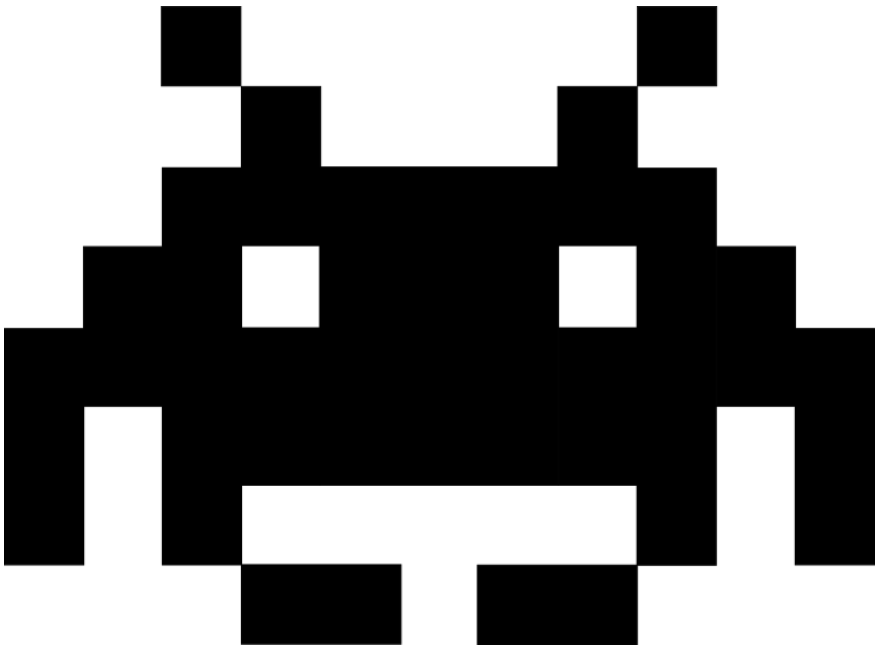

Space Invaders

Design Document

Syed Quadri
SE 456
March 13, 2017

Table of Contents

Introduction

The purpose of this design document is to describe the methodologies used to construct a *Space Invaders* clone, using modern architecture design techniques.

Problems

Throughout the design process, we were introduced to various design patterns and were tasked with implementing them into our code. The following are various architectural design problems that were encountered while creating the *Space Invaders* clone.

1. Organization

Problem

To create an efficient Space Invaders clone, we need first need to make sure that our data is organized. We need to know who holds our data and where does it live. By the time we are finished with creating the game, there will be hundreds of items to control and we can easily get lost if we need to debug and track where each code path leads to.

Solution

To solve this organization nightmare, we use Managers to hold and organize a group of data. The Managers hold no hardcoded data elements, and instead passes that task along to the specific node it is responsible for managing. Since Managers act as façade for our data, there should only be one Manager for each data group. This provides easy access anywhere within our program and we can have confidence that we are accessing the correct Manager, as there is only one.

Patterns Used

To achieve this solution, we will use **Singleton** pattern. The Singleton pattern ensures that a class has only one instance and provides a global point of access to it. In our program, we have set up several Managers to manage the textures, images, sprites, sounds, and game objects, to name a few. Each of these Managers can only be instantiated once. For example, there should only be one sprite Manager to control the sprites. If we had multiple sprite Managers, we could easily get confused in which sprite Manager we were accessing.

Within each of our Manager classes, we utilize lazy initialization of creating a private static <type>manager variable called pInstance. We set the constructor to private, so

that no user can “new” a <type>Manager object. Instead, we have a create method, that makes sure that pInstance is null before instantiating <type>Manager using “new”. This makes sure that if pInstance has already been created once, then it cannot be created again. Moreover, this guarantees us that there will only be one <type>Manager to control <type>nodes.

In addition, the Manager class has a basic API for adding, creating, finding, and removing specific nodes from an active linked list. We also made sure that the Manager class is an abstract class, making it reusable for the various data groups. Enumeration was used to make sure there are no typos and to increase speed while working, helping organize the data even more. Please refer to figures 1 and 2 to see how the managers were organized and an example of a singleton image manager, respectively.

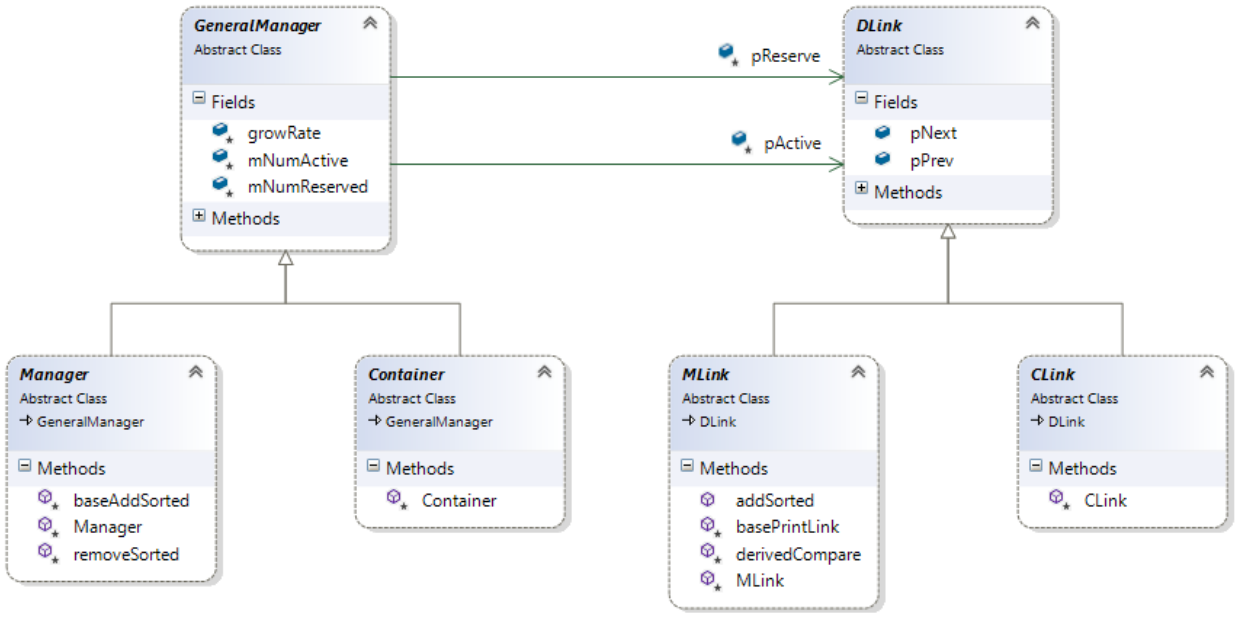

Figure 1 - Manager Organization

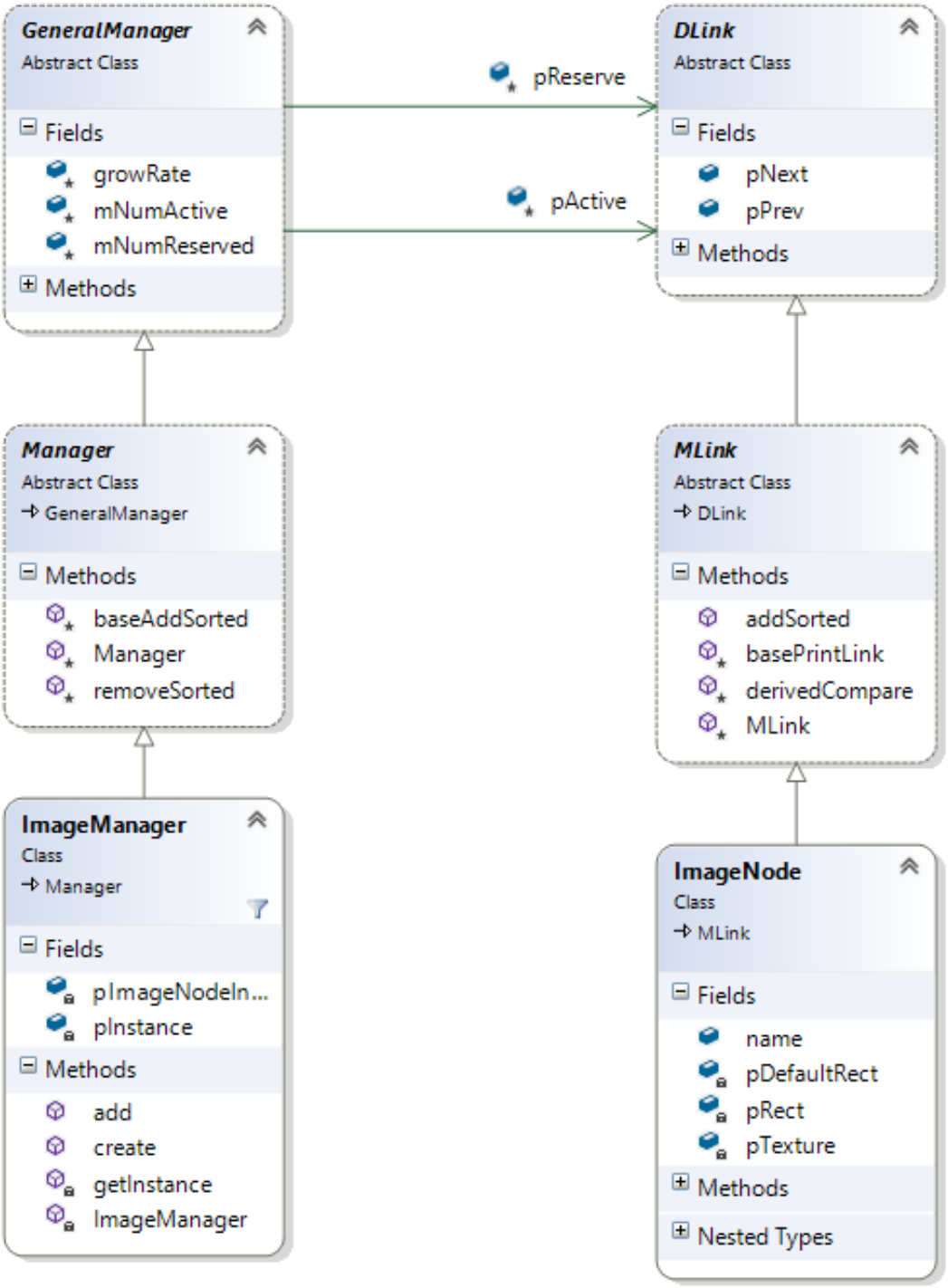

Figure 2 - Singleton Design Pattern

2. Creation

Problem

We need to create several alien objects to display on screen, 55 aliens to be exact. Many of these aliens are similar to each other, as we will have 11 Squids, 22 Crabs, and 22 Octopuses. There must be an easier way to create these 55 aliens, rather than creating each one individually, along with other game objects, such as the shields. Developers often instantiate an object before they even know how they intend on using it. This would cause a mess later down the line.

Solution

We solved this problem by having a "factory" that creates several of the same type of objects given unique input parameters.

Patterns Used

The **Factory** design pattern is a method, to instantiate an object, as it is a creational pattern. They hide the variation, which is handled through factories instead of explicit means, such as creating a red car versus a blue car. The color of the car is the only difference, as the car itself is the same. Factories keep the object cohesive, decoupled, and more importantly testable.

Within our program, we have created an Alien factory to create the 4 types of aliens: Squid, Crab, Octopus, and Grid. When we call the Alien factory's create() method, we are sending in as parameters, the Alien type, and the initial position values. Once inside the create method, we have a switch statement to see which Alien type we need to create. Once inside a case, we instantiate the Alien using new. The Alien then gets added to the PCS tree (see section E), and then attached to the sprite batch. In addition, we created a shield factory to create the 4 shields. Each shield is comprised of a column, which in turn is comprised of 10 shield blocks. The shields again are relatively the same, except for their position. Collaborating with the factory pattern, we'll use proxy objects for repetitive objects, as we'll mention in the next section. Please refer to figure 3 for the UML diagram showing the AlienFactory, which is used to create different types of aliens.

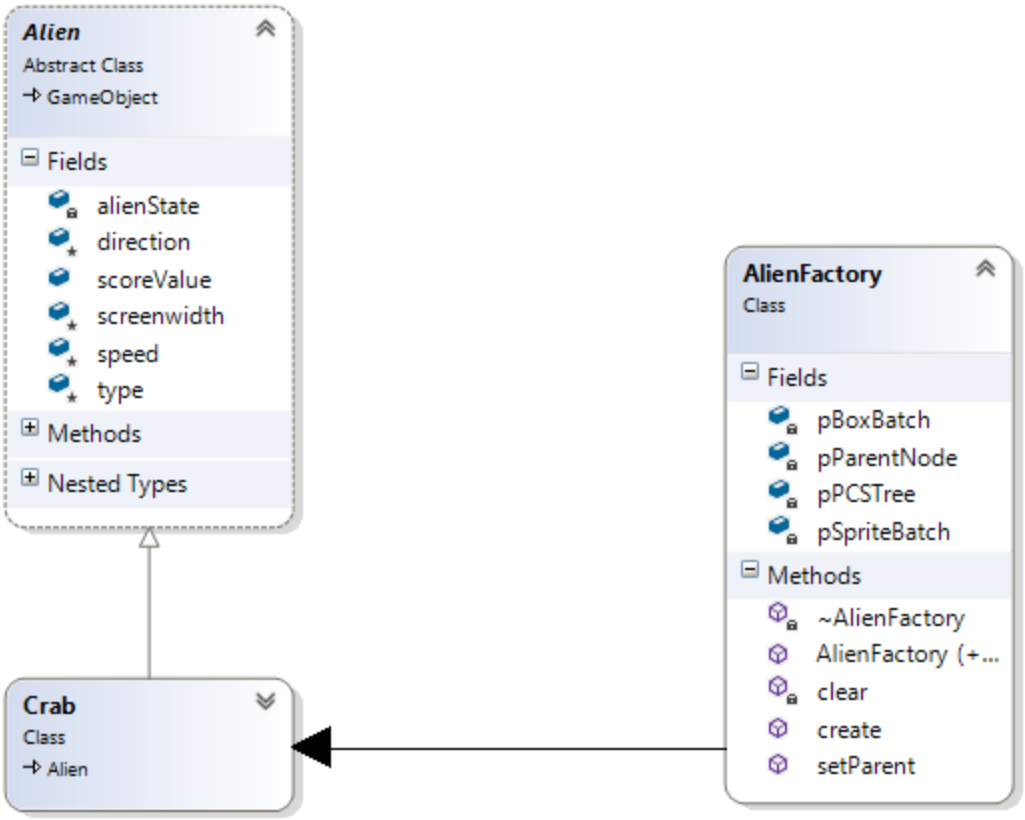

Figure 3 - Factory Design Pattern

3. Reuse Sprites

Problem

In our *Space Invaders* game, we need to create several sprites of the same type, but with different position coordinates (x, y). Creating 11 of the same Squid sprites or 22 of the same Crab and Octopus sprites can accomplish this, but then we would be creating 55 resource hungry objects, which only differ by their x and y values. Going this way could hinder our games performance, especially if our resources are limited. We also need to be able to share objects with other classes.

Solution

We solved this problem by using proxies for the sprites that the alien GameObjects will use, along with flyweights for sharing the sprites that proxiesl have a reference to.

Patterns Used

The **Proxy** design pattern provides a placeholder for another object to control access to it. It's as if we are adding a wrapper to protect the object from undue access. In our program, we have defined ProxyManager and ProxyNode classes. The ProxyNode inherits from the SpriteBase class, which derives the SpriteNode and SpriteBox classes. We are using the ProxyNode to represent the SpriteNode, by varying a sprite's coordinate position (x and y values). The ProxyNode is paired with a game object within the Alien Factory, within its base constructor, given the name of the SpriteNode name enumeration. Please refer to figure 5 to see how the GameObject class uses a ProxyNode to get SpriteNode data.

Additionally, the **Flyweight** design pattern downgrades an object to the lowest level of granularity, making it more flexible, but expensive in terms of performance. Flyweight objects are used to support a large number of objects that have part of their internal state (intrinsic) common, where the other external state (extrinsic) is stored by client objects and passed to the Flyweight when it's operations are invoked. In our program, we use Flyweights when we create the spriteNodes, ImageNodes, and textureNodes, as they all have common data that they share and are reusable. The font system does this in the same way, where it shares the texture and image of the character. Please refer to figure 4 for a UML diagram of how the flyweight was used to create different subclasses of aliens.

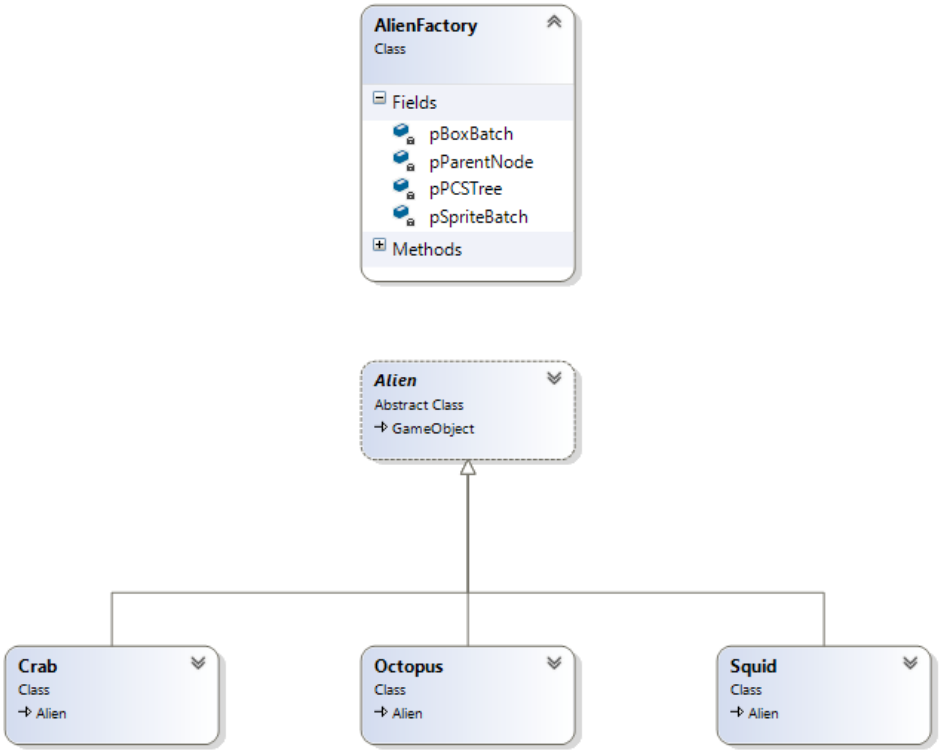

Figure 4 - Flyweight Design Pattern

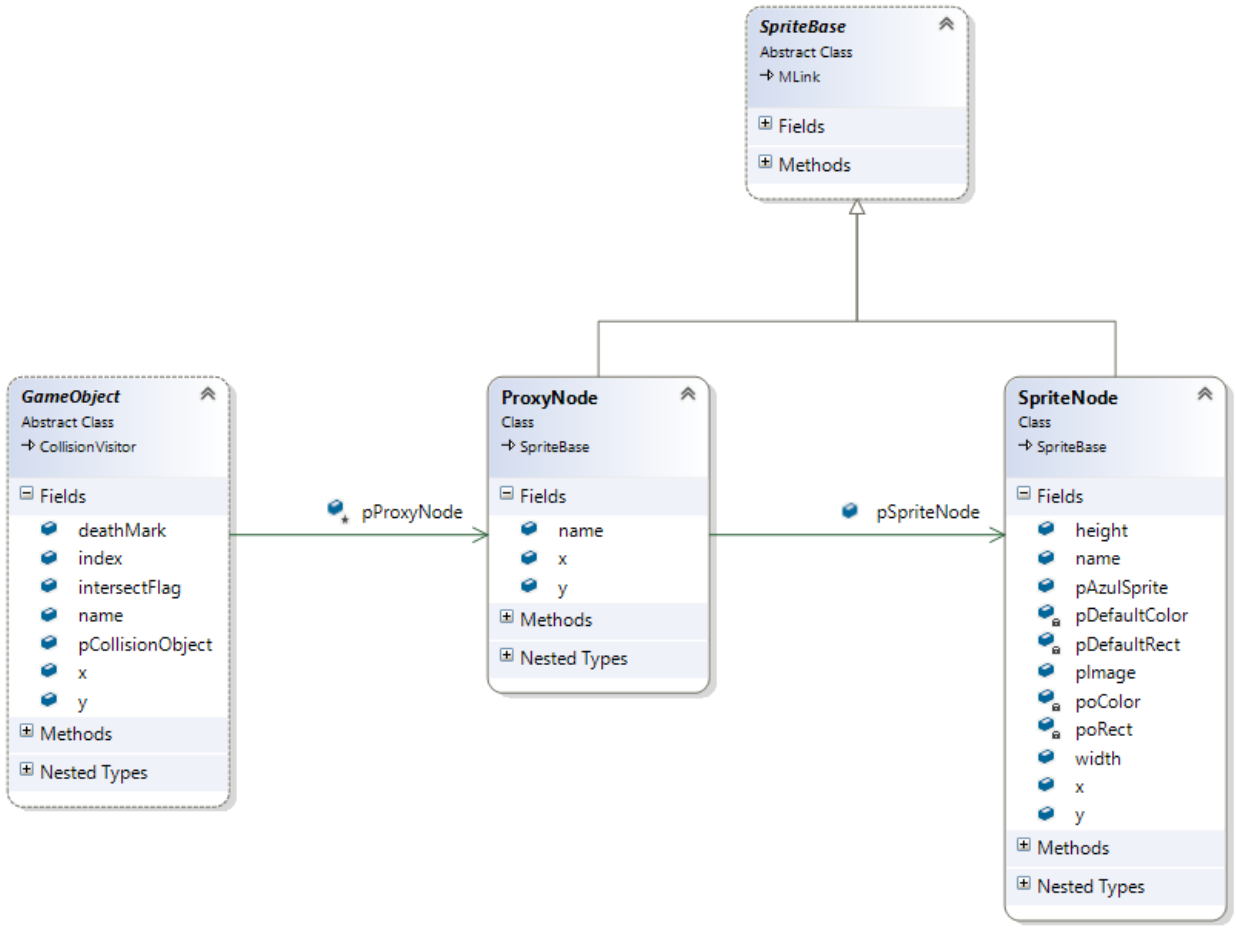

Figure 5 - Proxy Design Pattern

4. Special Cases

Problem

There are times throughout our program, where we want an object to implement some behavior, through an interface, but it will not possess that behavior. What happens? We get a null pointer exception and our program gives us an error. For example, in our game we need to create a Wall game object, but it needs to be invisible, making our draw method invalid.

Solution

To solve this problem, we will be using an object that does nothing and acts as safe object using the Null Object behavior pattern.

Pattern Used

The **Null Object** design pattern does one thing… nothing! The null object is used to encapsulate the absence of an object by providing a default "nothing" behavior. In our program, we create a NullGameObject class which simply acts as placeholder for our conditional checks, such as when the object needs to compute it's update method. As there is nothing defined in the update method, it simply returns. The main takeaway is that our code doesn't flag us with errors. Refer to figure 6 for the UML diagram of the NullGameObject.

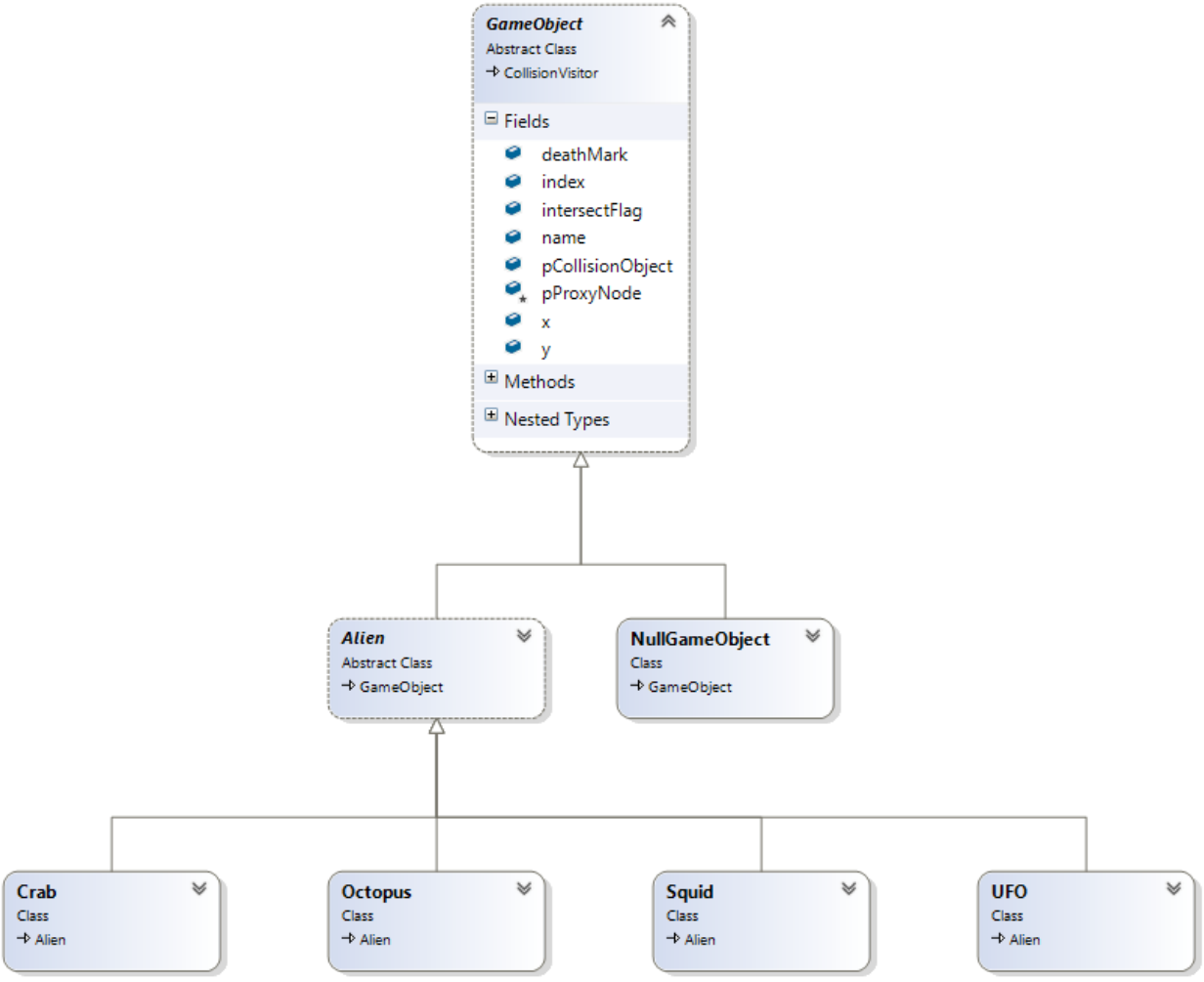

Figure 6 - Null Object Design Pattern

5. Collision

Problem

As we have several game objects in the scene (aliens, UFO, bombs, missiles, walls, and player ship), we need a quick efficient way to discriminate collision decisions, without using conditionals and switch statements. It needs to be extensible and scalable at the same time. In a scenario where a collision does occur, which object collided with whom?

Solution

To solve this problem, we will be creating accept methods in each of our collidable classes, which can in turn tell us who collided, using the Visitor design pattern.

Patterns Used

The **Visitor** design pattern delegates an operation to be performed on the elements of an object. It lets you define a new operation without changing the classes of the elements on which it operates. A classic technique for recovering lost class type information. Visitor basically acts as double dispatch, doing the right thing based on the type of two objects, as they communicate through a similar operation.

In our program, we only use the double dispatch portion, where each game object has an accept() method. In addition, each object will have various visitedBy<type>() methods, if that object would be prone to having a collision with object of type <type>. So, when a missile and an alien collide, the alien will accept the missile through its visitedByMissile() method, which in turn will inform the missile that it has been visited by an alien and continue with the collision process. Please refer to figure 7 to see how the Missile and MissileRoot classes use the CollisionVisitor class.

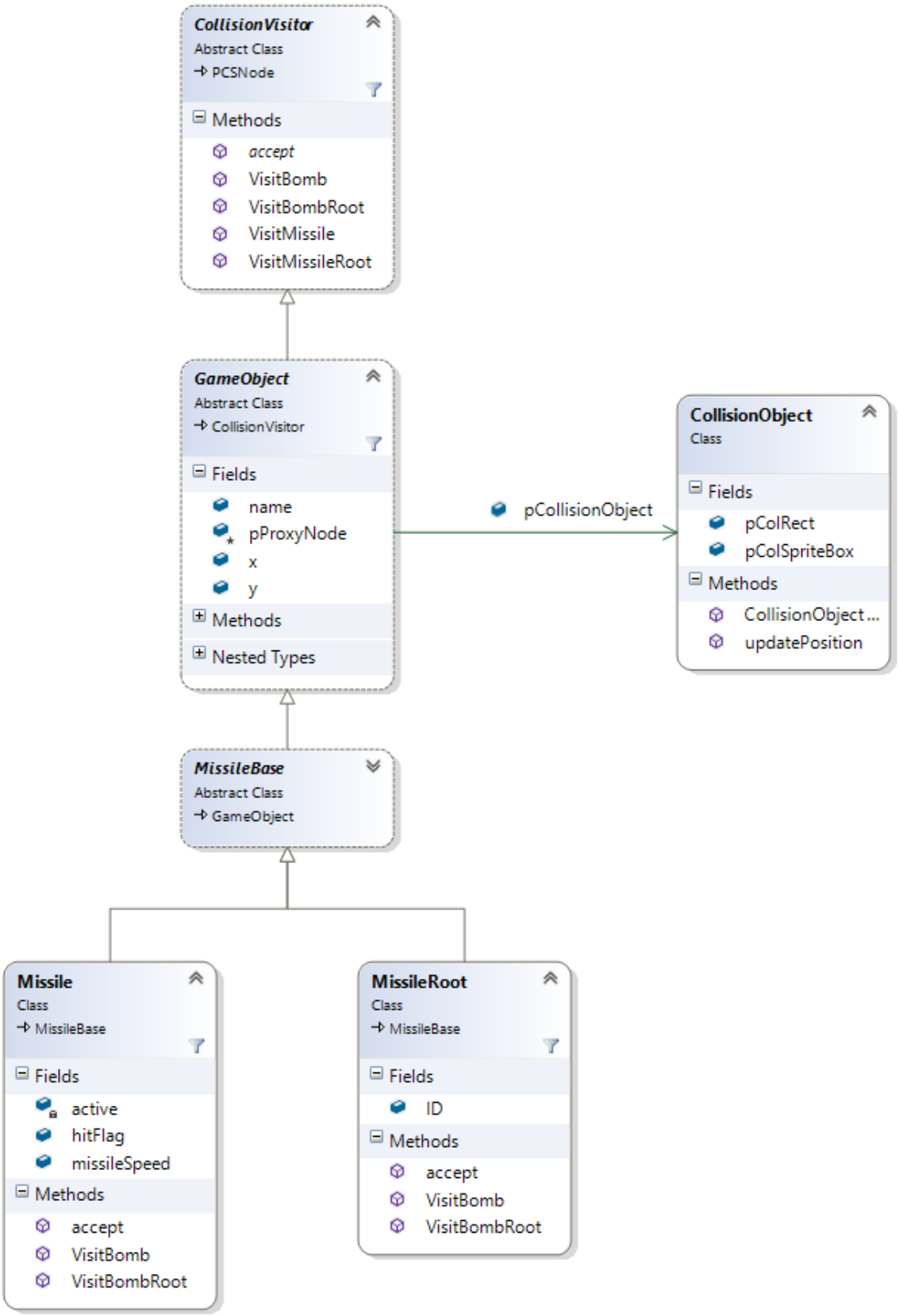

Figure 7 - Visitor Design Pattern

6. Notification

Problem

Now that we have a system where game objects collide with each other, we need to figure out when this occurs so that we can act accordingly. For Example, when the alien grid collides with one of the side walls, the grid shifts down and changes directions. Or when a UFO collides with a missile, we want to play the UFOs death animation and remove it from the scene. In essence, we want a system to handle the delegation and still adhere to encapsulation principles.

Solution

To solve this issue, we assign observers to certain collision pairs using the observer pattern, so that when a collision event does occur, the observer handles the processing. This way, the collision system is not involved in the processing and the game system is not just polling every update to see if the two objects collided.

Pattern Used

The **Observer** design pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents (Observers) are notified and updated as well. This pattern uses a Subject, which represents the common abstraction. The Observers represent the variable abstraction. The Subject invokes the Observers to do their tasks, while each Observer can call back to the Subject as needed.

In our program, we define several different types of observers and are assigned to a subject (CollisionPair). When the collision pairs are processing, and a collision event does occur, then within that visitor of the appropriate collision, we notify all the observers associated. This delegates the work to the observer which has its own notify() method that does all the necessary work for the collision processing. The Observers can call on the subject to see which game objects collided, using those elements in the collision process as well. Please refer to figure 8 to see how the CollisionSubject class owns a list of CollisionObservers, which notifies the AlienBombReadyObserver, ShipReadyObserver, MissileObserver, and AlienGridObserver whenever the appropriate collision is detected.

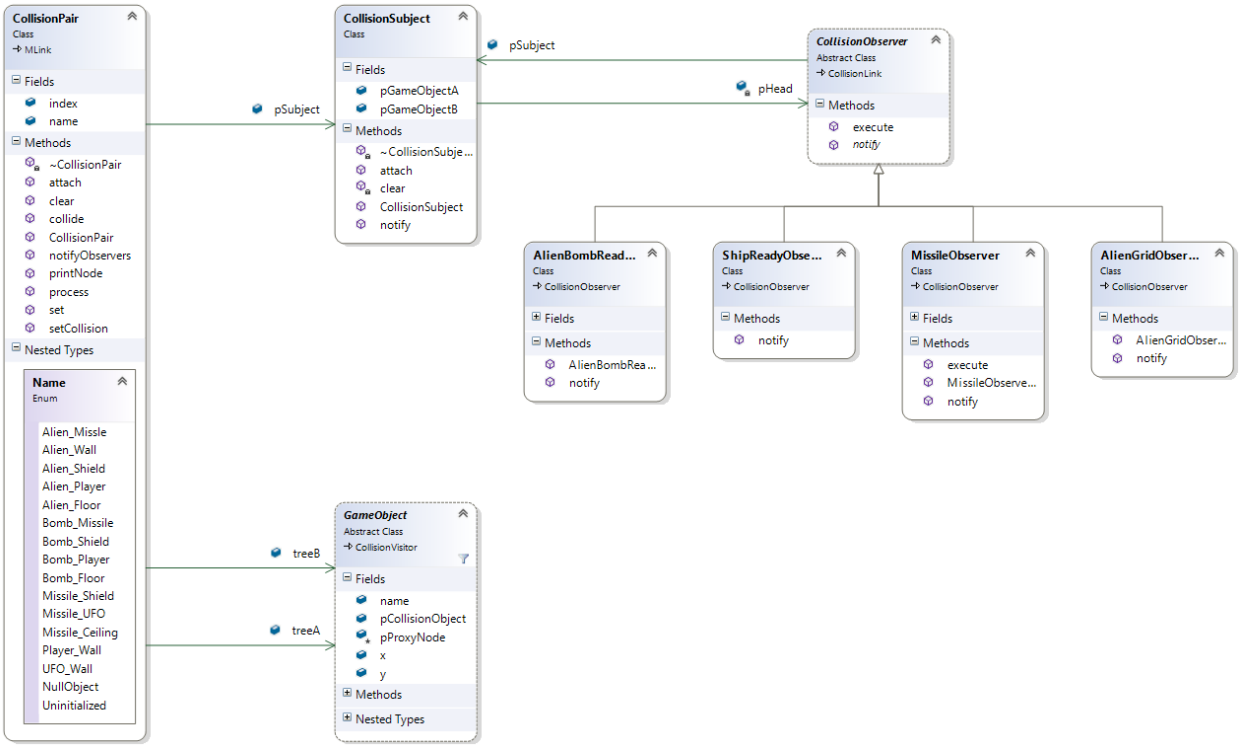

Figure 8 - Observer Design Pattern

7. Time Events

Problem

Our TimeManager will add TimeEvents in a priority queue. Once the Timer needs to update every frame, it needs to execute that event regardless of its what the operation is, as the TimeManager doesn't need to know the specifics. If we had different classes inside each of these events, do their own unique operation, it can get confusing very quickly.

Solution

To resolve this problem, we created a separate abstract class that other classes will derive from. The main method, execute(), will be the primary one the Timer will care for and compute whatever is inside that method. The Command pattern was used to construct this.

Patterns Used

The **Command** design pattern simply encapsulates a request as an object. This allows the requester to invocate a method, regardless of what type the object is, so long as it has a definition for the method.

In our program, we have an abstract Command class, with only one abstract method called execute. The Animation event class inherits from the Command class, thus allowing it to override the execute method. Within the execute method, the animation and movement of the sprite is conducted. The TimeManager takes in TimeEvents, which takes in a Command class as an input parameter. As of now, there is only one type of event (Animation), but later we will add collision, sound, and various other events. All these events will share the execute method form the Command class. Please refer to Figure 9 for the UML diagram of the Timer system, showcasing the use of the Command class through the AnimationSprite class.

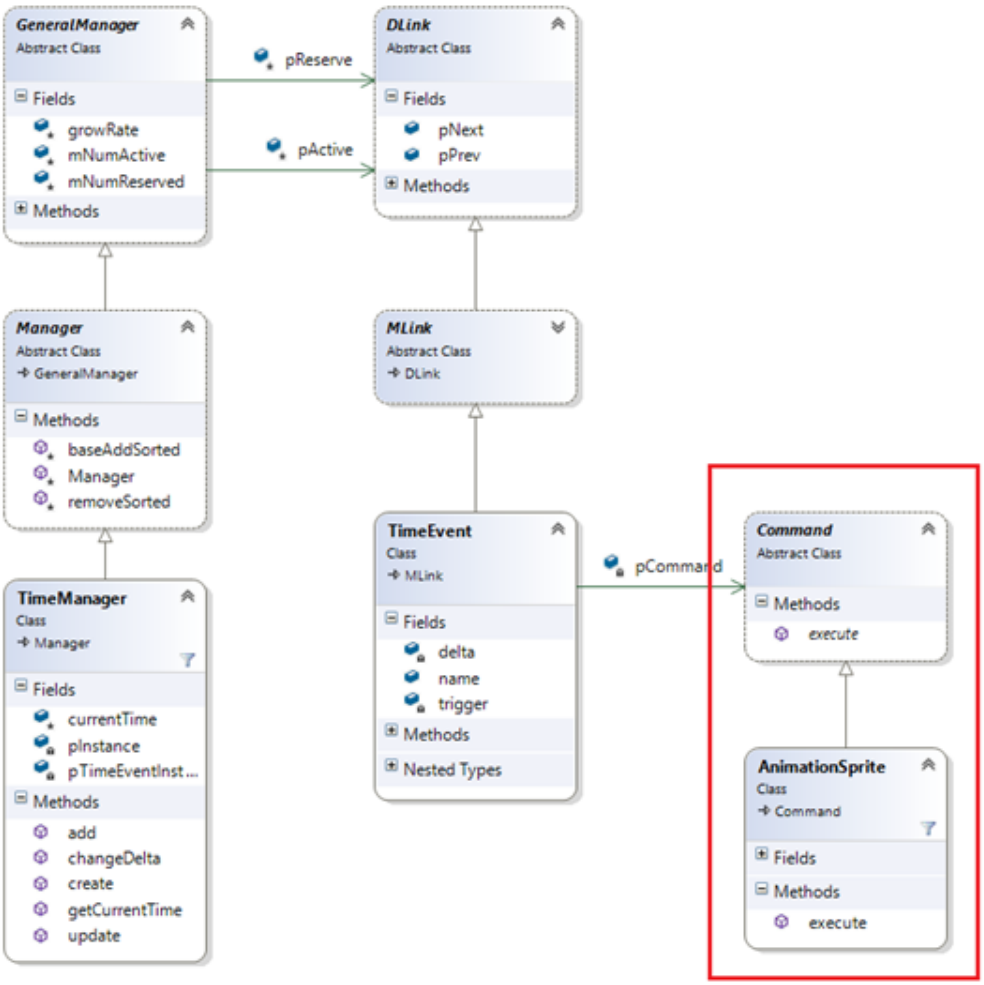

Figure 9 - Command Design Pattern

8. Getting Data

Problem

In our *Space Invaders* clone, we have several data elements in unique data structures. We have data nodes within a linked list and game objects within a PCS(Parent Child Sibling) tree (see section 12). Its straight forward to iterate through a linked list, but the PCS tree is unique and can get complicated when having to iterate or reverse iterate through such a tree. We need to access the data and still be able to change the holding container freely, without direct access links or any knowledge of the internals of the container.

Solution

To solve this dilemma, we need define an abstraction that allows us to iterate through the container, giving us control and encapsulating complexity of the storage mechanism. This is done using the Iterator design pattern.

Patterns Used

The **Iterator** design pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. This decouples the iterator from the container class, allowing polymorphic traversal.

In our program, we use the PCS tree as mentioned. When updating all the game objects, we use the ReverseIterator to go through all the objects of that tree, in reverse order. For example, in the AlienGridTree, we start off with the aliens, then go to the columns, and finally the grid itself. We also use a ForwardIterator to find a game object in a specified tree. It may seem like the iterator doesn't do much, as you could easily use the data directly, but then you wouldn't be able to control access and restructure your container without changing your entire code. Please refer to figure 10 for the UML diagram of the ForwardIterator and ReverseIterator classes, as they are a subclass of the Iterator abstraction for the PCS tree container.

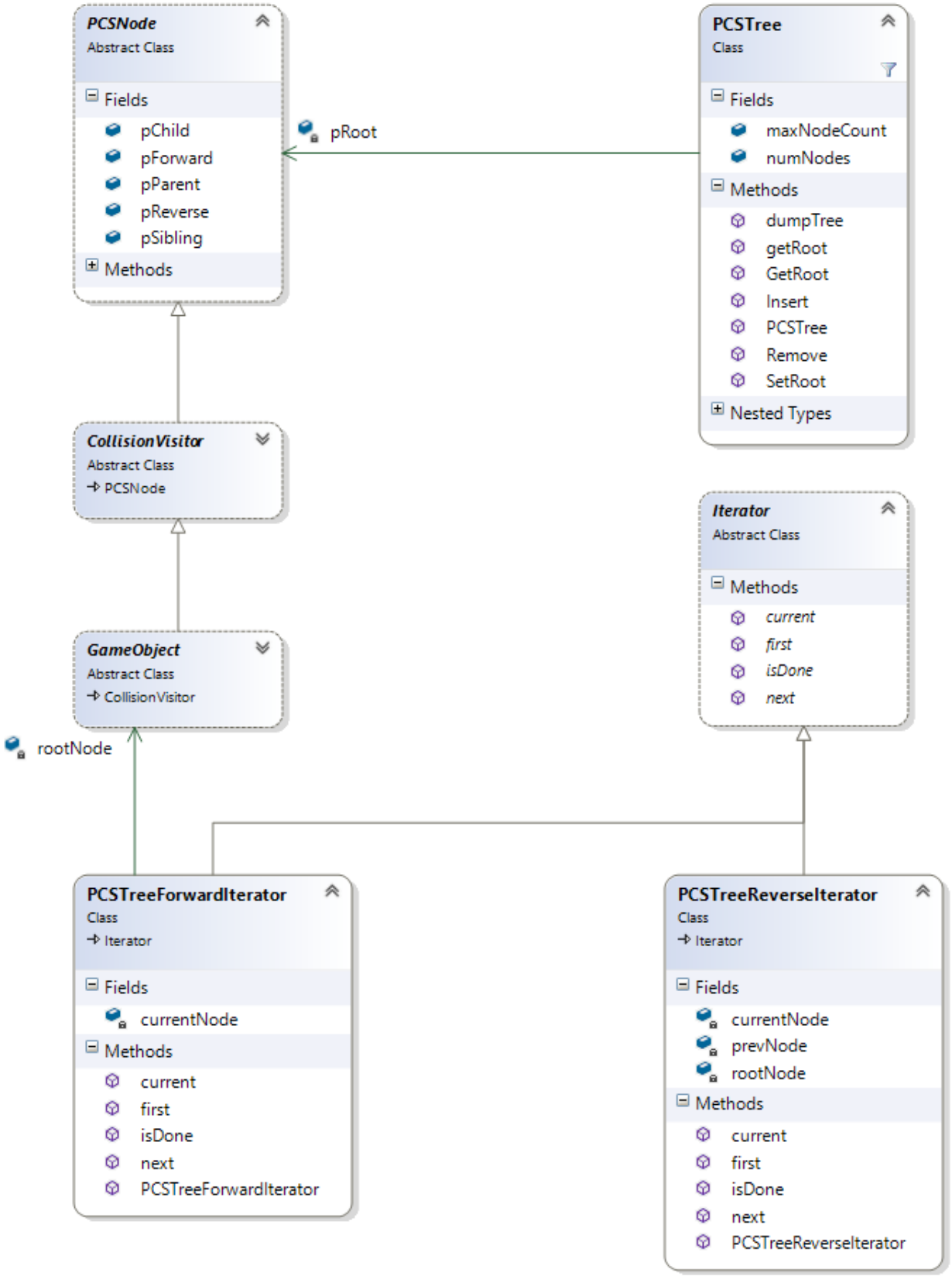

Figure 10 - Iterator Design Pattern

9. Special Behaviors

Problem

In our game, we have aliens that will drop 3 different types of bombs randomly: ZigZag, Dagger, and Straight. The bombs themselves all behave the same way, but the sprites they use and the way their images rotate are unique. Thus, we need to find a way to specialize each bomb's behavior, without risking code repetition.

Solution

To solve this problem, we organized the specialized bombs under an abstraction using the Strategy pattern.

Patterns Used

The **Strategy** design pattern defines a family of methods, encapsulating each one into its own class, and ultimately making them interchangeable. This lets the method vary independently from the client that uses it. As mentioned above, it requires capturing the abstraction to an interface and burying the implementation details in the derived classes.

In our program, we have a Bomb class that uses an abstract class's fall strategy to drop the bomb. The FallStrategy abstract class has 3 derived classes, one for each of the different bomb types (ZigZag, Dagger, and Straight). Within each of those derived classes, it overrides the fall method to its own unique behavior. For example, the ZigZag bomb fall method simply flips the sprite image horizontally, while the Dagger bomb flips it vertically, and the Straight bomb does nothing. This way when an alien drops a bomb, it simply calls the dropBomb() method and unbeknownst to the alien, the FallStrategy abstract class delegates the work to the appropriate bomb type. This makes it easy to extend if we need to add more bomb types, keeping the behavior isolated from the others. Please refer to figure 11 for the UML diagrams showing the bomb class and how it uses the 3 different fall strategies for dropping a bomb.

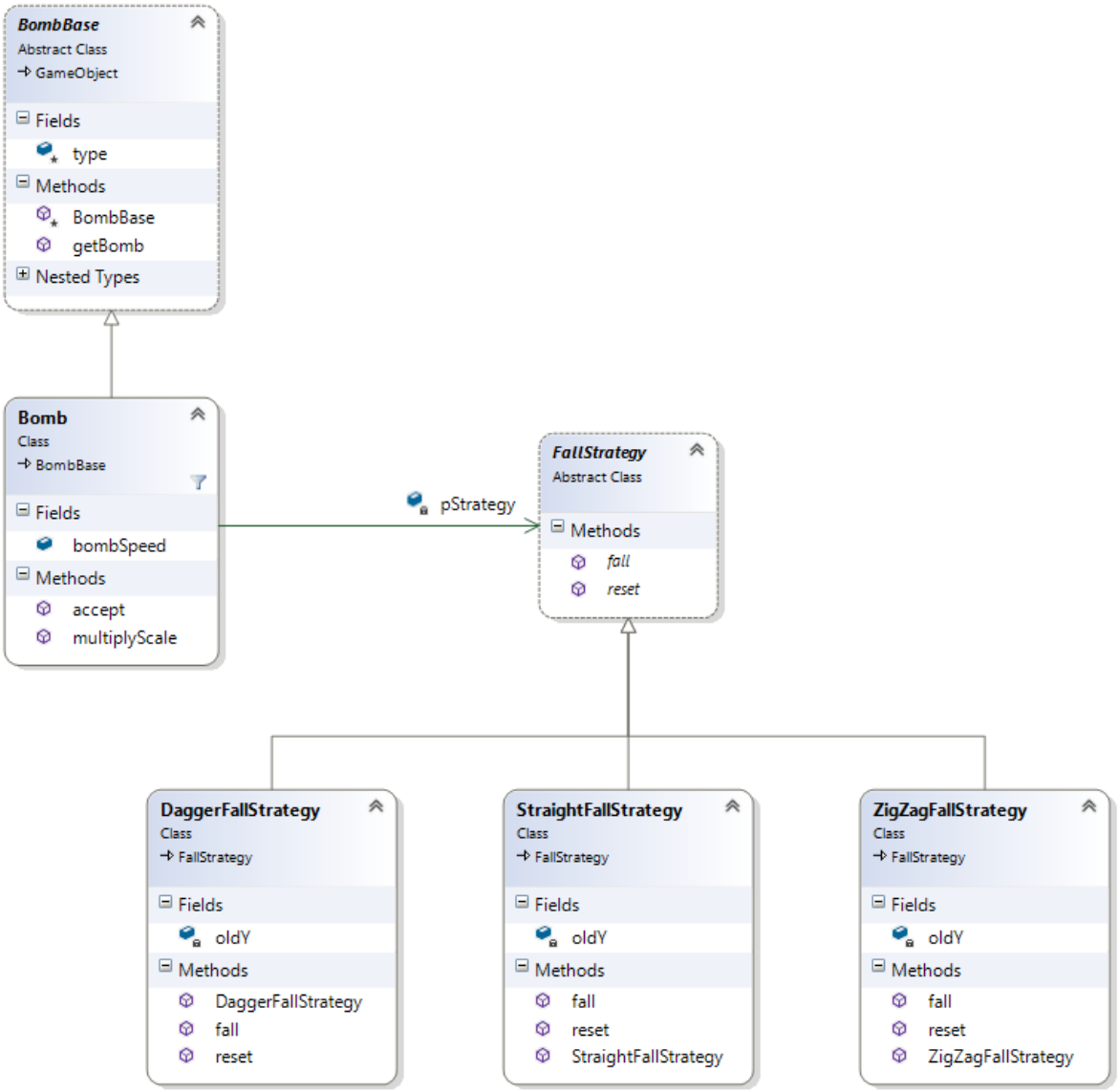

Figure 11 - Strategy Design Pattern

10. Ordered Time Events

Problem

In our game, we need to have an ordered list of TimeEvents, so that they occur in succession. For example, if they weren't ordered, we could have an alien sprite swap its image to explosion and then shoot out a missile, which doesn't make a lot of sense.

Solution

To resolve this issue, we use a Priority Queue data structure to hold the TimeEvents based on their respective trigger times.

Patterns Used

Not necessarily a design pattern, the **Priority Queue** is data structure that acts as an ordered list. The basis of the ordering can be a variable with each of the objects of the list. In our program, we have the TimeManager take in TimeEvents, as mentioned above. When an event is added to the TimeManager, it takes in a delta time value. The events are added to the list based on their trigger time, which is the current time value plus the delta time. The lowest trigger times will be added to the front of the list, while the highest trigger times, will be added to the back. Once an event executes its operation, it is removed from the list. Please refer to figure 12 for the UML diagram of TimeManager class, which uses the Manager's addSorted() and removeSorted() methods to construct a priority queue.

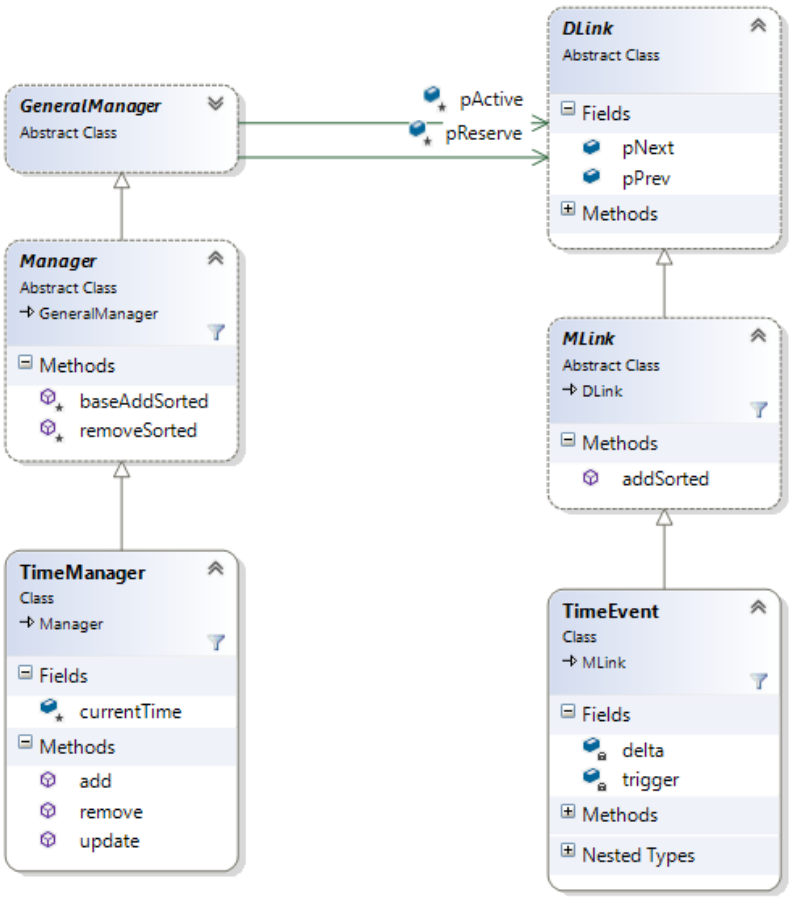

Figure 12 - Priority Queue used by the TimeManager through TimeEvents

11. Modes

Problem

In our *Space Invaders* clone, we need to have a mechanism for handling the different modes of the game. The game flow utilizes different modes as it switches between the intro screen to the actual game screen and then into the game over screen, repeating again. Our ship has different modes too, as it fires a missile it can't fire it again until it is destroyed. Thus, we have missileReady and missileFlying states. We need a way to effectively switch between modes without using conditionals and switch statements.

Solution

To solve this problem, we will be using the State pattern to control the different modes of the Ship, Alien, and game flow.

Patterns Used

The **State** design pattern allows an object to alter its behavior when it's internal state changes, making it appear as though it has changed its class. This essentially creates an object-oriented state machine, rather than a procedural state machine.

In our program, we created an abstract ShipState class which handles the keyboard presses of the right, left, and spacebar keys. When the ship is created it's state is set to ShipMissileReady. Within that state class, the ship can move left, right, and fire a missile. However, once the ship does fire a missile, the ship's state is change to ShipMissileFlying, where pressing the spacebar no longer launches a missile. The ship will remain in this state until the missile is destroyed by either colliding with an alien, bomb, UFO, or the ceiling wall. Within the ShipReadyObserver, which is assigned to any collision pair involving the missile, the ship's state is set back to ShipMissileReady once the missile is destroyed. This constant loop makes sure the ship can only fire one missile at a time and yet still operate with its other keys. This same effect was put into the alien as well, but uses bombs instead of missiles, making sure that each bottom alien of each column can only fire one bomb until it is destroyed. Again, this is how the game flow system works, as it switches between its 3 states (intro, game, and gameover screens). Please refer to figures 13 and 14 for the UML diagram examples of how the Ship and Alien classes use states to drop bombs/missiles, respectively.

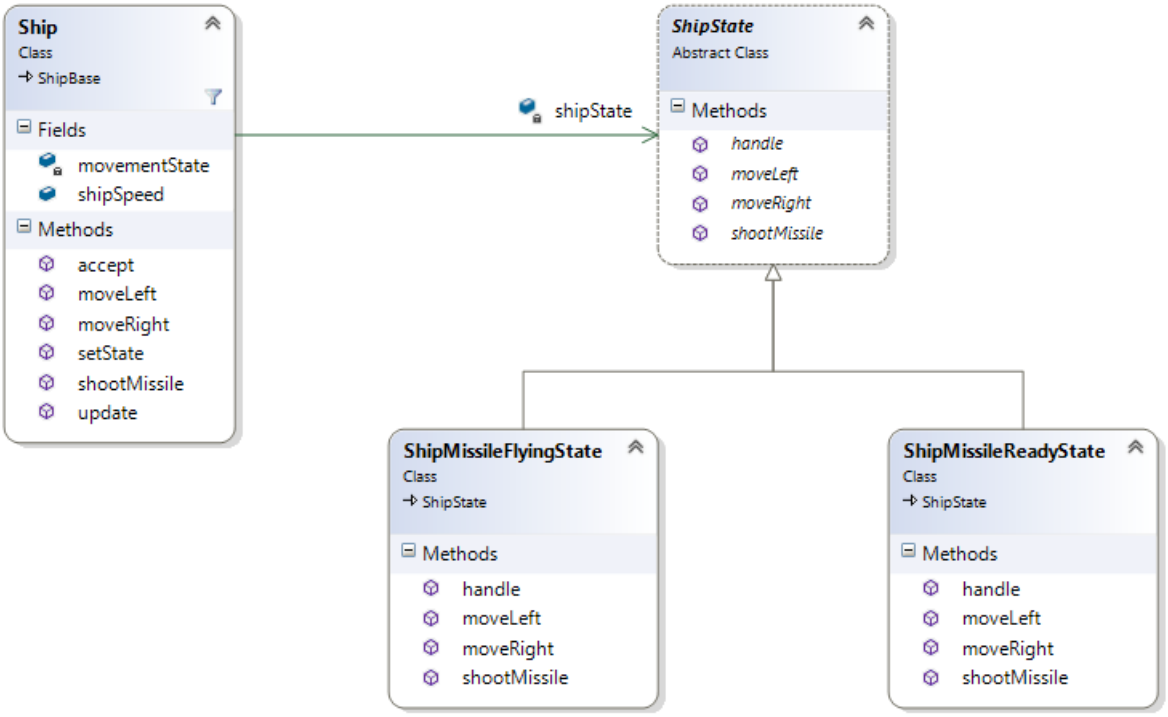

Figure 13 - State Design Pattern

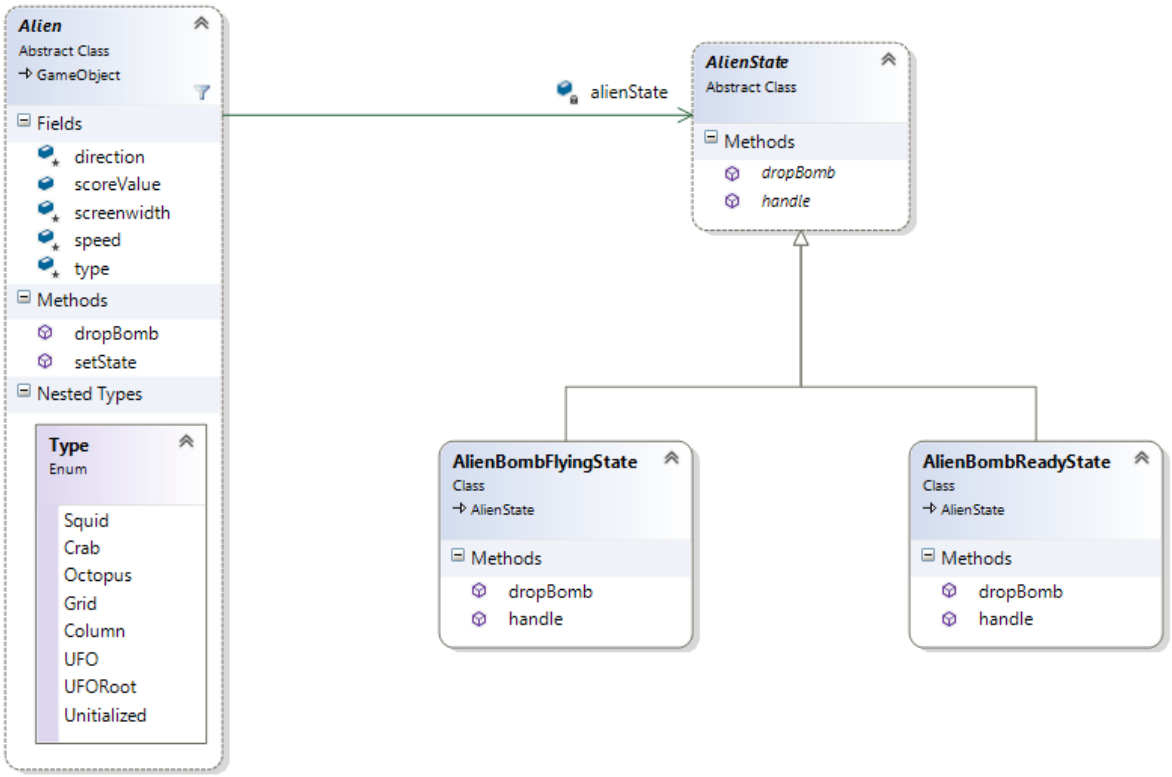

Figure 14 - State Design Pattern

12. Hierarchy

Problem

In our game, we have several game objects that do similar work, such as the 55 aliens and the grid. One such work movement. So how do we move the alien grid and aliens themselves together? Another scenario is if one of the aliens in a column dies. How do we let the grid readjust for this missing alien?

Solution

To solve this problem, we need to create a container that will handle each object in its own manner, using the Composite pattern and PCS tree structure.

Patterns Used

The **Composite** design pattern allows you to treat individual objects and composition of objects (composites) uniformly. This essentially allows us to represent part-whole hierarchies. The components can be further divided into smaller components, while the hierarchy remains intact.

In our program, we use the Composite pattern through a **PCS (Parent Child Sibling) tree**. The Parent acts as the root of the tree and can have many children. Each child can either be a parent to another child, or have siblings that share its height level on the tree. Every game object will inherit form the PCSNode class, making everything seen on screen part of some hierarchy. We can see this with the AlienGrid and aliens. The AlienGrid acts a parent to 11 columns. Each of the columns, has 5 aliens attached to it (1 Squid, 2 Crabs, and 2 Octopuses). As the GameObjectManager computes its update method, it first goes to the parent of the entire tree, the AlienGrid, and then computes the movement for each of the columns, which in turn computes the movement for each specific alien within the grid. If an alien is removed from play, the grid readjusts itself by removing that alien from the tree. If an entire column of aliens is killed, then the entire column child is removed, thus decreasing the size of the alien grid, allowing it to correctly calculate when the grid hits one of the side walls. Please refer to figure 15 for the UML diagram of the PCSTree and PCSNode class, which is used by all GameObjects in the scene.

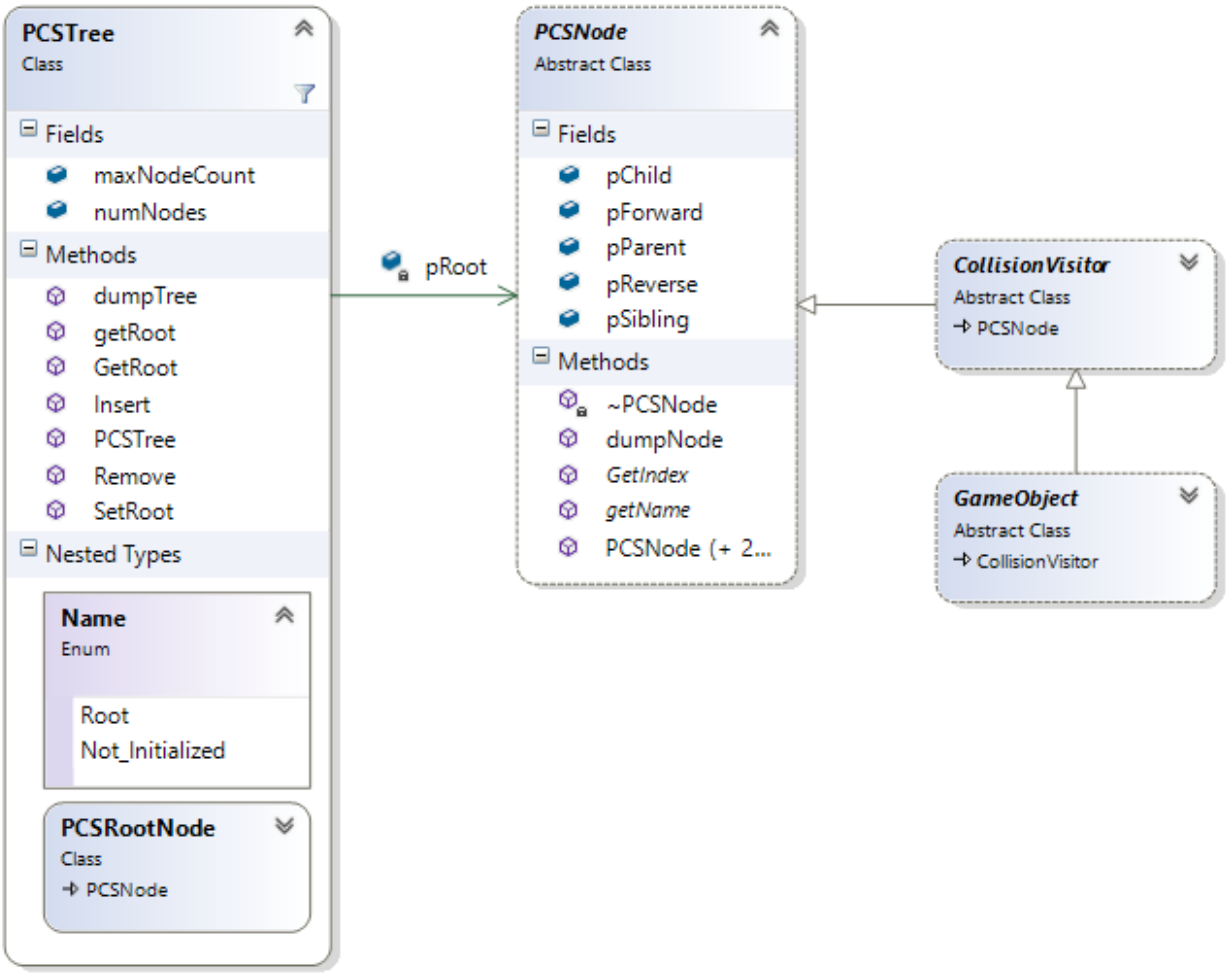

Figure 15 - Composite Design Pattern via PCS Tree

Conclusion

Overall, the design process for creating a *Space Invaders* clone requires the use of several design patterns, each solving unique developmental problems. Adhering to basic Object Oriented design principles, our game can be robust, extensible, and performance light. There is always room for improvement. In our case, we could have used a memento pattern to reset the game whenever the player dies and starts a new game. Refactoring is a continuing process and finding every bug is impossible. However, we can reduce the effort by making sure our program is structurally sound with the use of design patterns and following basic OO principles.