DESIGN DOCUMENT

SE 456 – Architecture of Real-time Systems

SPACE INAVDERS

Demo

By

Shoeb Mohammed Khan

Contents

Introduction:

The document discusses the development process of Space Invaders game using efficient techniques and various design patterns. It has listed the features and design patterns that were used and implemented during the game's development.

The game was developed without using any game engine, it only used the Azul framework, which is a lightweight software development framework designed to render textures. The use of the Design patterns allowed for efficient and streamlined development of the game, as it provided a set of systems and functionalities which will required in the absence of a game engine.

The Project’s main objective was to create a 5 x 11 alien grid that moved from left to right and top to bottom over time. The player controlled a ship at the bottom of the screen and needed to move horizontally to shoot the aliens. The game had three screens - Home, Play, and Game Over - and players could start the game by pressing a key. A font system was used to display scores and the number of lives remaining.

To make the software more robust, we utilized various design patterns such as the Factory pattern, Singleton pattern, and Observer pattern. The Factory pattern was used to create different types of aliens, while the Singleton pattern ensured that only one instance of the game was running at a time. The Observer pattern was used to notify the scoreboard whenever an alien was killed, or a player lost a life. The scoreboard added points for each type of alien's death, and the game featured aliens that randomly dropped bombs.

Overall, the development process of the Space Invaders game showcased how efficient design patterns and leveraging can make game development more streamlined and effective. The game's implementation of design patterns and best software design practices makes it a real-time software architecture masterpiece.

Proxy Design Pattern

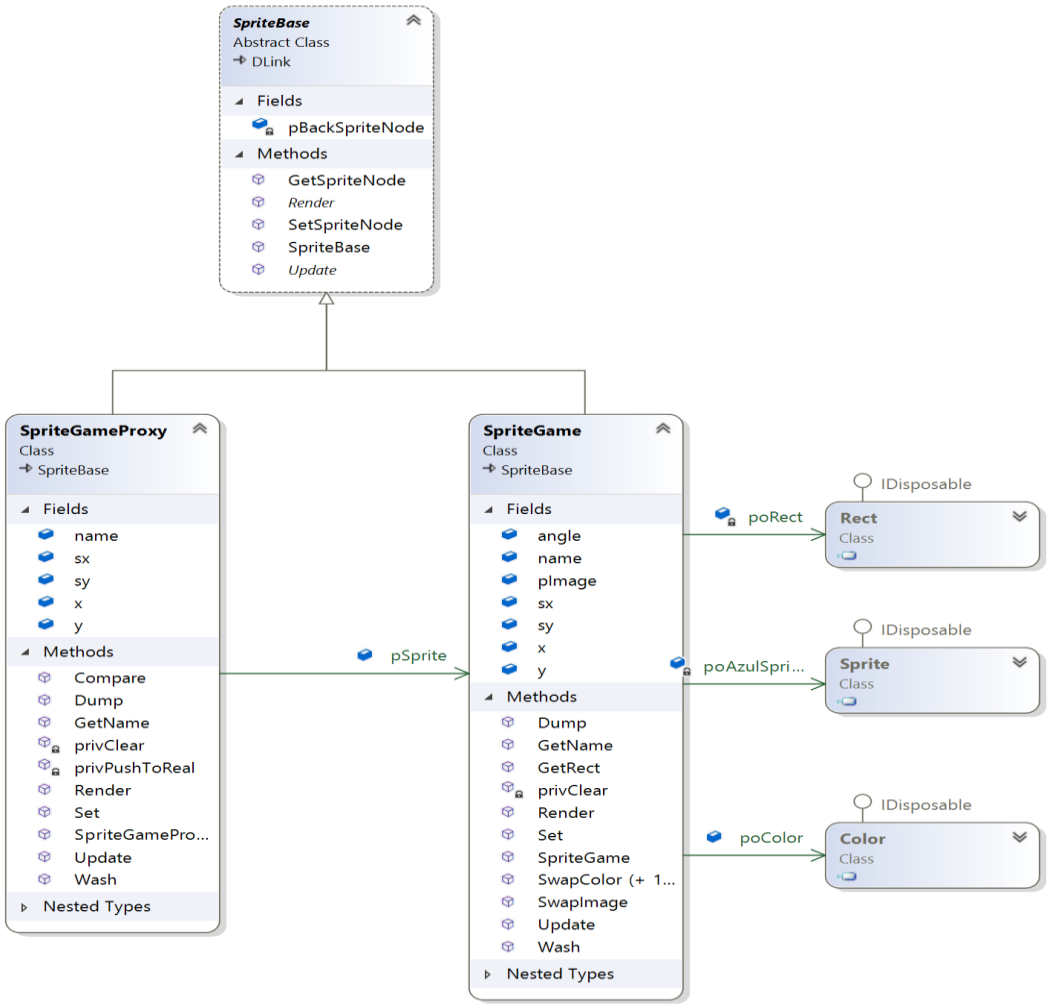
Problem:

We need to create 55 Aliens in Space Invaders by a combination of 11 Squids, 22 Crabs, and 22 Octopuses. Additionally, a similar issue with the shields was reusing the same sprite multiple times.

Solution:

My approach is to avoid creating multiple sprites with excessive data. Instead, we can reduce unnecessary overhead by using a proxy pattern. The proxy pattern acts as a reference to the real sprite but holds only a small amount of information.

Pattern Description:



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

The Proxy pattern is a design pattern that provides a substitute or placeholder for another object. It allows us to provide a level of indirection between the client code and the real object. The Proxy can control access to the real object, limit access to its resources, and provide a simplified interface to the client code. It's useful for improving performance, protecting sensitive information, and simplifying the client interface.

Key Object-Oriented mechanics:

Implementing the proxy pattern is a straightforward process since it closely resembles the real subject, with the only difference being that it holds less data. Consider an abstract class with abstract methods, and a subclass called Subject that extends that abstract class and contains additional information. If the client requires the creation of similar objects with slight variations in information, a proxy subject class can be used to point to the original subject class with only the necessary information. This is possible because the proxy has access to the original subject.

The Proxy pattern has three main components: the Base Subject, which is an interface for the abstract methods; the Proxy Subject, a concrete class that extends the Base Subject and holds a reference to the Real Subject; and the Real Subject, another concrete class that holds actual data.

Uses:

The Proxy pattern was used in Space Invaders to create similar sprite objects with different positions using the SpriteProxy class. Rather than creating multiple sprite objects, a sprite proxy was used to link to the original sprite, reducing the overhead of the original sprite class. The original sprite pointed to Azul Sprite, Color, and Rect, all of which required significant amounts of memory.

In contrast, SpriteProxy only contained position information and a pointer to the real sprite, resulting in less memory usage. Additionally, a SpriteBoxProxy was created to do the same thing but with less memory usage.

Iterator Design Pattern

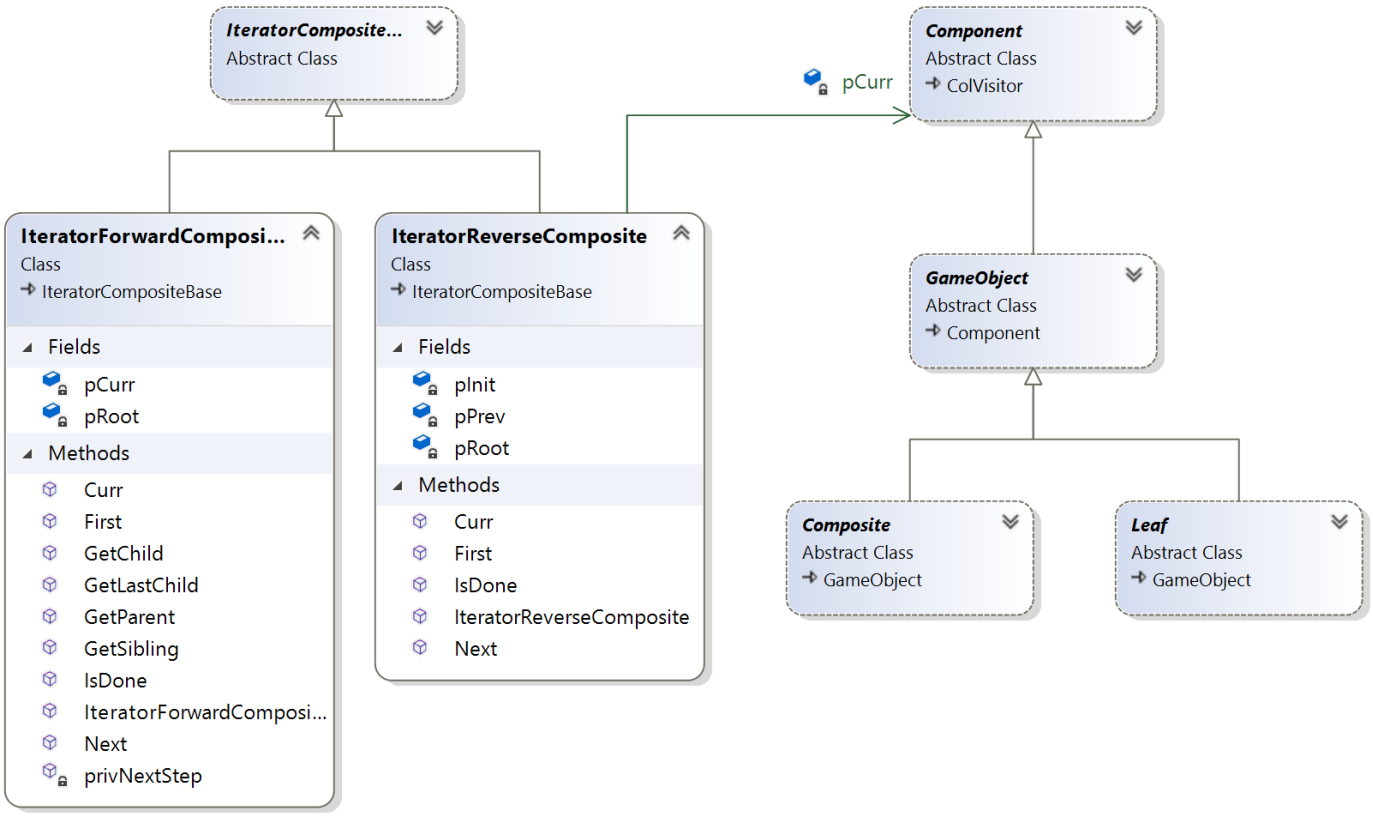
Problem:

In the development of Space Invaders, we utilized different data structures such as Linked Lists and Trees to store nodes like sprites and game objects. However, remembering the methods for navigating to each node within the data structure was challenging.

Solution:

To address the issue of complex data structures and the difficulty of navigating through their nodes, I implemented an iterator pattern in Space Invaders. This pattern enabled sequential navigation through the nodes of various data structures while concealing their underlying structure. Examples of its use included iterating over the alien tree, using a sprite batch manager to draw sprite batches sequentially, and other similar tasks.

Pattern Description:



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

The iterator pattern is an effective technique for sequentially iterating through aggregated objects without requiring knowledge of the object's structure. Its primary purpose is to simplify the complexity of aggregate objects and minimize decoupling on that object.

Key Object-Oriented mechanics:

The initial stage of implementing the iterator pattern involves creating an abstract iterator, which consists of abstract methods such as First(), Current(), Next(), and IsDone(), to access the elements within the data structure. The next step is to extend the abstract class and implement the abstract methods to iterate through the aggregate object.

There are three main components of the iterator pattern, including the Base Iterator, which functions as an interface for clients to sequentially access the elements within the aggregate object. The Concrete Iterator is a concrete class that extends the Base Iterator and provides the logic for the contract methods. Lastly, the Aggregate is a class that holds various data structures such as Linked Lists and Trees.

Uses:

The iterator pattern was extensively utilized in Space Invaders, particularly in iterating over the composite trees built for Aliens. To achieve this, two types of iterators were created, namely Forward Iterators and Reverse Iterators, which are subclasses of the BaseCompositeIterator class. In the game, the Forward Iterator is used to display and draw Aliens from the Composite by iterating through the entire structure from group, grid, columns, and game objects (Aliens). On the other hand, the Reverse Iterator is used to remove elements from the bottom of the tree. Since the composite is maintained as a tree, it is suggested to delete elements from the bottom of the tree.

In addition, DLinkIterator was created to iterate over a double linked list, and SLinkIterator was used to iterate over a single linked list. The Double Linked List is used in managers, and accessing the elements in managers is made easier by retrieving the iterator and accessing them effortlessly.

State Design Pattern

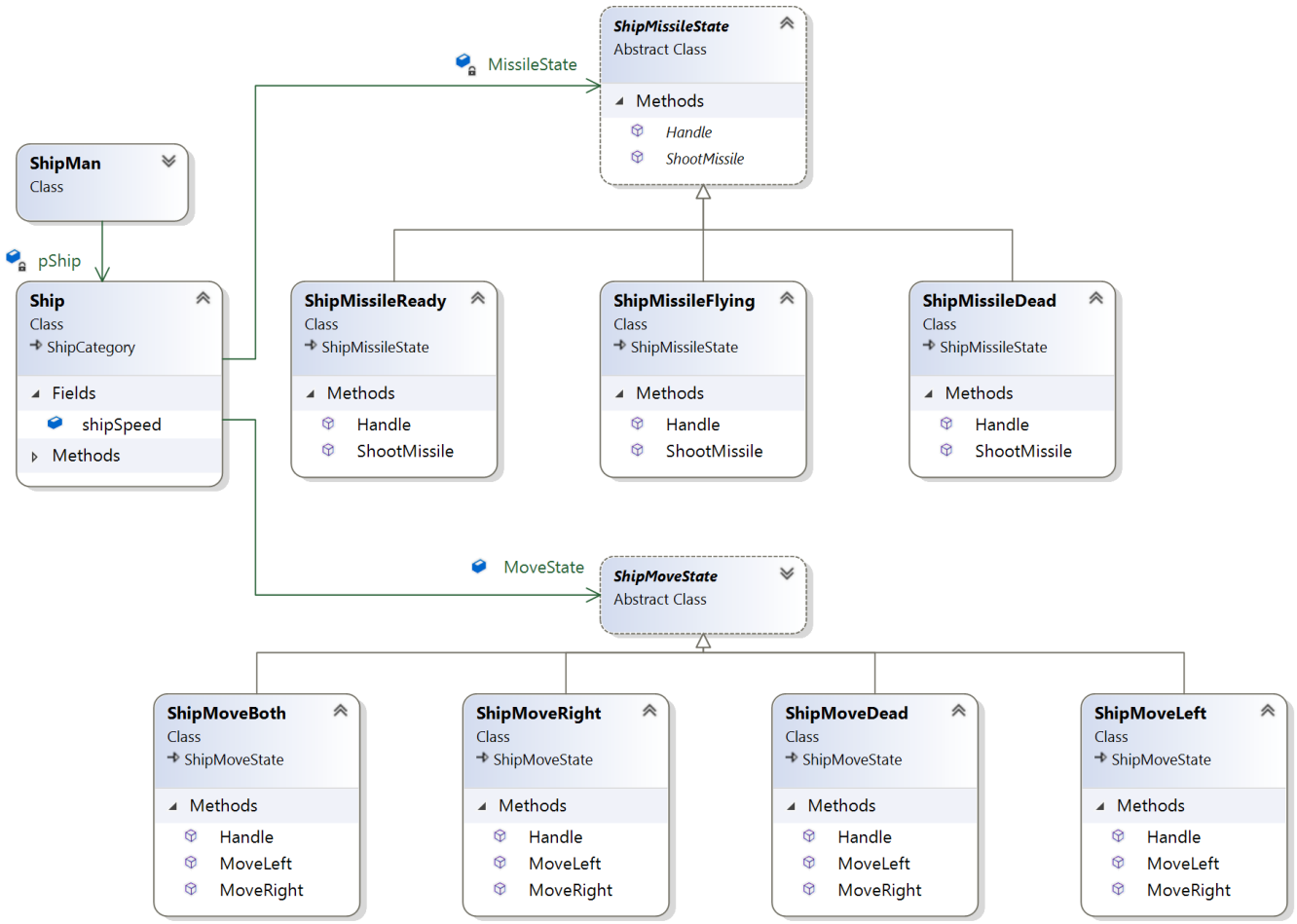
Problem:

The Space Invaders game presents a challenge in managing the state of the missile. Specifically, the game allows the ship to launch only one missile at a time, requiring careful management of the ship's inputs.

Solution:

To manage the missile state in the Space Invaders game, I employed the state pattern within the ship class. By keeping track of the missile's current state and modifying the ship's state accordingly, I was able to effectively manage missile firing using the appropriate methods.

Pattern Description:



The above UML diagram is an example for State pattern to change Ship State.

The State design pattern is used to manage an object's behavior based on its different states, without changing the class itself. It helps avoid having several conditional statements that depend on the object's state, by splitting them into separate classes. This pattern is useful for reducing the number of objects created for different behaviors and is similar to the Strategy pattern. However, the key difference between the two is that in the State pattern, the context has a replaceable connection and can change the state behavior by modifying the context's current state.

Key Object-Oriented mechanics:

State pattern is a useful tool for managing the behavior of an object based on its different states. To implement this pattern, you should start by creating an abstract class for the state and subclasses for the different behaviors by inheriting from the abstract class. In the context of the state class, you can manage the current state and trigger the respective state methods. You can easily change the current state of the context by replacing the link with the new link, which changes the object reference.

The necessary classes for this pattern are the Context class, which holds the current state and acts as an interface for the clients to perform actions and hide the internal behavior changes; the State class, which is an abstract class or interface that defines the methods that need to be overridden by base classes; and the Concrete States, which are the subclasses for the abstract class State and implement their own behavior logic.

Uses:

In order to limit the number of missiles that the ship can shoot in the Space Invaders game, I implemented the state pattern. I created an abstract class called ShipShootState, which contained abstract methods for different behaviors, and then created two subclasses called ShipReady and ShipMissileFlying, each with their own specific behaviors that override the abstract methods. By keeping track of the current state of the missile in the Ship class, I was able to restrict the ship to only being able to shoot one missile at a time until the previous one was destroyed.

I also used the state pattern to manage the movement of the ship. By creating a third, I was able to allow the ship to move in both directions. When the ship reaches the left or right wall, the state changes and only one direction of movement is allowed. This approach allowed me to avoid using conditional statements and effectively manage both missile and ship movement in the game.

Strategy Design Pattern

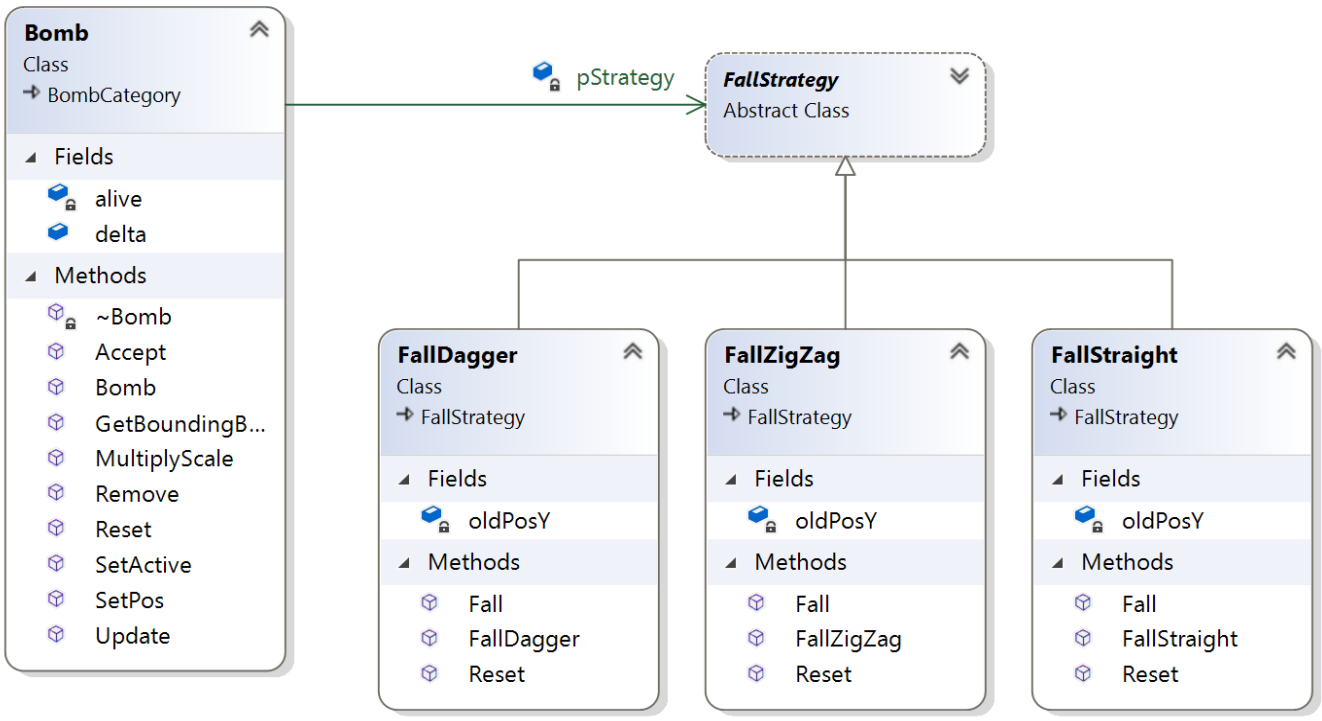
Problem:

In Space Invaders, there is a need for the aliens to drop various bombs randomly, as per the game requirements.

Solution:

By implementing the Strategy pattern, I was able to establish three unique strategies for bombs, each with its own specific behaviors. This made it easy to define and modify the behavior of each bomb without having to alter the overall structure.

Pattern Description:



The above UML diagram is an example for State pattern to change Ship State.

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.

Key Object-Oriented mechanics:

Designing and implementing a strategy that resembles the state pattern is a straightforward process. To start, create an abstract class with an abstract method. Then, create different subclasses that override the abstract method based on their specific usage. This allows clients to access the same abstract method while behaving differently depending on the context.

The following are the key components of the Strategy pattern:

• Base Strategy: This serves as an interface and provides the abstract methods that must be implemented.

• Concrete Strategy: These are concrete classes that extend the base strategy and add their own behavior.

• Context: This has access to the methods and contains an object reference to the Concrete Strategy.

Uses:

I utilized the strategy pattern to implement three distinct bombs that are dropped by the aliens. Normally, creating three different classes for three distinct behaviors and corresponding Bomb classes can be complex. However, by leveraging the strategy pattern, I was able to create a Bomb class that contains an instance of the BombStrategy.

As a result, removing any edge conditions and complexities, we can easily create three different instances of bombs with unique behaviors while maintaining the same abstraction. Additionally, extending Bomb's behaviors is straightforward - by creating a new subclass for the Base Strategy and implementing the BombFall and Reset methods.

Factory Design Pattern

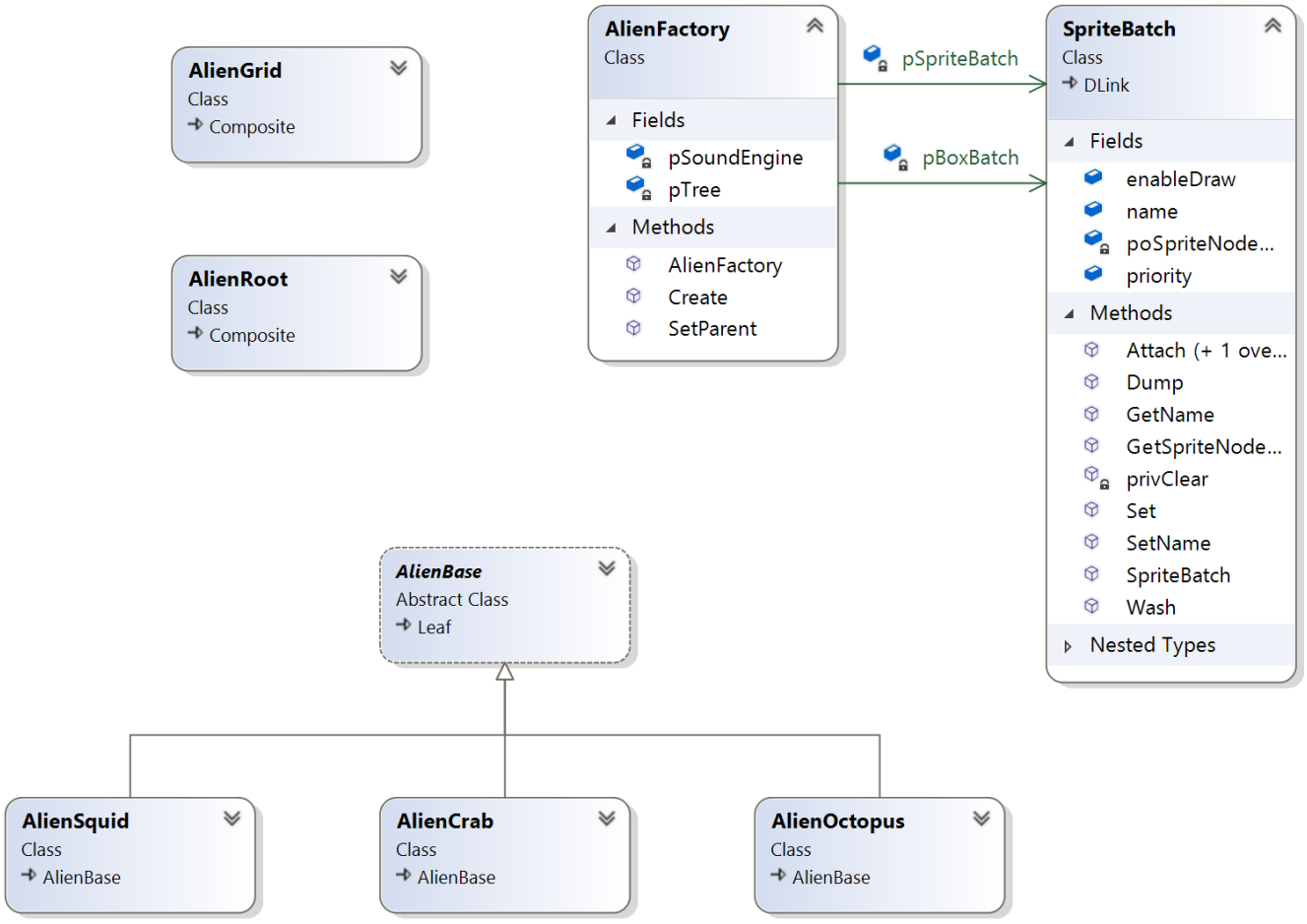
Problem:

To create a large set of objects with basic commands we had to use a lot of conditionals, the problem is to avoid it using an object oriented approach

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

The Factory pattern is a creational pattern that simplifies the process of creating instances by concealing the creation logic from users. It offers a unified interface for generating different instances of related classes without exposing the specific concrete classes.

Key Object-Oriented mechanics:

To create objects from related products without worrying about their internal representation, you can use the abstract factory pattern. This pattern involves implementing an abstract factory that the concrete factory extends. The concrete factory can create various objects from related products. Here are the key components of this pattern:

Base Factory: This component acts as an interface that provides abstract methods to create different products.

Client: To create instances easily, you can use the concrete factory and call the interface method with different parameters.

Concrete Factory: This component is a concrete class that extends the base factory and creates different products based on input.

Uses:

I utilized the factory pattern in two instances: AlienFactory and ShieldFactory. Due to the composite pattern that holds the 55 aliens together, creating different types using separate classes became challenging for the user. To tackle this issue, AlienFactory was implemented to aid in the creation of various instances such as Crab, Squid, Octopus, and others, based on arguments. This was done using switch statements with multiple cases, adding instances to the GameObjects, and linking them to the composite via a tree.

Another factory that was used is ShieldFactory, which functions similarly to AlienFactory, but instead creates shields with Gird, Column, and Bricks composite trees. I implemented the Create method in both factories, creating AlienGroup and ShieldGroup. Consequently, whenever I require it, I can simply use the Create method to create the same, making it a straightforward process.

Composite Design Pattern

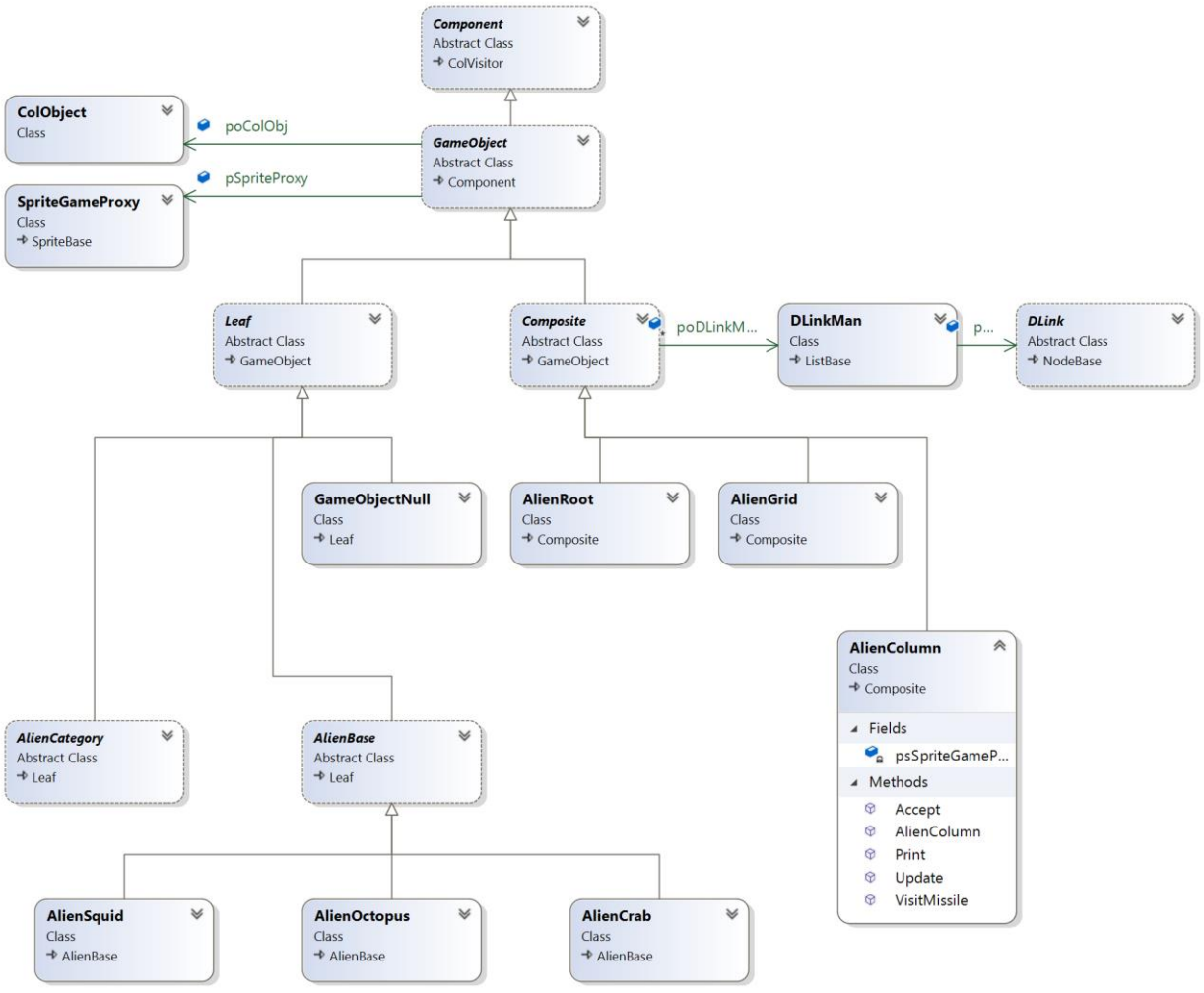
Problem:

We encountered an issue where we were unable to move all the aliens synchronously, and we needed to address this problem by treating all aliens uniformly.

Solution:

We employed the composite pattern to address this problem by treating the group of 55 aliens as a unified collection and incorporating a grid layout.

Pattern Description:



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

The Composite pattern is a structural design pattern in object-oriented programming that allows you to compose objects into tree structures to represent part-whole hierarchies. It enables clients to treat individual objects and compositions of objects uniformly. In other words, it enables the client to treat a group of objects as a single instance of an object, allowing the objects to be manipulated in a similar way. The Composite pattern consists of a component interface, a leaf class that represents the individual objects in the composition, and a composite class that represents the collection of objects in the composition.

This makes adding commands is a straightforward process, and the key distinction between the command and observer patterns is that commands can be added back to the list, enabling them to execute the same command multiple times, and can also be utilized to reverse actions.

Key Object-Oriented mechanics:

To achieve uniform treatment of both leaf and composite objects, we maintain the composite concept. We create a component interface along with two classes, "Leaf" and "Composite." The Composite class has child elements, while the Leaf class does not. By using the component interface, the user can easily access and modify the elements in the composite.

The Composite class holds a list of children that is linked to the component interface. We can add as many children as we want to the composite object iteratively.

• Component: This object acts as the interface for all composite objects.

• Composite: This is a concrete class that has multiple children and recursive behavior, connected to the component interface.

• Leaf: This is a concrete class that has no children and performs its specific behavior.

Uses:

In order to maintain the structure and ensure that all game objects are treated uniformly, we have implemented the composite pattern for all game objects. The game involves 55 aliens that are organized in a grid structure with 5 rows and 11 columns. We have created an Alien Group that acts as the parent of all grid structures and cannot be deleted. The Alien Grid contains the Alien Columns, each of which contains a different type of alien. The Aliens themselves are the leaves in this case, while the Group, Grid, and Columns are composites. We can also use Iterators to easily access and iterate over the data in the composite.

Moreover, I have used this pattern for all GameObjects in the game, including Missiles, Ships, Bombs, and Shields. This allows the game to uniformly organize all objects and makes them iterable.

Visitor Design Pattern

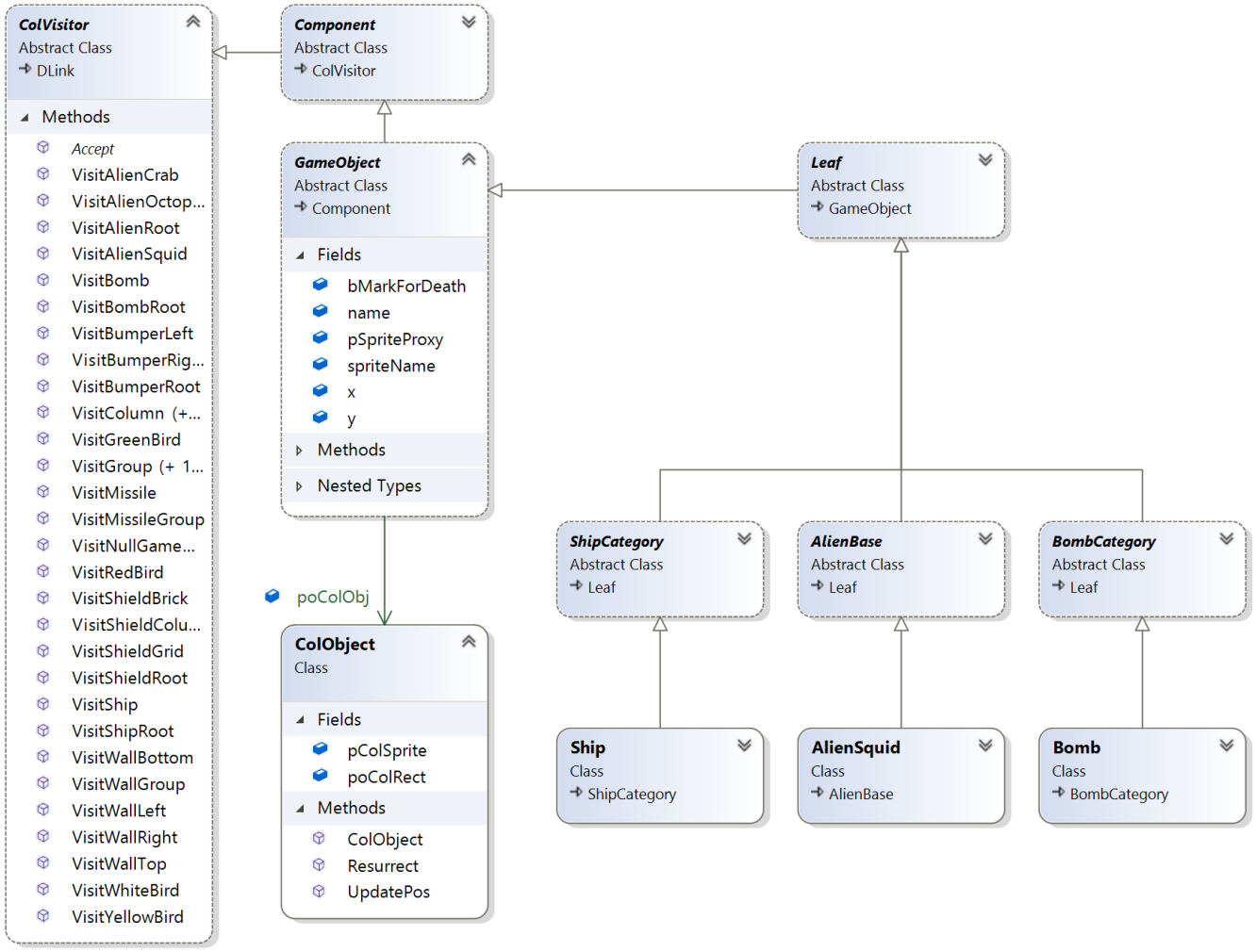
Problem:

I developed the collision mechanism for the game elements, but a significant challenge is figuring out what to do and which objects were involved when two distinct entities collide.

Solution:

To address this challenging problem, I applied the Visitor design pattern to GameObjects to achieve a double dispatch on both objects that had collided with one another, this will help me know which objects just collided.

Pattern Description:



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

The Visitor pattern assists in defining operations on the elements it is associated with, making it easy to add new operations. The key objective is to determine and apply equality to both objects, while also carrying out double dispatch.

Key Object-Oriented mechanics:

The standard implementation of the Visitor pattern involves an abstract Visitor class and concrete visitors with Visit methods for each element. When two elements visit each other, their respective Accept methods in the Element classes and the Visit methods in the Concrete Visitors are called. However, we employed a modified version of the visitor pattern for the collision system in Space Invaders. Since all objects are subclasses of Game Object, the Visit and Accept methods are combined in the same class, and the collision system will trigger the corresponding Accept method in the Game Object and execute double dispatch. The Visitor pattern in our implementation consists of three components: the Base Visitor, which serves as an interface for the abstract Visit methods; the Concrete Visitor, which is a concrete class that implements Visit methods and indicates that the object has a visitor; and the Concrete Element, which is a concrete class that extends the Base Element and has an Accept method that accepts the visitor object.

Uses:

The Visitor pattern proved instrumental in resolving the challenging issue of determining which objects had collided in our game. To achieve this, I created a Collision Visitor with abstract Accept and Visit methods. The CollisionVisitor implements the GameObject interface, while the various GameObject subclasses such as Bomb, Missile, Ship, ShieldBrick, or Aliens implement the required Accept and Visit methods. For instance, we know that the Ship will collide with the Bumpers and the Bomb, To trigger the notifiers, I linked the visit methods in the GameObject with the observer pattern, but only the corresponding Visit methods and the Accept method are implemented in the Ship class. By using virtual visit methods in the CollisionVisitor, we don't have to implement all of the methods in the GameObjects, only the ones related to collisions.

Observer Design Pattern

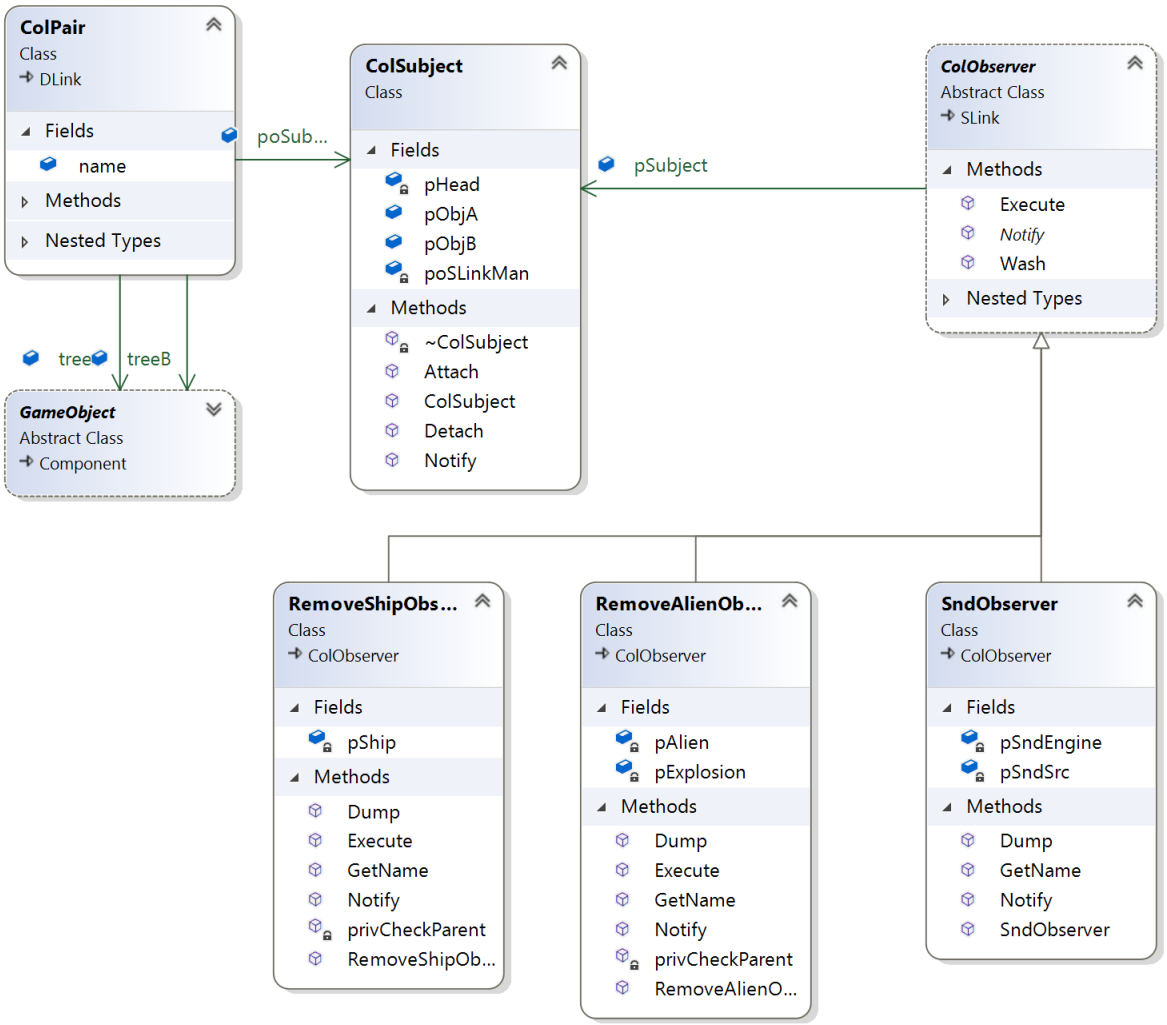
Problem:

The Observer pattern addresses the issue of notifying events to trigger actions. When two game objects collide, specific actions need to be carried out, such as removing an Alien, playing a sound, adding points to the score, and so on.

Solution:

To tackle this problem, we need to create multiple observers and link them to the collision pair. Therefore, when the event happens, it notifies all the observers that are associated with it.

Pattern Description:



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

The Observer pattern is a behavioral design pattern that establishes a one-to-many dependency between objects, such that when the state of one object changes, all its dependents are notified and updated automatically. In this pattern, the object that notifies its changes is called the "subject" or "observable", while the objects that receive the notifications are called "observers". The Observer pattern allows for loose coupling between objects, making the code more modular and flexible.

Key Object-Oriented mechanics:

The Observer pattern establishes a one-to-many relationship between objects, allowing for loose coupling and flexible design. Observers register with a subject, and are automatically updated when the subject's state changes. This pattern consists of an abstract Observer class, multiple concrete Observer classes, and a Concrete Subject class that holds the state and notifies its observers. The Observer pattern provides a way for objects to communicate and react to events without tightly coupling their implementation.

Uses:

There are many actions that need to be performed when game objects collide, such as removing an alien, adding a splat effect, adding score, and so on. To perform these actions, we need a way to notify the relevant parts of the program when a collision occurs.

This is where the Observer pattern comes in. The basic idea is that we have a set of Observers that are interested in some event, and a Subject that triggers that event. Whenever the event occurs, the Subject notifies all the Observers that are interested in it, and they can perform whatever action they need to.

In the Game, I created many Observers for each set of actions to be performed when a collision occurs. For example, there might be an AlienObserver that removes an alien and adds a splat effect when an alien collides with something, and a MissileObserver that removes a missile and adds score when a missile collides with something.

To make this work, I created a CollisionSubject that has a list of Observers that are interested in collisions. Whenever a collision occurs, the CollisionVisitor's Visit method delegates the work to the CollisionSubject, which then triggers the Notify() method on all of the Observers that are associated with that subject.

By dividing the code into multiple Observers, we can easily reuse them and attach them to multiple collision pairs. For example, if we want to remove a missile when it hits the top of the wall, we can simply attach the MissileObserver to that collision pair.

Overall, the Observer pattern makes it easy to implement actions that need to be performed automatically whenever a specific event occurs in our game.

Singleton Design Pattern

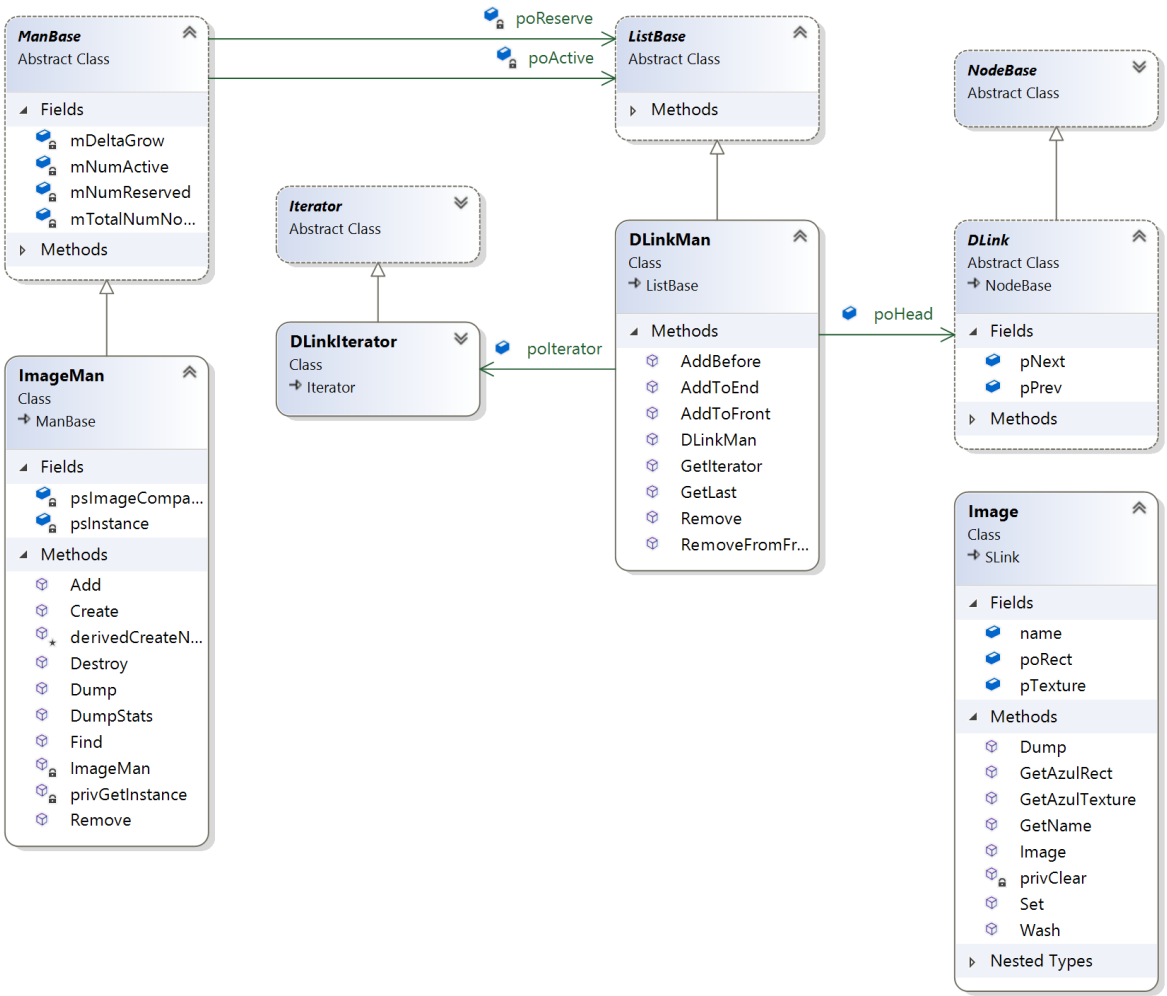
Problem:

Object pooling was utilized to establish a system that manages data. Managers are essential for adding, removing, or finding objects in various locations. However, it can be challenging to acquire an instance of the manager every time it is needed.

Solution:

To simplify accessing manager instances across the entire game, we utilized Singleton managers that allow for global access. Since Singleton managers only have one instance, we made it static and provided a global access point so that it can be accessed from anywhere in the game. This way, we can easily add, delete, or locate objects in the game without having to repeatedly create and access manager instances.

Pattern Description:



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

The Singleton pattern is used when a class should have only one instance, providing a global access point for easy access to everything. There are two ways to initialize the Singleton: either create the instance beforehand or create the instance only when it is needed.

Key Object-Oriented mechanics:

T The Singleton class usually has a private constructor that ensures that no other object can be created from outside the class. The only way to access the Singleton object is through a public static method, which returns the single instance of the class.

This pattern also makes it easier to access methods and properties of the class because they can be accessed directly by the class name rather than creating an instance of the class first.

There are two ways to initialize the Singleton instance - eagerly or lazily. In eager initialization, the Singleton instance is created when the class is loaded, whereas in lazy initialization, the instance is created only when it is first accessed. Eager initialization can improve performance, but it can also lead to unnecessary memory usage if the Singleton is never used. On the other hand, lazy initialization can save memory but may lead to performance issues if the Singleton is accessed frequently.

Uses:

Singleton managers are classes that are frequently used and can be found throughout the game. To make them more accessible and reusable, I implemented the Singleton pattern for the Managers. This is done by using static methods, which allows the methods to be accessed from anywhere in the game without creating a new instance every time.

By using the Singleton pattern for the Managers, it's easier to add, remove, or find nodes as needed. Instead of creating a new instance every time, the methods within the class retrieve the private instance and perform the required operations. This approach reduces the overhead and increases the efficiency of the game.

Object-Pooling Design Pattern

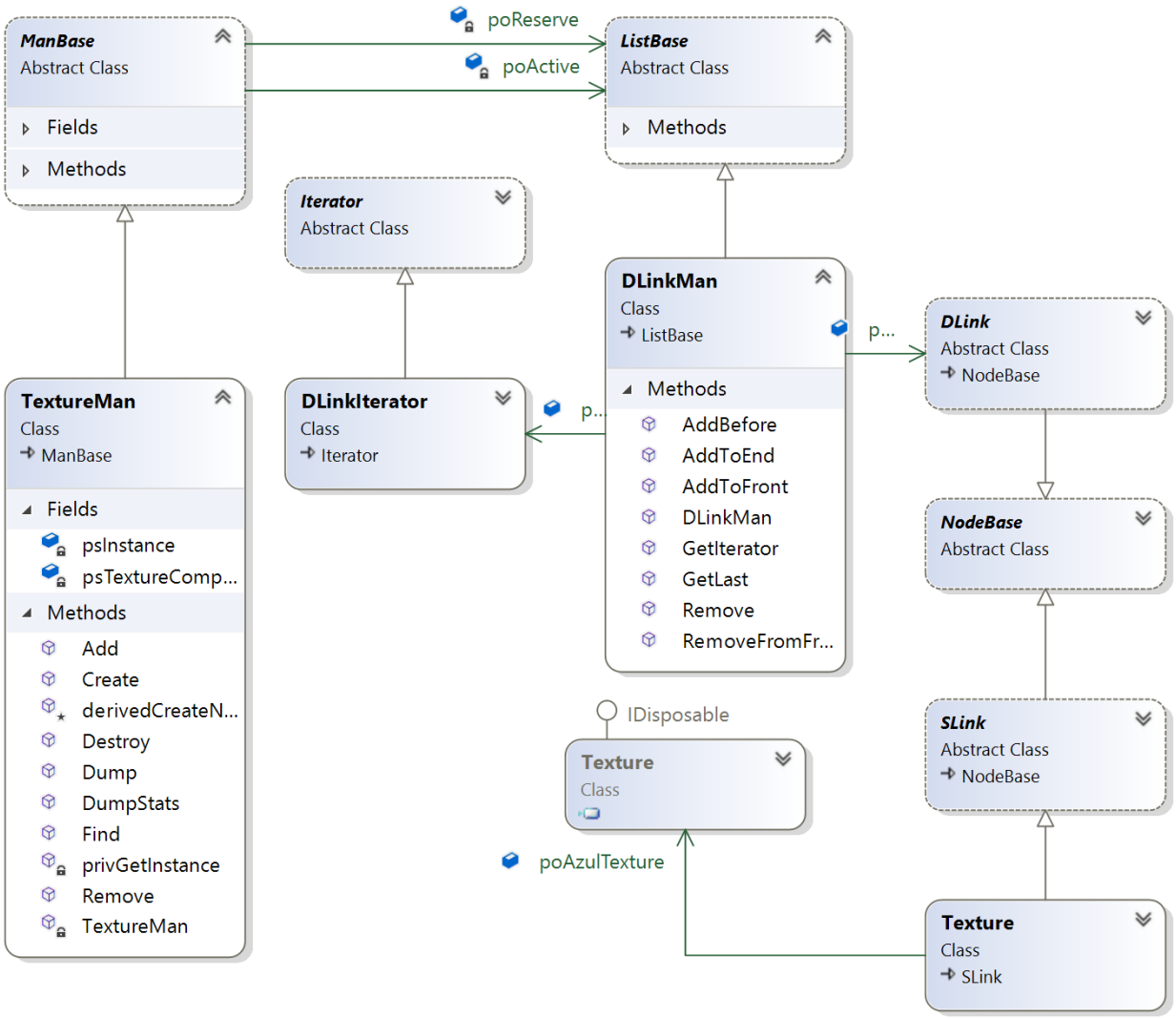
Problem:

Creating new objects can be costly in terms of performance in games. In Space Invaders, there were many objects that needed to be created.

Solution:

To optimize performance, object pooling was used to recycle existing objects instead of creating new ones. This helped reduce the overhead of object creation and improved the game's overall performance.

Pattern Description:



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

The concept of object pooling involves reusing objects instead of creating new ones, which can be costly in terms of performance. This approach enables the sharing of objects among game objects as needed, making it easier to add, remove, and locate objects in an efficient and effective way.

Key Object-Oriented mechanics:

Object pooling is a technique used to efficiently reuse objects by keeping removed objects in a reserve list and assigning them for reuse when a new object is required. This is because creating and initializing new objects can be costly. The pool checks its reusable pool for objects before creating new ones. By reusing objects, we can improve efficiency and avoid the unnecessary removal of objects by the garbage collector.

Uses:

The use of object pooling through managers is a common architectural technique implemented in all nodes. For instance, a TextureManager is created in the UML diagram to manage textures efficiently by reusing existing objects instead of creating new ones. The manager consists of two active list pointers - poActiveList and poReserveList. When a new texture is needed, the manager searches for available nodes in the reserve list, removes a node, adds it to the active list, and sets its values as needed. To remove a node, the manager removes it from the active list and returns it to the reserve list for later reuse. DoubleLink lists are predominantly used in the managers to optimize the process of adding and removing nodes. In the event that the reserve list is empty, the manager creates the necessary nodes and adds them to the list.

Adapter Design Pattern

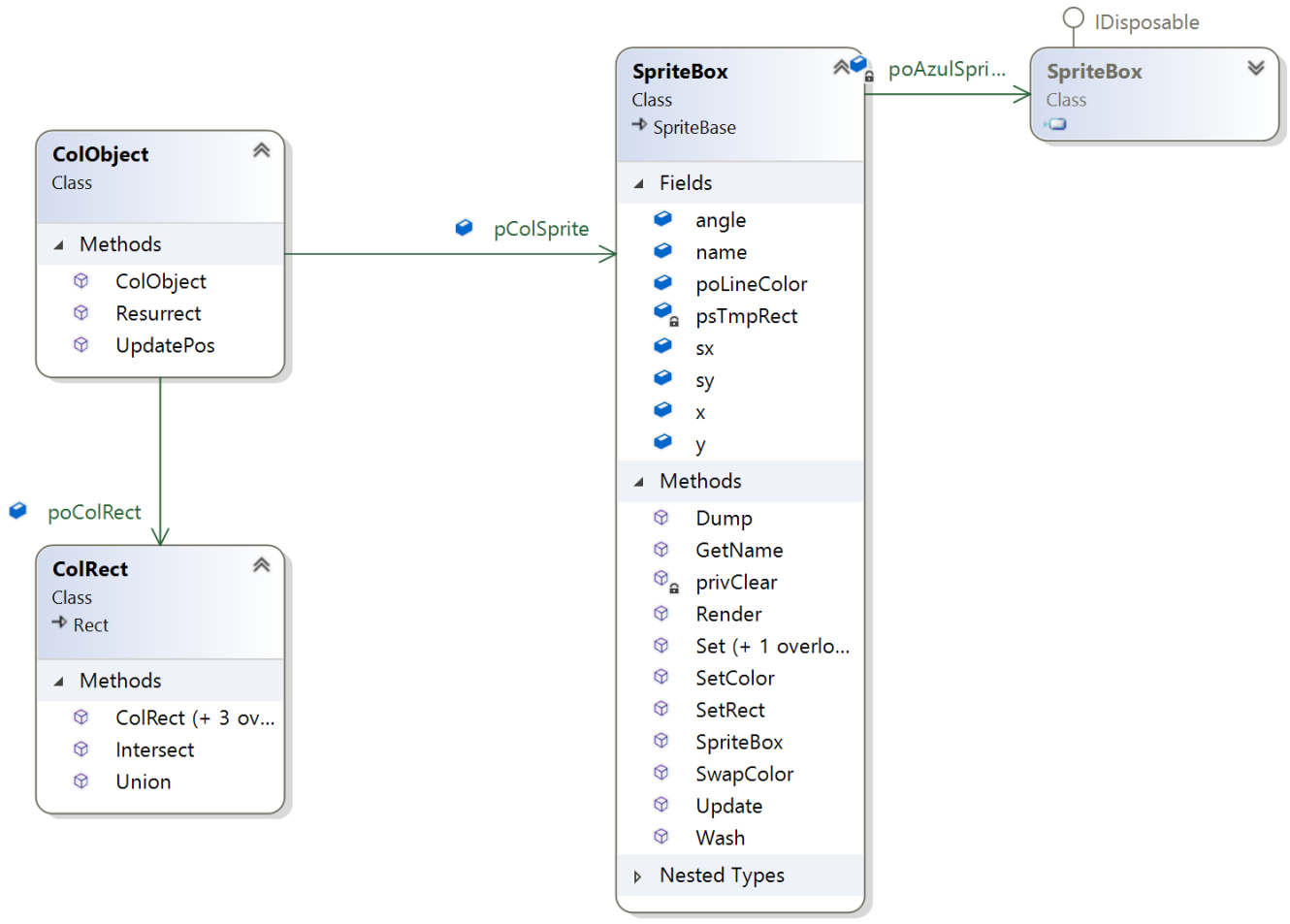
Problem:

In the game, the Azul framework is employed to handle tasks such as loading textures, updating sprites, and displaying game-related graphics on the screen. However, using the framework requires users to memorize complex method names.

Solution:

To simplify the usage of the Azul framework, we developed multiple adaptor classes for Sprite, Image, and Texture. These adaptor classes serve as a universal interface, bridging the gap between the client and the Azul framework.

Pattern Description:



The above UML diagram is an example for Adapter on the SpriteBox.

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.

In the context of software development, an adaptor class is created when the client needs to use a third-party library that has a different interface than the one the client is using. By creating an adaptor class, the client can use the third-party library without having to modify its existing code.

The adaptor class sits between the client and the third-party library, providing a translation layer to convert the third-party library's interface into one that the client can use. This conversion process makes it possible for the client to interact with the third-party library seamlessly, as if it were designed for the client's use from the beginning.

Key Object-Oriented mechanics:

The adaptor pattern serves as a mediator between two distinct interfaces or classes, facilitating communication between them. This pattern is implemented through an adaptor class that extends the target class and implements the required methods to convert and invoke the real class functions.

Uses:

In the game, we have utilized Adapter classes to interface with the Azul Framework, allowing us to use its objects, such as Image and Sprite, more efficiently. These Adapter classes act as a wrapper class, simplifying the process of using the framework's functionality in the game development process.

Command Design Pattern

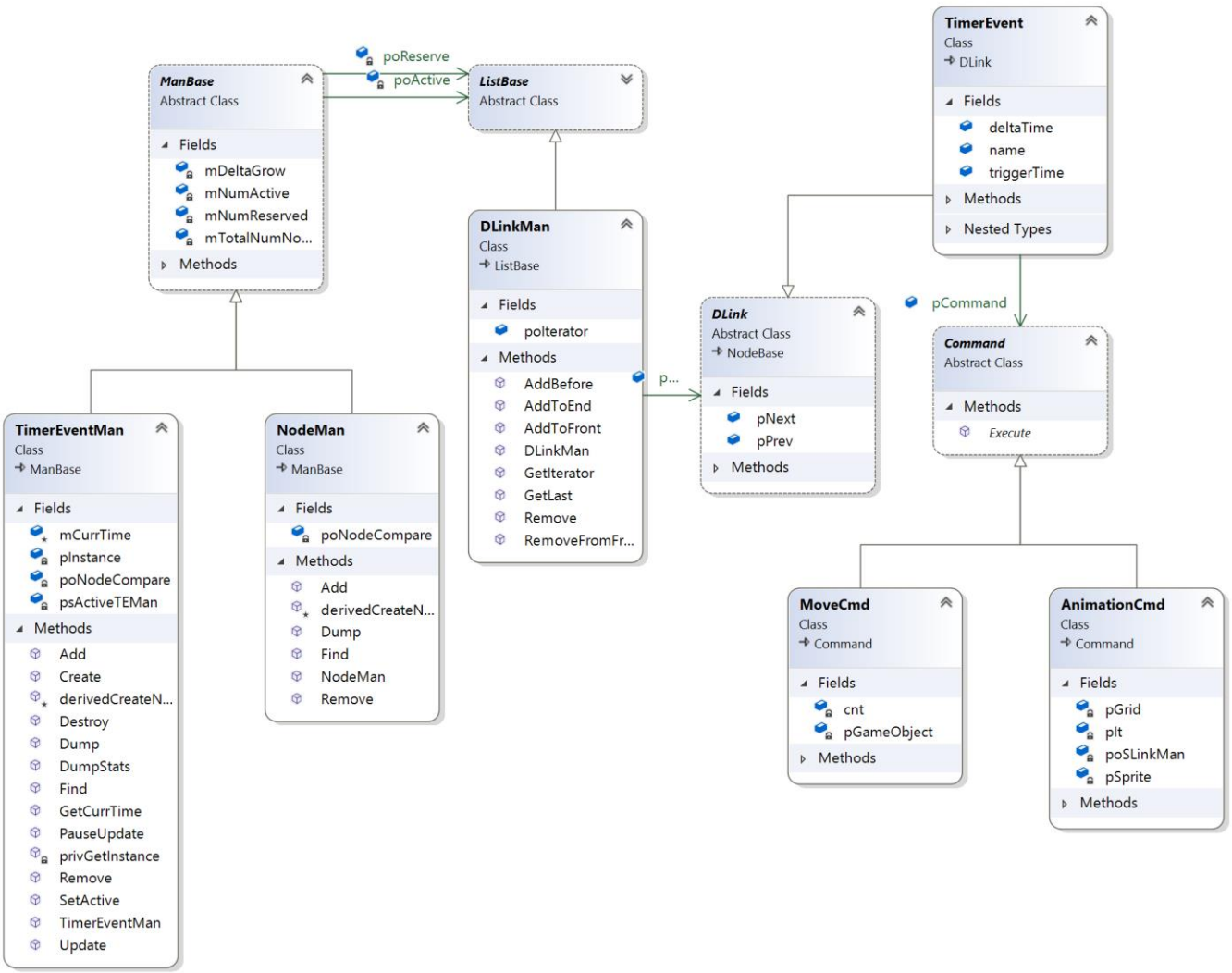
Problem:

We needed to execute some statements with specific timing, such as triggering the animation for alien marching, Animating the Ship while dying, and similar actions.

Solution:

To execute the required actions based on timer events in the Space Invaders game, we utilized the Command pattern with a priority queue implementation. This approach enables us to perform the actions in a specific order by assigning them priority values in the queue. By using this pattern, we can ensure that the actions are executed at the correct time, improving the game's overall functionality.

Pattern Description:



The above UML diagram is an example for Command Pattern in the TimerEventManager.

The Command pattern is a design pattern that encapsulates requests as objects, enabling separation of concerns between the object that invokes an operation and the object that performs it. It allows objects to be parameterized with various requests and executed at different times, and often includes an undo operation. This pattern is commonly used in GUIs, game development, logging, and transactional systems. Command pattern can be viewed as a way of implementing callback methods that hold requests to be executed at a specific point in time. Rather than being invoked immediately, these requests are encapsulated within objects and can be executed later when needed.

Key Object-Oriented mechanics:

The Command pattern can be implemented by defining an abstract class and several concrete commands that override the execute() method. The base Command class acts as an interface for the method being executed, providing abstraction. The Concrete Command class is a concrete implementation that overrides the execute() method and performs an action on the receiver. The Receiver class is a concrete class that carries out the action based on the request. In this pattern, the commands act as callback methods that call methods on the receiver class.

Uses:

There are several actions that need to be performed at specific times. These actions include animating the aliens, moving them synchronously, and performing other tasks. To handle these actions based on timer events, we implemented the Command pattern on the priority queue.

In this pattern, we defined several concrete commands, such as the Animation Command and Move Command, which override the execute() method. Each command is added to the TimerEventManager based on the priority of timeToTrigger, which ensures that these events are referred to sequentially based on time.

By using this pattern, we were able to easily add new timer events to the TimerEventManager or remove them as needed. For instance, in the Animation command, we swapped the image on the sprite and then added the same command to the timer with delta time. This ensured that the commands ran indefinitely until they were manually removed from the TimerEventManager.

One of the benefits of using the Command pattern is its flexibility. We can pause events, add multiple commands at a specific time, and perform other actions as required. Overall, the Command pattern provides an elegant solution for handling timer-based events in Space Invaders.

Summary:

In this project I made a clone of Space invaders game from scratch using multiple design patterns to create a real-time data-driven application, the development process presented its own unique challenges. Taking a total of 10 weeks, during which I followed best practices in software engineering to ensure the project was completed efficiently and effectively.

Throughout the development process, I utilized Iterative Development with weekly sprints, Testing, Continuous Integration, and a final design document. These best practices enabled me to create a fully functional game that met the project's specifications and requirements.

We Used Basic C# with Visual Studio 2019 Enterprise for coding, which proved to be a robust and reliable combination. The project also involved using Helix Visual Client - Perforce for version control, which enabled me to manage the codebase. I used the Azul GUI Framework to create the graphical user interface, which allowed me to develop the game's visual elements quickly and efficiently.

One of the most significant challenges in creating the game was trying to implement the software design principles and pattens to improve the quality of software by making it easier to modify. While I couldn’t test or implement all the specifications due to time constraints, I did my best to bring the game to life and provide an enjoyable gaming experience for the players.

However, as with any project, there is always room for improvement. For instance, I would like to enhance the Alien movement mechanism so that the speed increases over time, making the game more challenging as it progresses. I also noticed some bugs that need fixing and that the Alien Grid Respawn could be improved to make the game more exciting. Additionally, I would like to add a two-player mode, which would allow players to compete against each other, adding an extra level of fun and excitement to the game. I also plan to include graphical noise-based dissolve effects on the Shields and the UFO with its animation, which drops bombs, to make the game more visually appealing and engaging.

Overall, this project was a significant undertaking for me, and one of the most challenging and largest projects that I have completed to date. I am extremely grateful for the assistance provided by Professor Keenan, whose guidance and support were invaluable throughout the development process. Creating this game has taught me many valuable Software architecture skills and helped me understand how to design and implement real-time software architectures and has given me a deeper understanding of the software development process, which will be invaluable in my future software engineering endeavors.