# Space Invaders Design Documentation

William “BJ” Blair

03/20/2024

# Contents

# Introduction

During the quarter of SE 456 at DePaul, students were tasked with a final project of re-creating the arcade classic Space Invaders using modern, object-oriented software principles and design patterns. The project was developed in C# using the custom Azul framework for low-level graphics, windowing, and input, while IrrKlang was used for audio. For the code architecture, at a high level, the project contains a top-level SpaceInvaders class which inherits from the Azul.Game parent class, and contains Initialize, LoadContent, Update, Draw, and UnloadContent methods, which Azul uses as part of the framework. Within the SpaceInvaders class there are three main scene contexts which each implement their own versions of these methods for the different gameplay stages, and switches between the three as necessary. These scene contexts are the Select scene for the title and game start, the Play scene for the main gameplay, and the Over scene for the game over screen when the player loses all of their ship lives. The details of the design patterns used internally for each scene's logic are presented in the remainder of this document.

# Iterator

**Problem**

Within our game there are multiple methods used to store collections and groups of objects. Some examples of these groups include aliens, sprites, textures, images, game objects, and more. We frequently need to march through each item in a given group one at a time, applying operations or different logic at each step. Depending on the container used, the process for moving between elements is different. Ideally, we would like our code which uses these object groups to be agnostic to which type of container they are stored in, and have a uniform way for walking through the data – the logic applied to each data item shouldn't care what data structure is used to store them.

**Iterator Pattern**

To abstract the element operations and the movement between elements, we can use the iterator pattern. This allows us to step through a group of objects using a standardized interface, regardless of how the elements are grouped and stored. The client using the iterator is given a simplified traversal method and can focus instead on applying operations to the group data. An additional benefit of using the iterator pattern is that you can swap out the back-end container/storage method without changing the client front end, since the iteration method stays the same. The iterator pattern works by having the storage container create an iterator object, which provides a set of abstract methods for stepping between elements, accessing the current element, and checking if there are any elements left. The client then requests this iterator object from the container it wishes to access and calls these methods for moving through the group.

**Space Invaders Implementation**

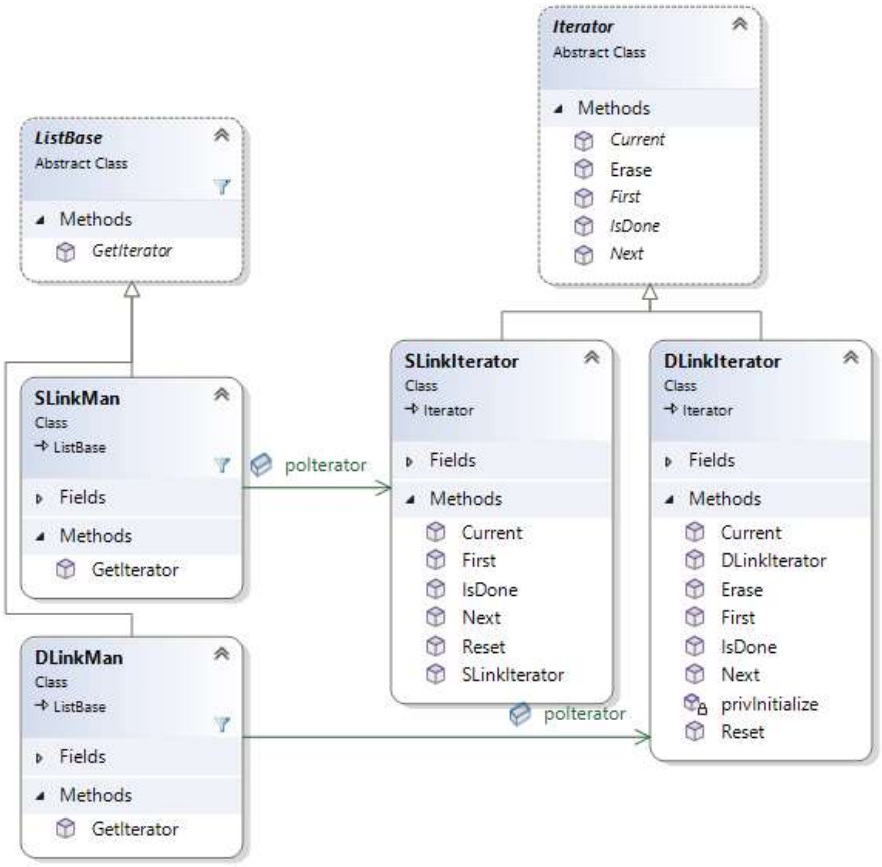


*Figure 1: Single/Double Linked List Iterator UML*

Within Space Invaders, the iterator pattern is used in a couple different locations. One example is the SlinkMan and DLinkMan classes, as shown in Figure 1. The ListBase and ManBase classes use an iterator to abstract the use of either a single or double linked list (SlinkMan and DLinkMan). The Iterator abstract class requires the method First to set the iterator object to the beginning of the group, Next to step forward in the list by one item, Current to access the current entry in the list, and IsDone to check if there are any more elements remaining to step through. The client (ManBase) obtains an iterator object to use on the list by calling GetIterator, which is agnostic to which list implementation is underlying. In ManBase, this is used to step through its reserve and active object lists. Similarlay, the Iterator pattern is implemented to access groups of Components. The IteratorCompositeBase is the abstract iterator parent class, while the IteratorForwardComposite and IteratorReverseComposite classes are concrete implementations used to traverse a Composite tree in both forward and backwards directions. Without the iterator pattern, this would be a complicated and messy process intertwined with the logic we are trying to apply on Composite objects.

# Strategy

**Problem**

Each of the aliens in Space Invaders has the ability to drop bombs towards the player ship. There are three varieties of bombs which display unique behavior; however, they all share the fact that they fall towards the player, hit the shield/ship/player and explode/are destroyed, and are animated. The general behavior of each bomb type is the same, but there are specifics to each one. We would like to avoid code duplication for shared behavior/logic between the bombs while still maintaining their individuality. We can abstract and forward this unique behavior outside of the shared bomb logic by using the strategy design pattern.

**Strategy Pattern**

Using the strategy pattern, a client can abstract an "algorithm" to store specific logic and behavior in an object, which is set at runtime. The shared logic and data is stored in a client class while different implementations of a given algorithm can be created and swapped in and out as desired. This eliminates the need to copy shared code between different implementations and also lets us change specific behavior/which algorithm is used on the fly. The pattern works by defining an abstract strategy class or method with a standardized algorithm interface, which is implemented by concrete strategy implementations. The client then contains one of these implementations as an object, which forwards the call to the specific algorithm to said object. The strategy object can also be changed and set when desired as previously mentioned.

**Space Invaders Implementation**

The strategy pattern is used in the Space Invaders implementation to implement the three different alien bomb behaviors, whose UML is shown in Figure 2. The abstract strategy interface is named FallStrategy, which dictates the methods Fall and Reset. The Fall method in the concrete implementations (FallStraight, FallDagger, and FallZigZag) contains the logic for the unique way each of the bomb types behave as they move down the screen. There is then a single Bomb class, which stores a specific FallStrategy in its pStrategy member. The Bomb class calls Fall on the pStrategy member during its Update method to forward this unique behavior to the strategy object. The game logic then to randomly have aliens drop bombs only needs create a Bomb instance and provide it which strategy to use for falling.

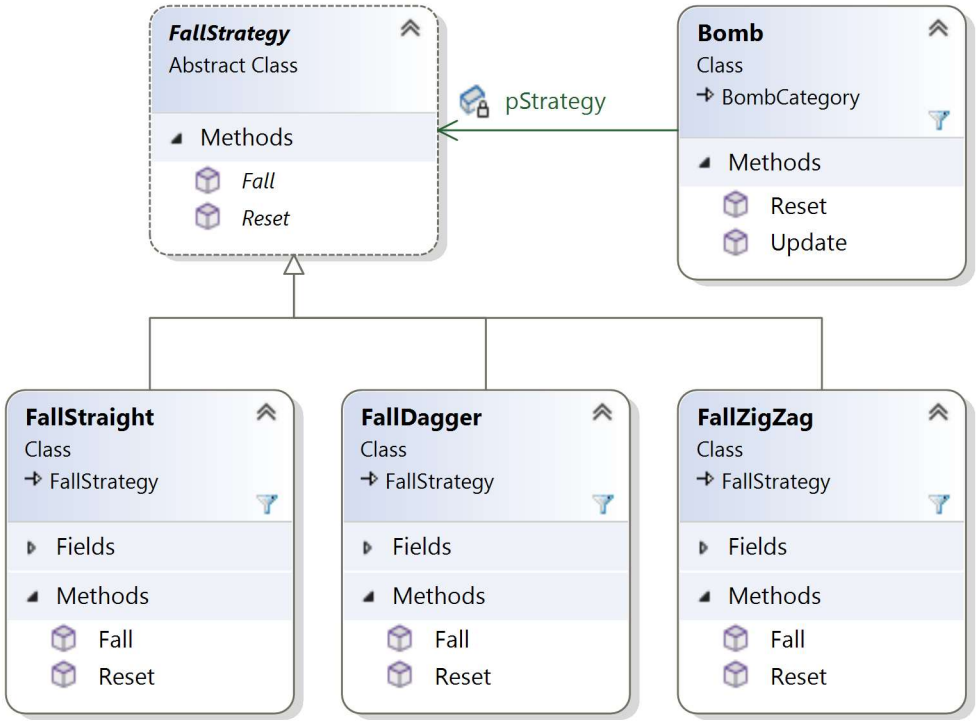


*Figure 2: Bomb Fall Strategy UML*

# Factory

**Problem**

In our Space Invaders implementation, we need to create large groups of similar but slightly different objects with a shared parent class – namely, the 55 alien enemies. There are three different alien types: Squid, Crab, and Octopus. These are all subclasses of the AlienBase parent class, which in turn is a Leaf and then GameObject. We could manually create each alien instance, but that would take a lot of code and typing and is prone to errors. Additionally, what if we want to change something about how an alien is instantiated, or any post-processing applied after creation? We would have to go back and modify 55 different object creations. Instead, we could wrap the creation of the different alien types in a factory.

**Factory Pattern**

A Factory in design pattern terms is used to instantiate objects of a shared class type without specifying specific subclasses. The creation of the specific object is done within the factory, which is supplied with the type object to create along with any other information desired. This allows us to change internally how and which subclasses are created and initialized in a single, internal location (the factory). We can also perform additional steps besides object creation without duplicating code for each instance.

**Space Invaders Implementation**

For the enemy aliens, we define an AlienFactory class to perform alien initialization. Its UML is shown in Figure 3. The AlienFactory Create method returns a generic GameObject and internally creates an AlienCrab, AlienOctopus, AlienSquid, AlienColumn, or AlienGrid subclass, all of which make up our group of different aliens and their containers. It additionally activates the sprite and collision box for each object. In the main game, the Play scene creates an AlienFactory, then, looping over each alien position, uses the factory to create the alien grid, columns which go inside the grid, and the aliens which go inside each column. By using the factory we save a lot of typing and extra code instead of going through the creation and initialization of each alien one-by-one.

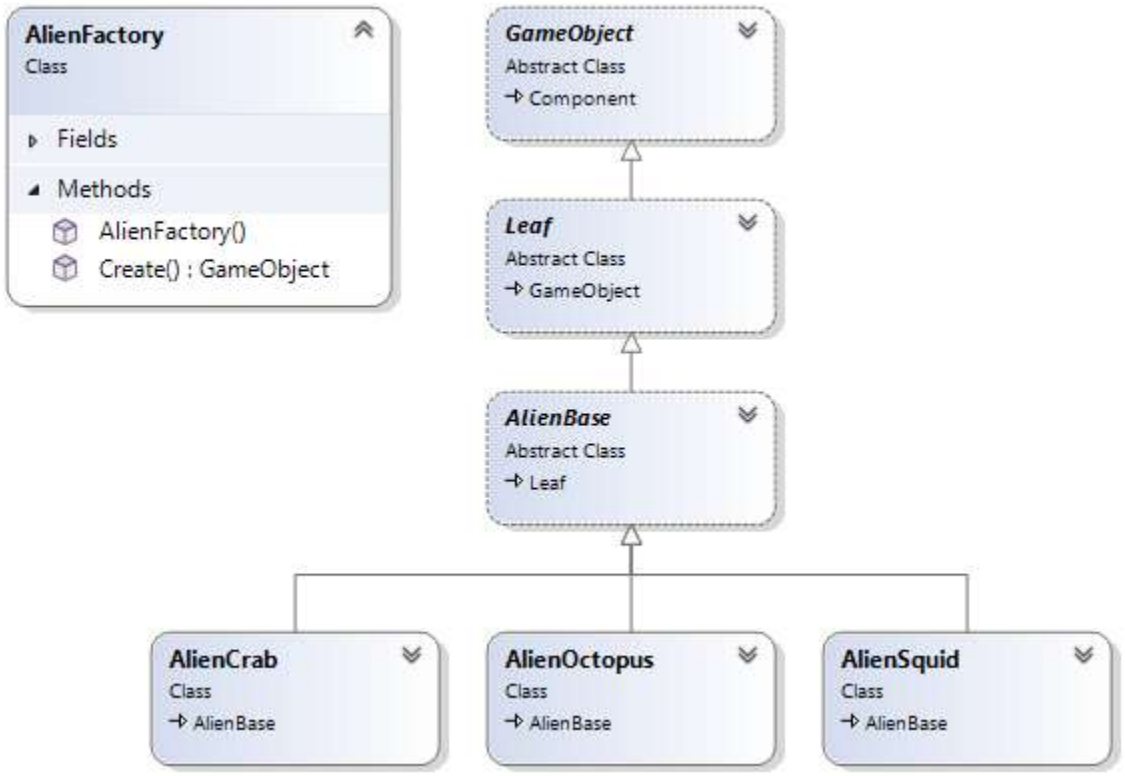


*Figure 3: Alien Factory UML*

# Proxy

**Problem**

In Space Invaders, the player faces a large group of moving, animated aliens which constantly head downwards on screen towards the protagonist's ship. Their movements and animations are all synchronized such that each alien's position and animation change simultaneously at a fixed rate during gameplay. To implement this functionality, we need a unified and efficient way to control and render each alien once per frame. Because every alien contains the same functionality, we can treat them as a single group and iterate through them, applying the same logic for update and

render in a unified manner, while separating entity-specific data and shared data common to all aliens. For this, we can use the Proxy design pattern.

**Proxy Pattern**

The Proxy design pattern, in general, is used to define an interface between client code and the real subject the proxy represents. A proxy acts as a placeholder for the real deal, while the client interacts with it as if the proxy was indeed the original. A benefit from using the proxy pattern is that we save memory by not storing duplicate data contained in the original in each of our proxies. We also increase execution speed by reducing the data needed to be swapped in and out of the cache, using the common data shared among proxies.

In code, the proxy pattern is implemented by defining a base abstract parent class/interface which both proxy and actual subject implement. The client then accesses the proxy through the parent class's interface, so it does not know or care whether it is using a proxy or the original. Internally, to implement the actual subject's behavior, the proxy first forwards any specific data it stores to the actual subject, and then has the actual subject perform its operation.

**Space Invaders Implementation**

In my Space Invaders implementation, the proxy pattern is used to represent the enemy aliens group as the SpriteGameProxy class, the UML of which is shown in Figure 4. The SpriteGameProxy is a placeholder for a SpriteGame. There is a concrete SpriteGame object for each alien enemy type (Squid, Crab, and Octopus), while multiple SpriteGameProxies are used to represent the group of aliens, of which there are duplicates of these types. Both the SpriteGameProxy and SpriteGame implement the SpriteBase interface, so the client code can treat them the same for both updating and rendering. The SpriteGameProxy stores each alien's specific screen coordinates (x and y), while the SpriteGame stores the shared alien texture and engine sprite data. To update and render the SpriteGameProxy, it forwards its x, y position to the actual SpriteGame via its pSprite pointer in the privPushToReal method, and then calls update and render on the actual SpriteGame. To store and iterate through all aliens, a SpriteBatch contains multiple SpriteGameProxies in a linked list by the client (SpriteNodeManager). The SpriteNodeManager walks through the list once per frame to update and draw the proxies in a single call from the client, which as far as it is concerned is interacting with actual SpriteGame, whose location is actually in an object pool kept by the SpriteGame Manager.

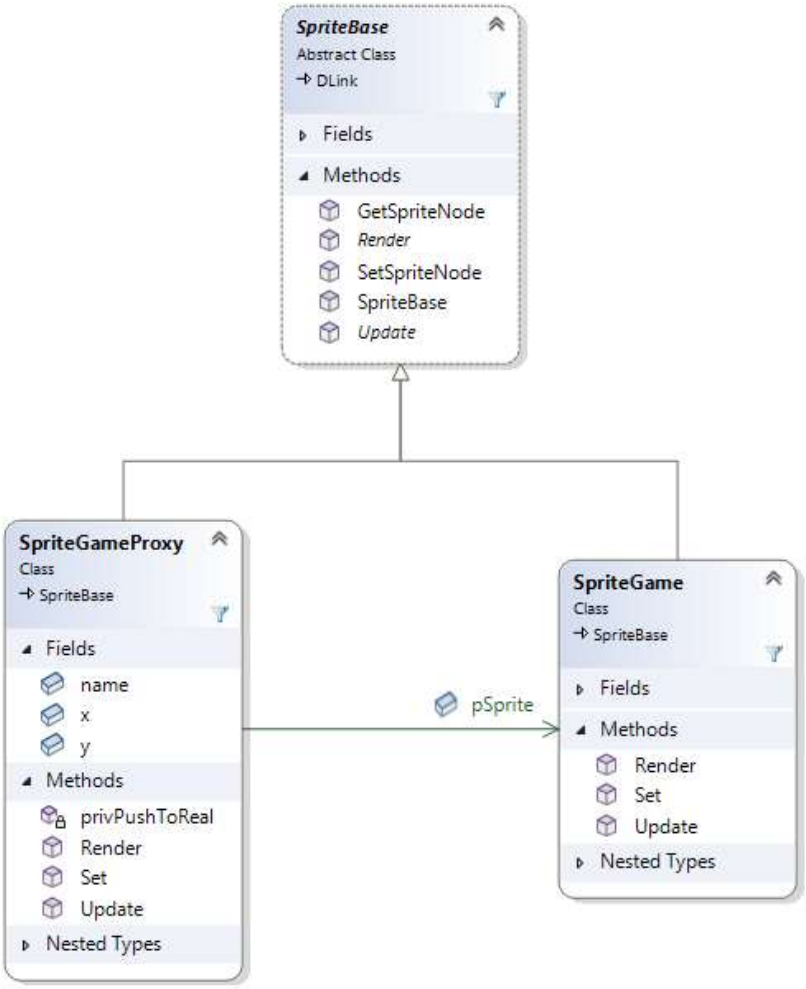


*Figure 4: Sprite Proxy UML*

# State

**Problem**

During the gameplay of Space Invaders, there are three major sections to be performed. First, the title screen and player selection. Second, the gameplay loop of destroying aliens and ship destruction. Finally, the game over screen. We need to be able to cycle between these different game stages depending upon the player's actions and our desired logic and behavior. All of this needs to be done within the same executable, as seamless as possible from the player's perspective, without de-initializing and re-initializing our core Azul library framework and backend. These three sections share common actions and needs: to draw images to the screen, update the game's simulation time, input, and sound, and move from one scene to the next. We can achieve this using the State design pattern.

**State Pattern**

Using the State pattern, a class (called a “context”) can use specific behavior for its internal state within state objects. Through a set of common methods, a class can forward the handling of these methods to its current state object. The class can store as many states as desired, which are subclasses of a shared parent state interface. A client calls one of the methods of the wrapper class, and the current internal state “handles” the request. By using a state object, the code is cleaner by removing the need for multiple switch cases or if-else statements to move to different behavior, which is instead always called from the current state object. Additionally, adding new behavior is made easier, as you simply define a new state class to use, instead of modifying existing code.

**Space Invaders Implementation**

In the Space Invaders implementation, we use the State pattern to handle the game logic for the different game stages of title/select screen, main gameplay, and game-over screen. We define a generic state interface, SceneState, which provides abstract methods for drawing to the screen, updating game logic, and moving between scene states. Its UML is shown in Figure 5. The three gameplay section state subclasses are defined in SceneSelect, ScenePlay, and SceneOver. The container Context class is defined in SceneContext, which stores instances of these three state objects, and references the current active one as the pSceneState variable. The main Azul game class contains the SceneContext, and upon its Update and Draw calls, forwards the logic to the SceneContext object, which in turn then forwards the logic to the current pSceneState. Internally, depending on the player gameplay or input, the current SceneState dictates when the context should transition to a different state.

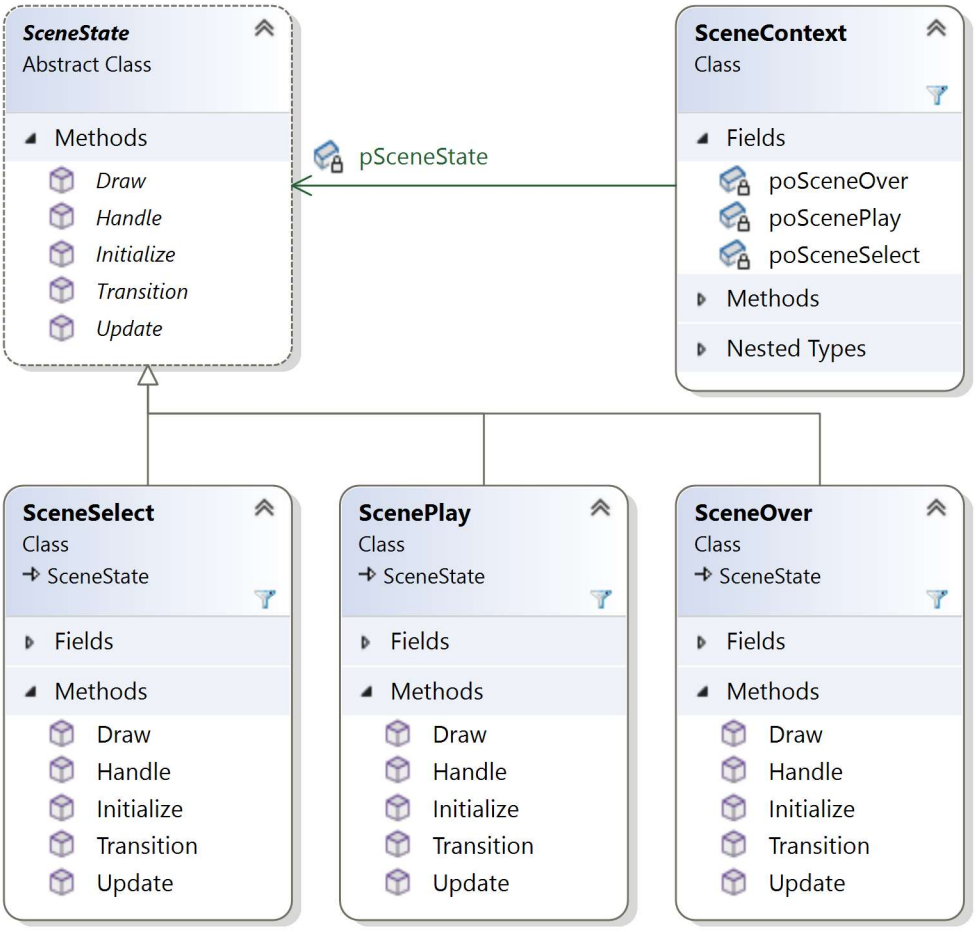


*Figure 5: Scene State UML*

# Visitor

**Problem**

For the gameplay of Space Invaders to function properly, we need to know when different objects collide on screen. For example, when the player shoots an alien, when an alien hits the player ship, and others. There are many different pairs of object types which can collide. However, all collision in our case is detected via the intersection of 2d rectangles with specific positions on screen. We need a generic way to detect collisions for many different combinations of object pairs, without giving ourselves a headache each time we want to introduce a new pair. We additionally need references to the two objects for further logic of collision handling depending on the two types.

**Visitor Pattern**

Using the Visitor design pattern, client code can have a class forward new or additional logic and behavior to a separate class by giving it a “visitor” object, which applies said behavior given a

reference to the original object. The result from the Visitor's perspective is a pair of objects: itself and the object it is "visiting," with which it can perform the desired action. The original object "accepts" the visitor, which defines abstract methods for the different class types it can visit. Adding the ability to visit a new object type requires adding a new virtual method to the visitor interface and defining the method, while adding an accept method to the class to be visited. This way, new behavior can be added to a class without changing its current implementation, resulting in less code breaking due to changes.

**Space Invaders Implementation**

The Visitor pattern in Space Invaders is used for collision via the ColVisitor class, which defines an overloaded Visit method for each object that can be collided with in game, as shown in Figure 6. Examples of objects shown which can be "visited" for collision include the AlienGrid, AlienBase, and Missile classes. In our case, who is visiting who for collision doesn't matter, so the objects can either do the visiting or accept a visitor. The ColVisitor class defines default implementations for each Visit overload, so each subclass needs only define the Visit method for objects it will actually interact with. In each visit method implementation, the visitor uses the references of itself and the object it is visiting to define a collision pair (in the ColPair) class, which is used in further logic to actually handle the collision of the two objects. For parent containing/grouping classes, such as AlienGrid, the Visit method attempts to define a ColPair using its child and the second object.

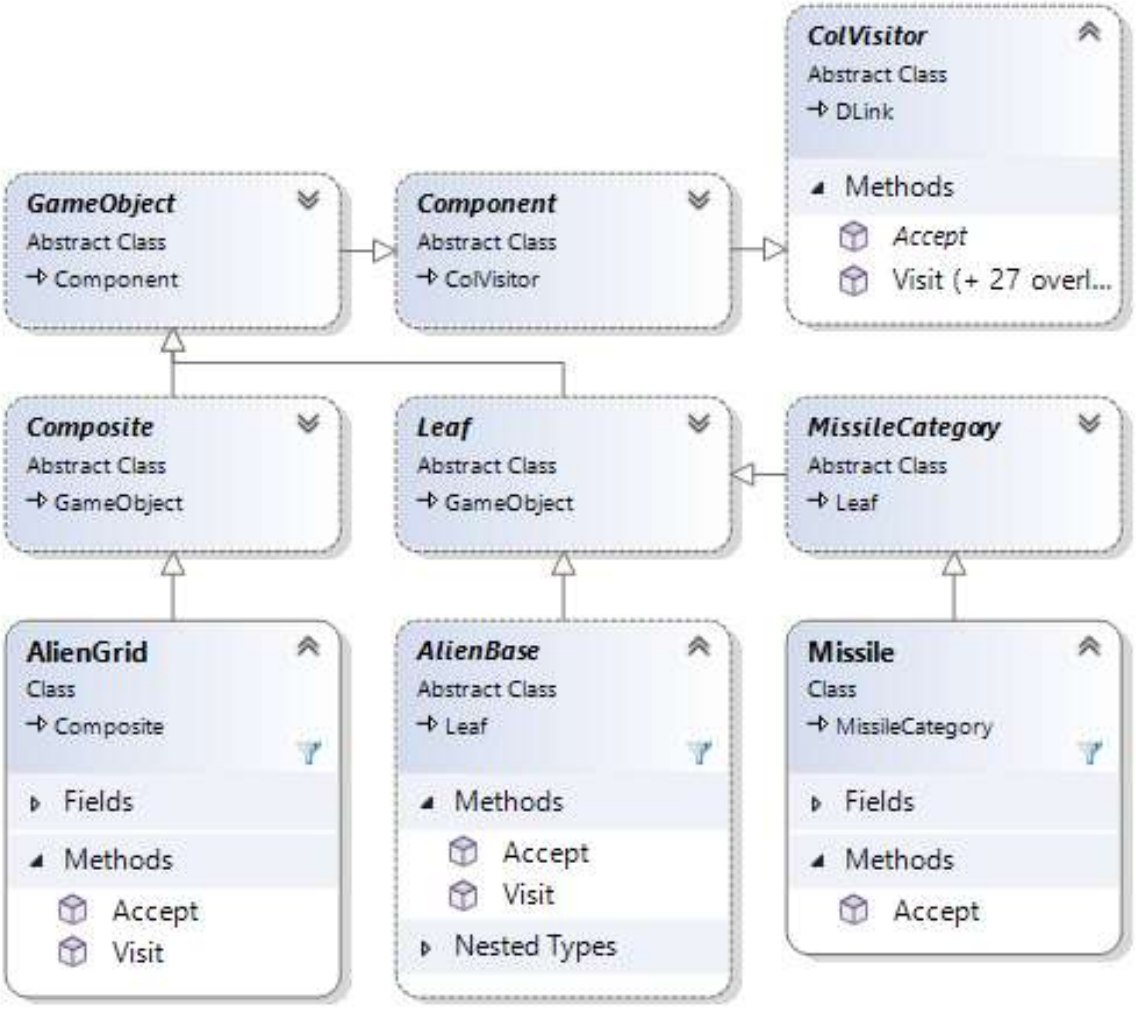


*Figure 6: Collision Visitor UML*

# Observer

**Problem**

There are many events which occur during Space Invaders which need to be handled accordingly for the game to function properly. For example, when the player presses the left and right buttons, the Ship needs to move; and when an alien drops a bomb on the player ship, it needs to explode and make the player lose a life. We need to define a cause-and-effect relationship in code, ideally without interleaving their logic too much, and allowing multiple actions to occur based on a single event. The game needs to notify other parts of game logic of the event so they can trigger actions in response.

**Observer Pattern**

The Observer design pattern allows us to define “subjects” which update one or more “observers” when an event has occurred regarding the “subject.” The subject internally stores a group of

observers, which register themselves with the subject in order to be added to its list. When an event occurs to the subject, it walks through its stored list of observers and notifies each one of the event. This allows for a clean separation of the event definition and its response. Also, we can define multiple event actions and add new ones as easily as adding a new observer object to the subject's list.

**Space Invaders Implementation**

An example use of the Observer pattern in my Space Invaders implementation is for collision handling. The ColObserver and ColSubject classes are used to define collision events and responses for when different object types overlap onscreen; in this example, ship missiles and aliens, as shown in Figure 7. First, the main game initialization registers observers with specific collision pairs. Next, During collision detection, the Visitor pattern as defined previously initializes a collision pair object (ColPair), which contains a collision subject (ColSubject) in the poSubject field. Next, the ColPair is used to notify any observers of that combination of objects by forwarding to the subject Notify method in NotifyListeners. For collision between missiles and aliens, these would be the RemoveMissileObserver and RemoveAlienObserver, which remove the missile and alien in the given collision pair, respectively. Similar observer logic is used in other Space Invaders locations, for example, when the player presses keys to move the ship or fire a missile.

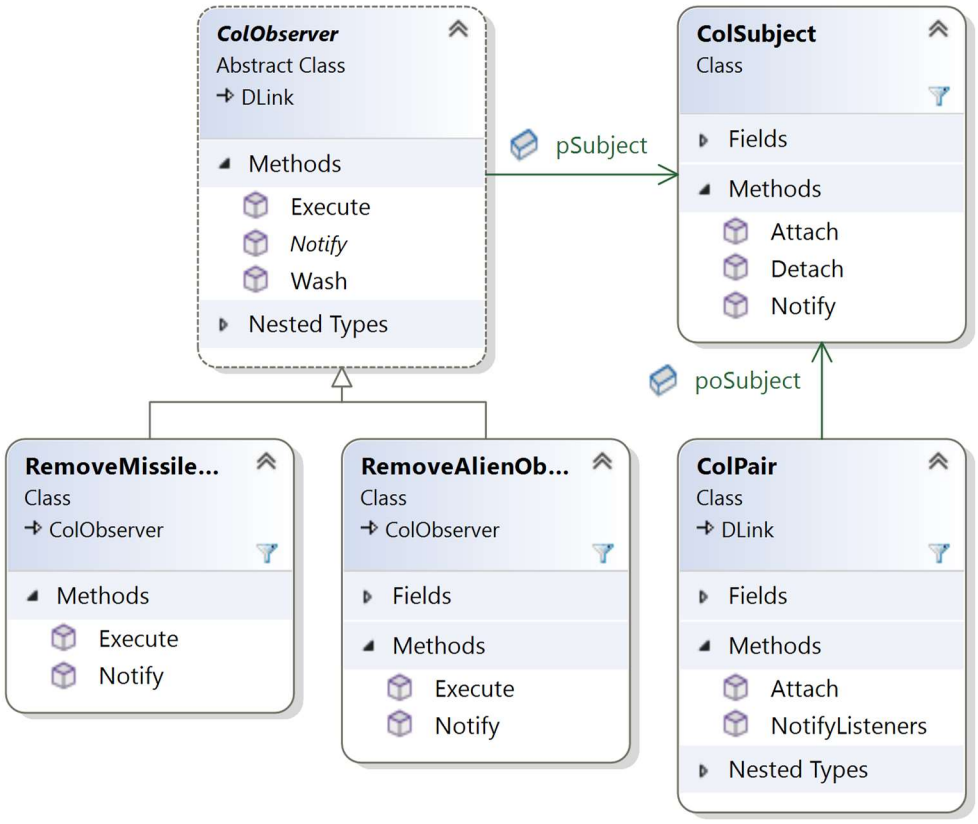


*Figure 7: Collision Observer UML*

# Command

**Problem**

There is a need within our Space Invaders game to schedule certain operations to happen in a specified time in the future or repeatedly at a given interval. The movement of the enemy aliens happens in a fixed interval over time, the aliens need to randomly drop bombs over time, among other things. We need to be able to store the logic for what happens in the future such that it can be saved for later use, when the desired time occurs. We also want to store these different happenings in a unified manner so that we don't have to keep track of time in multiple locations in our codebase, or be stuck having past events need to wait for future events to occur because of intermingling the two's logic.

**Command Pattern**

The Command design pattern can be used to define objects that store the logic and data needed to perform an operation at a desired future point. This Command object can be stored and invoked when necessary by a calling class. An execute method is defined within the Command object, which contains the action logic, and any necessary data for the logic is stored within member variables of the Command class. When the action is desired to be performed, the caller simply invokes the execute method of the Command. This allows us to separate timing logic from the specifics of the timed events, as well as store multiple timed events using a single, unified interface, without caring about the underlying action implementation.

**Space Invaders Implementation**

As the game progresses, the Timer Event Manager keeps track of the game time, which is continually increasing. It internally stores a list of Timer Events, which contain a time at which to execute and a Command reference to use for the execution. Two examples of these events are alien grid movement (to move all of the aliens on the screen) and sprite animation (to cycle the aliens between their different keyframe images). On each update, we note the game time which has passed, and iterate through the list of events and check if their execution time has passed compared to the total game runtime. If so, the event is processed, which calls Execute on its associated Command, and it is removed from the list of timed events. Commands which need to run continuously (such as the alien Grid Movement and Sprite Animations) re-register themselves with the Timer Event Manager at the next timestep in the future. The UML for these classes is shown in Figure 8.

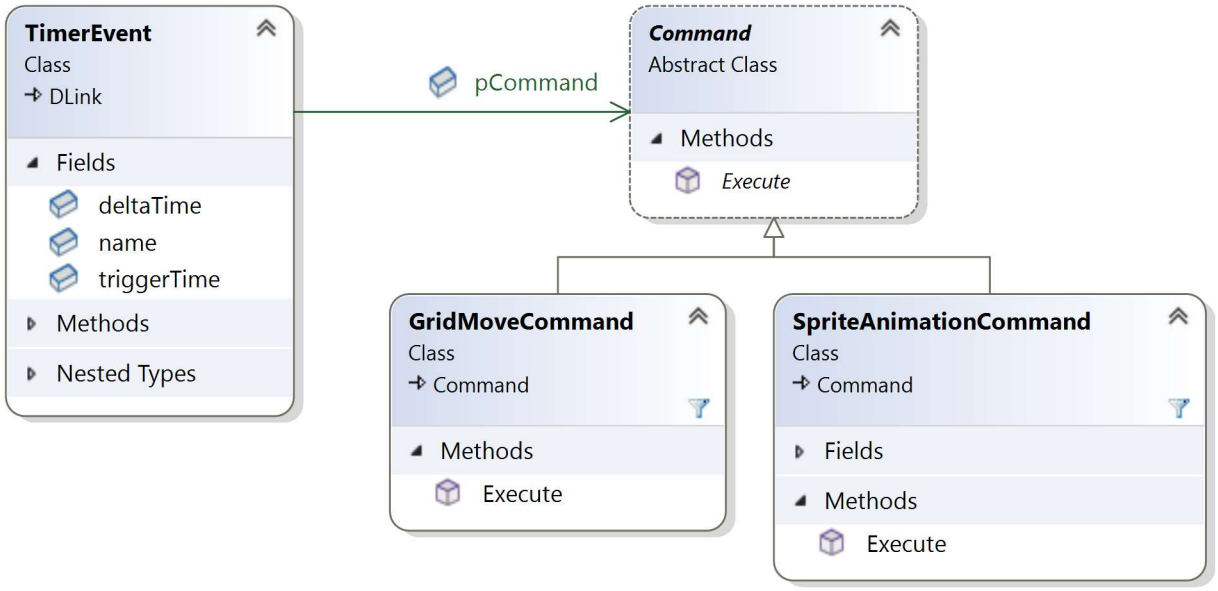


*Figure 8: Timer Event Command UML*

# Composite

**Problem**

There are multiple groups of objects within Space Invaders – alien enemies, player shields, and more. They are stored in a tree hierarchy, so we have parents and child nodes, and those child nodes can have their own children, and so on. This way, we can move and position elements based on their parents, perform early termination on collision detection, and other benefits. However, performing such logic over a tree structure can be tedious and complex when dealing with differences and checking between branches and leaf nodes, compared to storing all objects in a container such as an array. We want an easier, more unified way of performing operations on the tree, regardless of the type of node.

**Composite Pattern**

The Composite design pattern allows us to use tree nodes in a defined interface by treating both branches and leaves as the same type. Both branches and leaves are a “component” of the structure, while branches are “composites” which have one or more child components. Operations on composite nodes apply the same operation to all of its children, while for leaf nodes the operation is applied to just the leaf. This simplifies working with the tree structure as the client code doesn’t have to deal with differentiating between tree element types. In code there is a parent Component abstract class which defines the operations available, then there are both Leaf and Composite subclasses which implement these operations, for which the Composite applies to all of its children.

**Space Invaders Implementation**

Within my Space Invaders code, the Composite pattern is used to represent and group Game Objects. The example UML shown in Figure 9 is for the grid of enemy aliens, but it is used for other game objects such as collision walls and player shields. We define an abstract parent Component class, which contains Add and Remove methods for building the tree structure. The generic GameObject class is used to represent all object types within the game. The Leaf and Composite classes are used as defined for the pattern as nodes of the tree, and in this example AlienBase instances are the Leaf nodes, while AlienGrid and AlienColumns are the branches. In the main game initialization, we first create the Root and AlienGrid objects then add child AlienColumn and different AlienBase types from there. To update all elements in the tree, the GameObjectNodeMan manager calls Update on the tree root, which internally through the Composite structure automatically calls Update on all of its children, as opposed to having to walk the tree and checking for leaf nodes versus branches before applying the update.

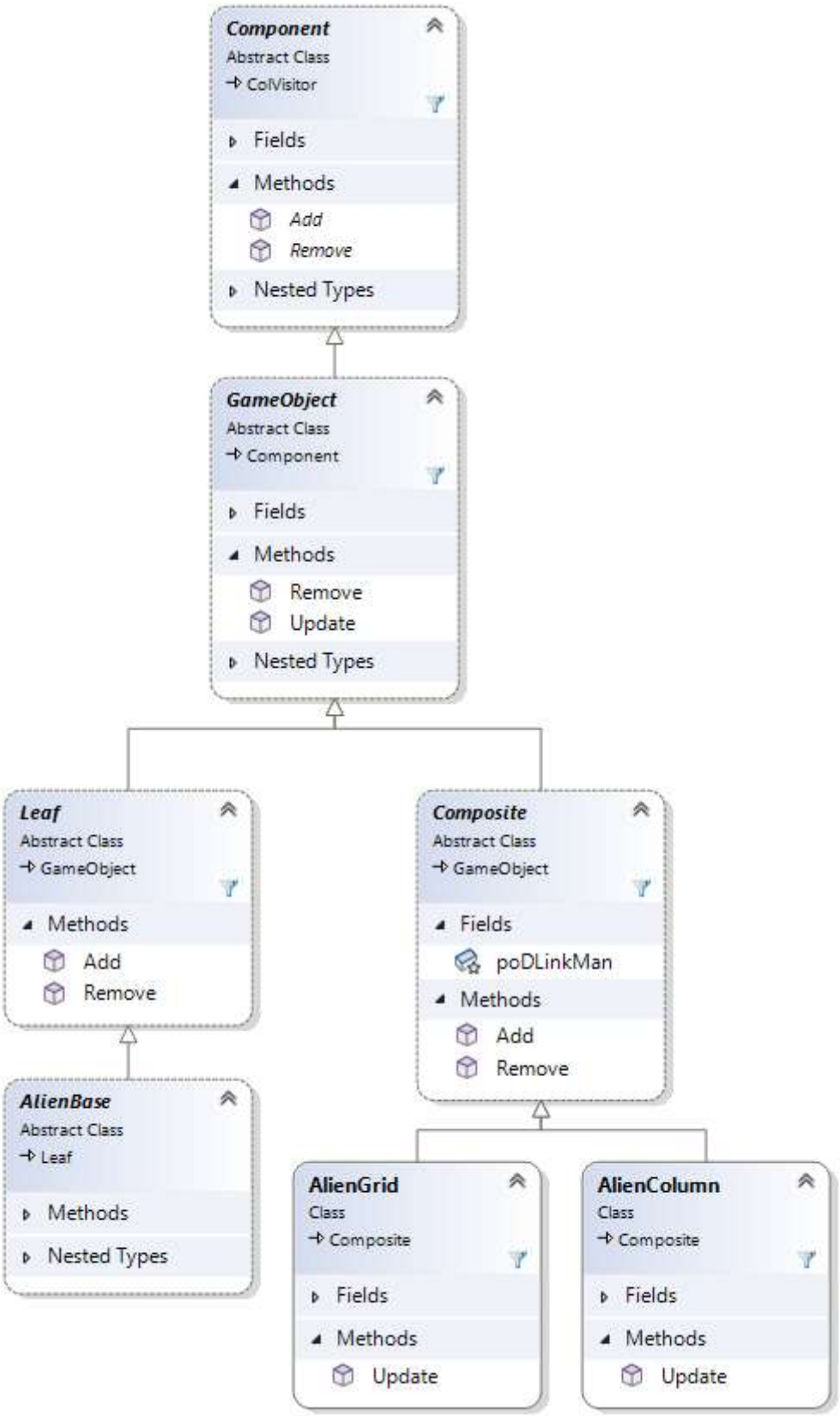


*Figure 9: Space Invaders Composite UML*

# Object Pool

**Problem**

During the logic of Space Invaders, objects need to be frequently created and destroyed, such as sprites, sounds, missiles, aliens, among others. However, allocating memory for every new item is costly time wise and could cause the performance of our application to suffer. We want a way to remove the need for these allocations while still being able to create and destroy objects. One way to achieve this is to pre-allocate the memory needed at program startup and re-use this memory for initialization and de-initialization of objects in game. This can be accomplished using the Object Pool design pattern.

**Object Pool Pattern**

Using an object pool, a group of objects of the same type, called a "pool," are pre-allocated on startup and reused over time accessing new objects of that type. When a new object instance is desired, instead of using new to allocate the object, a currently unused object instance is pulled from the pool and initialized as desired. When the object is to be destroyed, it is added back into the pool. To implement this, you need to keep track of which objects are free and currently in use, as well as providing a way to initialize the object besides its constructor. Similarly, any desired logic that would be handled in the objects deconstructor must be handled separately, prior to re-adding it back into the object pool. Although object initialization and deinitialization is complicated by using an object pool, the benefit is performance during runtime due to the removal of new memory allocation.

**Space Invaders Implementation**

Object pools are using in my Space Invaders implementation for each of the managers for different game resources, such as images and textures. In the UML diagram in Figure 10, the SpriteGameMan manager is shown. Each of the managers, i.e. the SpriteGameMan, are subclasses of the ManBase parent class. The ManBase handles the object pool by internally storing an active and reserve list of objects for the specific subclass type. In the SpriteGameMan case the list is a doubly linked list, which can be expanded if we run out of objects within the pool and a new one is desired. To obtain a "new" instance of an object, the client using SpriteGameMan calls the Create method, which in turn uses the ManBase base* methods to pull an object from the reserve list (for objects from the pool that are currently not in use) and add it to the active list (for objects that are currently in use). To "free" a SpriteGame object, the client of SpriteGameMan provides the object as an argument to Remove, which in turn causes the ManBase to move the object back from the active to the reserve list. For pool initialization, the constructor of SpriteGameMan specifies a reserve number of initial size of the object list, and a grow number to specify how many items to add to the pool if we run out of reserve SpriteGame objects to Create. The ManBase forwards the allocation of each reserve object to the subclass via the derivedCreateNode method, so that way the ManBase doesn't need to know the type of objects it is pooling.

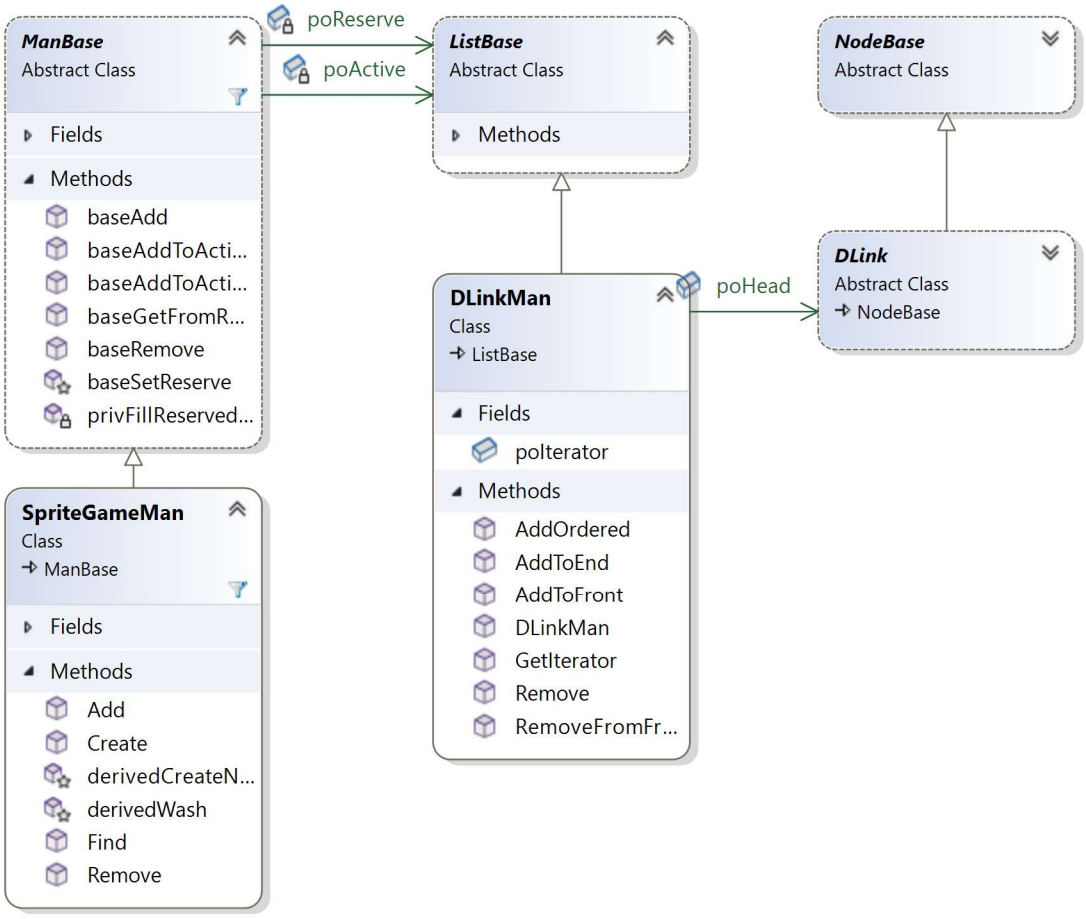


*Figure 10: Sprite Game Manager Object Pool UML*

# Post Mortem

Overall, I believe my Space Invaders project went smoothly and was generally successful. The core game requirements to my knowledge have been implemented and work as expected. What is missing are some smaller details related to accuracy against the original 1979 Space Invaders. Some of these include the additional title screen animation where there is demo gameplay and an alien which flips the ‘y’ character, gameplay start animations (player 1, blinking score), and game over screen details. A larger missing feature is the ability for two players to play the game.

As for how the project went during the quarter, things went pretty smoothly right up until the last two weeks approximately. During each week, I found it easier and less stressful to do a little bit of the week’s sprint assignment every day, spreading the work out in less time-consuming chunks. I was able to fight off procrastination successfully in terms of core sprint requirements. During the last two weeks, however, there had been unresolved bugs/issues that I had been putting off which came back to bite me, on top of the additional requirements to complete the remaining game features. This buildup of technical debt resulted in a higher workload for the end of the project, adding more stress, which also makes it harder to produce quality work. I would have been better off addressing the bugs immediately as they occurred, which if I remember correctly was a rule John Carmack and the Doom team followed.