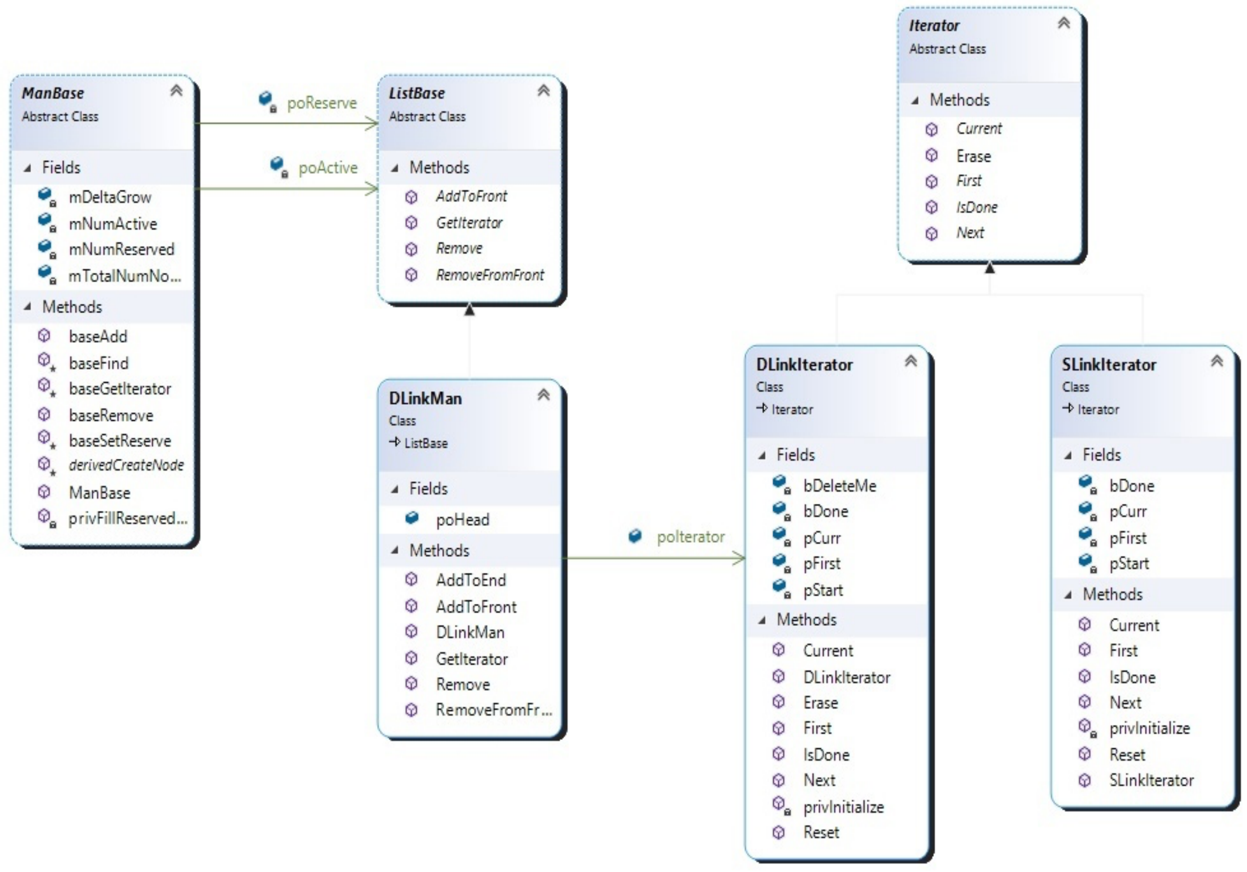
Artem Kirienko

Project Space Invaders
Engineering Design Documentation

Patterns Used in Project Space Invaders

0) Iterator Pattern
1) Strategy Pattern
2) Factory Pattern
3) Proxy Pattern
4) State Pattern
5) Visitor Pattern
6) Observer Pattern
7) Command Pattern
8) Composite Pattern
9) Object Pooling Method

Iterator Pattern



Motivation

In this project, we need to easily access elements of different collections (Linked Lists, Trees etc.) in sequential order. Most importantly, to enforce the idea of abstraction, we needed a way to access elements not only sequentially and easily, but also without revealing the internal behavior of the collection. Thus, we needed mechanics to be able to iterate through objects/elements in the way described above.

Intent

The point of iterator pattern is to do dirty works for us and internally access and pass objects of a collection and put it into an object, which keeps current object, first object and is able to find next elements and identify if a collection has been iterated trough to the end. A general-purpose component that enables collection abstraction and easy access outside of that collection.

Object-Oriented Working Mechanics

Iterator pattern provides universal interface for any type of collection. In this project, functionality of our iterator consists of 4 methods: first(), isDone(), next() and current(). Thus, we have an abstract base class which holds those methods as abstract. We derive concrete iterators from that class, thus forcing them to implement the 4 must-have methods.

We keep a pointer to the iterator in a data structure. DLinkMan for instance. Then, when we need to call the iterator, we call GetIterator() method on a specific DLinkMan and pass it the head of this DLinkMan. Them we can call other iterator methods to sequentially access elements of a linked list very easily. Iterator methods return an object of type NodeBase, which we then can cast down to DLink or Slink as we want.

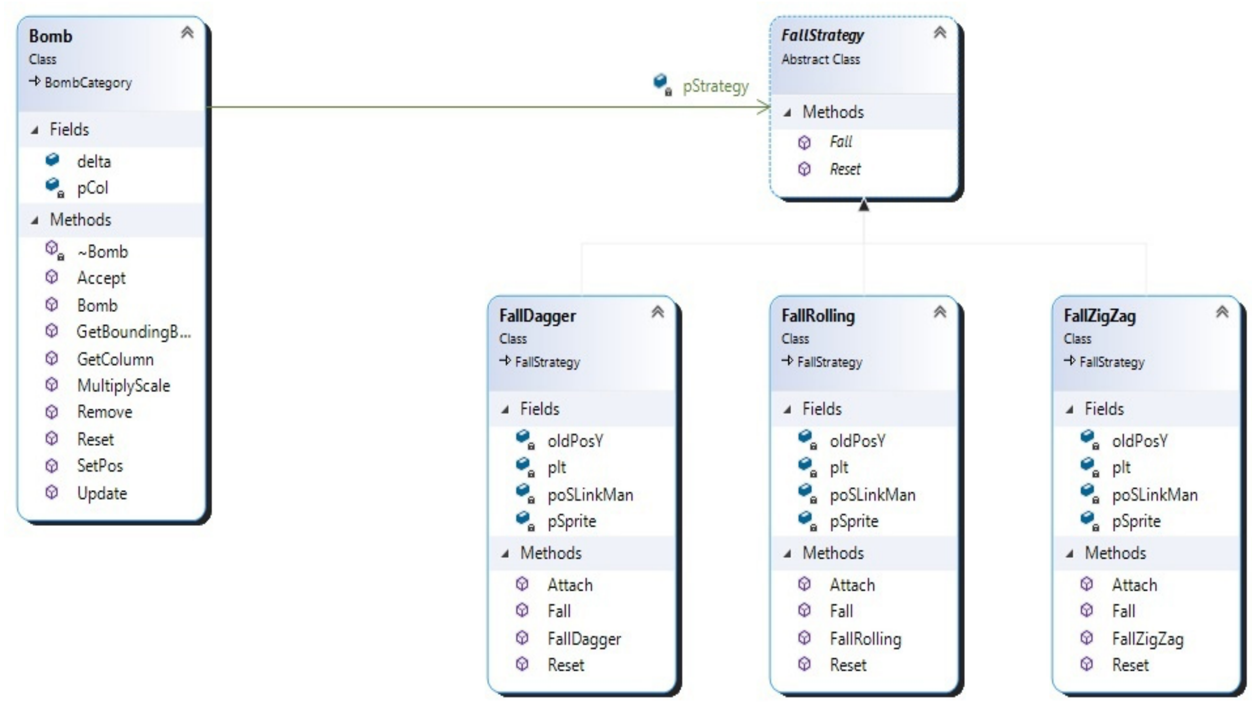
Discussion

This specific design is being used to iterate trough the nodes of data structures we use in our project. Those are Doubly Linked List, Single Linked List and a Tree (which is really a composite pattern, but that is discussed later in this paper). We often need to sequentially access nodes of those structures almost everywhere in our project. We need to traverse images in sequential order, collision objects sprites etc.

If we didn’t use that pattern, that would have meant we needed to access the nodes directly, which would mean breaking encapsulation. We also would have had a lot more difficult job accessing them in our Trees. Tree is a complex non-linear data structure, and we needed to access node siblings, children and parents, in addition to usual next, first etc.

I would say that the greatest problem with this pattern was to iterate through a tree(Composite), as it is a complex and non-linear data structure. It was hard to figure out the sequence in which we should iterate the Tree and also the way we access siblings, parents or children.

Strategy Pattern



Motivation

For this project, we needed aliens to drop different types of bombs in a random fashion. While falling algorithms for all bombs were the same, the animations were not. Thus, we needed different strategies for the falling bombs. In this project, we had 3 strategies for bombs called FallDagger, FallRolling and FallZigZag.

Intent

The point of strategy pattern is to use a specific strategy in the same context. In this project, when we drop a bomb, we need to randomly assign it a fall strategy, which could be either FallDagger, FallRolling or FallZigZag. Then, when we call bomb.fall(), it automagically calls the very fall() method we need for that specific strategy.

Object-Oriented Working Mechanics

We have an abstract class FallStartegy, which has Fall() and Reset() abstract methods, forcing derived classes to implement those. We derive 3 classes from FallStrategy: FallDagger, FallRolling and FallZigZag. Then, inside of a Bomb class, we have a data field of type FallStrategy to determine what strategy to use for a specific bomb.

We create bombs in BombSpawnEvent, where we randomly determine the type of a bomb and thus the appropriate strategy to use for that bomb. Depending on what strategy has been randomly selected, we create and set an object of the appropriate strategy class and pass it down to Bomb in the constructor used to create the actual bomb.

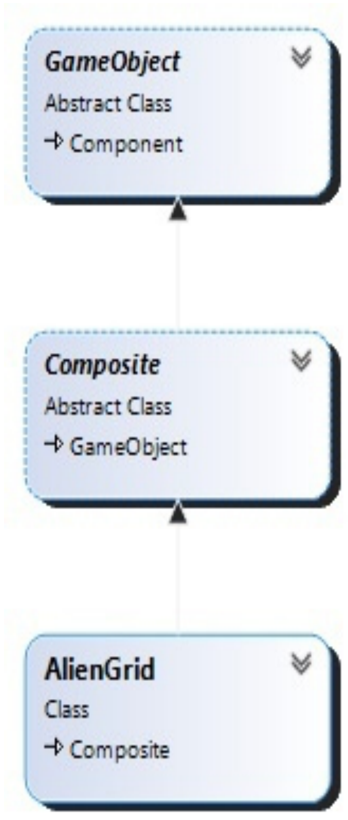
Finally, in the Update() method of Bomb class we call pStrategy.Fall(this). The call to Fall on a member field pStrategy calls an appropriate fall method, depending of the type of pStrategy, and the bomb automagically falls the way we assigned to it on creation.

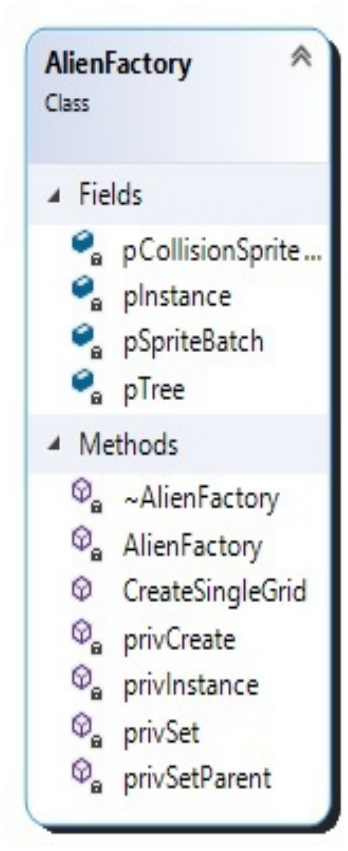
Discussion

In our case, strategies differ only in the images that are being swapped to animate bombs. Thus, we attach appropriate images to each strategy on its creation and then pass it down to bomb. Then, when Fall() method is called in Bomb's update(), it jumps to an appropriate strategy class and executes its Fall(), which iterates trough images appropriate to this strategy and swaps them to animate.

If we hadn't used that pattern, we would have stored 3 different lists of images in one Bomb and select from the three lists, which would only add if statements to our project. And we wanted to avoid if checks as much as possible. We also would have stored some kind of flag of what strategy we were using for a specific bomb, that would have added even more if statements. Thus, we eliminated unnecessary checks.

Factory Pattern





Motivation

We needed a way to create objects without exposing internals to the client, which in this case is the initialize function in ScenePlay. The whole project is based on abstraction, so we need to maintain it everywhere in the project. We also needed to create objects of the same type (GameObject), but with different specs and parameters. Including, Alien and Shield grids. Here lies the need for a Factory Pattern, which can create objects internally (enforcing abstraction) and in a single function call.

Intent

Factory creates a new object, with provided parameters and return that object through a common interface (casting to an object base abstract class). Client uses that object without being any the wiser of its concrete implementation.

Object-Oriented Working Mechanics

We have a regular Factory class, AlienFactory for instance, which has a method create(). In the case of AlienFactory class, we have a method CreateSingleGrid(), which creates a complete grid of aliens, including its tree (Composite) structure, column objects, and actual aliens with appropriate parameters (Images, position etc.).

It creates 11 columns of aliens and outs them together in a grid, keeping a tree member field. CreateSingleGrid() method calls another method for every object creation (Alien, Column or Grid). The method being called is privCreate(), where the actual creation happens. We pass required parameters to it, and it returns a newly created GameObject of appropriate type. We use new to create an object, and pass parameters (Sprite, position) into object constructor. Then we add it to a tree which is kept in the same AlienFactory class, activate its sprites and return GameObject to CreateSingleGrid() method.

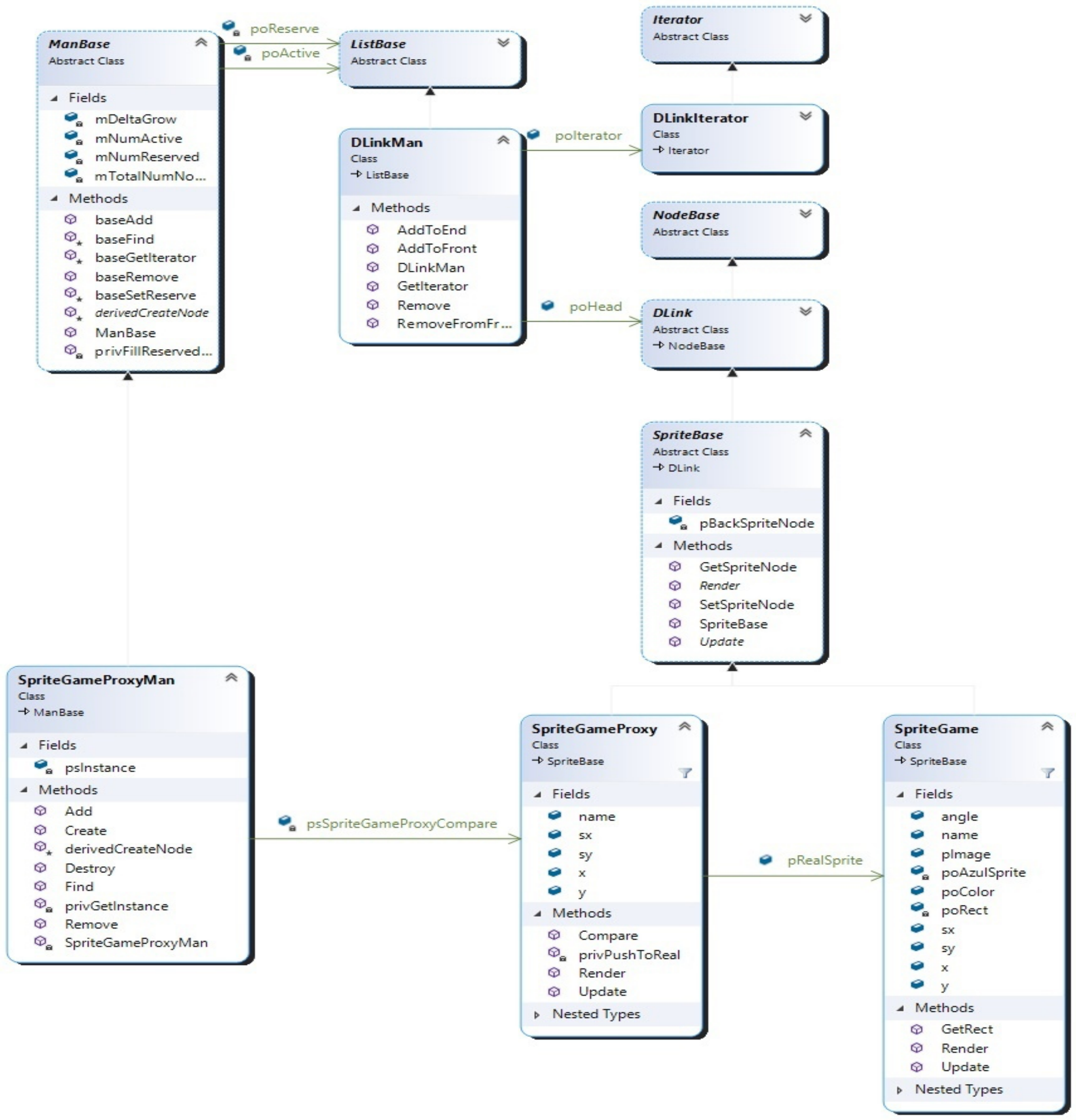
When all the columns are created, we return to the client a GameObject, which then could be cast down to AlienGrid. The same mechanics work for ShieldGrid.

Discussion

This design pattern is used to create a grid of aliens and shields, enforcing abstraction, because we can cast the return object to its abstract type. It is beneficial, as for each object we do not modify the factory, and we don’t change a single line of code in the client (ScenePlay). The client uses the product returned by the factory casted to an abstract type, without knowing its concrete type.

Not using this pattern, would have meant us writing the code in the create function everywhere it is called directly and thus exposing objects’ internals to the client (ScenePlay in this case), which would kill abstraction.

Proxy Pattern



Motivation

Sometimes we only need a few methods or data fields of a costly object. For instance, in this project we needed to draw the same sprites on the screen varying only in their position coordinates.

When we draw the same alien 11 or 22 times, we shouldn't create 11 or 22 different sprites. This increases the number of calls to new as well as space for storing those sprites. Same thing with shields, ships etc. Here comes the need for the proxy pattern.

Intent

The intent of this pattern is to create a placeholder for an object to be able to reference it and use only its small parts. To draw 55 aliens, we only need to draw the same sprite 55 times with different coordinates.

Object-Oriented Working Mechanics

We have a SpriteGameProxy class, which acts as a placeholder for a single unique real sprite. It holds a reference to a real sprite (pRealSprite in the diagram) and uses that reference to change x and y coordinates of the real game sprite.

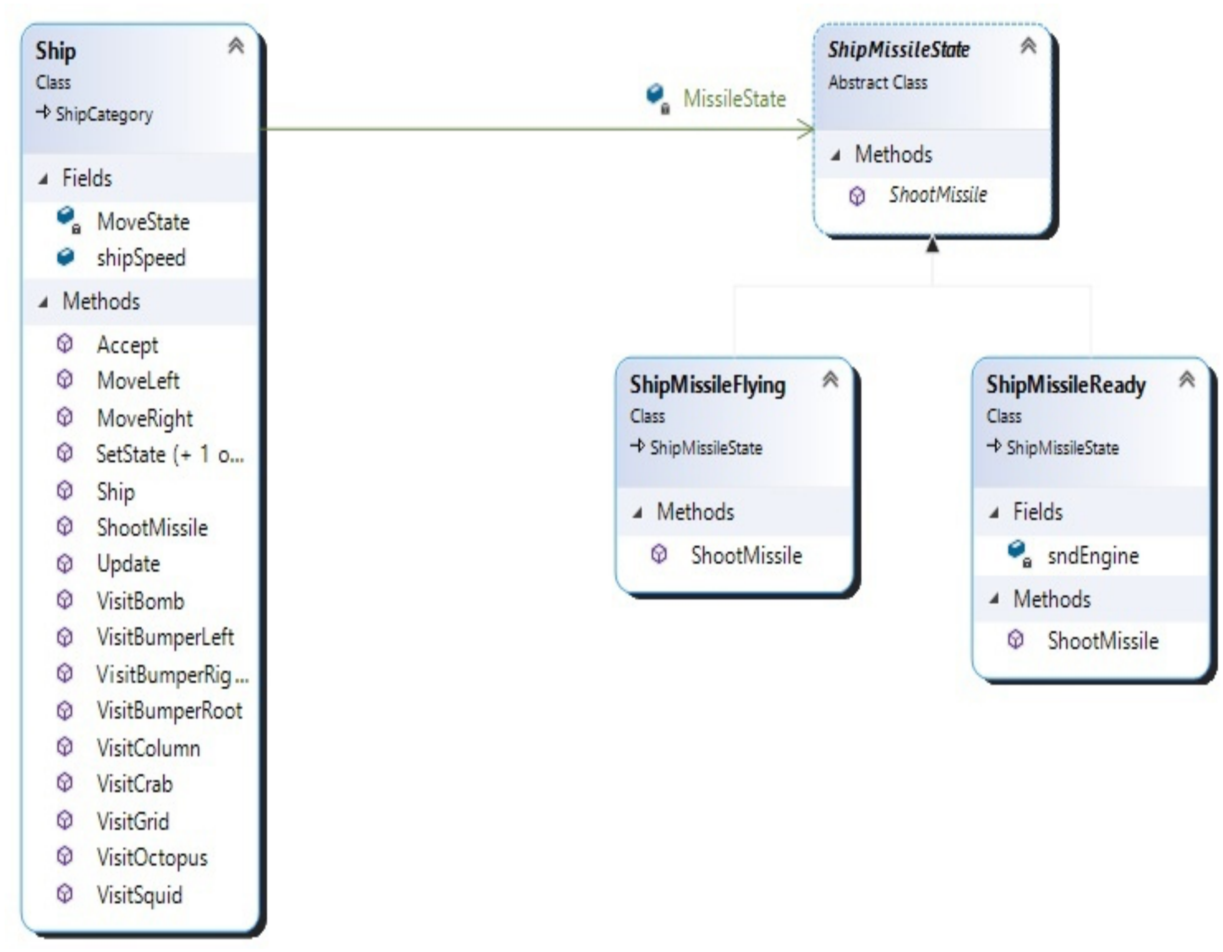
All our aliens are actually proxies. When update() or draw() methods are called on the proxies, x and y values of that proxy are "pushed" to a real sprite to which proxy points to by means of privPushToReal() method, then we update or render the real sprite with x and y we just pushed to it. Thus, we draw the same sprite for squid 11 times, crab 22 times and octopus 22 times using proxies.

Discussion

This pattern is used to get rid of unnecessary news and multiple sprites which are the same. We now use only one sprite and draw it as many times as we want just replacing x and y.

If we hadn't used that pattern, it would have meant creating 55 identical sprites, 54 additional news, which is bad for real-time programming. It also would have meant storing 54 additional sprites and accessing them by their index, which is a lot of tedious and redundant work.

State Pattern



Motivation

In project Space Invaders, a ship can shoot only one missile at a time. While missile is flying, that is until it hits an alien, shield, bomb or top wall, ship cannot shoot another missile. While the player can shoot a missile by pressing SPACE key, imagine if the player hits SPACE key while the previous missile is still flying (or active). Now we need some mechanism to control that only one missile could be on the screen at a time.

Intent

The intent of this pattern would, in this case, be to prevent multiple missiles appearing on the screen at the same time. Another thing to note is that this pattern prevents us to use multiple if statements.

Object-Oriented Working Mechanics

As always, we have an abstract class ShipMissileState, with two classes deriving from it. Those are ShipMissileFlying and ShipMissileReady classes. Inside of Ship class we have a reference to ShipMissileState class, which is used to shoot a missile. We set the state of a ship assigning an appropriate state object to MissileState data field of Ship class. Then, when a ShootMissile() method is called on SPACE key press, we call another ShootMissile() method on this data field, which jumps to the appropriate class. If state is Flying, it will jump to ShipMissileFlying class and call its ShootMissile() method, otherwise is calls the one in ShipMissileReady class. ShootMissile() method in Flying state class is empty (because we cannot shoot a missile while the previous one is still active), which is precisely how we force only one missile to be on the screen at a time.

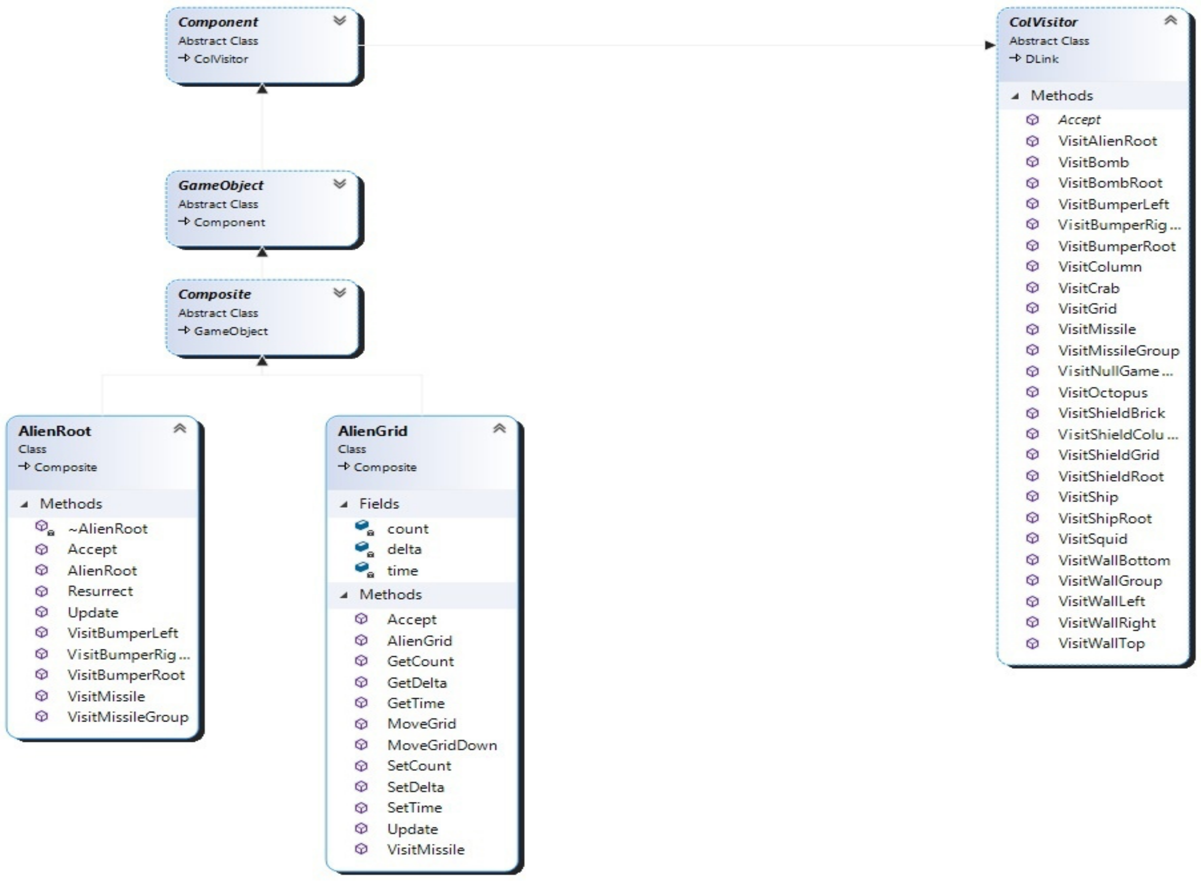
To sum up, the missile is in exactly one of two states (state classes) at any moment in the game. At SPACE key press we call exactly one of the two ShootMissile() methods depending on missile state. One of them is empty, which means we still call it, but it does nothing. Another one has the algorithm to actually shoot a missile.

Discussion

This pattern is used, as described above, to shoot a missile and most importantly, shoot one missile at a time. This pattern also uses abstraction like others which is the key to success in this project.

If we hadn't used that pattern, we would have arranged multiple if statements and also break the abstraction as Ship shouldn't know anything about how missile is being shot.

Visitor Pattern



Motivation

In this project we have different types of objects on which we need to perform certain operations. We have aliens, shields, missiles, bombs, ship etc. All these objects interact with one another in different ways. We need to check bombs vs ship, missile vs Aliens, aliens vs walls, bombs vs shields, missiles vs shields and so fourth. As we need to constantly check those collisions, thus performing collision operations on different types of objects, we need a mechanism to introduce and control such operations and to not change objects classes to perform them. Here comes the need for a visitor pattern.

Intent

Visitor pattern is supposed to introduce and control new operations on different types of objects, without changing classes of elements on which they operate. It also provides control over who collided with whom, as both elements have the accept() method implemented.

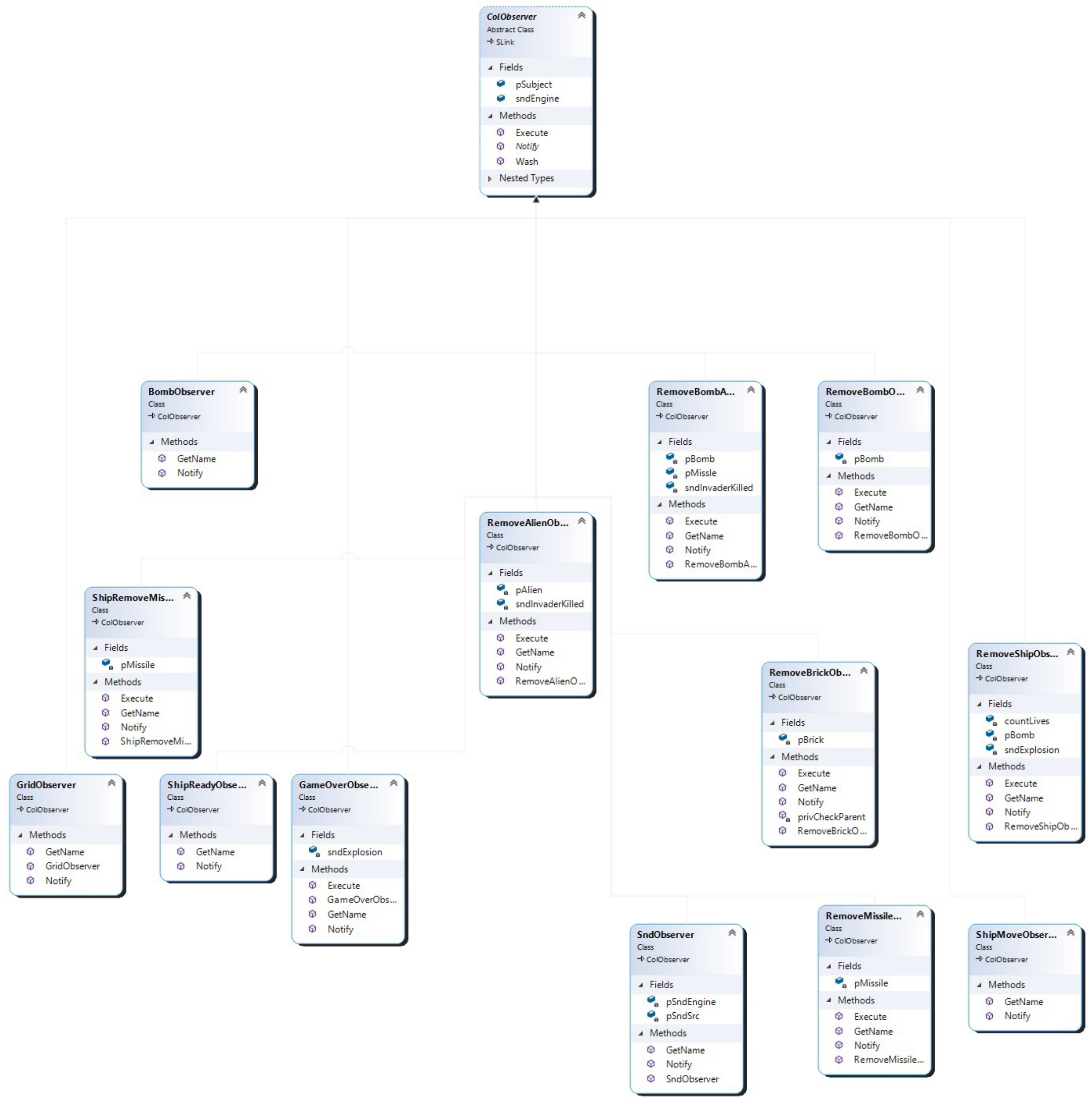
Object-Oriented Working Mechanics

This pattern has quite difficult mechanics. First, we have an abstract class ColVisitor, which introduces virtual visit methods and an abstract accept(). We make visit methods virtual, not to force derived classes to implement them as many of those wouldn't be needed in a specific class. This is also another way of debugging the project by putting asserts inside virtual implementations to catch an error if we are doing something wrong. We make the accept() method is abstract, because every derived class must implement it. We create objects of a class ColPair, which introduces a new collision operation on two roots of GameObjects and add it to ColPairMan. It in turn, processes those operations constantly and does the check for actual collisions. When two objects collide, an accept() method of object is called. At this point we have two objects that collided; thus we have to call the appropriate reaction to this event. We then call an appropriate visit method of the second object, passing to it the first one. Jumping in this method, we then perform appropriate operation.

Discussion

This pattern is being used to check collisions on different objects' trees. We add different pairs of GameObjects trees to ColPairMan and repeatedly check for them to collide or not. As our trees are complicated structures, we equip a smart and time saving approach. We check trees one node at a time starting from the root, which gives us the opportunity to do the early out, where we don't have to check whether alien column collided with a wall or not, because we already know AlienGrid has not, thus there is no chance of column colliding with a wall. We do that by having visit methods for roots, grids, columns (non-leaf composites in general). Thus, we go, node by node, level by level until we go down to the point where we have two objects that we want to collide (such as Alien vs Missile, Bomb vs Shield Brick, Bomb vs Ship etc.) and then, in that final visit method we do the actual collision and notify listeners to perform the operation needed (but that's next pattern). As for contrasting this approach to a non-design pattern solution, I can hardly point out that in the event we used another approach, it would have been hell to do all those checks by hand (and we have quite a few), refactoring classes completely, and tons of tedious work

Observer Pattern



Motivation

When we arrive at the point where two objects in a collision pair collided, we need to actually do something about it, perform appropriate reaction to this event. We need to inform objects that they have collided. Thus, comes the need for observer pattern.

Intent

The intention of this pattern is to inform the observable (in our case ColPair) that objects have collided and do something with this information. It also allows us to reduce dependencies of objects.

Object-Oriented Working Mechanics

We have an abstract class ColObserver with many concrete observers deriving from it. Abstract method notify() inside ColObserver class forces derived classes to implement it. In ColPair we keep a ColSubject object, which in turn keeps a linked list of observers attached to that ColSubject.

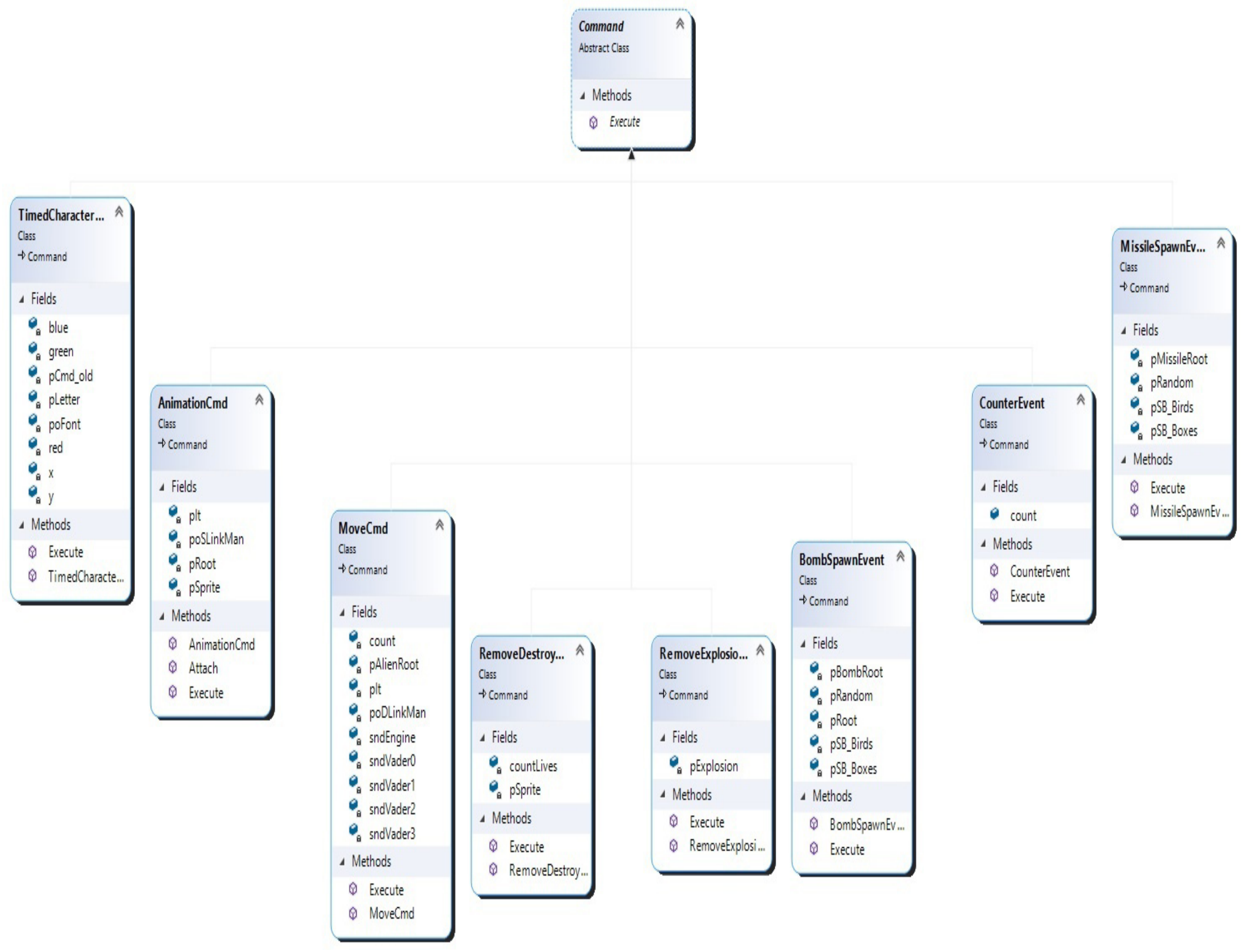
At the point where we have two objects collided in the last visit method for a pair, we call NotifyListeners() method, which calls notify() of every listener/observer that is attached to a collision pair. Then it does its job, which is the reaction to a collision. We have several observers for different types of collisions.

Discussion

While visitor pattern is used to check for collisions and introduce new operations on objects (collision), observer pattern is used to actually perform those operations. In ScenePlay class, when we create collision pairs, we attach every observer that performs an operation on this particular collision event. Then, when we set the collision between two concrete objects (Alien vs Missile), we notify the listeners/observers and they do all the work needed for this event (remove alien, remove missile, play sound etc.).

If we hadn't used that pattern, we would probably have to perform all operations inside visit methods, which would make our code messier and less maintainable, as we would have to do all the work inside visit methods. What if we had thousands of observers? (just to give an idea of how much code it would have been in a single method)

Command Pattern



Motivation

A lot of times in our project, we make requests for something to happen with certain time to happen at and different parameters. Alien animation is one example of this. We need to repeatedly execute one algorithm every half a second. We want to make it a special request to animate in 0.5 sec and animate again in another 0.5 sec and so on. The use of command pattern is highly beneficial here.

Intent

The intent of a command pattern is to encapsulate a request into an object and put it in a queue sorted by time (in this case) which executes when time comes for it to do so. It is also intended to support different requests in the same queue.

Object-Oriented Working Mechanics

We have an abstract class Command with abstract method execute(), which is used to execute a specific command. We also have many classes deriving from it, all implementing abstract execute() method as a contract.

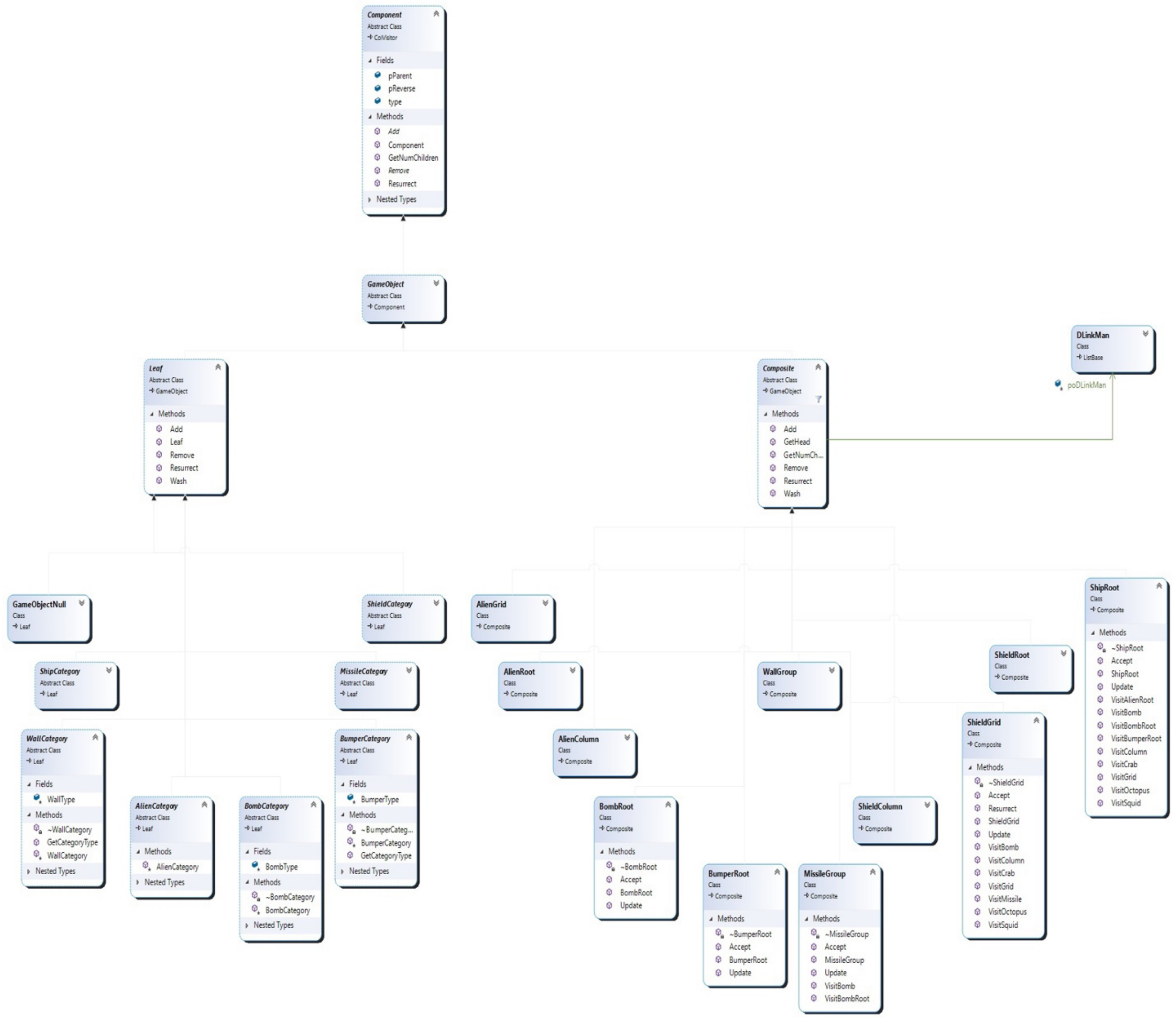
To get control over different commands, we create TimerEvent class, which is a representation of an event(command) to be executed at certain time. In addition, we have a TimerEventMan class managing TimerEvents and storing them in a queue(in our case a linked list). In some cases, when we need to reeat the command, after executing we add itself back to the queue.

Discussion

This pattern is being used a lot in our project. As we need to move aliens, animate them, print characters, spawn a bomb etc. Basically, everything that happens at a prearranged time, is encapsulated in a command. In timer event man update function, we only check those timer events in the queue, whose trigger time is less than or equal to current time, which gives us an early out, no need to check another 1 million nodes because we know they are in sorted order.

Well, if we hadn’t used that pattern, I suppose we would have set up some kind of flag to trigger events, and write all the code in ScenePlay, which is kind of our main class. Also, it would have been less maintainable.

Composite Pattern



Motivation

With hit boxes and collisions introduced to our project, we needed to maintain a hierarchy of objects. When we check for collisions, for example, we check the two roots first and then go down level by level, which gives us an early out. This hierarchy is basically a tree. So, we needed to construct objects in a tree. Composite pattern is the best one to use in this case.

Intent

The intent of a composite pattern is not only to construct a tree structure, but also to treat both branches and leaves uniformly, as they have many operations in common (check for collision, for instance).

Thus, composite construct a hierarchy that is part-whole, that is every component could be treated as part of tree structure or its own whole tree. Moreover, we need to easily add to and remove from a tree.

Object-Oriented Working Mechanics

Abstract class Component has a GameObject class as a child, from which we derive Leaf and Composite classes. Everything in our project that is a Leaf, that is alien, shield brick, missile, bomb, ship etc. is derived from the Leaf class, as they all have no children, they are the last in the hierarchy.

On the other hand, everything that has children or sub-structures in them, derive from the Composite class. Classes like AlienRoot, AlienColumn, BumperRoot, MissileGroup, BombRoot, ShieldRoot, WallGroup and more have substructures inside them, which are in turn split into another Leaf/Composite layout and so on until the leaf node has been reached that has no more children.

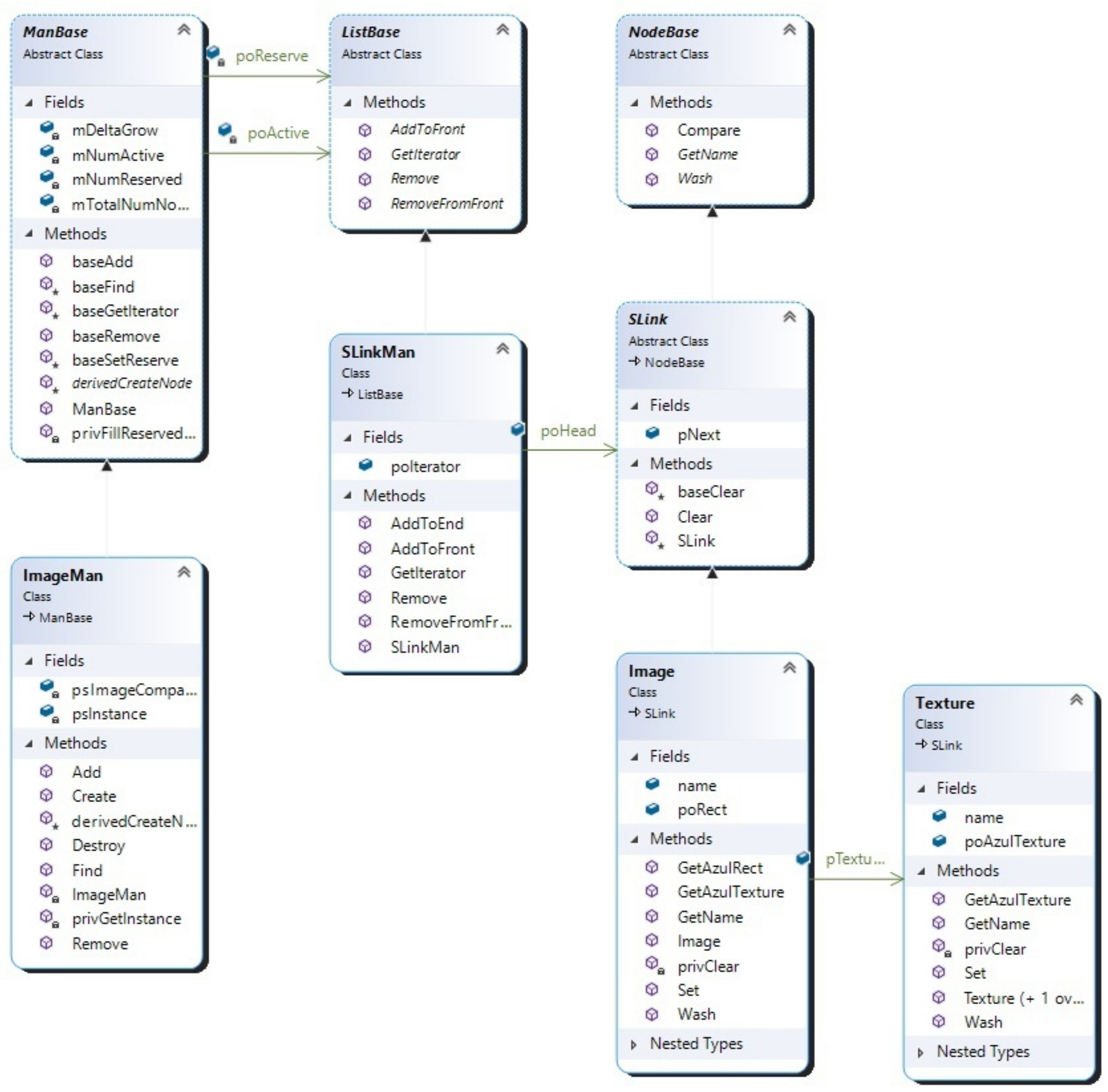
Composite class also has a linked list of its children stored in it. Thus, we can easily add to or remove from the tree.

Discussion

This pattern is being used for a collision system and makes it very easy to collide objects in a top-down approach, which is we collide trees first, if there is no collision between the two trees, then why do we need to check their children, it an early out.

If we hadn't used this pattern, it would have meant a lot of if statements and confusion with Leaf/Not-Leaf nodes, as we needed this early out for better performance. In fact, all of our design patterns shave off an enormous amount of ifs and confusion.

Object Pooling Method



Motivation

In real-time development, it is all about speed and performance. The most wanted speed-killer in the wild west is the new operator. When allocating objects on the heap, it is very time-consuming operation.

Also, C# is a language that has a garbage collector, which could at some point in our program take the memory back and release the object, if there are no references to it. From all said, we need a lifesaver in the name of Object Pooling Method.

Intent

The intent of Object Pooling Method is to reduce the number of calls to new in the first place. Also, it prevents C# garbage collector from releasing memory, because we always keep a reference to the object.

Object-Oriented Working Mechanics

In every manager that we have in our project, we keep two linked lists of objects. The first one, poActive, is a list with active objects that we are currently using. The second one, poReserve, is a list with uninitialized empty objects that are there until we need a new object in our active list.

Both lists are of one type ListBase, and in both lists we can add or remove. At the start of our game, we first prefill the reserve list with the number of objects specified in the create() function, and then take from it every time we need an object in our active list and add to the front of active.

Discussion

This method is used from a performance standpoint. With object pooling we reduce the number of calls to new, which is good as we save time, we also hold onto our objects, so they don’t get automagically release in the middle of the game.

Now, if we hadn’t used this method, we would have had a great decrease in speed, as number of news would go drastically up. We also, would have had a probability of objects being released early, and our game could have crashed.