SE-456 Real Time Software Architecture

Space Invaders Re-Creation

Design Document

Nicholas Edington
3-22-2020

1 CONTENTS

2 SPACE INVADERS RE-CREATION SUMMARY

2.1 INTENTION

This is a project to re-create the classic arcade game Space Invaders. This is built using C# and heavily relies on Object Oriented Design principals. The use of C# libraries was heavily limited for this project. C# standard system and diagnostic libraries were used along with the Azul library for the creation of sprites and playing of sound files. Otherwise, no collections, templates or other advanced features we utilized. The intention is to create a real-world representation of several design patterns built from the ground up for the creation of this game.

This document represents the major design components used in this implementation. Additionally, heavy detail is given for many of the specific design problems encountered during the development. Explanations are provided for the traditional design patterns used to solve these problems in addition to how they were specifically used within the context of this product.

2.2 OVERALL DESIGN

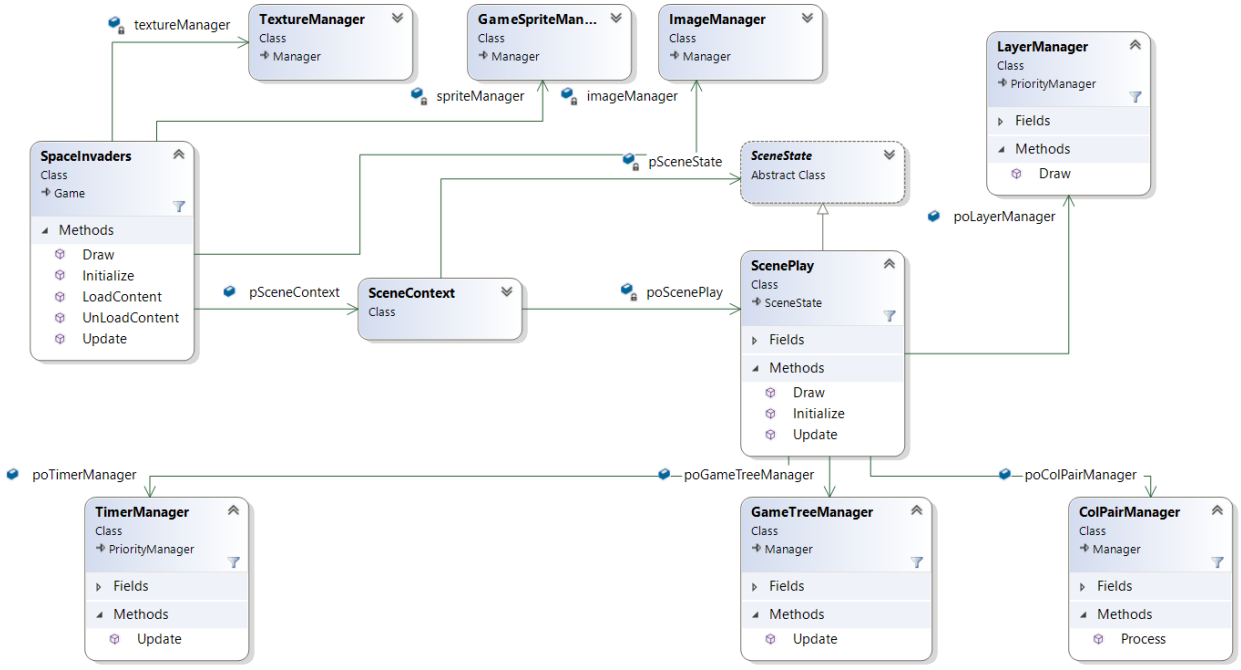

Figure 1:Overall design

The core game loop in handled in the Game class. This handles periodic updates, initialization and drawing to the screen. The TextureManager, ImageManager and GameSpriteManager use the Azul library to convert texture files into sprites that can be drawn to the screen.

Switches in context are handled between this Game class and the SceneContext. The SceneState, shown with the ScenePlay example above, use instance of the other core systems to perform actions in their respective contexts.

The TimerManager is responsible for event handling based on the game time. It is updated during each Update call on the SceneState and executes the appropriate operations given the updated game time

The GameTree manager contains tree representations of all the game objects used by the play state. These are updated in the Update call to represent movement of the game objects.

Many of the states have individual LayerManager instances which are utilized upon the call of Draw. This allows these states to have their own scene drawing state to represent large changes between the visuals between scenes.

The ColPairManager contains all the colliding GameObject trees in pairs. Upon the Update call the ColPairManager object's Process method is used to determine if any collisions occurred between GameObject instances.

3 COMPONENTS

3.1 GAME SPRITES

3.1.1 Summary

Sprites refer to the graphics used to render images to the game screen. In this project there are 3 primary classes of sprites: game sprites, box sprites and font sprites. Game sprites are the sprites that represent the traditional game graphics, such as the aliens, shield bricks and the player's ship. All sprites utilize the Azul game library.

3.1.2 Creating and Managing Sprites

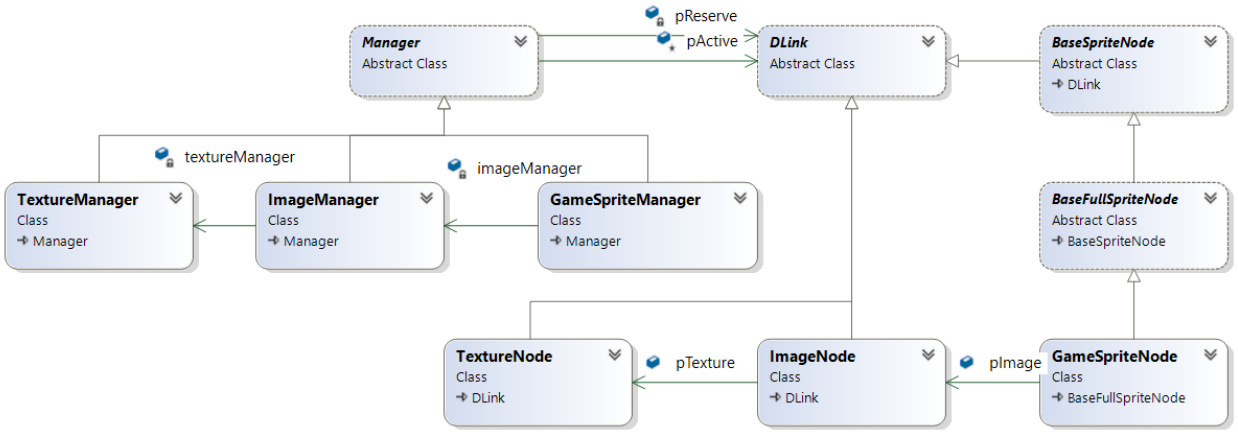

Figure 2: Creating and Managing Sprites

Game elements are pulled from textures which come from texture files. Then an image is created by specifying a rectangle on the texture. This is finally sized into a game sprite, ready to be placed on the screen.

The creation and ownership of the textures, images and sprites are all controlled by their individual managers which implement the object pool and singleton patterns as described in sections *Problem: Creating and Destroying* and *Problem: Single Instances* respectively. Because of this, the ImageManager

can the TextureManager to build the ImageNode objects with and a similar relationship can be used for the GameSpriteManager.

3.1.3 Using Game Sprites

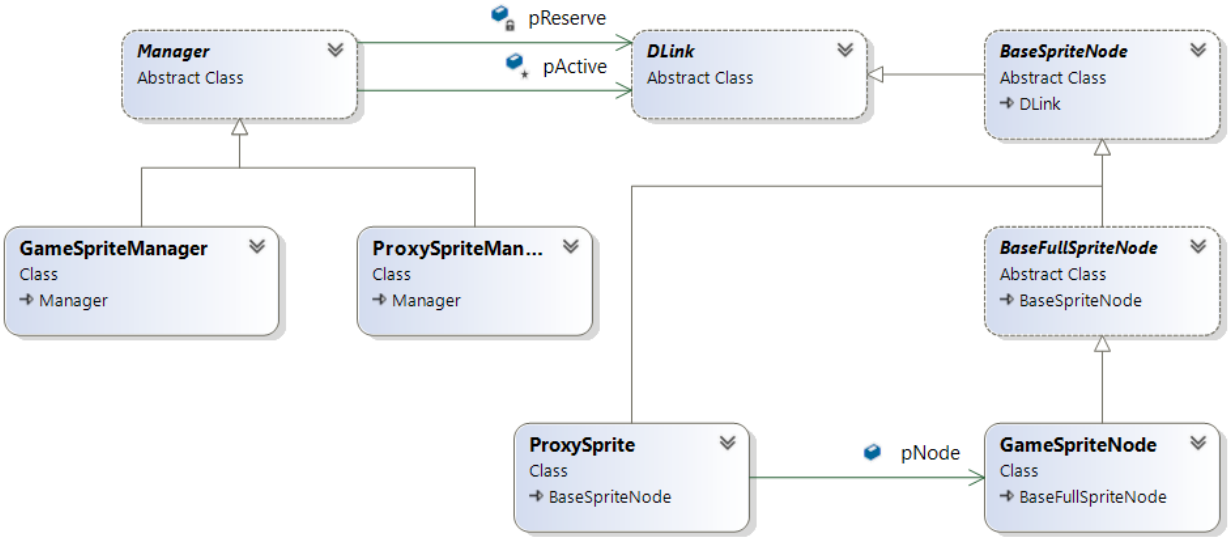

Figure 3: Using Sprites

After creation, GameSpriteNode objects are not accessed directly but rather though proxy sprites, which contain references to them. The ProxySpriteManager functions similarly to the other managers but provides ProxySprite objects which implement the proxy pattern as described in the section *Problem: Distributed Control of Game Sprites*.

3.1.4 Problem: Creating and Destroying

3.1.4.1 Description

Creating and destroying game sprites and the related objects are expensive processes. Every time one of these objects is created with the "new" keyword and allowed to go out of scope, the C# garbage collector is free to perform the expensive destroy process at any time, potentially slowing down gameplay. Additionally, some game sprites, like the aliens for every level, may need to be created several times during play.

A creational pattern is needed to control the instantiation and deletion of these objects and reduce the number of objects needed.

3.1.4.2 Object Pool Pattern

The object pool pattern involves a reusable pool class for all objects that have expensive instantiation and deletion. This object pool class manages a reserve list of one type of object. Any time a new object is needed, instead of it being created directly, the reserve pool is polled. If the reserve pool does not have any objects, it grows itself by pre-determined size by creating new instances of the objects it manages. Then the newly created or already existing object is returned to the client. Once there is no more need for the object, it can be returned to the reserve pool and stripped of any relevant state for later use.

3.1.4.3 UML

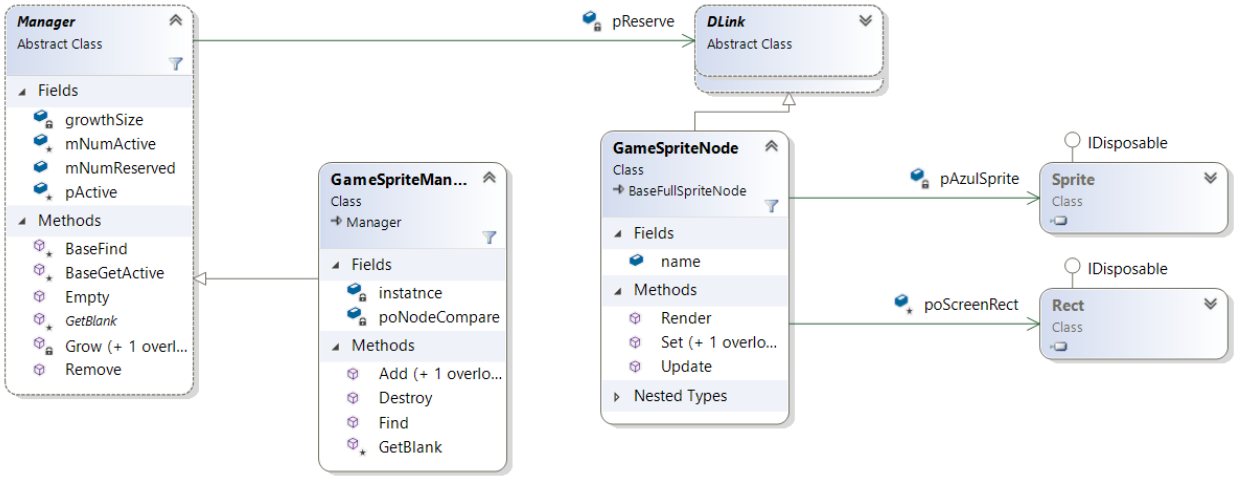

Figure 4: Object Pool Pattern

3.1.4.4 Use in Game

This pattern is used frequently throughout the game. For this component, there is a manager for textures, game sprites, images, and proxy sprites. Each of these follow a similar pattern as shown in Figure 4: Object Pool Pattern.

In the diagram the Sprite and Rect objects are from the Azul library. The instantiation of the sprites as well as of these objects are intended to be mitigated by this pattern. For this project, most of the object pools are implemented as singletons. This ensures there is only one poll for each type of object and makes the pools globally accessible.

Additionally, there is an active list used here. This maintains the link between the manager and all its objects currently being used. Through use of the Find method, it makes all active object also globally accessible.

The Manager defines the handling of a linked list of DLink objects which represents the reserve list. It contains instructions for growing the reserve list if needed. Growth utilizes the GetBlank method implemented by the subclasses which defines the minimal creation of the object type being pooled.

When a client requests that a new sprite be created, the BaseAdd function handles pulling from and/or growing the reserve list. The object is placed on the active list and then the manager calls the object's Set function to make the recycled object comply with the client's request. Finally, the object can be returned. When any object isn't needed anymore, the manager's Remove function can be called with the object. That will cause it to be cleaned and returned to the reserve list.

3.1.5 Problem: Single Instances

3.1.5.1 Description

For Space invaders, object creation, input handling, event dispatching, along with many other elements must be carefully managed to avoid memory leaks and state mismatches. Additionally, the control of these elements must be easily distributed so that they can interact with the many other systems.

This requires a way to ensure that for certain classes there is only ever one instance in existence. It is also pertinent that these instances are globally accessible to allow many objects access to them without needing to be passed their references. This prompted the use of the creational singleton pattern.

3.1.5.2 Singleton Pattern

The singleton pattern uses a private constructor to limit the instantiation of the class it is applied to. This instance is stored as a static variable and it can be accessible as a public member or via a public static GetInstance method. It is possible to utilize lazy instantiation, in which the constructor is not called until the first invocation of the GetInstance method, and then the instance is returned as is on subsequent calls. Furthermore, it is possible for all the classes' methods to be static and use the GetInstance method on each call to acquire the instance to perform operations on.

3.1.5.3 UML

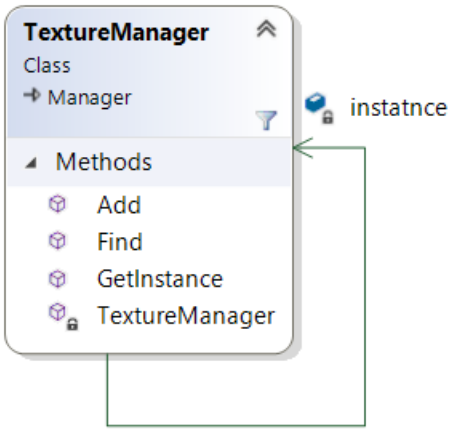

Figure 5: Singleton Pattern

3.1.5.4 Use In Game

For this component, the pattern was used for all the object pools. The TextureManager, is accessible to all other objects via the GetInstance method and so it can ensure a single shared pool of textures. In the case of this program, the singletons' methods other than GetInstance and Destory are not static so that not all the manager's method implementations require a call of the GetInstance method.

3.1.6 Problem: Distributed Control of Game Sprites

3.1.6.1 Description

Many game sprites need to be replicated on the screen in different positions such as the 55 aliens or the hundreds of shield bricks. There needs to exist instance data for the locations of each of these sprites, but it should not be necessary to create a GameSpriteNode for each of them, even if they are controlled by object pools. This leads to the use of the structural Proxy pattern.

3.1.6.2 Proxy Pattern

The proxy pattern allows for a surrogate of a complex object, which itself has minimal instance data. This way, the proxy instances can use all the features of the object they are a proxy to. Additionally, if there are changes to the subject, all proxies are automatically updated with the changes.

3.1.6.3 UML

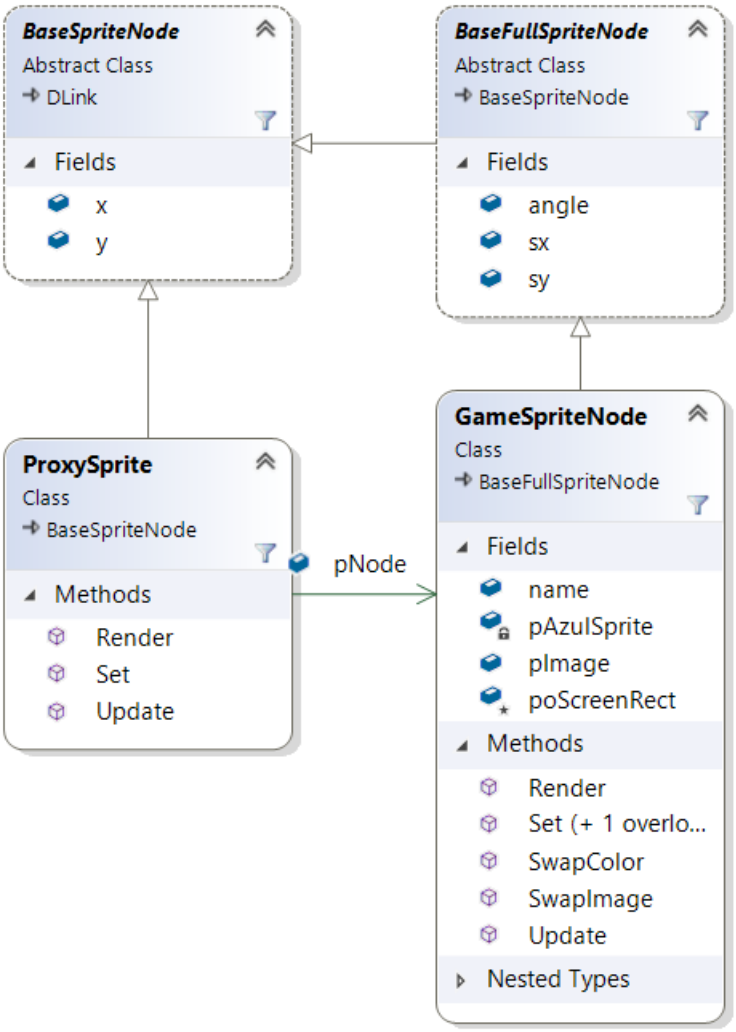

Figure 6: Proxy Pattern

3.1.6.4 Use in Game

The ProxySprite object contains a reference to a GameSpriteNode. The ProxySprite can still be treated as a BaseSpriteNode but does not have the complexity that comes with the GameSpriteNode. Each ProxySprite may have unique x and y coordinates but they do not require any of the other features of the full object.

The Update and Render methods of the ProxySprite, update and render the GameSprieteNode as needed. This allows for the need of only a single full sprite for each kind of game sprite on the screen, which is essentially “stamped” several times over each render.

3.2 LAYERS

3.2.1 Summary

By default, sprites appear in the order that they are rendered, so ones that are rendered later will be above those that were added earlier. For this project layers are used to better control the order on the screen, as well as allow for easy hiding and showing of groups of sprites.

3.2.2 Layer Details

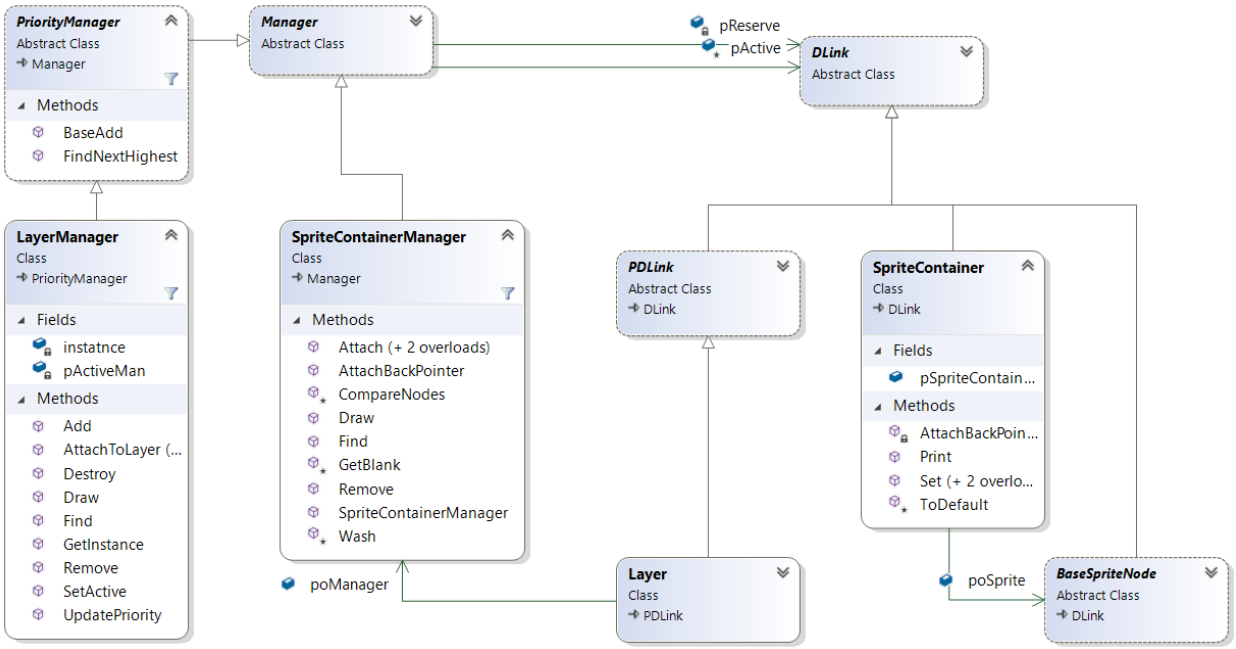

Figure 7: Layers

The LayerManager has a pool of Layer objects. It can be called upon to add new layers, attach a sprite to a layer and to draw all its layers. When adding new layers, each is given a priority and inserted into the active list according to that priority. This in turn determines the rendering order of each layer when the Draw function is called. This priority is managed using the PriortyManager and PDLink super classes.

Each Layer object contains a reference to a SpriteContainerManager, which it calls the Draw method of when instructed by the LayerManager.

The SpriteContainerManager is itself a pool of SpriteContainer objects, which ultimately point to sprites that are rendered for the layer.

Together this means that sprites can be added via the LayerManager to a specific Layer, where it will be rendered in order, along with the other sprites of its layer. The order of these layers can be updated as the program runs, or they can be configured to be hidden during runtime.

3.3 COLLISIONS

3.3.1 Summary

Many of the objects in the game trigger operations when they collide. For instance, when a missile hits an alien, a sound is played, the colliding objects are removed, the player's points go up and a splat animation occurs. This requires the use of the visitor pattern to handle the different operations and the observer pattern to trigger events related to the collisions.

3.3.2 Collision Objects

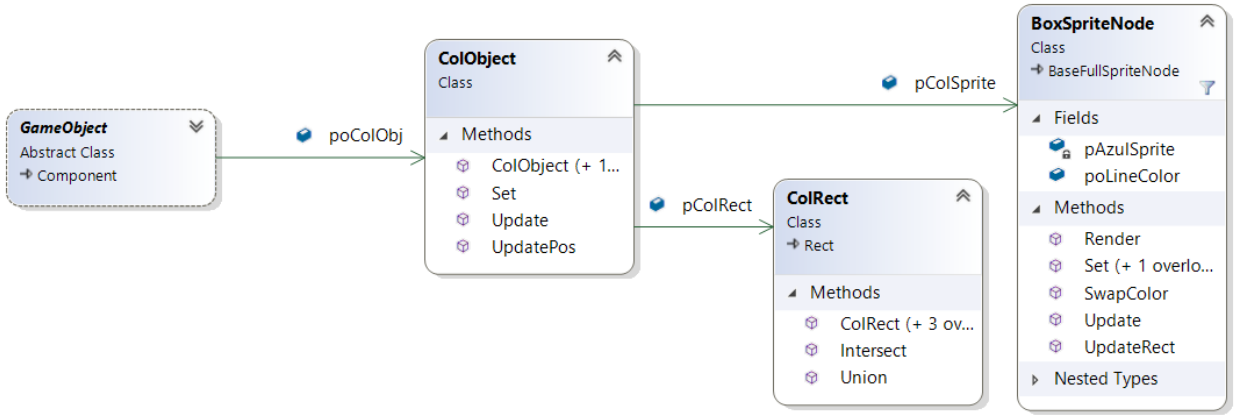

Figure 8: Collision Object

Every GameObject has a ColObject. This object is responsible for managing the collision space on the screen for the GameObject. The space is represented by the ColRect, owned by the ColObject. The rectangle can tell it is intersecting with any other rectangle with the Intersect method and can be expanded by adding a different rectangle with the Union method. The GameObject updates the ColObject as it moves around on the screen.

This object also contains a BoxSpriteNode, which is analogous to the GmaeSpriteNode but instead renders a box. There is a BoxSpriteManager just like the GameSpriteManager as well. Because of the above-mentioned layer system, these boxes can be toggled on and off to show the collision regions for each GameObject.

3.3.3 Collision Pairs

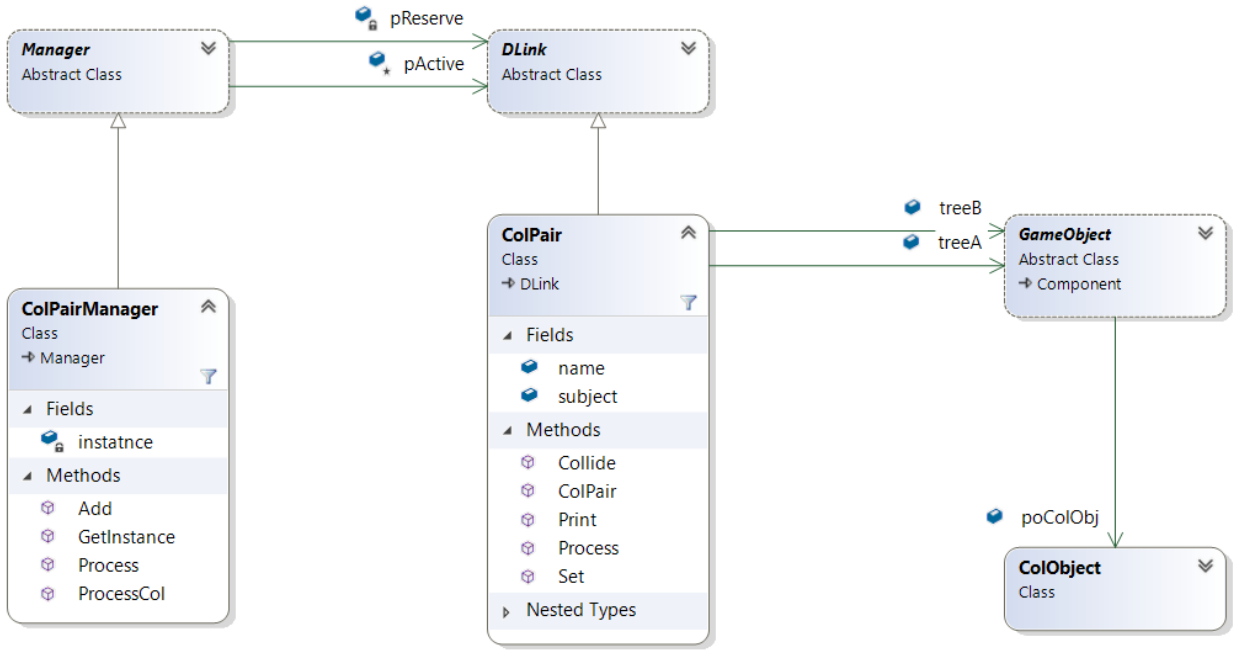

Figure 9: Collision Pair

A ColPair contains two GameObjects which should be roots of composite GameObject trees. These are then managed by a singleton object pool, the ColPairManager. Game objects are added to the manager creating a ColPair. When it is time to process collisions, the ColPairManager instances' Process method is called to run through all the ColPair objects using the visitor pattern as described below.

3.3.4 Problem: Extensible System of Interactions Between Many Object

3.3.4.1 Summary

Space invaders requires a robust collision system which can handle interactions between several types of objects including aliens, alien columns, missiles, shield grids and many more. Each new type has the potential to add n more interactions, where n is the current number of object types. Additionally, the structure of the objects may vary significantly. A simple switch statement structure could easily get out of hand, resulting in algorithms that are difficult to read and update. Instead an architecture where each interaction could be treated separately would be easier to maintain. For this purpose, I used the behavioral visitor pattern.

3.3.4.2 Visitor Pattern

The visitor pattern is a behavioral pattern that allows each interaction to be treated the same by the client code, inherently discriminated by the type of objects, and implemented by the participating objects. The client code in the visitor pattern only needs to recognize that an interaction between two objects is occurring. Then, the visitor pattern determines which operation is performed to handle the interaction. This results in these operations being able to be created without changing the structure of the objects or the overall structure of the interactions. For instance, an operation to handle the interaction between a missile and a shield, only requires the creation of the operation. The visitor pattern only needs to be updated to include the interaction, and the client does not need to be aware of any change at all.

3.3.4.3 UML

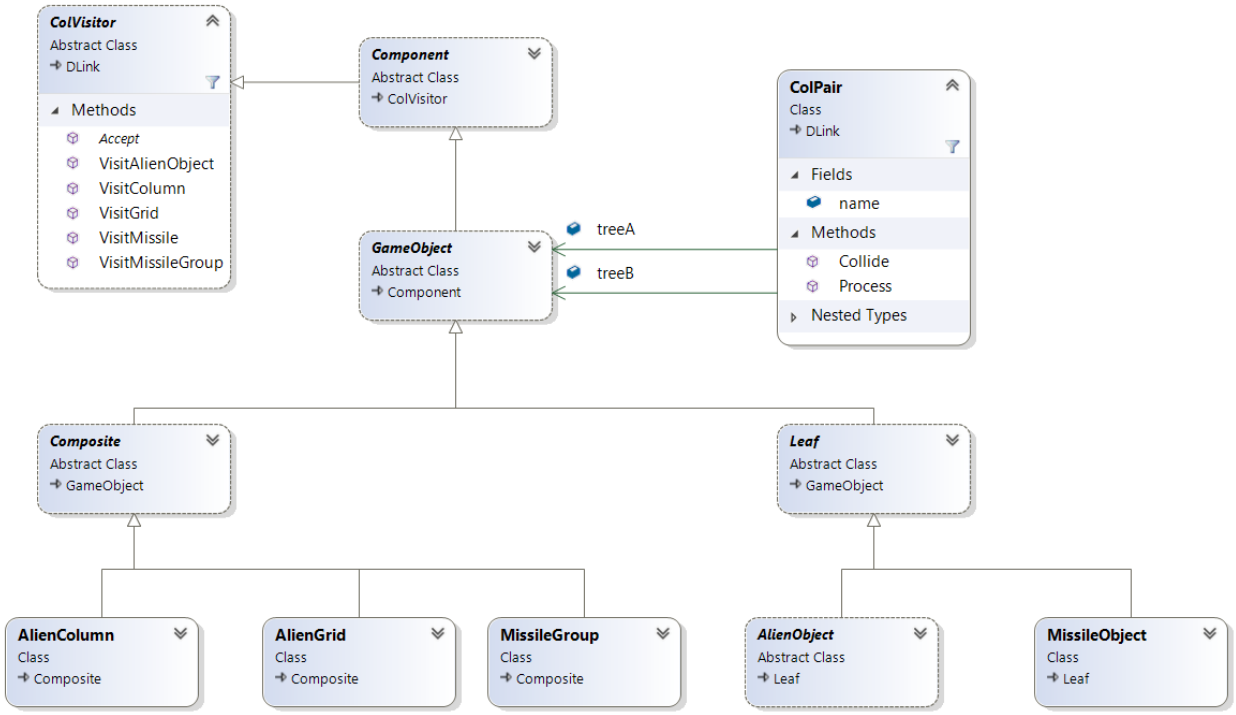

Figure 10: Visitor Pattern

3.3.4.4 Use in Game

The client, in this case the ColPair, can treat all implementations of the ColVisitor the same. In order to handle an interaction, it only needs to know about these is that they have an accept function. This function allows an implementation to accept another implementation. Beyond the client's scope, there should be one visit method for each class of object that needs to interact. The visit method represents an operation to be performed when visited by the class of object referenced by the visit method. Each object that needs to interact, can then implement all the visit methods referencing objects it can interact with. For instance, a WallTop, SieldBrick and AlienObject, all have operations for interacting with a MissileObject. With these visit methods in place, each implementation of the accept method simply calls the visit method referencing itself on the accepted object.

As stated above, this architecture removes the need for a complicated conditional structure, by instead using distinct method signatures and having those signatures only being used by the classes that are referenced by them. Each implementation of the pattern is then only responsible for calling that method signature in its Accept method and implementing the possible interaction operations in its visit methods.

For this implementation of the visitor pattern, GameObject inherits the abstract ColVisitor class. The GameObject is an element of the game which has the potential to represent just itself in the Leaf subclass or represent the composition of several other GameObject instances, as the Composite subclass. This leads to the need a way to implement interactions, agnostic of the structure of any individual GameObject. With the visitor pattern, the containing ColPair can process a collision by having any GameObject accept any other GameObject, without need for update.

To simplify the update process further, visit methods are all implemented in the base abstract ColVisitor class. These implementations will throw errors if they are called but they satisfy the contract for all concrete classes without requiring them to implement unnecessary visit methods, such as a ShieldBrick visiting a WallTop. When a visit method is required, a concrete class only needs to override the base class.

3.3.5 Problem: Triggering Actions

3.3.5.1 Summary

Many events need to occur in the game as a result of other events. When a missile hits the top wall, it must be removed, the explosion should happen, the flying sound should stop, and the ship should be allowed to fire again. However, only the collision objects would know that the collision happened, and it should not be their responsibility to perform these actions, nor should it have to change its code each time a new function needs to happen after a collision. This leads to the behavioral Observer pattern.

3.3.5.2 Observer Pattern

The observer pattern defines a one-to-many dependency between a single subject and many observers. Each time a new operation is required, a new observer can be created or instantiated, and tied to a subject. The subject code does not need to change for each addition, instead it only needs to call a notify method and then all observers will call their operations.

3.3.5.3 UML

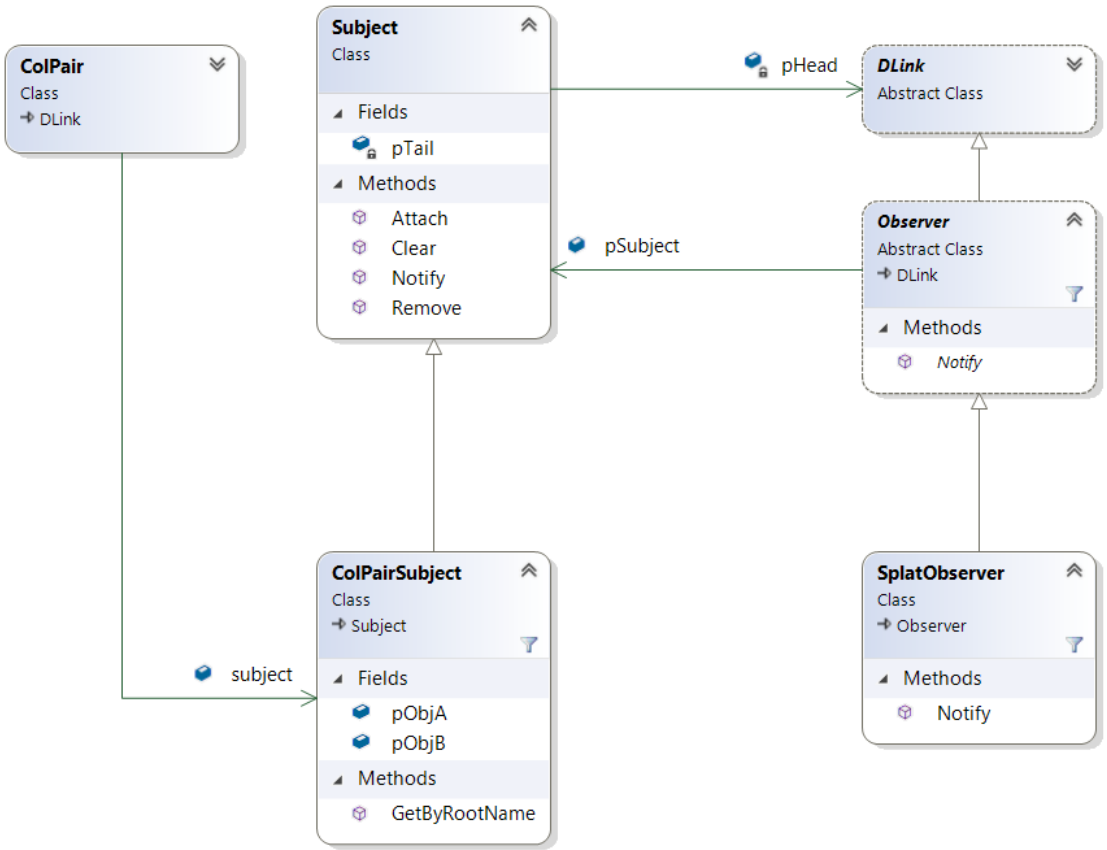

Figure 11: Observer Pattern

3.3.5.4 Use in Game

For the collision system each CollisionPair owns a ColPairSubject. Each subject has a list of Observer objects which are watching its state. When a collision occurs, the ColPairSubject calls its Notify method, which in turn, invokes the Notify method of all the attached Observer objects.

In the diagram, the SplatObserver is used as an example. It is responsible for playing a splat animation after a collision happens. An instance of this class is attached to the subject of each CollisionPair that triggers a splat, so that it gets notified of a collision.

It should also be noted that in this implementation, each observer also contains a reference to the subject it is observing. This allows the observer to consider the state that the subject is in when calling Notify without needing to adjust the signature of the Notify method.

3.4 GAME OBJECTS

3.4.1 Summary

Apart from text and some splat animations, everything drawn on the screen is a GameObject. These objects combine the collision system and graphics system into a single point from which the core elements can be manipulated. To achieve this goal, it must be possible to easily implement operations on single game objects as well as collections to create them efficiently.

3.4.2 Game Tree Manager

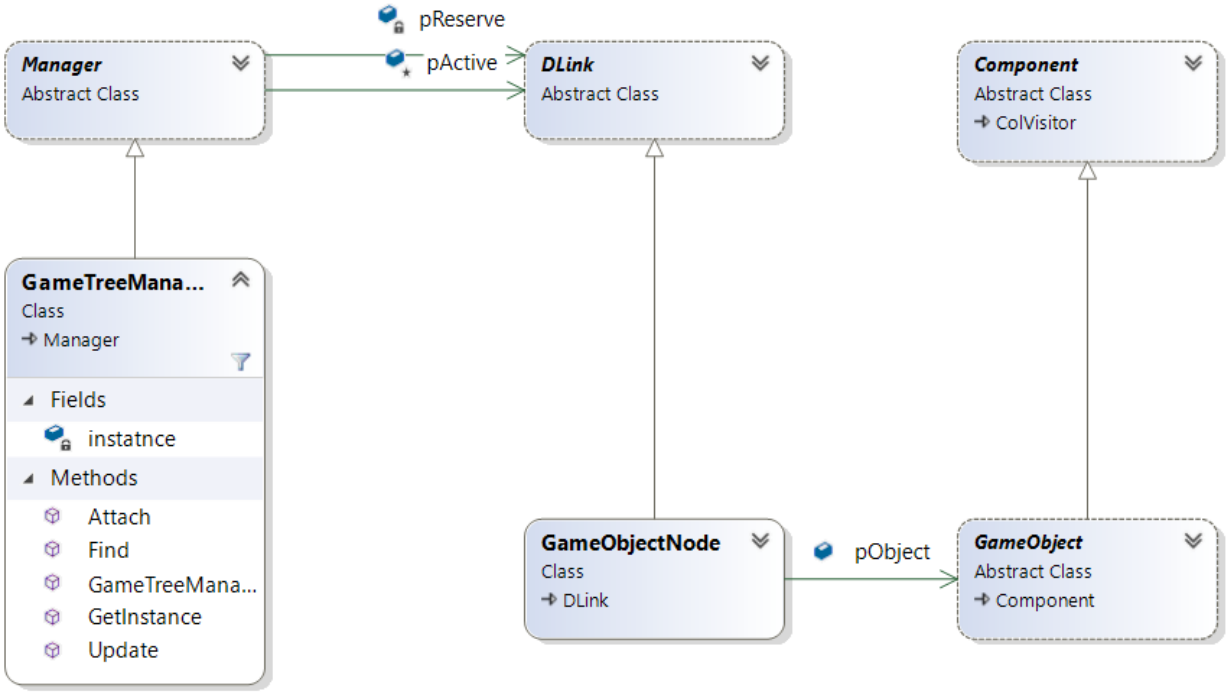

Figure 12: Game Tree Manager

All the GameObject instances are aggregated into trees using the composite pattern as described below. While providing the benefits described in that section, this also allows for a relatively small number of top-level objects to be managed at a broad scope. The GameTreeManager is a pseudo singleton object

pool which is responsible for these aggregations. This is the client to the iterator pattern in the section Problem: Performing Operations on Collections Regardless of Structure. The update method utilizes the iterator to call update on all the GameObject instances, using the attached GameObject trees as access points.

3.4.2.1 Singleton Factories

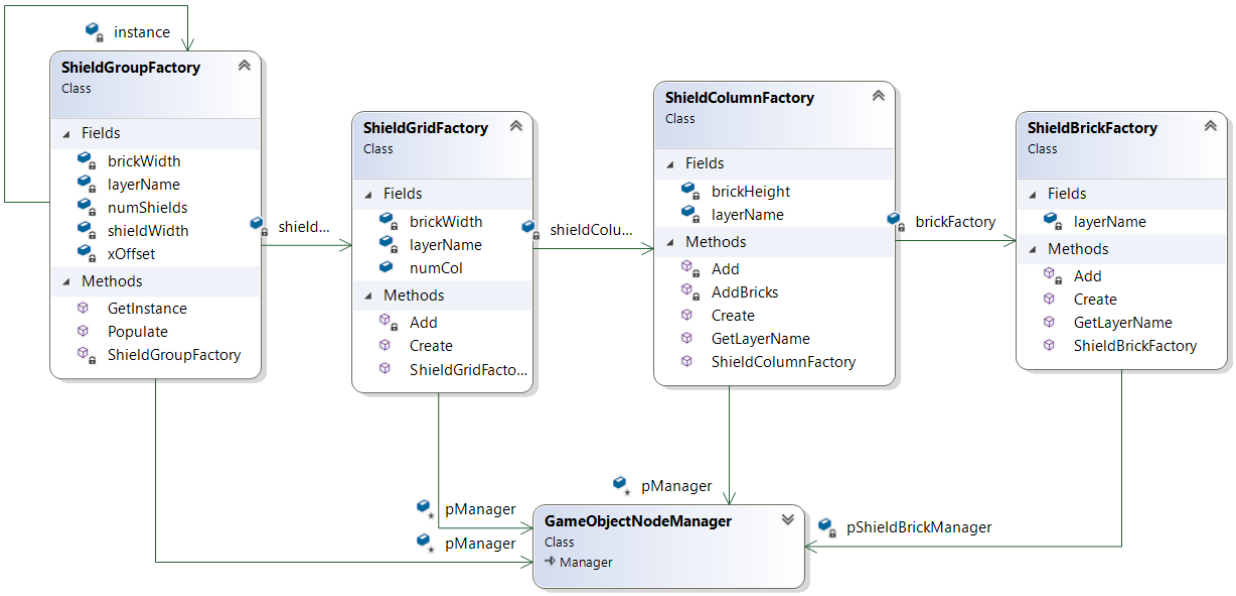

Figure 13: Singleton Factory

The ShieldGroupFactory and the AlienGridFactory are both singletons and factories. These factories have references to the factories for the lower levels of their respective object trees. This allows for the levels to have independent classes to implement their creation methods but for it to be possible for the client to construct the entire tree using a single Create method. Each factory has a reference to an object pool for creation in the form of a GameObjectNodeManager reference. These types of pools are better described below in Problem: Repeated Creation. Additionally, the singleton pattern provides global access to the factories responsible for creating the objects while ensuring there is only one instance of each factory and by extension one instance of the GameObjectNodeManager pool for each type of object that must be created.

3.4.3 Unique Objects

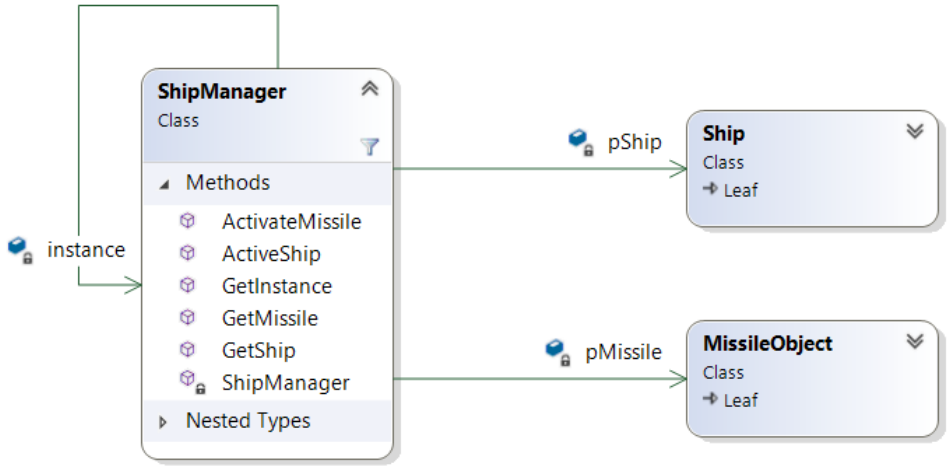

Figure 14: Unique Objects

Some game objects such as the Ship, the MissileObject and the Ufo are unique. This means that there is only ever one instance of them in the entire game. In these cases, a singleton manager is used but an object pool would add unnecessary overhead. Instead the manager always maintains a reference to the single objects and thought the global access to the manager, these objects can be manipulated, shown or hidden from the screen. In the case above, the ActiveShip and ActiveMissile methods are responsible for adding their respective GameObject instances to the appropriate GameObject tree and attaching their sprites to the appropriate layers.

3.4.4 Problem: Treating Single Instances and Collections of Objects Uniformly

3.4.4.1 Summary

Many of the objects in Space Invaders are naturally grouped such as aliens in a column and grid. Many operations have to be performed on both the groups and the individual objects. However, it would be cumbersome to need to implement unique methods for each level of grouping. This prompts the use of the structural composite pattern.

3.4.4.2 Composite Pattern

The composite pattern allows objects to be represented as a tree structure where each element has the potential to contain none to many other elements of the same type. Each of these components can be treated uniformly by the invoking clients. This limits the need for unique implementations between levels.

3.4.4.3 UML

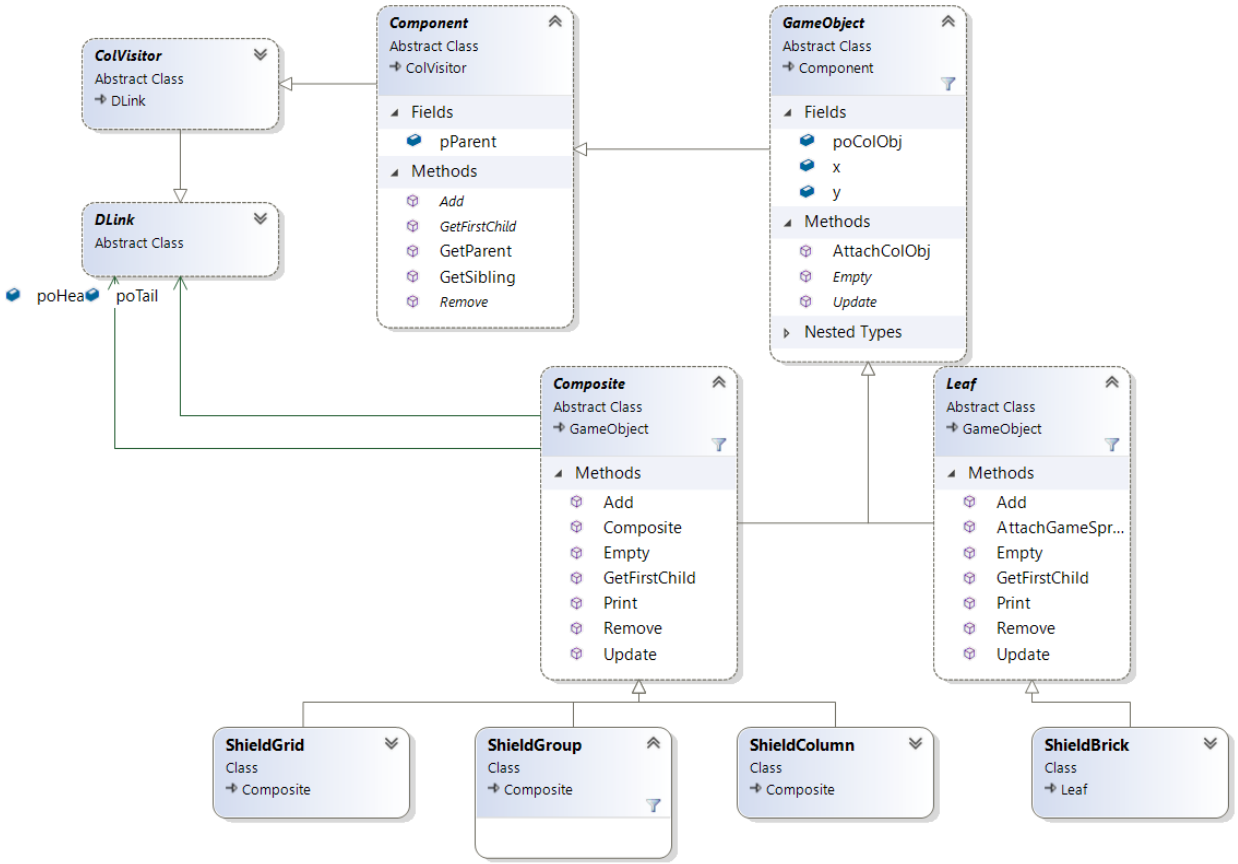

Figure 15: Composite Pattern

3.4.4.4 Use in Game

All pieces of the composite pattern inherit from an abstract class called Component. This class along with the GameObject class serve as the contracts that all subclasses must adhere to. The Composite class represents a GameObject which can be given more instances of the Component Class. Because of this relationship, those instances can either be of the Leaf type or the Composite type, meaning they themselves could contain more Component objects.

Just like Composite objects, Leaf objects are instances of the Component class, but they don't contain any other instances. These generally have GameSpriteNode object nodes associated with them and therefore represent what is seen on the screen. The Composite objects then represent the containing boxes around collections of Leaf objects. This allows something like a ShieldGrid to be composed of many ShieldGroup objects which in turn are collections of ShieldColumn objects and so on. However, because they are all instances of Component and GameObject, they can all be called upon at any level with something like the Update method without the client being aware of this composition.

3.4.5 Problem: Performing Operations on Collections Regardless of Structure

3.4.5.1 Summary

Some operations such as updating sprite and collision object positions need to be able to be performed on large groups of GameObject objects. However, the client invoking this method should not be

responsible for knowing their composite structure and implementing an iteration over them. Instead the iteration should be encapsulated in a behavioral iterator pattern.

3.4.5.2 Iterator Pattern

The iterator pattern allows a client the ability to access all elements of a collection one at a time without requiring knowledge of the structure. An abstract Iterator class can be implemented by one or more concrete iterators. There needs to be one concrete iterator for each aggregation that needs to be iterated over. With this, a client with access to the iterator for a given aggregate can use the simplified contract of the Iterator class to step sequentially though all the elements. Meanwhile, the concreate iterator implements the way in which the steps through the collection are performed.

3.4.5.3 UML

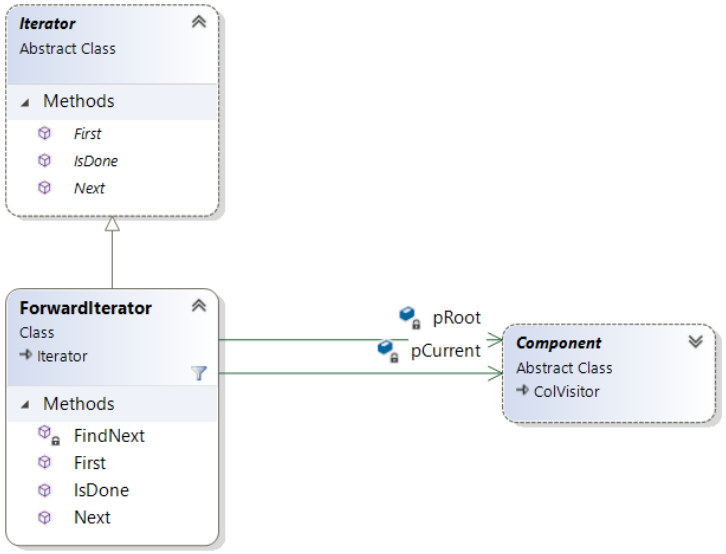

Figure 16: Iterator Pattern

3.4.5.4 Use in Game

Currently in the game there is only one implementation of the Iterator class, the ForwardIterator. The ForwardIterator is responsible for iterating over Component objects. As explained above, Component objects can be references to one or many GameObject instances in a tree structure. However, when an operation needs to be performed on one of these tree structures such as updating positions, it would be cumbersome to have the invoking class implement the iteration. The ForwardIterator manages the sequential stepping through of the trees but returns objects as if walking a list. This also allows for the order in which elements of the GameObject trees to be managed by the ForwardIterator.

3.4.6 Problem: Repeated Creation

3.4.6.1 Summary

Objects like AlienObject must be created frequently as there are 55 of them for each level. Furthermore, these must be structured into columns and then a grid. The creation process is complex for this as well as for shields and bombs, and therefore there should be the use of the creational factory pattern to encapsulate these operations.

3.4.6.2 Factory Pattern

The factory pattern utilizes a Factory class which is responsible for creating objects with complex instantiation. For instance, if an object requires repetitive operations before it can be created, a Factory class can implement those operations and allow the client to provide minimal input to create the object. This also allows for different types of objects which share a common interface to be created from the same Factory object.

3.4.6.3 UML

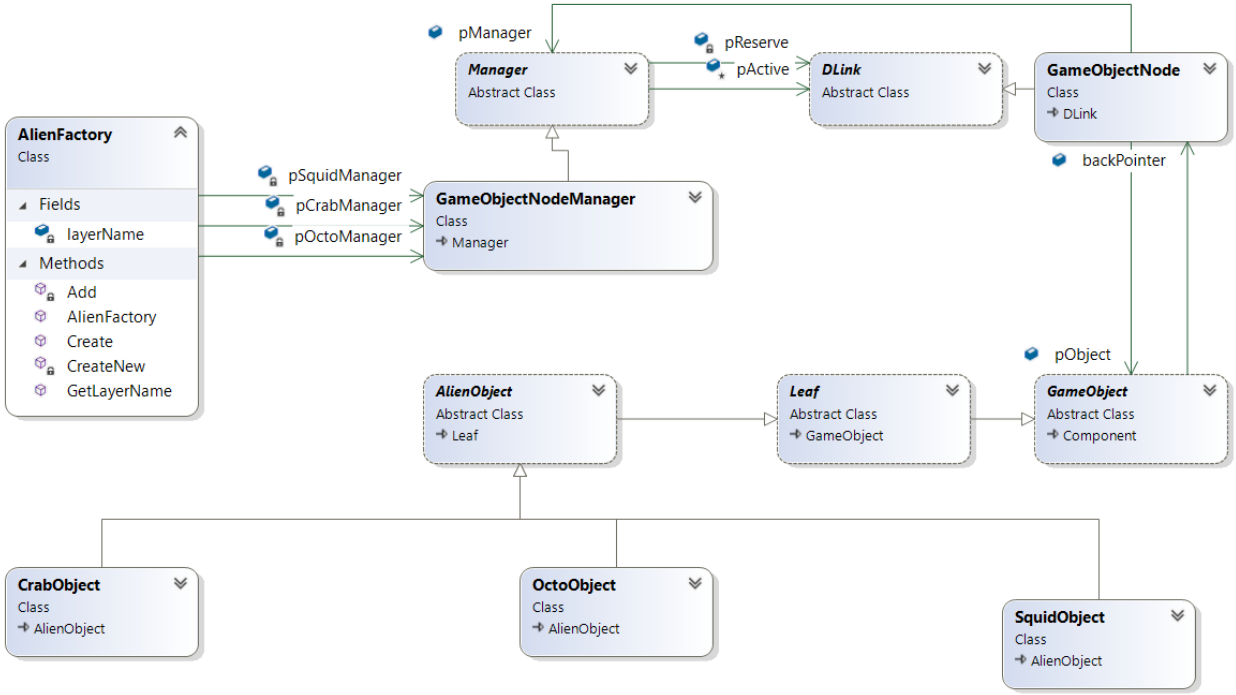

Figure 17: Factory Pattern

3.4.6.4 Use in Game

Factories are used in this project in several cases including for bomb creation, shield creation and alien creation. All the factories have a Create function. In the above case, the AlienFactoy implementation of the Create function returns an AlienObject. The Create function handles the process of determining which type of AlienObject to create based on parameters passed into the method. This means the client does not need knowledge of the different types of AlienObject instances nor does it have to know how the type is decided beyond what is passed into the Create method.

For this project, the AlienFactory as well as many other factories, use instances of the GameObjectNodeManager. Each instance is a static object pool which manages the instances of the objects created by the factory. This, combined with the back pointers from the GameObject class to the Manager class, allows for every instance created by the factory to be recycled and used again later. This is very important because for GameObject classes like aliens, many are created each time the level changes or the player loses. Having these object pools, means that no more than the maximum number of aliens on the screen at time, will ever be created. The same applies for Bomb objects, ShieldBrick objects and many more.

When the Create method is called for the AlienFactory, the CreateNew function is invoked which prompts the factory to check the reserve list of the appropriate manager and return a recycled object if there is one, or causes the manager to grow, and provide a newly created object. However, because the factory pattern abstracts the creation process, the client does not need any knowledge of this process.

3.5 EVENTS

3.5.1 Summary

For space invaders many of the operations need to be triggered relative to time. Aliens cycle though a marching animation about once a second at the start of each level, explosion splats fade away after a short period of time, and bombs drop at a specified interval. To accomplish this, the game requires a system for managing timed events and for having those events trigger operations.

3.5.2 Timer Manager

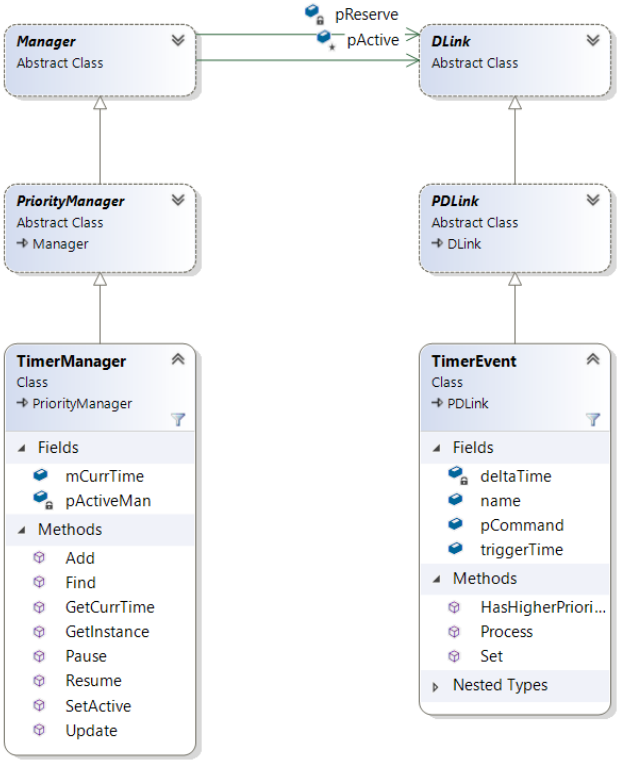

Figure 18:Timer Manager

The timer manager is another pseudo singleton object pool. It contains a linked list of TimerEvents. These events are created with a delta time and a Command. The delta time is the amount of time between when the TimerEvent was added to the TimerManager with the Add function, and when the TimerEvent should call its Process method. This process method is tied to the Command, which is discussed further in Problem: Parameterize Operations. This object pool is slightly unique, in that, it processes the active events, meaning that it uses each of the active nodes once, before recycling it back to the reserve list.

The client to the TimerManager calls its Update function regularly with the current game time, which prompts it to check if any TimerEvent objects should be triggered. To simplify this checking process, TimerManager is a PriorityManager, like the LayerManager. The priority that the PDLink TimerEvent objects are created with are their trigger time attributes. These are computed by adding the delta time to the current system time to create the game time that the event should be triggered. The TimerEvent objects are added to the PriorityManager in order of when they should be triggered. This means that when checking for what events should trigger, the TimerManager only needs to check the front of its active list and walk that list until it finds a TimerEvent with a trigger time later than the current game time. While walking to that point it can trigger every event on the way and be sure that it triggered all relevant events.

3.5.3 Problem: Parameterize Operations

3.5.3.1 Summary

To trigger operations based on time, a system like the above TimerManager is needed. However, it would not be feasible to implement unique TimerEvent classes for each type of operation needed to be triggered in this manner. The TimerManager object pool requires homogeneous objects and re-implementing the TimerEvent code for each operation would be cumbersome. The project needs a method to encapsulate these operations and trigger them based on time and this method is the behavioral command pattern

3.5.3.2 Command Pattern

The command pattern refers to the wrapping of an operation in an object. This object then can be used as parameters to other object which give the operation attributes. This can be used to allow for undo and redo systems or in the case of this project, events ordered by time. The Command object is only responsible for triggering the desired call, while the objects it is passed to, can trigger the Command Execute method as needed.

3.5.3.3 UML

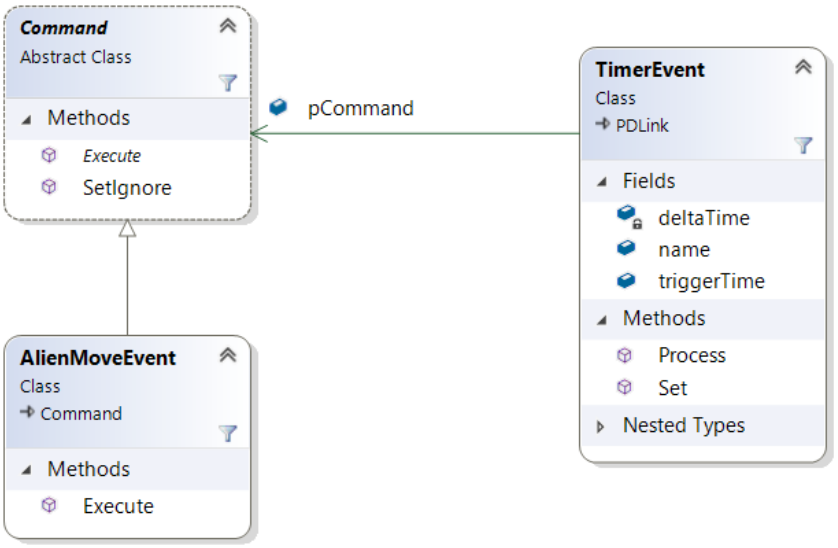

Figure 19: Command Pattern

3.5.3.4 Use in Game

For this game, the abstract Command class defines the interface through which the encapsulated operations are triggered. In the above example, the AlienMoveEvent is a Command that advances the aliens on the screen. This advancement is triggered by the Execute method that it implements. This command is passed into the TimerEvent object via the Set method along with the delta time for the event to trigger. This allows for the AlienMoveEvent and the many other Command classes to be agnostic of the concept of trigger time or the TimerManager system. Also, the TimerEvent is agnostic of the operations being performed so as many implementations of the Command class can be created as needed without needing to update the TimerEvent.

3.6 SCENES

3.6.1 Summary

Outside of the core gameplay, there are several states the game cycles through including the select screen, the game over screen and the paly screen. In this program these are divided into SceneState classes which control their own instances of the game systems.

3.6.2 Pseudo Singletons

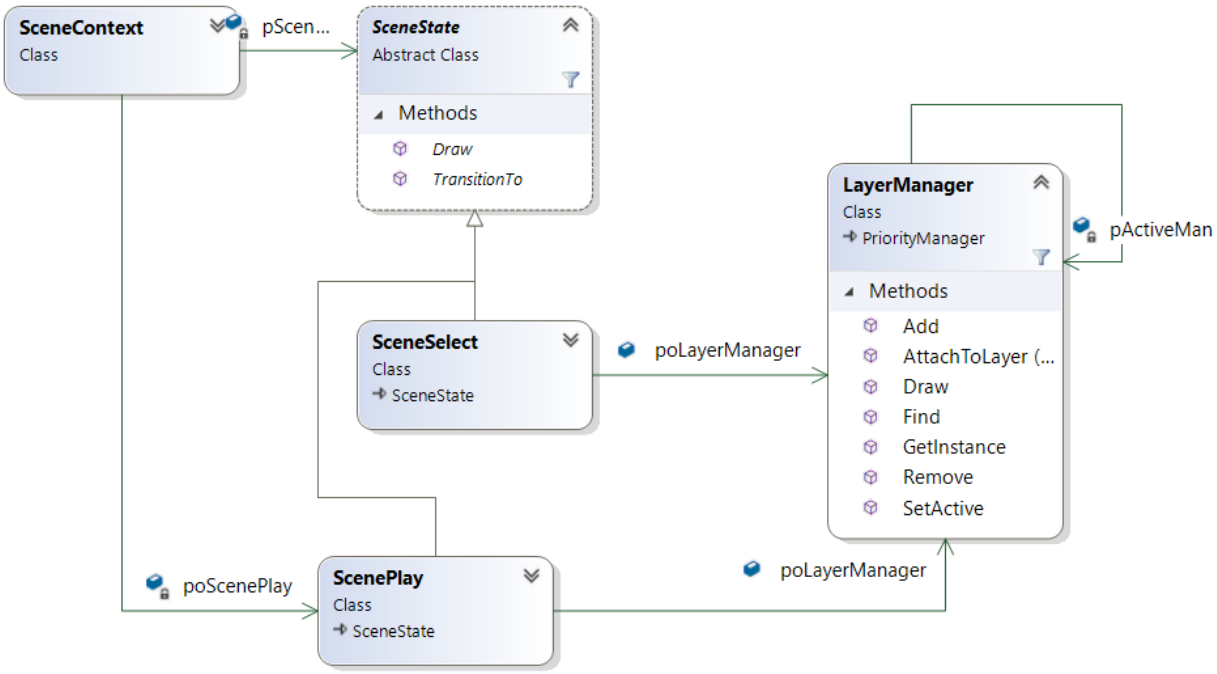

Figure 20: Pseudo Singletons

In earlier sections the concept of the pseudo singleton was mentioned. These exist in the program because rather than requiring unique behaviors of the systems for each state of the game, it is far more maintainable to give each state unique instances of the systems which they can utilizes as needed. The above example shows that the SecenSelect class and the ScenePlay class reference unique instances of the LayerManager objects. This means that when the Draw method is called on them, they can us the Draw method of their unique LayerManager instance to populate the screen rather than needing to tear

down and repopulate the layers according to the scene. This is used for many other systems including the sound system, event system and collision system.

In some cases, scenes can even share Manager instances to represent super and sub states such as when the OverScene displays the text “Game Over” overtop the layers of the ScenePlay state. The OverScene has it own unique TimerManager instance so that it does not continue the core gameplay loop but still can time the drawing and removal of the text.

These pseudo singleton classes work by holding a self-referencing pointer to the current active instance. Each scene is then free to call the static SetActive method of the Manager implementation with its individual instance of the class. The manager then will return the new active instance when the static GetInstance method is called, similarly to the full singleton manager classes. This means the active instance is globally accessible to all the distributed systems, and those systems can be agnostic of the state change.

3.6.3 Problem: Large Changes to Game Loop Behavior

3.6.3.1 Summary

As the game progresses, certain events cause major changes in behavior to the core game loop. For instance, when the player loses all their lives, they are returned to the select screen, which itself could lead back to gameplay when the player selects to start. This could be handled by if and switch statements, however, because of the amount of changes to behavior required and the distributed nature of the game control, it would be extremely cumbersome to add new states. Instead a way of encapsulating the behavior of different states is used via the behavioral state pattern.

3.6.3.2 State Pattern

The state pattern represents different states of operation as classes. The top-level State class defines operations all its subclasses can perform but leaves the implementation of those operations to the individual subclasses. The client then only requires knowledge of when to trigger a state change. A Context class can be used to contain the instances of the various states, and to handle the switches between each object. The client then can call the operations of State instance provided by the Context class without knowledge of the different implementations.

3.6.3.3 UML

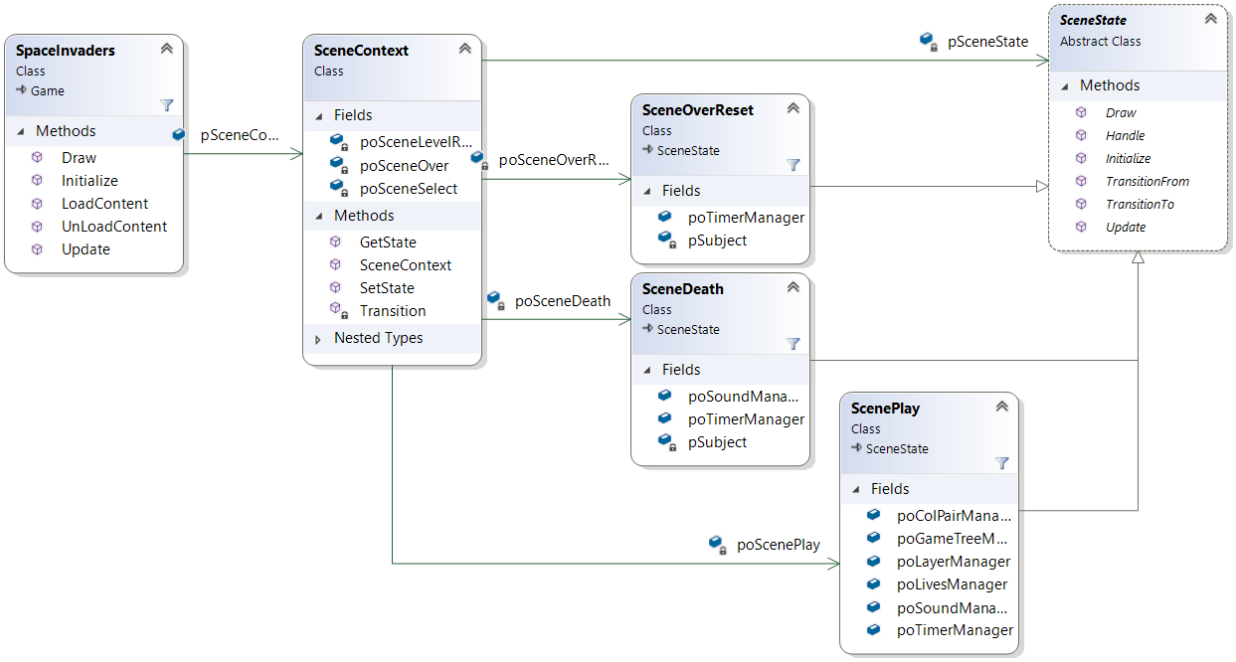

Figure 21: State Pattern

3.6.3.4 Use in Game

Scenes in the game make use of this state pattern. The client Game class has logic to trigger Draw, Initialize, and Update operations. It also has a reference to a SceneContext. This SceneContext holds references to several implementations of the SceneState class. This includes the SceneOverReset, SceenDeath, and ScenePlay. Each of these implement the SceneState Draw, Handle, Initilize, TransitionTo, TranisitonFrom an Update methods. These are the methods with which the client and the SceneContext control the SceneState implementations. For instance, the ScenePlay class, runs through the collision system, the layer system and several others during every update while the SceneOverReset simply uses the timer system to perform reset operations.

Whenever the Game needs to call an operation on the current state, it first calls the SceneContext object's GetState method which returns the current active SceneState instance.

Some states need to perform actions directly upon being transition to or transitioned from, such as the SceneDeath which triggers the ship destruction animation and sound as soon as it is entered. For this purpose, the SceneState class also has TransitionTo and TransitionFrom events. When the SetState method is called on the SceneContext object, the current active state's TransitionFrom method is carried out and then the active scene is switched and that state's TransitionTo is called.

4 CONCLUSIONS

4.1 COMMENTS ON FINAL PRODUCT

The development of this project was carried out over the course of an 11-week class lead by Professor Keenan of DePaul University. While there remain many opportunities for improvement as discussed in Opportunities for Improvement below, the product overall provides a playable re-creation of Space Invaders with nearly all components represented.

The restrictions imposed along with the heavy use of Object-Oriented Design resulted in robust and relatively maintainable game engine. This was accomplished despite spanning over 100 classes and containing thousands of lines of code. Over the course of carrying out this project and creating this documentation, I was able to highlight and greatly advance my understanding of Object-Oriented Design principals and traditional design patterns. The large scope of this project allowed for heavy realistic practice building software architecture. I feel the product and this document show my grasp of many complex software architecture concepts as well as an ability to implement their uses.

4.2 OPPORTUNITIES FOR IMPROVEMENT

4.2.1 Generic observers and commands

Observer and Command implementations make up a considerable portion of the classes in this project. Later in the project I developed a few implementations of Observer and Command that could be used across several instances. For example, when aliens, missiles, bombs, the ship or a UFO is destroyed, an explosion sprite is shown on the screen which then decays after about a second. The SplatObserver accepts an argument for the name of the splat sprite. It then polls the GameSpriteManager for that sprite and displays it on the screen. Then a RemoveFromLayer command is added to the TimerManager that points to the sprite and removes it after a specified decay time. This reduced the need for individual implementations for each type of splat, thus reducing repetition and complexity in the project.

A few other cases could use a similar pattern such as the movement command implementations, for which there are unique cases for aliens, bombs, the UFO and missiles. These could instead use a single MoveEvent that invokes a Move command on the objects they are moving.

4.2.2 Singleton Bomb Factory

The ShieldGroupFactroy and AlienGridFactory classes are singletons allowing for global access and the guarantee that there is only ever one instantiation of them. In the case of the BombFactory class, it is created by the DropBombEvent object. The DropBombEvent is only created once but this is more by coincidence than by architectural design. It is quite possible that another class or instance of the DropBombEvent could eventually require access to the BombFactory. As the project stands today, this would require another instance of the BombFactory to be created for each independent use. This could be mitigated by making the BombFactory a singleton.

4.2.3 Naming and Comments

As the project has evolved, many naming conventions have developed while others have become obsolete. The MissileObject is an example of this as the postfix "Object" is not used in most of the other GameObject implementations. This along with many other class names could be combed through to be improved.

Additionally, with the crunch time during the final weeks of the project, comments have not been well maintained or added. Many comments need to be updated or added to ease maintenance and readbility of the code.

4.2.4 More Clear Separations of Responsibilities

With vast number of features covered in this project, some specific interactions were a low enough level that a clear organization was not apparent. For instance, while top level GameObject implementations such as the ShieldGroup or WallGroup are responsible only for reacting to collisions, the AlienGrid has knowledge of the direction it is marching and implements the method for moving the grid downwards. For this and a few other cases, commands, observers and/or managers could be used more in order to relegate these responsibilities in a more organized manner.

4.2.5 Additional State Pattern Use

While developing a few more potential uses for the state pattern emerged. The UfoManager handles the launching of the UFO GameObject, similarly to how the ShipManager handles the launching of the MissileObject. However, the ShipManager utilizes the state pattern to handle the launch command, in which the MissileFlying state does not trigger the missile to fire but the ShipReady state does. This simplifies clean up of the screen if a missile is still present and handling of the sound that plays for the time the missile spends on the screen. This is done without the need of any if statements to handle the states. The UFO shares many similar behaviors such as launch conditions, clean up, and sound playing, which could benefit similarly from the state pattern.

4.2.6 Details on Lower Level Components

There are several other components to this program such as the sound and input systems. However, time constraints prevented details on those systems from being added to this document. This document could be improved later by adding those elements along with the patterns they re-used or implemented.

4.2.7 Alternative Modes and Debugging Features

It is possible in this project to show and hide the bounding boxes for the collision objects, however, there are several other opportunities for easier display and debugging. For instance, when checking behavior for the UFO being destroyed, I have tended to adjust the y position of the UFO launch point. This makes it easier for me to hit the UFO and test its destruction. A similar feature to the bounding boxes could be added that allows for a hidden way to move the UFO launch point. This would reduce the need for code changes when debugging

4.2.8 Visual Polishing

Several visual elements of the overall game are not perfectly re-created. In the classic game, when the UFO is destroyed, along with the explosion animation, the points earned from the UFO are also

displayed. Additionally, the original game used color filters to color the sprites on screen based on their y positions. Time constrains put these features out of scope but in the future, these could be added.

4.2.9 2-Player Mode

Again, due to time constraints, the ability for two players in one round of the game was not achieved. In the future, this feature could be added by utilizing the scene system and displaying alternating scenes for the players' turns.