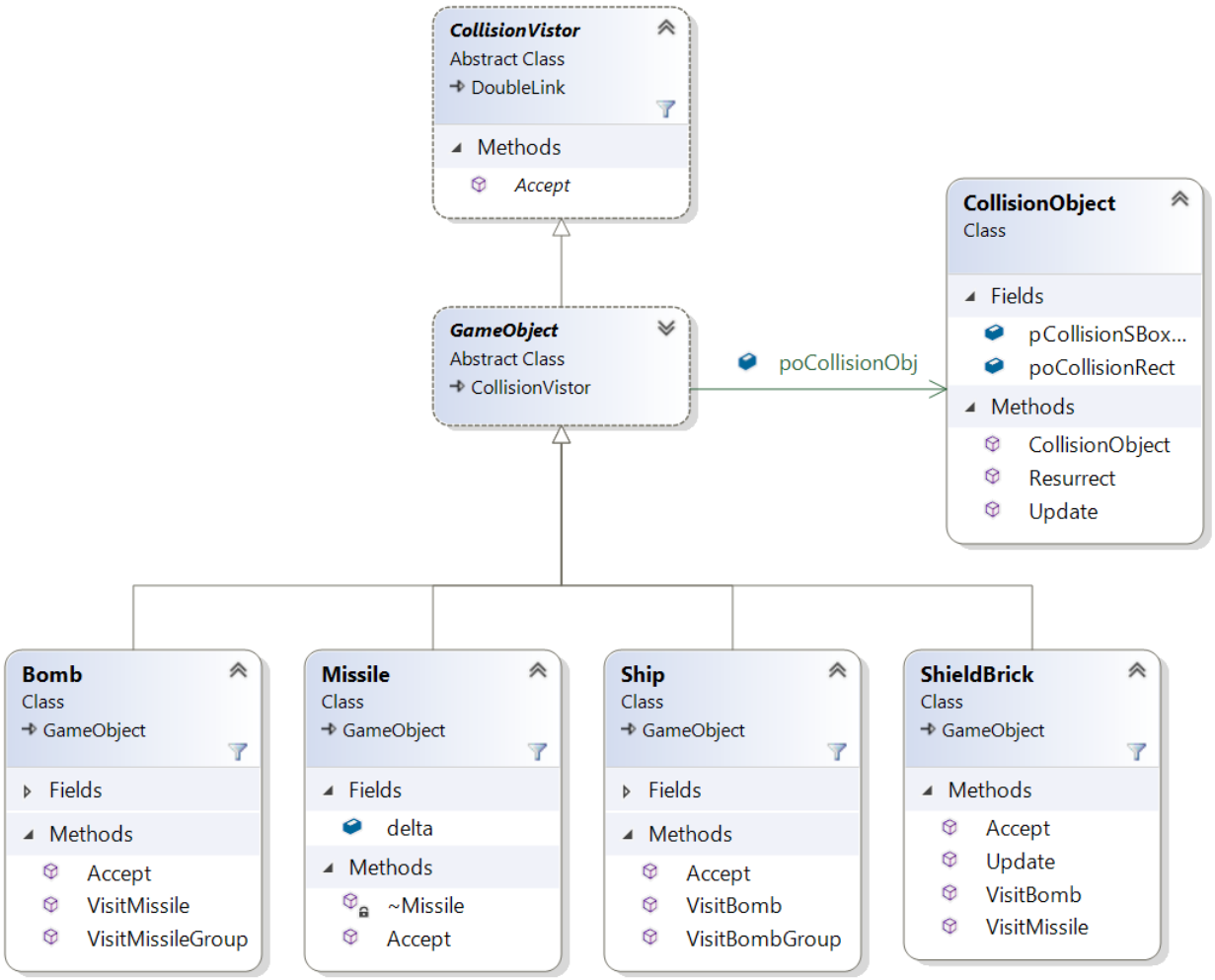
The Visitor pattern aids in the definition of the operation on the elements to which it is linked. It makes it simple to create a new operation on the elements. The main goal is to decide on and apply equality to both objects, as well as to perform double dispatch.

Key OO Mechanics:

The basic Visitor implementation consists of a Visitor abstract class and concrete visitors with Visit methods for each element. As a result, whenever two elements visit each other, the respective Accept methods in the Element classes and the Visit Methods in the Concrete Visitors are invoked.

In Space Invaders, however, we used a modified visitor pattern for our collision system. Because all of the objects are subclasses of Game Object, the Visit and Accept methods are in the same class, and the collision system will invoke the respective Accept method in the Game Object and perform double dispatch.

- Base Visitor: It serves as an interface for the abstract Visit methods.
- ConcreteVisitor: This is a concrete class that has Visit methods and indicates that the object has a visitor.
- Concrete Elements: It is a concrete class that extends the Base Element and has an Accept method that accepts the visitor object.



The above UML diagram is an example for Visitor pattern on the GameObjects

Uses:

The Visitor pattern helped in resolving the most difficult problem of determining which object collided with which other object. In our game, I implemented a Collision Visitor with abstract Accept and Visit methods. The CollisionVisitor implements the gameobject, and the respective GameObjects such as Bomb, Missile, Ship, ShieldBrick, or Aliens implement the Accept and required Visit methods.

We know that the Ship will collide with the Bumpers and the Bomb, but only the respective Visit methods and the Accept method are implemented in the ship. Because I used virtual visit methods in the CollisionVisitor, it is not necessary to implement all of the methods in the Gameobjects. As a result, we only implement the collision-related methods.

To call the notifiers, I basically linked the visit methods in the gameobject with the observer pattern.

2.7 Observer Design Pattern

Problem:

When two game objects collide, certain actions must be taken, such as Remove Alien, Play Sound, Add Score, and so on. The Observer solves the problem of notifying events to perform actions.

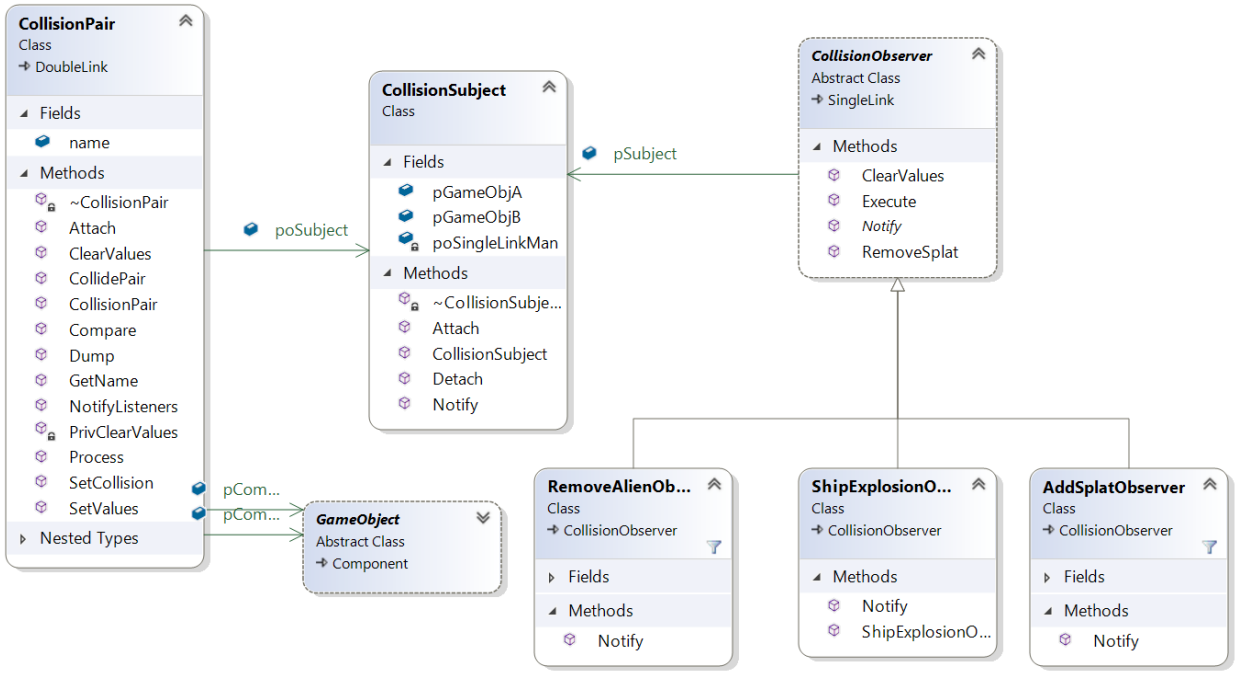
Solution:

To address this issue, we must create multiple observers and attach them to the collision pair. As a result, when the event occurs, it notifies all observers who are associated with that event.

Pattern Description:

The Observer pattern is defined as a one-to-many dependency on a current state change. As a result, whenever an event occurs, it triggers all of the events that are dependent on that event.

This makes it simple to attach as many observers as possible to the object.



The above UML diagram is an example for Observer pattern on the Collision Pair

Key OO Mechanics:

The Observer abstract class is used to create the basic implementation of an observer, and multiple observers can be extended from the base class. Observers are linked to each subject. These linked concrete observers are automatically called and updated based on the state of the Subject. For this pattern, each concrete observer class implements the Notify method, which is called automatically based on the event.

- Base Observer: It serves as an interface for the Notify method, providing abstraction.
- Concrete Observer: A concrete class that overrides the Notify method and performs some action independent of the object.
- Concrete Subject: It is a concrete class that contains the state and has a pointer to it in the observer. As a result, observers can update the subject state as needed.

Uses:

In Space Invaders, I've created many observers for each set of actions to be performed and updated whenever the game objects collide. When a collision is detected, the Visit method delegated the work to the subject, which then triggered the observers. In our case, the Collision Subject Notify() method notifies all observers who are associated with that subject.

Including all of the code in one class is normally inconvenient. As a result, by dividing into multiple observers, these can be reused and added to multiple collision pairs. For example, whenever an Alien and a Missile collide, the game requires us to remove the alien, add the splat, and remove the missile. As a result, we can simply connect the observers to that pair.

Furthermore, I can extend the observers and reuse the same observer, such as removing a missile when it hits the top of the wall. This pattern aids in the implementation of actions and updates automatically whenever a specific event occurs in our game.

2.8 Command Design Pattern

Problem:

Certain actions in Space Invaders must be triggered at specific times, such as animating the alien marching, removing the splat after 0.5 seconds, and so on.

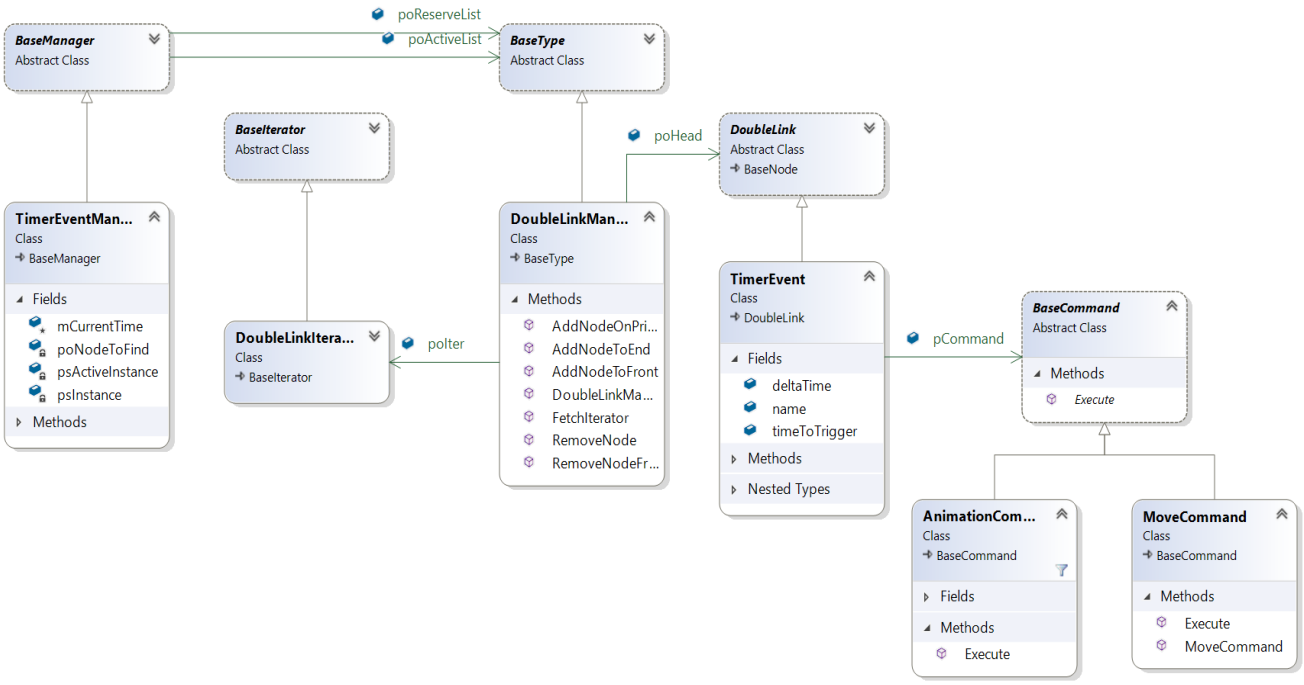
Solution:

We implemented the Command pattern on the priority queue, which executes the actions in order, to perform those actions based on timer events.

Pattern Description:

The command pattern holds the request to be executed at a specific point and then executes it. In oops concepts, these serve as object-oriented callback methods.

It is very easy to add any commands, and the main difference between the command and the observer is that the commands can be added back to the list, allowing them to execute the same command multiple times, and they can also be used to undo actions.



The above UML diagram is an example for Command pattern on the TimerEventManager

Key OO Mechanics:

The first step is to define an abstract class and several concrete commands that override the execute () method. The primary mechanism is that these commands are simply callback methods that call methods on the receiver class.

- Base Command: It serves as an interface for the method being executed, providing abstraction.
- Concrete Command: A concrete class that overrides the execute method and performs an action on the receiver.
- Receiver: It is a concrete class with the intention of carrying out the action based on the request.

Uses:

I created several commands in Space Invaders, such as the Animation Command, which animates the aliens, the Move Command, which moves the aliens synchronously, and many more. Each command is added to the TimerEventManager based on the priority of timeToTrigger.

As a result, these events are referred to sequentially based on time, which solves the problem of delayed actions. We can easily add new timer events to the TimerEventManager or remove them.

For example, in the Animation command, we swap the image on the sprite and then add the same command to the timer with delta time. As a result, these commands will run indefinitely until they are manually removed from the TimerEventManager.

Furthermore, the ability to pause events and add multiple commands at a specific time is very flexible.

2.9 Composite Design Pattern

Problem:

We have a problem moving the aliens synchronously together in Space Invaders and must treat all of them uniformly.

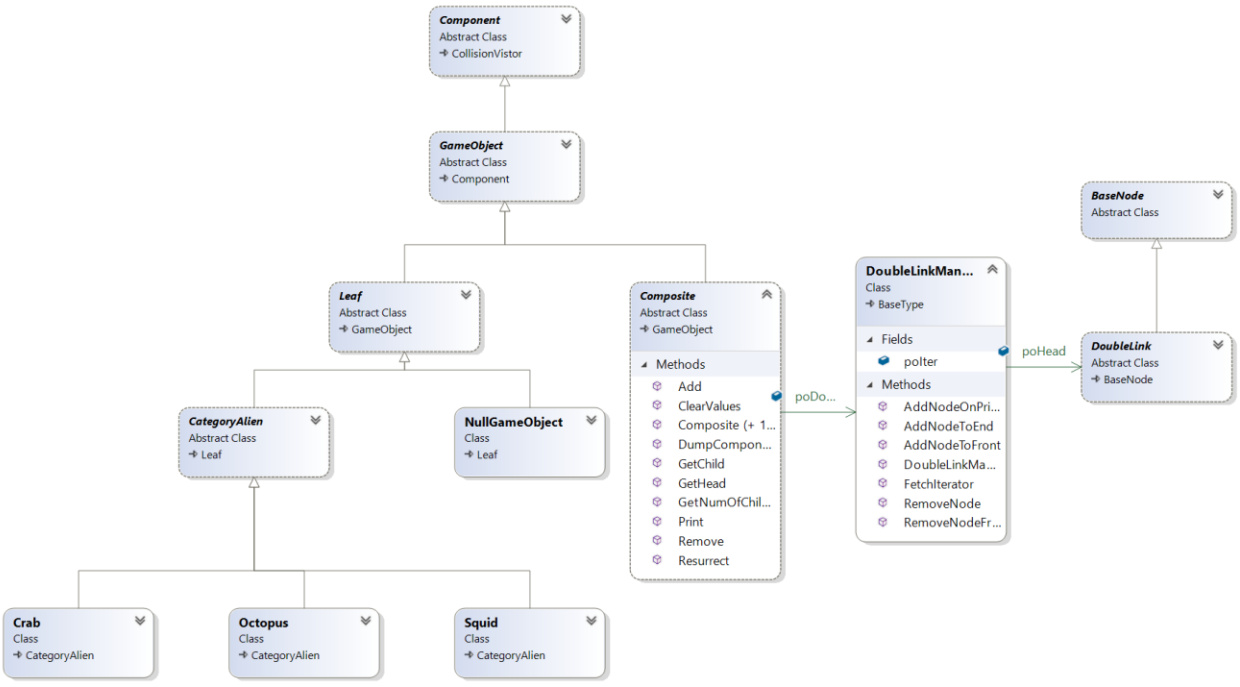
Solution:

To solve this issue, we used the composite pattern to treat all 55 aliens as a single collection and implemented a grid layout.

Pattern Description:

The Composite pattern views the collection of objects as a tree, with each object playing a role in the overall hierarchy. There could be multiple leaves and composites attached to a tree in this case. Using this pattern, the user can access and treat each object in a consistent manner.

It is very easy to add any commands, and the main difference between the command and the observer is that the commands can be added back to the list, allowing them to execute the same command multiple times, and they can also be used to undo actions.



The above UML diagram is an example for Composite pattern on the GameObjects

Key OO Mechanics:

To treat both leaves and composites similarly, the concept of composite is maintained. We make a component as well as two classes called "Leaf" and "Composite." Composite has children, whereas Leaf does not. Using the component, a user can easily access and change the elements in the composite.

The Composite contains a list of children that is linked to the component itself. We can iteratively add as many children as we want to the composite.

- Compoent: This object serves as an interface for all composition objects.
- Composite: A concrete class with multiple children and recursive behavior that connects to the component.
- Leaf: It is a concrete class with no children that performs its related behavior.

Uses:

To maintain the structure and treat all game objects uniformly, all game objects are treated as composites in the game. We have 55 aliens divided into 5 rows and 11 columns, just like in the game. We created an Alien Group that serves as the parent of all grid structures and cannot be deleted. Following that is the Alien Grid, which contains the Alien Columns, each of which contains a different type of alien. Aliens are the leaves in this case, and the Group, Grid, and Columns are composites. Using the Iterators, we can also easily access the data and iterate over the composite.

Furthermore, I used this pattern for all GameObjects, including Missiles, Ships, Bombs, and Shields. This makes it easier for the game to organize all of the objects uniformly and easily accessible.

2.10 Object Pooling Design Pattern

Problem:

In any game, performance is everything. And making the object from new is costly. Many objects had to be created and handled in Space Invaders.

Solution:

To improve game performance by reusing objects created by managers through the concept of object pooling.

Pattern Description:

Object pooling is used to reuse objects and share them among game objects as needed, rather than creating unnecessary objects that are very expensive in terms of performance.

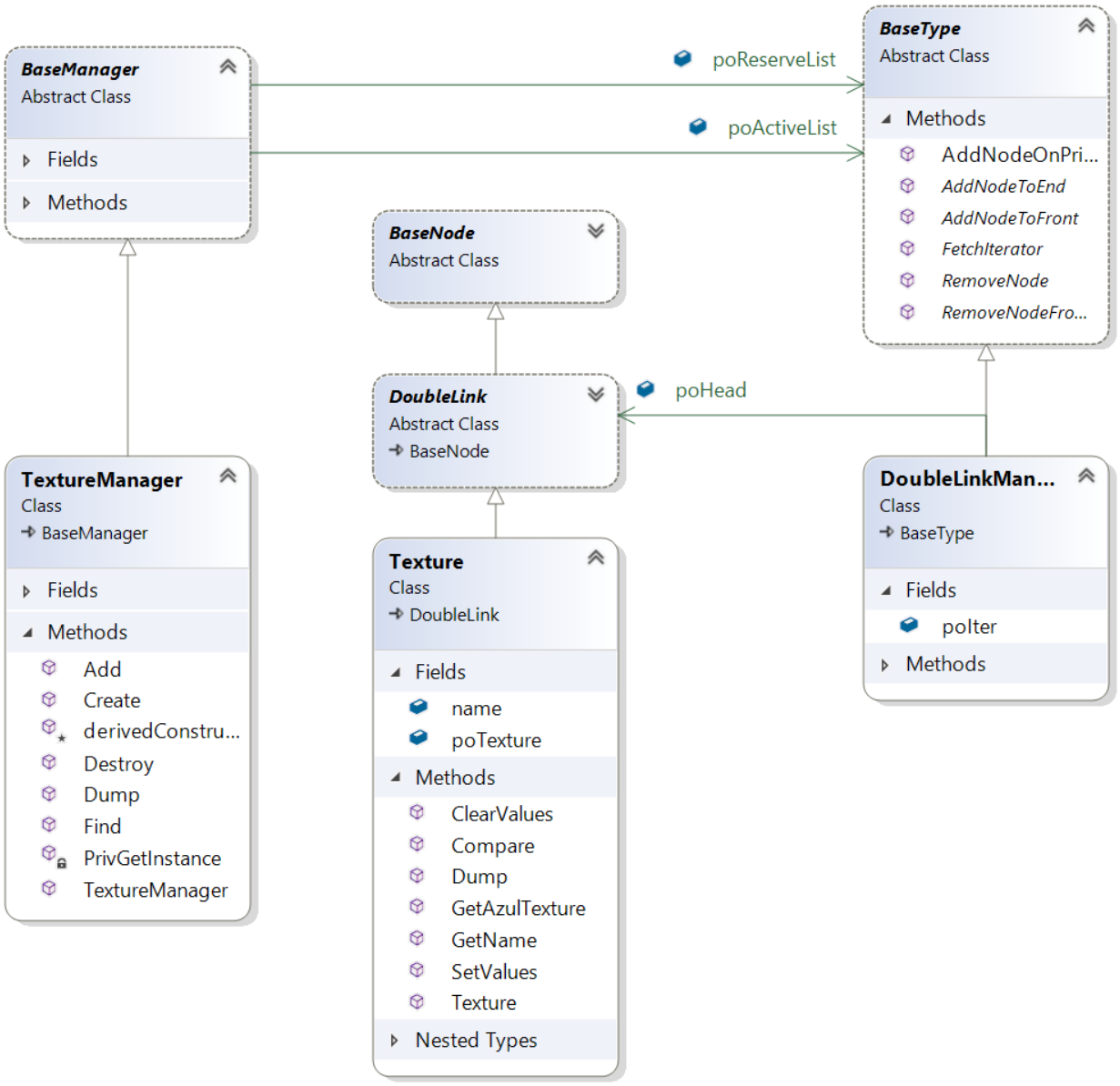
Using this pattern, it is simple to add, remove, and find objects in an efficient and effective manner.

Key OO Mechanics:

Because it is very costly to create new objects and initialize them. Object pooling will create objects as needed by keeping the removed objects in a list. As a result, when a new object is required, it will check the reserve list and assign it for reuse. We can reuse the objects more efficiently this way.

When the client is required to create a new object n object-oriented mechanics, the object pooling will check in its reusable pool and provide the object instead of creating an unnecessary new object.

Furthermore, unused objects are marked for removal and are collected by the garbage collector at an inconvenient time. As a result, it is always preferable not to delete the objects and instead reuse them until they are no longer needed.



The above UML diagram is an example for Object Pooling pattern on the Texture Manager

Uses:

Object pooling is used in all nodes in the game by using Managers and the best architectural technique. I created a TextureManager in the UML shown to manage the Textures. As a result, instead of creating texture objects, we can reuse them.

The Manager has two active list pointers, poActiveList and poReserveList. When a new texture is required, the manager checks the reserve list and, if nodes are present, removes them from the reserve list and adds them to the active list with the given values.

If we want to remove a node, the manager removes it from the active list and returns it to the reserve list. The manager is effectively reusing the objects in this manner. If the reserve list is empty, it creates the necessary nodes and adds them to it.

I mostly used DoubleLink lists in the managers to add and remove nodes more efficiently than single linked lists.

2.11 Singleton Design Pattern

Problem:

We used object pooling to create a manager to organize the data. Managers are required to add, delete, or locate objects everywhere. It's difficult to get a manager instance every time you access it.

Solution:

We used Singleton managers to give it global access to the entire game. Because managers only have one instance, making it static and allowing the global access point to access it from anywhere simplifies things.

Pattern Description:

If a class has only one instance, the Singleton pattern is used to provide a global entry point to everything for easy access.

There are two kinds of initialization: creating the instance first and creating the instance only when it is needed.

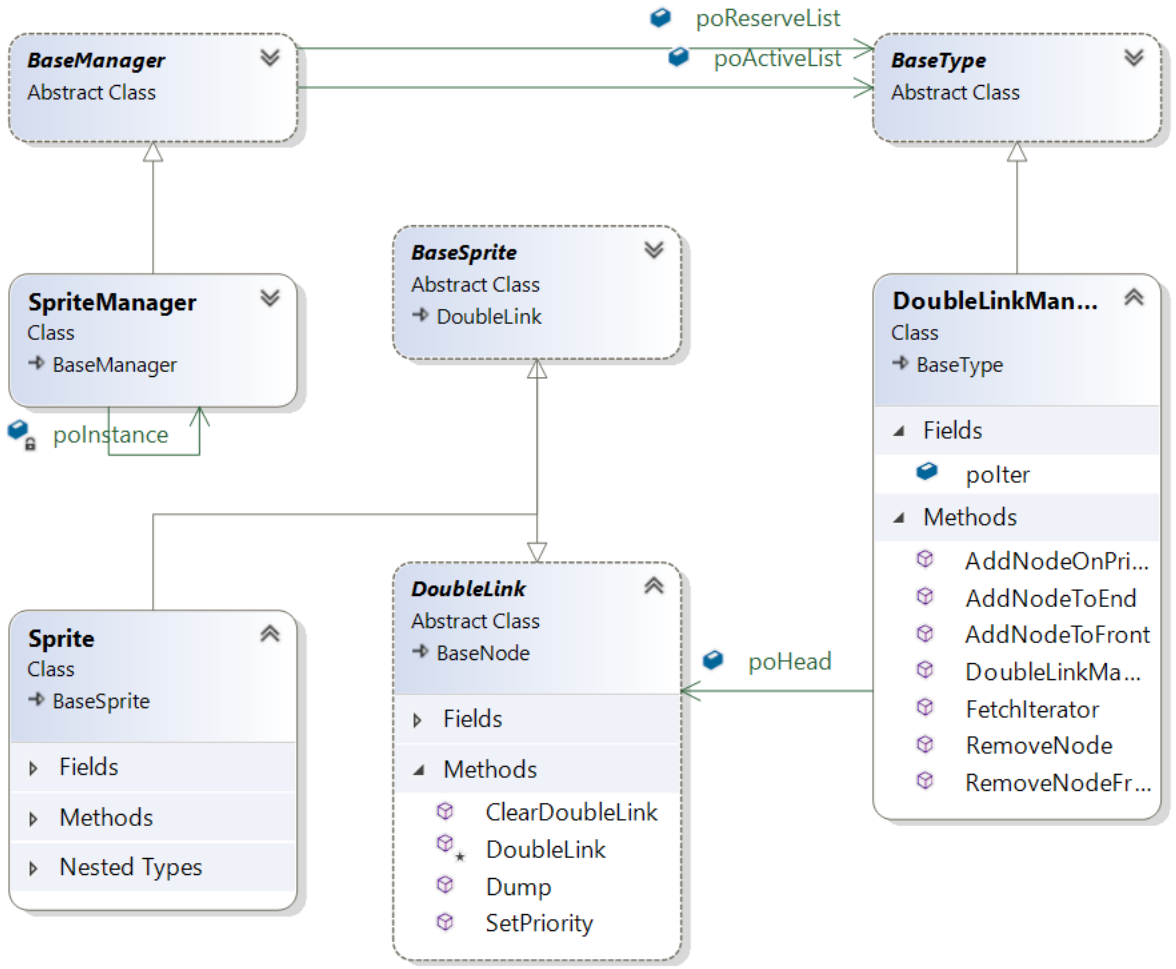
Key OO Mechanics:

Singleton is a class that has only one instance and needs to be accessed from everywhere. It provides global access so that methods can be easily accessed by class name rather than instance.

The instance is kept private within the class and is created when the class is first created.

Uses:

Managers in the game are distinct and reusable classes that can be found throughout the game. I made the Managers Singleton by static methods to make it easier to access from anywhere. As a result, whenever I need to add, remove, or find nodes, I can quickly access the methods by using the classname. The methods within the class retrieve the private instance and perform the appropriate operations.



The above UML diagram is an example for Singleton pattern on the Sprite Manager

2.12 Flyweight Design Pattern

Problem:

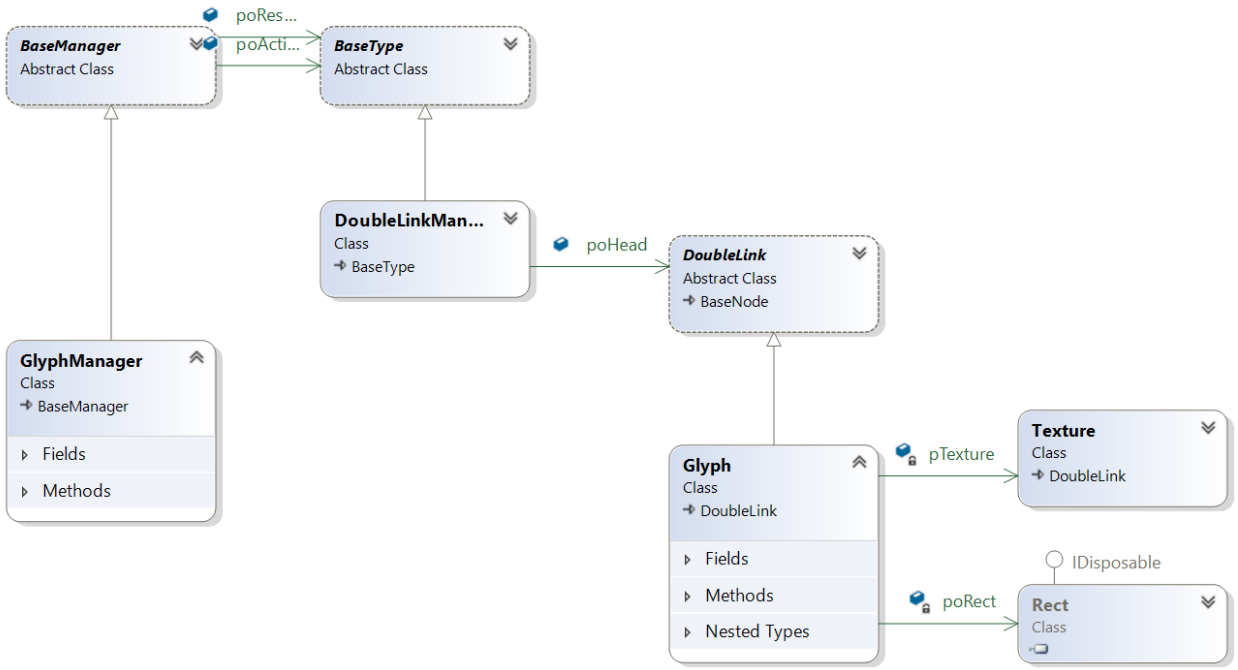
We used multiple characters in the game to display the scores, points, and lives in different scenes. Furthermore, for each level, we must recreate the shields and aliens on the screen, which reduces efficiency.

Solution:

We reuse the objects by sharing them when needed by using the flyweight. When it came to fonts, I used the Flyweight pattern to load all of the characters at once and reuse them as needed. This is a reworked flyweight. Furthermore, the Ghost Manager serves as a fly weight in order to recreate the shields and aliens.

Pattern Description:

The flyweight aids in the reuse and creation of objects as needed. It contributes to increased efficiency by reusing objects. The difference between object pooling and fly weight is that the reusable objects in fly weight are immutable.



The above UML diagram is an example for Flyweight pattern on the Glyph

Key OO Mechanics:

The main moto of Flyweight is to create objects only when they are needed and to reuse similar objects that already exist. This increases the game's efficiency.

Uses:

It is used throughout the game, such as in Glyph, where we load all of the characters into the class and reuse them as needed in the scenes.

In addition, it is used as a Ghost manager. We use the Ghost manager to restore the removed game objects to the Ghost manager's active list. As a result, whenever we need to create aliens or shields, we first check with the ghost manager to see if one is available, then reuse or create a new one.

We improved the performance and efficiency of the space invaders by doing so.

2.13 Adaptor Design Pattern

Problem:

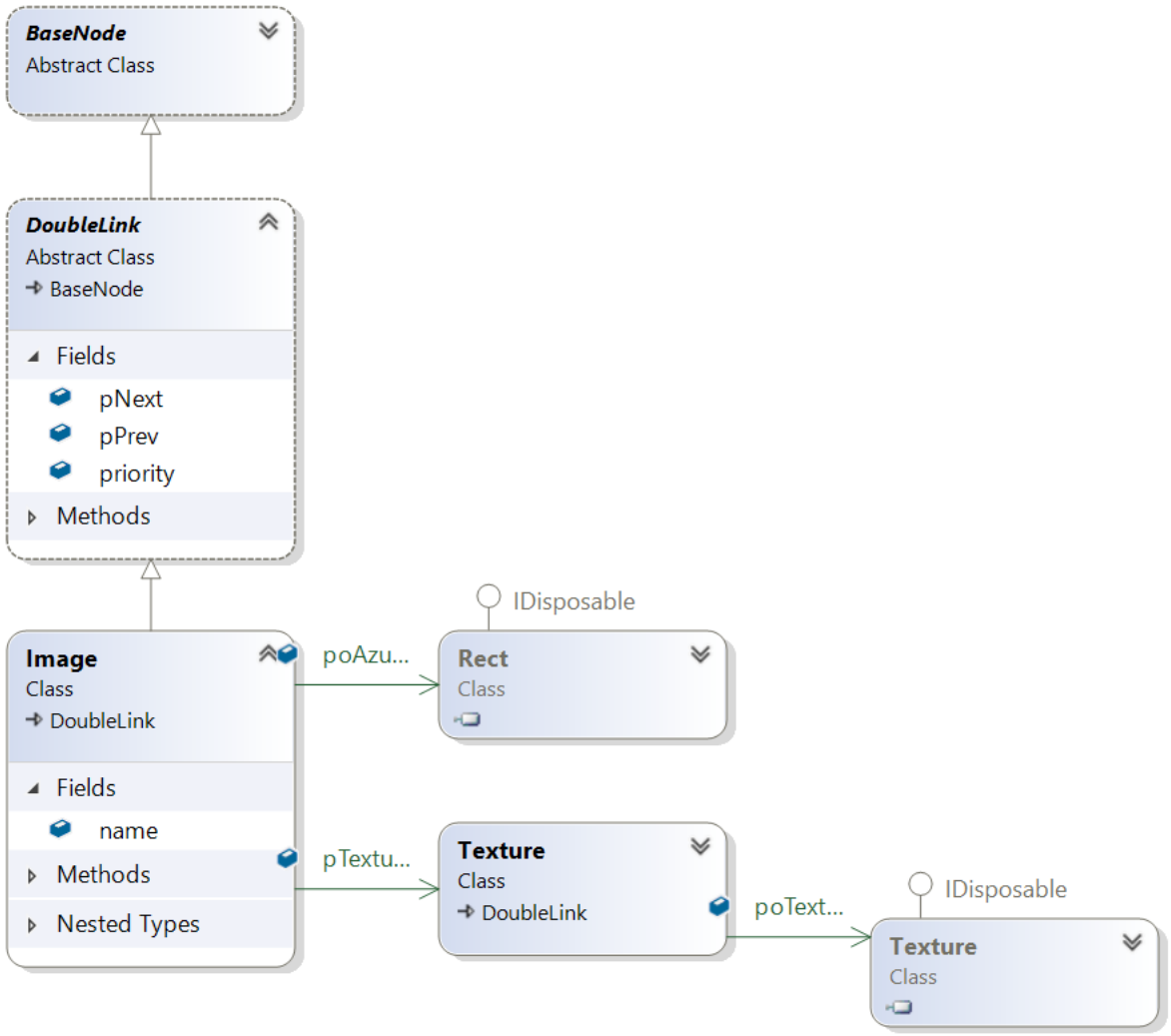
The Azul framework is used in the game to load textures, update sprites, and display game-related sprites on the screen. It is necessary for the user to remember the complicated method names in order to use them.

Solution:

In this case, we created several adaptor classes for Sprite, Image, and Texture that act as a common interface between the client and the Azul framework.

Pattern Description:

The adaptor converts one interface to the other so that clients can use it. It acts as a bridge between two unknown classes, allowing them to communicate.



The above UML diagram is an example for Adaptor pattern on the Image adaptor

Key OO Mechanics:

The adaptor serves as a go-between, establishing communication between two different interfaces or classes. The adaptor class extends the target class by implementing the methods that convert and invoke the adaptee's methods.

In this case, the client can use the adaptor class to achieve the same results as the target interface.

Uses:

I created Texture, Image, and Sprite adaptor classes in the game to communicate with the Azul framework while hiding the underlying complexity of creating the objects. The Image class contains pointers to the Azul Rect, and the Texture class contains pointers to the Azul Texture.

2.14 Null Object Design Pattern

Problem:

There may be some special cases to handle, such as when an object is removed or does nothing.

Solution:

We use the Null Object pattern to handle these special cases by replacing them to check the edge conditions by creating a Null class that does nothing. As a result, all of the objects can be created in the same way, eliminating the need to write multiple test conditions.

Pattern Description:

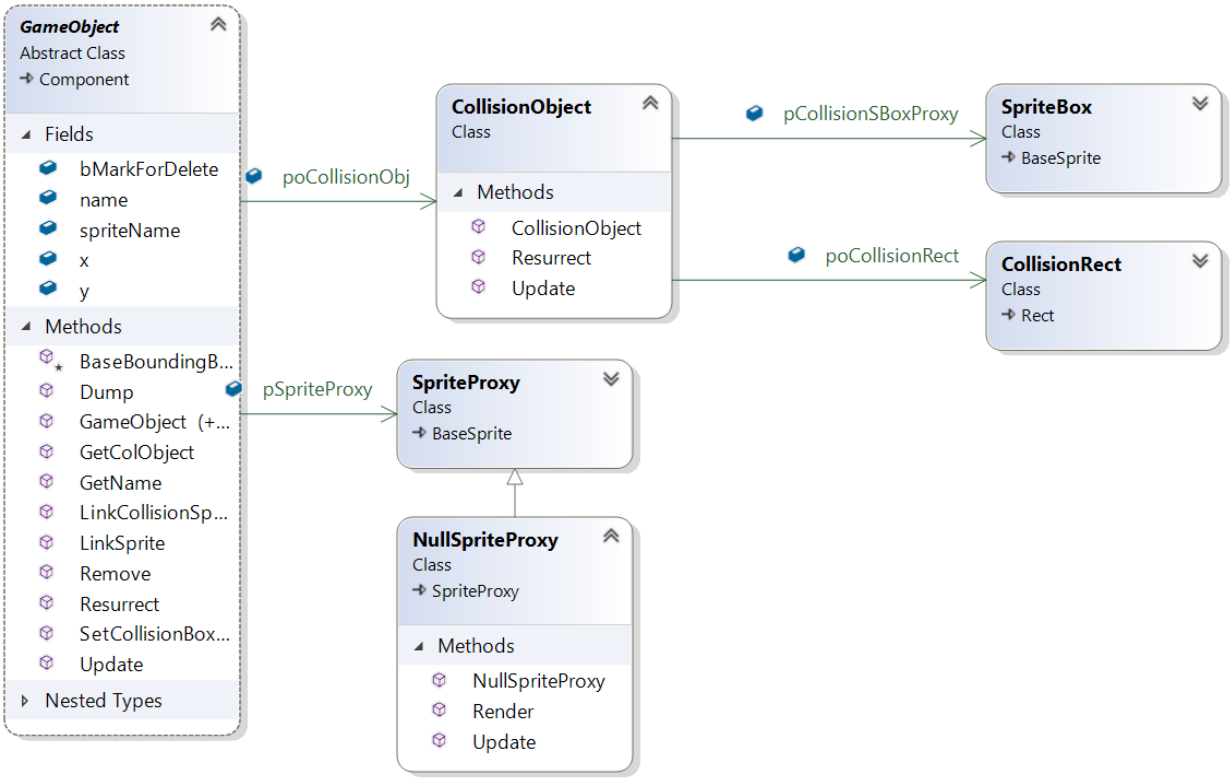
The Null Object pattern defines the Null class, which is used to represent null objects. As a result, there are no special conditions to test in the system.

Key OO Mechanics:

For the Base class, the Null Object class is created, along with a Real Object that does nothing. As a result, Client can use the object without fear of null reference pointer exceptions. To accomplish this, Null Object must extend the base class and implement all methods that return nothing.

Uses:

I created a Null Sprite Proxy in Space Invaders that overrides the Render and Update methods, but the actual implementation does nothing. As a result, I reduce the number of edge conditions and unnecessary conditions.



The above UML diagram is an example for Null Object pattern on the Sprite Proxy

2.15 Template Design Pattern

Problem:

We needed to create nodes for each manager or different functionality that needed to be implemented in the subclasses for the Managers.

Solution:

We used the Template pattern in the BaseManager by providing the skeleton that needs to be implemented, such as the creation of nodes by subclasses that are Managers from the BaseManager.

Pattern Description:

The Template pattern creates an abstract skeleton for the base class, which delegated work to its subclasses.

```
// Overriding methods
2 references
protected override BaseNode derivedConstructNode()
{
    // LTN - SpriteManager
    Sprite pSprite = new Sprite();
    Debug.Assert(pSprite != null);

    // Return a newly created Sprite
    return pSprite;
}
```

The above code snippet is an example from the SpriteManager that implements the method that is an abstract method in the BaseManager class

Key OO Mechanics:

The Template pattern requires subclasses to implement the base classes' abstract methods. Here, for the base class that ensures the client's method contract, As a result, whenever a subclass extends the base class, it must also extend the base class's methods.

Uses:

In Space Invaders, I've created many abstract classes that must be implemented in the subclasses. In the BaseManager, for example, I've created a template for the derivedConstructNode method, which must be overridden by all of its sub classes.

Similarly, the ForwardCompositeIterator and ReverseCompositeIterator classes must implement the First(), Current(), Next(), and IsDone() methods in the BaseCompositeIterator.

3. Post-Mortem:

I developed this Space Invaders game using design patterns to create a real-time data-driven application. We developed this game from Week 1 to Week 10 using best software engineering practices such as Iterative Development with weekly sprints, Testing, Continuous Integration, and a final design document.

The following technologies were used to create this game:

- Simple C# with Visual Studio 2019 Enterprise
- Perforce – Helix Visual Client for version control
- Azul GUI Framework

I did my best to bring this game to life by implementing the necessary specifications and adhering as closely to the real game as possible. I haven't tested or implemented all of the specifications due to time constraints.

This game requires some enhancements and features as follow:

- Graphical noise base Dissolve effects on the Shields
- Credits on the Screens
- Implement the additional edge condition for collision system between aliens and the wall bottom.
- Improve the random bomb drop per alien column to reduce the edge conditions
- Additional Stress test required on the game to find the more edge conditions

To summarize, the Space Invaders with design patterns for SE 456 is one of the most challenging and large projects completed to date. I am extremely grateful for all the assistance provided by Professor Keenan to accomplish this project.