SE 456 Space Invaders
Design Document

Mohsin Ismail

03/20/2018

Link to Demo Video

TABLE OF CONTENTS

1 Introduction

1.1 Purpose

This document outlines the specifications and design pattern implementations used to create the Space Invaders game. While much of the underlying functionality makes use of the Azul software package provided in advance, this document will describe the overall design and high-level view of the game, the various components and design patterns implemented, how the implementations function within the game and gameplay, and possibilities for further improvement.

1.2 Goals

The goal of this project was to recreate the original Space Invaders arcade game as faithfully as possible, while making use of design patterns to write more efficient code. The Space Invaders game consists of a 5 x 11 grid of aliens that animate and move left and right on the screen, moving down slightly and reversing direction when hitting the left or right edge. A player ship at the bottom of the screen can move left and right, and shoot missiles to try and destroy the aliens one by one. The player has 3 lives to get the highest score possible.

To meet the relevant specifications, this game is designed to stably run for at least 2 full levels in the 1-player mode, and likely many more after that. The game was thoroughly tested under numerous conditions and stress tests in order to provide the smoothest possible experience.

2 Design Patterns

2.1 Object Pool Pattern

The first task was to be able to manage the various types of objects that would be created within the game, by keeping the objects organized and being able to traverse a list of the objects present. To solve this, we used the Object Pool pattern utilizing the abstract DLink class, which provides doubly-linked list functionality, and from which other classes could derive from. An abstract Manager class that would provide functionality to interface with the DLink class is also created.

In the Figure below, it's shown that the Image class inherits from DLink, which implements its own doubly-linked list functionality. We also have a corresponding ImageManager class which derives from

the abstract Manager class and provides base functionality for adding/removing from the associated linked list, as well as a number of other methods.

Upon instantiation of a manager, the abstract Manager class's constructor is first called. The Manager class contains two references to DLink objects, with one reference representing the first DLink on an "active" list, and the other representing the first DLink of a "reserve" list. When the Manager is created, a set number of objects are created right away and added onto the manager's Reserve list.

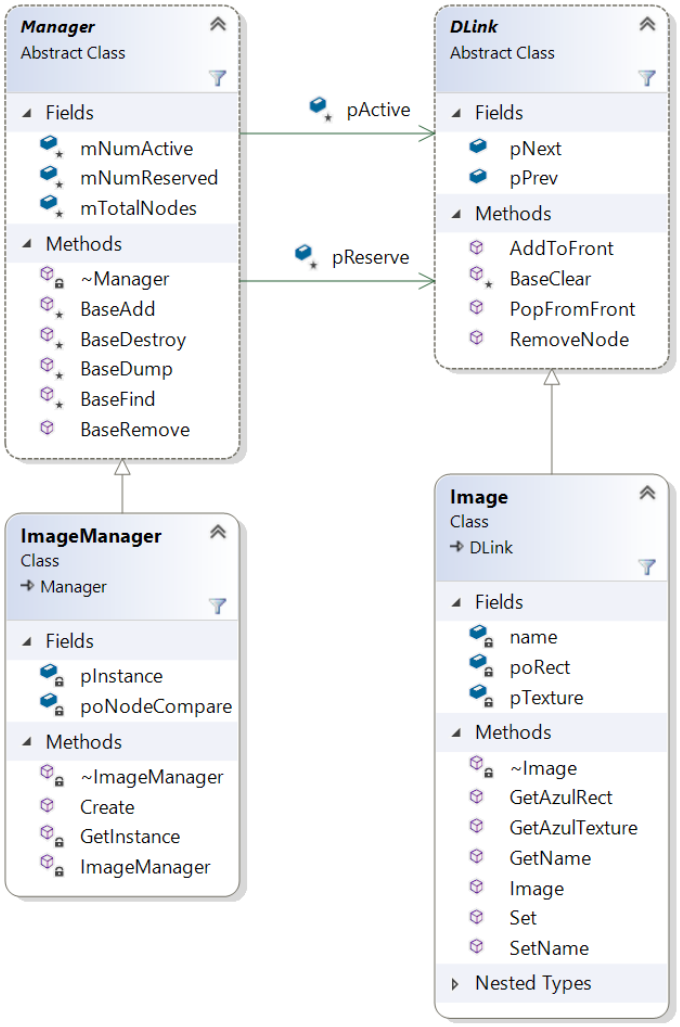

Figure 2.1.1. Example of the Manager design pattern used to manage Image objects

If an Image object needs to be created, rather than calling new Image() to instantiate a whole new object, we can call the ImageManager's Add() method. Here, instead of creating a new object, the Manager will check if there are any existing objects on the Reserve list. If one exists, the Manager will remove that object from the Reserve list, add it to the Active list, and return the object to the function caller. If the Reserve list is empty, the Manager will re-populate the Reserve list and then perform this

process. Similarly, when an Image is no longer needed, rather than deleting the object directly, we can call the ImageManager's Remove() method, which will remove the Image object from the Active list, reset all of its member variables, and add it back to the Reserve list.

This design pattern is applied throughout the milestone project for a number of different classes. While different types of specific manager classes may include their own methods specific to the object type they are managing, all managers utilizing this pattern are able to easily manipulate the underlying list used to keep track of the objects.

2.2 Singleton Pattern

For a majority of the objects in the game, it was necessary to have a single manager that can manage its objects across all classes and be accessible from anywhere within the game. For example, having a single ImageManager that would serve as a repository for all of the images that may be used and reused within the game. To achieve this, the Singleton pattern was implemented in most of the various manager classes. This pattern provides a way to design classes with a single point of access to a single instance of that class, limiting the ability to create new instances elsewhere within the game.

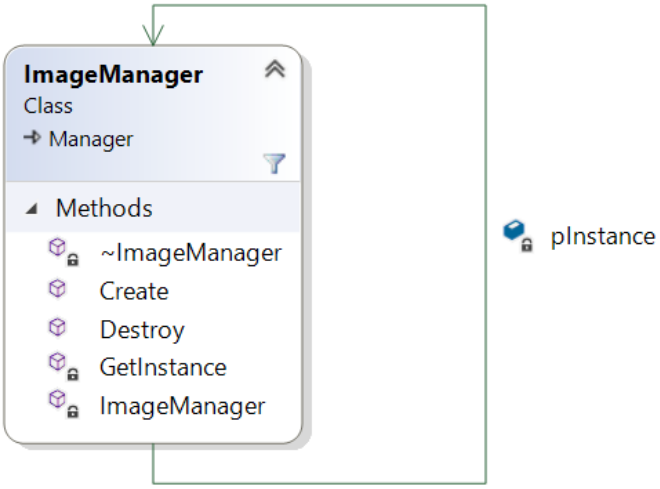

Figure 2.2.1. Example of the Singleton design pattern applied to the ImageManager class

In this Figure, the ImageManager class implements the Singleton pattern. The class contains a reference to an ImageManager, pInstance, which is initially set to "null". The class constructor is private, preventing any instantiations from outside of the class itself. The Create() function first checks that pInstance is null, and then creates a new ImageManager object using the private constructor, which it then assigns to pInstance. The GetInstance() function, by contrast, checks that pInstance is not null, and

then returns the ImageManager pInstance. This accomplishes the goal of making sure that there is only one ImageManager present at any given time.

Within the game, this singleton pattern proves very useful. When the game initializes, several managers of different types utilizing this pattern are created, and then reused throughout the rest of the game execution. As a result, we can be sure that the ImageManager created with 6 Image objects is the same ImageManager that may be referenced in another class, with access to the same 6 Image objects.

2.3 Adaptor Pattern

Figure 2.3.1. Example of the Adaptor design pattern used to wrap a legacy Azul.Sprite class in a new interface

The Adaptor pattern solves the issue of reliably implementing the legacy Azul package classes while also extending the functionality as necessary for this project. Most of the underlying objects that are rendered on the screen make use of Azul's own Sprite, Color, or Rect classes. However, at different times during the game's execution, we may want to change some of the attributes of our objects and, as a result, the underlying Azul objects.

The Sprite class in this project is an illustrative example of the Adaptor pattern in use. The true object that will be updated and rendered on the screen is the Azul.Sprite object. In order to be able to manipulate the attributes of this sprite, we wrap the underlying object in a new interface with new methods (the Sprite class). The rest of the classes in the game then interact with the Azul.Sprite through this Sprite class we create, essentially converting the interface of the Azul.Sprite class into one that the rest of the game can understand and work with.

2.4 Composite Pattern

When dealing with the multitude of GameObjects that are created and manipulated during the game, it becomes increasingly useful and necessary to implement a hierarchical structure for these objects. This is where the Composite pattern comes into play. Shown in Figure 2.4.1 below, this pattern mainly comprises of the Component, Composite, and Leaf classes, and behaves similar to a tree structure. In a tree consisting of nodes, there are nodes that act as intermediaries and contain further nodes beneath them, and nodes that have no children and are the "leaves" of the tree. This is exactly how the Composite pattern is implemented: a Component defines the highest level of abstraction, from which both Composites and Leafs are derived. Composites act as the intermediary nodes and Leafs are the final leaf nodes of the data structure.

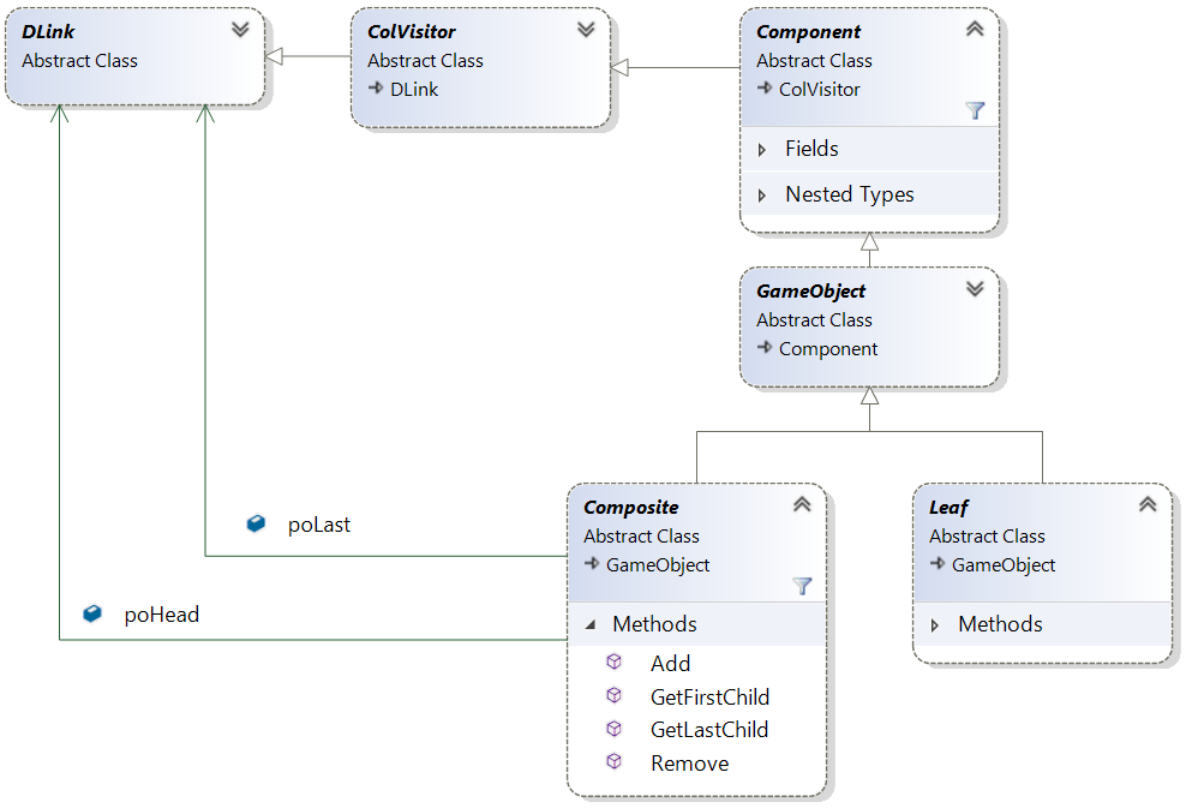

Figure 2.4.1. Example of the Composite design pattern used to create a hierarchical structure of GameObjects

In the Figure above, there are additional classes that are interwoven with this pattern, providing functionality specific to the game's mechanics. The Composite and Leaf classes, rather than deriving directly from Component, are derived instead from the GameObject class, which then derives from the Component class itself. Component then Derives from the ColVisitor class, which in turn derives from the DLink class. Although the ColVisitor class is not part of this pattern, the chain of derivation is

shown to illustrate that both Composite and Leaf indirectly derive from DLink and contain doubly-linked list functionality.

In line with the intended behavior of this pattern, the Composite class also has two references to DLink objects, as shown in the Figure above. Practically, this allows for a Composite object to contain a reference to a head node, the first of any number of children objects which can be both Composite and Leaf objects.

```
public abstract class Component : ColVisitor
{
    8 references
    public enum Type...

    27 references
    public abstract void Add(Component c);
    3 references
    public abstract void Remove(Component c);
    3 references
    public abstract void Print();
    3 references
    public abstract Component GetFirstChild();


    public Component pParent = null;
    public Component pRevNext = null;
    public Type type = Type.Unknown;
}
```

Figure 2.4.2. Code snippet from the Component class, showing the contractual methods that both Composite and Leaf classes must implement

Within the game, this pattern is used in creating a hierarchical grid of Alien objects, organized into AlienColumns. The Alien class derives from the Leaf class and inherits its inability to hold any children objects, while the AlienColumn and AlienGroup classes derive from the Composite class. Similarly, the 4 Shields in the game are actually ShieldGroup objects, which contain multiple ShieldColumn objects, which then contain many ShieldBrick "leaf" objects. This way, when Aliens or ShieldBricks are added or deleted, the Composite pattern's inherent structure allows for easy ways to deal with such situations.

2.5 Iterator Pattern

The Iterator pattern provides a way to sequentially access the members of an aggregate structure without having to deal with the underlying functionality of the structure. Ideally, aggregate structures or collections of objects would be entirely decoupled from the algorithms used to iterate through the structures. However, within this game, the Iterator pattern is designed to interact only with structures that adhere to the Composite pattern.

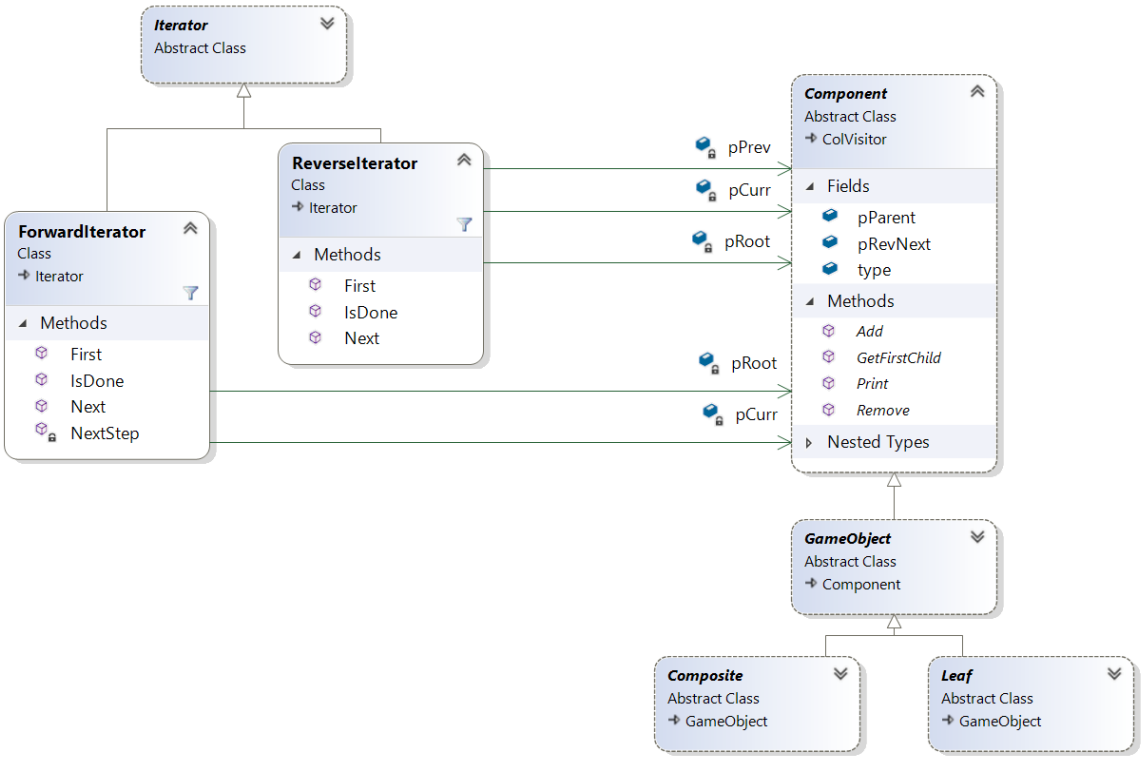

Figure 2.5.1. Example of the Iterator design pattern used to sequentially access Components arranged in the Composite structure

The ForwardIterator is created by taking a Composite object as a reference argument. The iterator then iterates through the specific tree structure in a depth-first manner, returning all child nodes of an object before moving on to the next sibling. The ReverseIterator iterates through the objects in a Composite structure in the exact opposite manner of the ForwardIterator. Together, these iterators are used to cycle through GameObject structures when every object needs to be updated or adjusted in some way.

```
private Component NextStep(Component pNode)
{
    Component pParent = GetParent(pNode);
    Component pChild = GetChild(pNode);
    Component pSibling = GetSibling(pNode);

    if (pChild != null) pNode = pChild;
    else
    {
        if (pSibling != null) pNode = pSibling;
        else
        {
            while (pParent != null)
            {
                pNode = GetSibling(pParent);
                if (pNode != null) break;
                else pParent = GetParent(pParent);
            }
        }
    }
    return pNode;
}
```

Figure 2.5.2. Code snippet from the ForwardIterator class showing the algorithm used to retrieve the next node

Within the game, this pattern is used when moving whole groups of objects, or checking for precise collisions. The grid of Alien objects is moved at every interval by creating a new ForwardIterator with the AlienGroup passed as a parameter, and updating the position of every object that the iterator returns. This process is replicated for all bombs that are rendered on the screen as well. When detecting collisions, the root nodes of two Composite structures are first checked; if there is a collision, an Iterator can return the child node of the containing object to check for a more specific collision. If there is no collision, an Iterator can return the next sibling object to check for collision instead.

2.6 Factory Pattern

Throughout the game, there are numerous instances of various classes being created at any given time. GameObjects and TimeEvents, specifically, are created with more variation in the type of data that will be passed in as parameters. Each of these types of objects then have to be attached to their respective Managers, and in the case of GameObjects, have to be attached to one or more SpriteBatches as well. If we were to manually write all of these steps every time a new object was created in any function, we would be re-writing a lot of code. To alleviate this, and to simplify the process of object creation, the Factory Pattern is used.

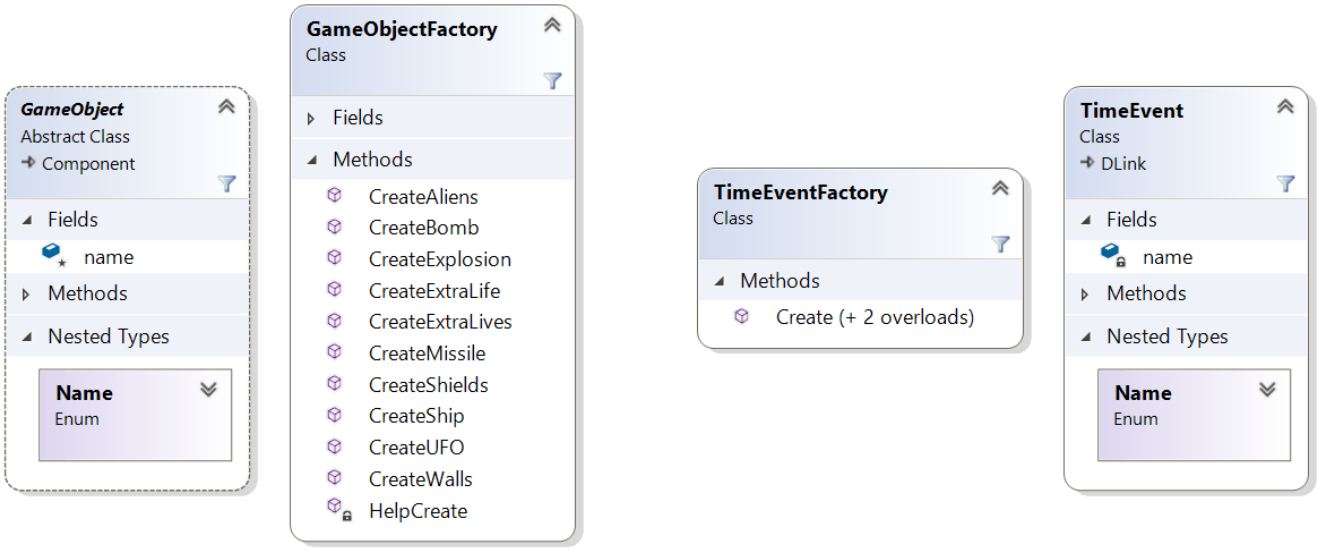

Figure 2.6.1. Example of the Factory design pattern applied to the creation of GameObjects and TimeEvents

The example above shows the implementation of the Factory design pattern used in the GameObjectFactory and TimeEventFactory classes. Shown in the Figure above, the GameObjectFactory contains a number of methods that can be used to create many types of GameObjects, based upon the specific method called and parameters passed in. Similarly, the TimeEventFactory class contains two overloaded Create() methods, each of which takes a specific set of parameters and creates a specific TimeEvent.

While this pattern won't provide any visual benefit to the person playing this game, this pattern simplifies the process of creating an object, especially when there are multiple actions involved. For most GameObjects created, the object has to be attached to the relevant Manager, SpriteBatch, or Composite object. Using this pattern, we can delegate these actions to the underlying functionality of the corresponding Factory, and know that every action that needs to be done is taken care of.

Another benefit this pattern provides is greater extensibility when adding new features further down the line. If we add another object type to the project that requires its own specific actions when creating a new instance of that object, we can easily modify the factory by adding a new method. Additionally, if at a point it becomes necessary to attach every GameObject created to an additional SpriteBatch, or a second Manager, only the Factory would need to be modified, since it acts as the central point through which most objects are instantiated.

2.7 Proxy Pattern

We reuse Sprite objects in a number of places throughout the project. For example, the grid of Alien objects starts out with 55 GameObjects, but only 3 unique Sprites. The problem is that Sprite objects hold many different data, such as pointers for Azul.Image, Azul.Sprite, Azul.Color, and Azul.Rect, as well as their individual position and scale variables.

If we were to create 55 individual Sprites that would each be updated and rendered, the project will still run. However, we would be reusing a lot of data that doesn't change for similar Sprites. In our grid of Alien objects, for the two rows of Crab sprites, the only difference among the 22 Sprite objects are the x and y variables (i.e., the starting position of the Sprite). There would likely be 21 copies of the same Azul.Image, Azul.Sprite, and Azul.Rect pointers. To solve this problem and make the project more efficient in its use of game objects, the Proxy pattern is used.

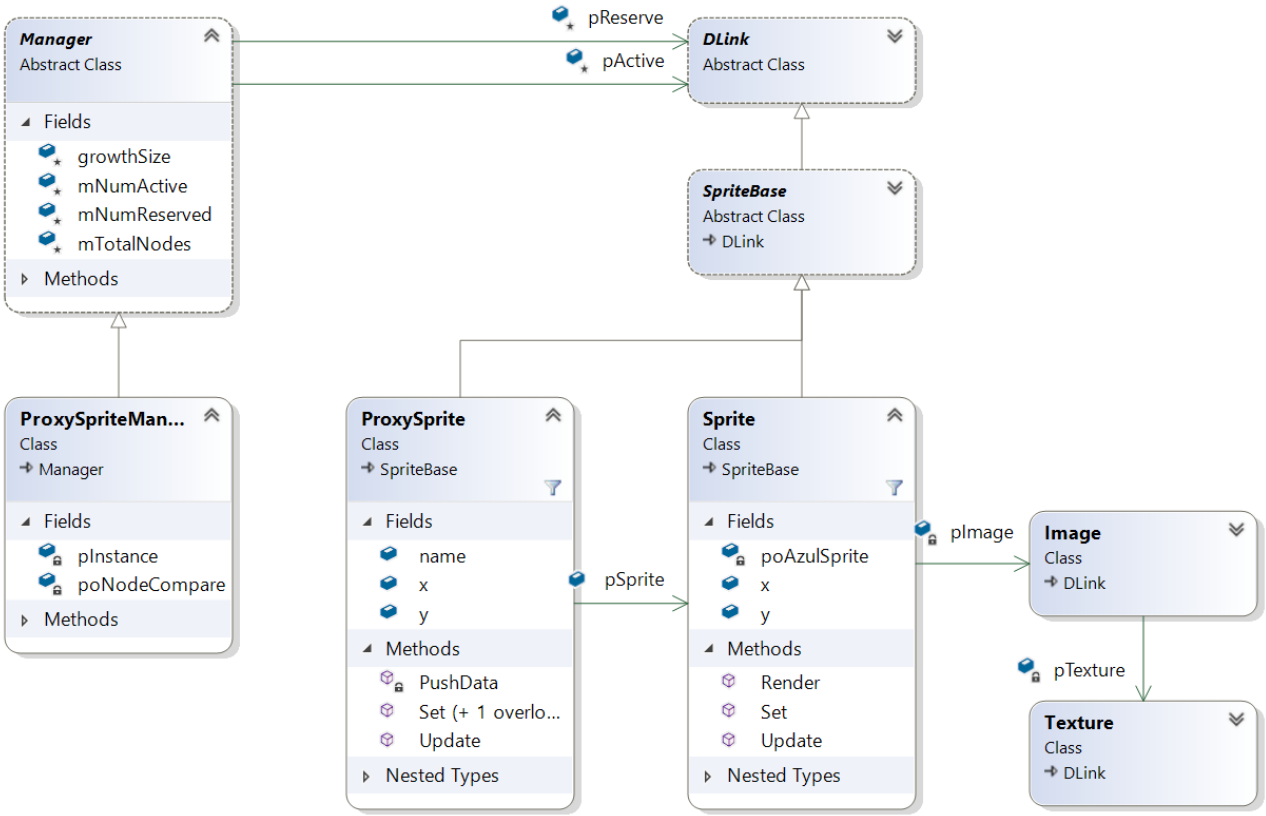

Figure 2.7.1. Example of the Proxy design pattern used in creation and management of Sprite objects

The Figure above shows the ProxySprite with a reference to a Sprite object, while containing its own unique data. The Sprite class contains the overlapping data, such as the Azul.Image pointer to the Image it references, that will be reused among the various ProxySprite objects that all reference the same Sprite. Within the game, this means that rather than needing to initialize 55 different Sprite

objects, we initialized 3 Sprite objects and then created 55 ProxySprite objects.

The ProxySprite class contains methods for pushing its unique data to the Sprite object it references when needing to update and render all of the sprites present on the window. It's at this moment of rendering where the necessary data for the ProxySprite is swapped into the Sprite object and drawn on the screen, after which it is reused by the next ProxySprite that references it.

2.8 State Pattern

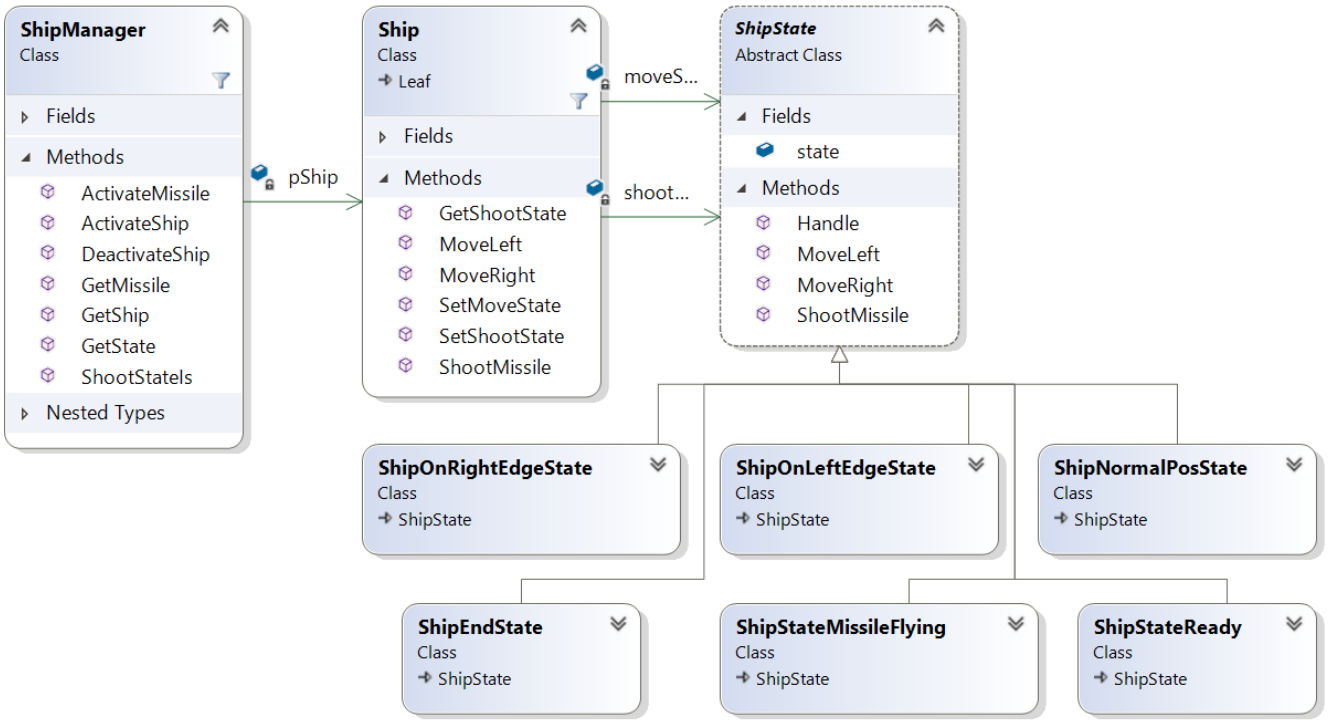

Figure 2.8.1. Example of the State design pattern used in managing the various Ship states during the game

Over the course of the gameplay, the player ship is bound by certain rules: the ship cannot shoot more than one missile at a time, and the ship cannot move past the left and right wall boundaries. However, since the ability to move the ship and shoot a missile is controlled by key inputs, the game should always capture the input and call the relevant function for the Ship. To allow for this type of functionality, we implement the State pattern.

This pattern allows for the Ship to have a reference to a certain state class and its underlying methods, and to define those methods in variable ways. In the game, there are six ship states; three that account for the various cases of ship movement, two that account for the cases when a missile is shot, and one for the case where the ship is destroyed. These states all derive from an abstract ShipState class, which

defines the contractual abstract methods that the subclasses need to implement: Handle(), MoveLeft(), MoveRight(), and ShootMissile(). For example, the ShipNormalPos state defines object movement for both the MoveLeft() and MoveRight() functions, indicating that the Ship is not colliding with a wall, whereas the ShipOnLeftEdge state defines object movement only for the MoveRight() function, indicating that the Ship is colliding with the left wall, and should not be moved any further left.

Additionally, because each state can implement specific functionality for each contractual method, they can be written in such a way that that the Ship state can be switched from within the states themselves. In contract to the ShipNormalPos state, the ShipOnLeftEdge state's MoveRight() method moves the ship position right, and also calls switches the state of the Ship back to ShipNormalPos. This ability to have the states affect which state will be called next for the relevant object is a useful feature of this pattern.

This State pattern is also one of the first noticeable design patterns, and one that integrates with the execution of the game most directly. Similar to the original Space Invaders game, we want to have distinct states that the game cycles through:

- Attract state – this will serve as an introduction to the game, and a point at which the player can enter a key and start the gameplay.
- Playing state – this will consist of the actual gameplay, where the various objects in the game will constantly be updated and rendered.
- GameOver state – the game will enter this state after the game ends, and will pause any animations or movements happening on the screen. A red label with the words "GAME OVER" will be displayed. From here, entering a key will lead the player back to the Attract state.
- Init state – this state is the one the game is initialized with, loading resources and files and automatically switching to the Attract state. The Init state is only called once.

By giving the game a reference to a GameState, and having a GameStateManager manage the game's states, we can have the game behave differently according to the state the player is currently in. And again, a key mechanism is the ability to switch the game states from within the states themselves after having met some condition, and this is where the State pattern excels.

2.9 Observer Pattern

Often times during the game's execution, there are various interactions that can take place from a number of detectable collisions. When certain interactions take place, we may want to update many different parts of the game. For example, if an Alien collides with a Missile, we would want to:

- Delete the Alien
- Delete the Missile
- Update the Score
- Reset the Ship state
- Speed up the rest of the Aliens in the grid

A number of these actions may also be done for other detected collisions as well, such as deleting the Missile when a Missile collides with the top Wall. To prevent rewriting code in multiple places, we make use of the Observer pattern in the collision detection process.

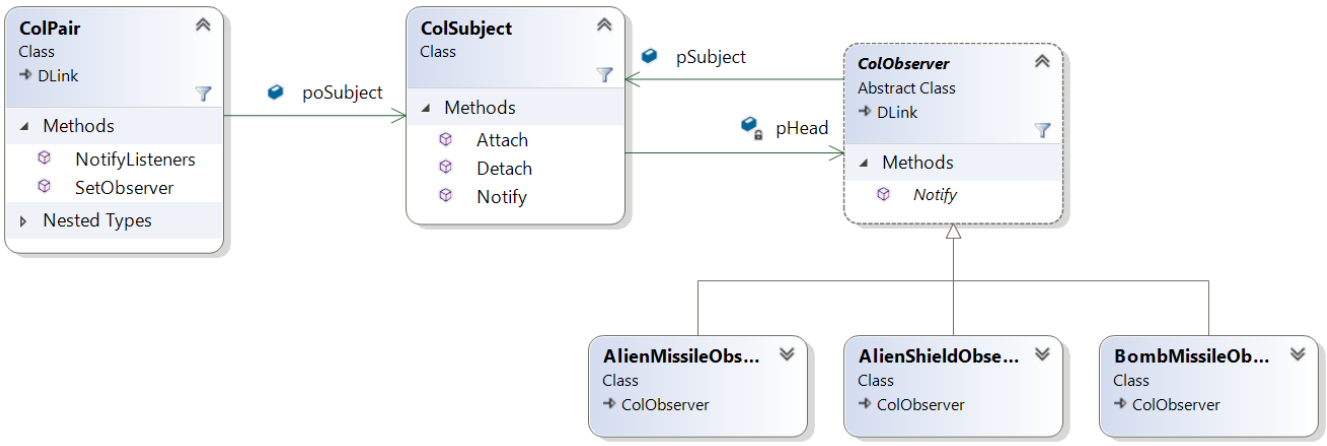

Figure 2.9.1. Example of the Observer design pattern used in processing various events upon collision detection

Upon creation, each ColPair is instantiated with a ColSubject object. We can then call the Attach() method in ColPair to add any number of ColObserver objects to this ColPair object's ColSubject. In Figure 2.9.1 above, there are 3 types of observers shown, all of which derive from the abstract ColObserver class. Contractually, every observer type implements a Notify() method, which performs a specific action (e.g., Deleting the Alien and Missile in the AlienMissileObserver().

When a collision is detected and an action needs to be done according to the types of the two GameObjects that collided, the ColPair's NotifyListeners() method can be called, which will in turn call Notify() on the ColPair object's ColSubject. This Notify() method will then iterate through all of the ColSubject's observers and call each one's Notify() method.

These observers can be attached in any combination to any ColPair object, allowing for easy reusability and simplicity when needing to determine what actions to take for specific collision events.

2.10 Visitor Pattern

When checking for collisions between different pairs of GameObject types, we want there to be unique actions taken for each specific collision. However, the ColPair class, responsible for holding the GameObject.Name enum values for the two GameObjects to be collided, retrieves the respective GameObject objects from the GameObjectManager when a collision is checked. However, the catch is that we want our ColPair to be generalized enough to be able to initiate a collision response for any two GameObjects that it may have, while simultaneously being able to do start a specific collision response based on what the specific type of object the two GameObjects are.

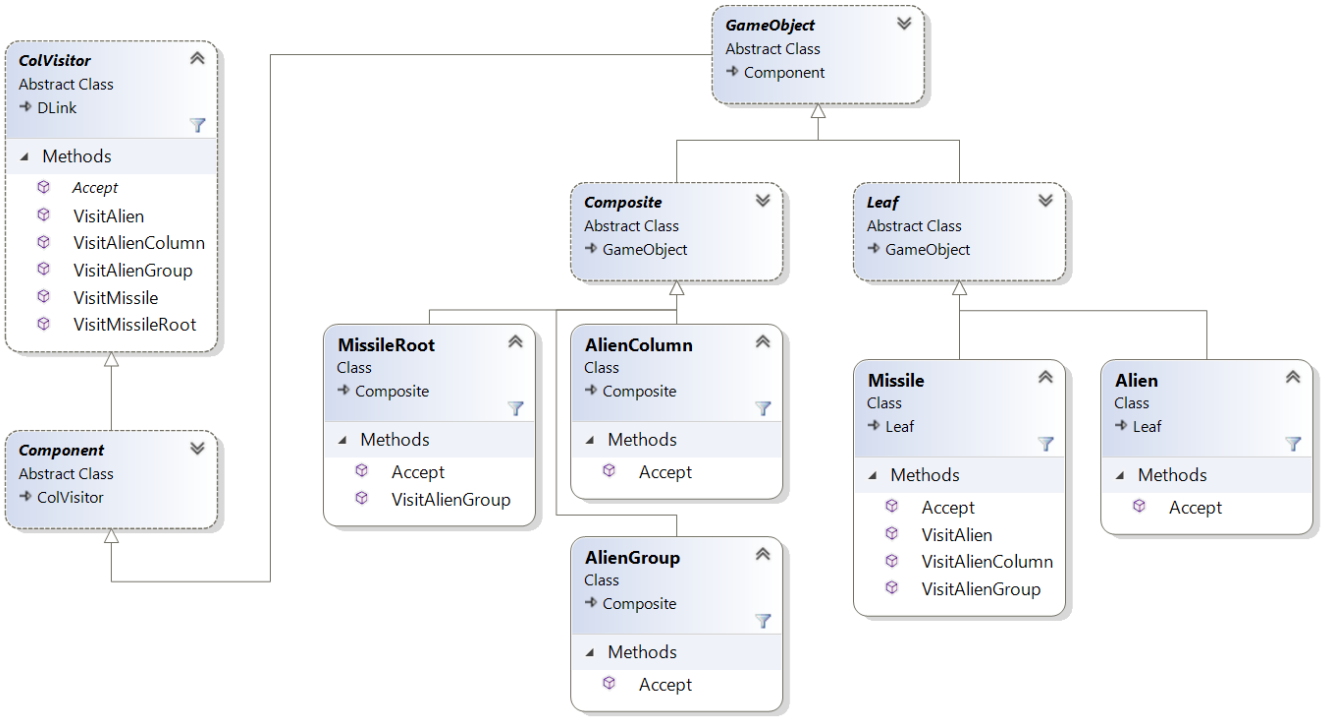

Figure 2.10.1. Example of the Visitor design pattern implemented for detecting collision between an AlienGroup and MissileRoot

This is where the Visitor pattern proves to be immensely useful. In Figure 2.10.1 above, the specific implementation of the Visitor pattern is shown with respect to Alien and Missile objects. Because all GameObjects derive from the Component class, which in turn derives from the ColVisitor class, each GameObject indirectly inherits from ColVisitor. The ColVisitor class contains a contractual Accept()

method that must be implemented in every concrete subclass, and a number of virtual methods that can be overridden in subclasses.

As an example, when a collision is detected by Alien and Missile objects, Alien's Accept() method is called, with the Missile parameterized as a ColVisitor object. Because Accept() is ColVisitor's method that the Alien class overrides, it's impossible to say within an Accept() method what the two interacting ColVisitor objects actually are. Here, in AlienGroup's Accept() method, we will call the MissileRoot's VisitAlienGroup() method, which is allowed because VisitAlienGroup() will have been implemented as a virtual method in the abstract ColVisitor class. Once in the MissileRoot's VisitAlienGroup() method, we know that the parameter passed in is an AlienGroup because of the specific function that is called. We also know now that we are in the MissileRoot class once the class's method is called.

```
// Wall.cs
6 references
public override void VisitAlienGroup(AlienGroup a)...

3 references
public override void VisitShip(Ship s)
{
    ColPair pColPair = ColPairManager.GetActiveColPair();
    Debug.Assert(pColPair != null);

    pColPair.SetCollision(s, this);
    pColPair.NotifyListeners();
}

8 references
public override void VisitMissile(Missile m)...

10 references
public override void VisitBomb(Bomb b)...

3 references
public override void VisitUFO(UFO u)...
```

Figure 2.10.2. Code snippet of the overridden functions implemented to detect collision in the Wall.cs class

The Figure above only shows a subset of the classes that make use of the Visitor pattern. Most GameObjects in this project use the Visitor pattern in an effort detect collision in an easier fashion. This pattern also integrates smoothly with the Observer pattern's functionality, since for each specific collision pair we detect, we want specific actions to be performed.

2.11 Command Pattern

At certain time intervals, we want to perform some action, such as animating a sprite object or moving the grid of Alien objects. To achieve this, the Command pattern was implemented allowing for different classes of actions to be associated with a time event.

The diagram above shows a TimerManager singleton class that can hold and manage multiple TimeEvents on a doubly-linked list in a sorted fashion. Each TimeEvent object references a Command object. The abstract class Command incorporates a single abstract method Execute(), which the derived classes are required to implement. This abstraction allows us to create multiple TimeEvent objects that can perform a different type of action, rather than having specific TimeEvents for every action needed to be performed on an interval, and managing each type separately.

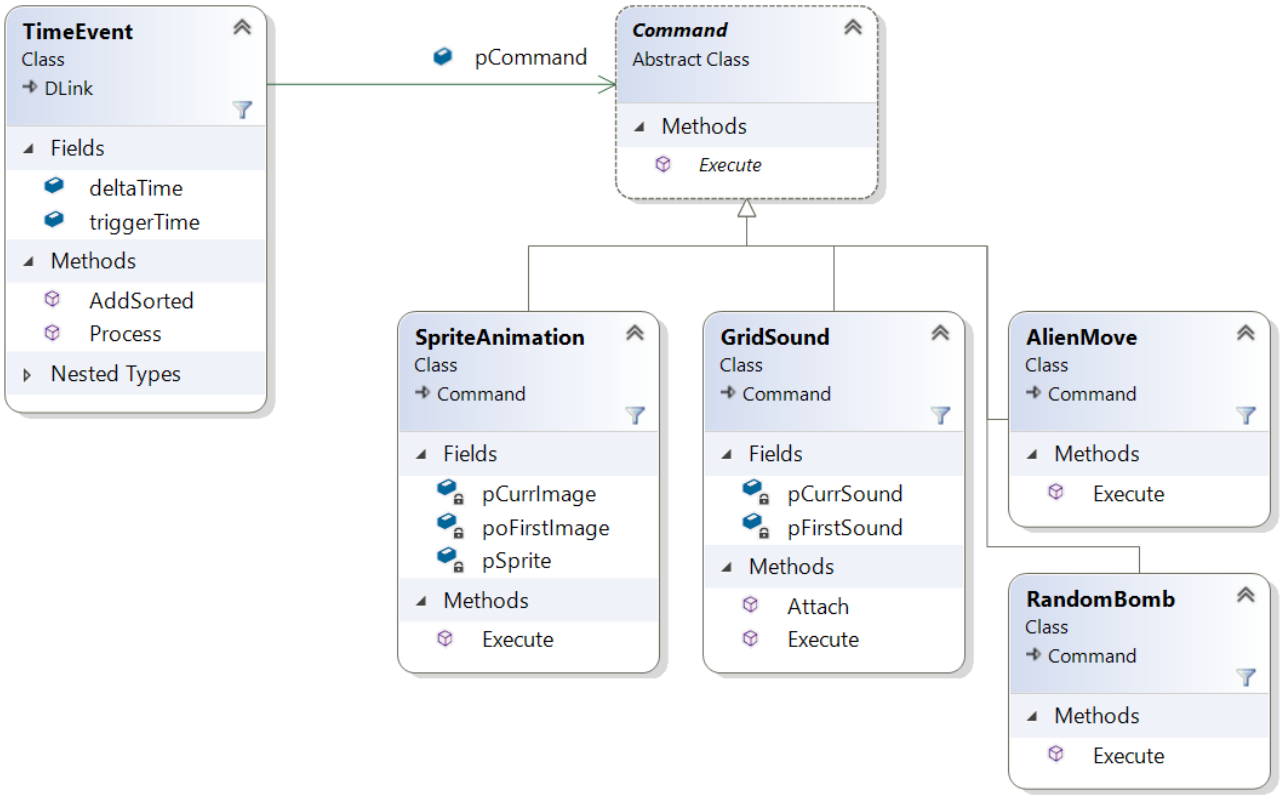

Figure 2.11.1. Example of the Command design pattern used to generalize executable commands associated with TimeEvents

Within the game, this allows for the singleton TimerManager to manage all actions that need to be performed on a timer. Because TimeEvent inherits from DLink, this also allows for a single call to TimerManager's Update() method to iterate through the list of TimeEvents and call TimeEvent's Process() method, which in calls its referenced Command object's Execute() function, regardless of what kind of derived Command it is.

3 Discussion

The process of designing this game included using a number of design patterns to fix a certain problem before implementing one that worked the best. After spending a considerable amount of time integrating these patterns into the project, it became clear that having these structural components within the code proved to be more useful as more and more features and components were added. The process of designing a large-scale project would invariably become more difficult if there was minimal forethought and conceptual planning taking place beforehand.

Each design pattern, while solving a particular type of problem, comes with tradeoffs. In this project, the design patterns implemented were written somewhat specifically to accommodate the related classes. This limitation made it difficult to add certain features. For example, this project does not contain any 2-player functionality because in order to add that feature in much later, the design patterns that were already written to accommodate a single-player system would have to be rewritten, and many classes would need to be refactored.

This only shows that these design patterns are only frameworks on how a particular section of a system could become more efficient. Going forward, it will be an even more nuanced process when implementing design patterns in larger systems in a way that maintains compatibility and improves extensibility.