Design Patterns

Object-Oriented Space Invaders in C# by Brian Keschinger

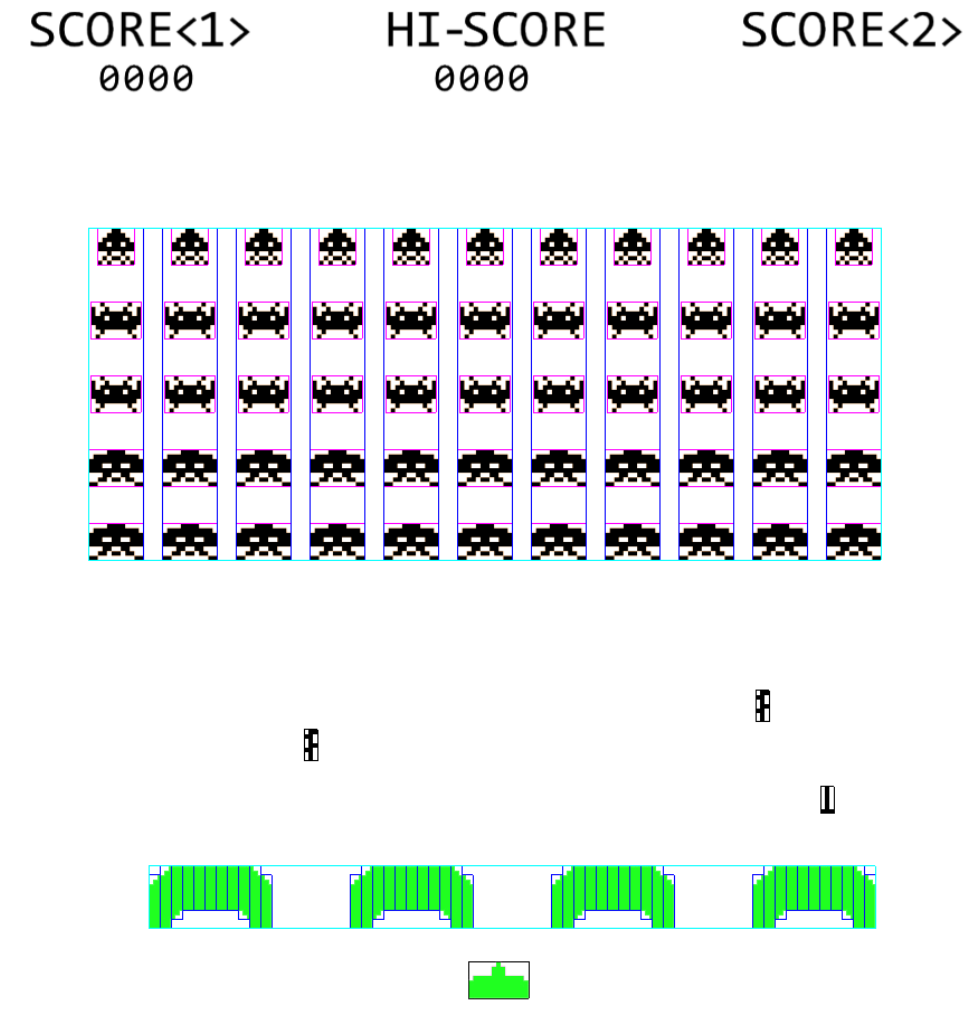
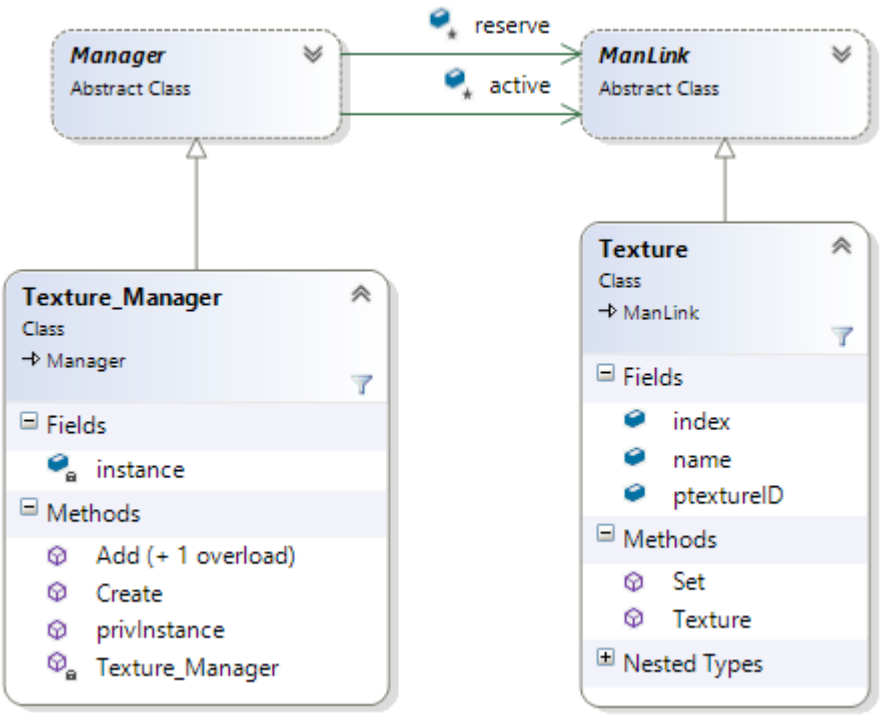


Table of Contents

Singleton



A Singleton class makes sure that there is only ever one instance of that class created. That instance is then able to be accessed globally through static methods. All of the managers in Space Invaders are Singletons to be able to manage their particular list of respective nodes.

In the above example, Texture_Manager is a Singleton, so there is only ever one Texture_Manager, which is stored in the instance variable. Our constructori s private so it can never be called outside of the class.

```
private Texture_Manager(int numReserve, int reserveGrow)
    : base(numReserve, reserveGrow)
{
    // Do nothing
}
```

To first use the Texture_Manager you're forced to call the Create() method.

```
public static void Create(int reserveNum = 3, int reserveGrow = 1)
{
    // Do the initialization
    if (instance == null)
    {
        instance = new Texture_Manager(reserveNum, reserveGrow);
    }

}
```

Notice that the Create() method is static so it can be called globally. If the Create() method is called more than once, it will simply do nothing because the instance has already been created.

Now whenever we want to use the Texture_Manager's instance, we use the privInstance() method.

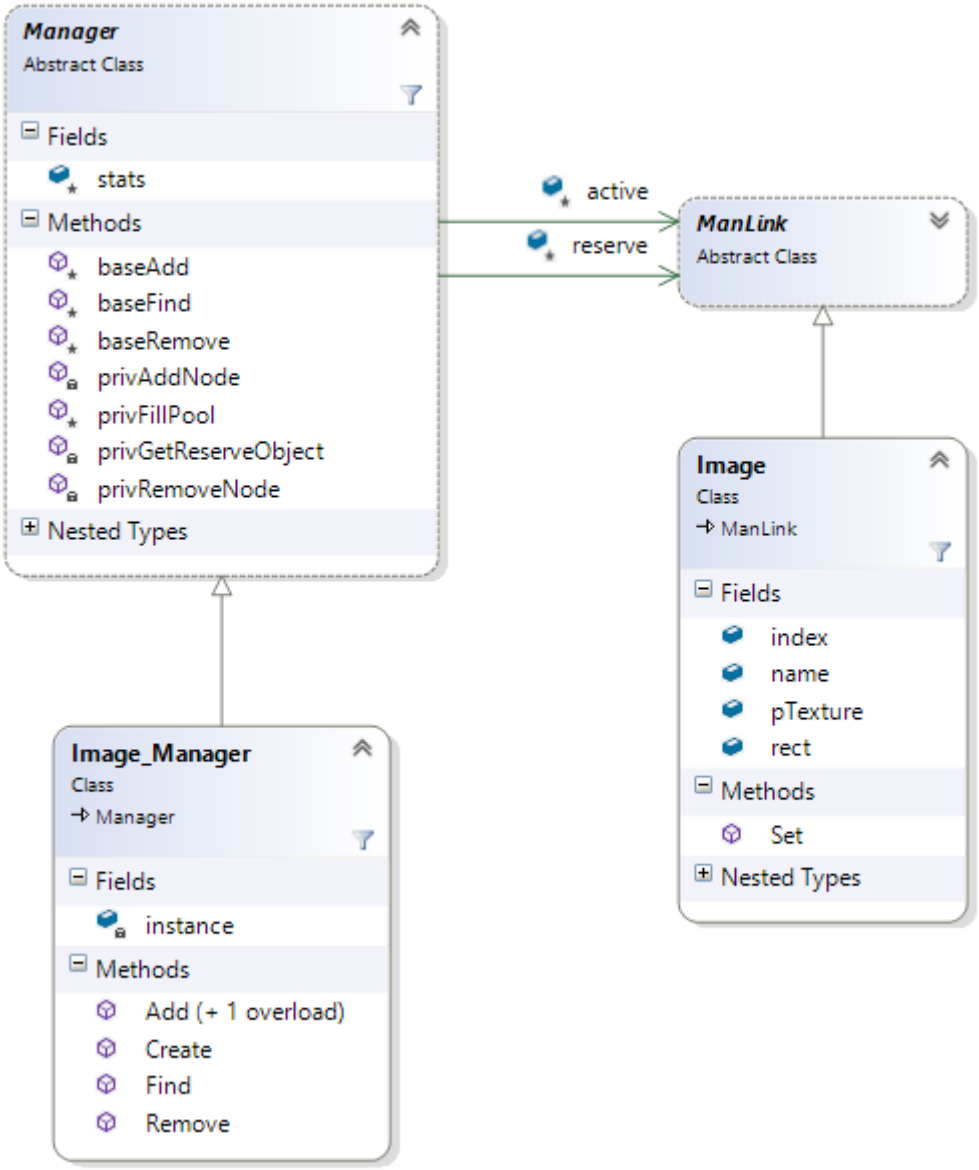
```
public static Texture_Manager privInstance()
{
    Debug.Assert(instance != null);

    return instance;
}
```

The privInstance() method will be used throughout the Texture_Manager to ensure we're only ever using that one instance of Texture_Manager.

For example, if we wanted to add a Texture to the Texture_Manager, we would call Texture_Manager.Add() with the proper parameters.

```
public static void Add(Texture.Name textureName, Index index, string fname)
{
    // Get the instance
    Texture_Manager tMan = Texture_Manager.privInstance();

    // Go to Man, get a node from reserve, add it to active, then return it
    Texture pTex = (Texture)tMan.baseAdd();

    // Initialize object
    pTex.Set(textureName, index, fname);
}
```

Manager



The Manager pattern is a way to manage nodes. With the Manager pattern you can do anything you want wiith the nodes globally (add, remove, update, etc.) as the Manager owns the nodes that are stored in a ManLink (double-linked list).

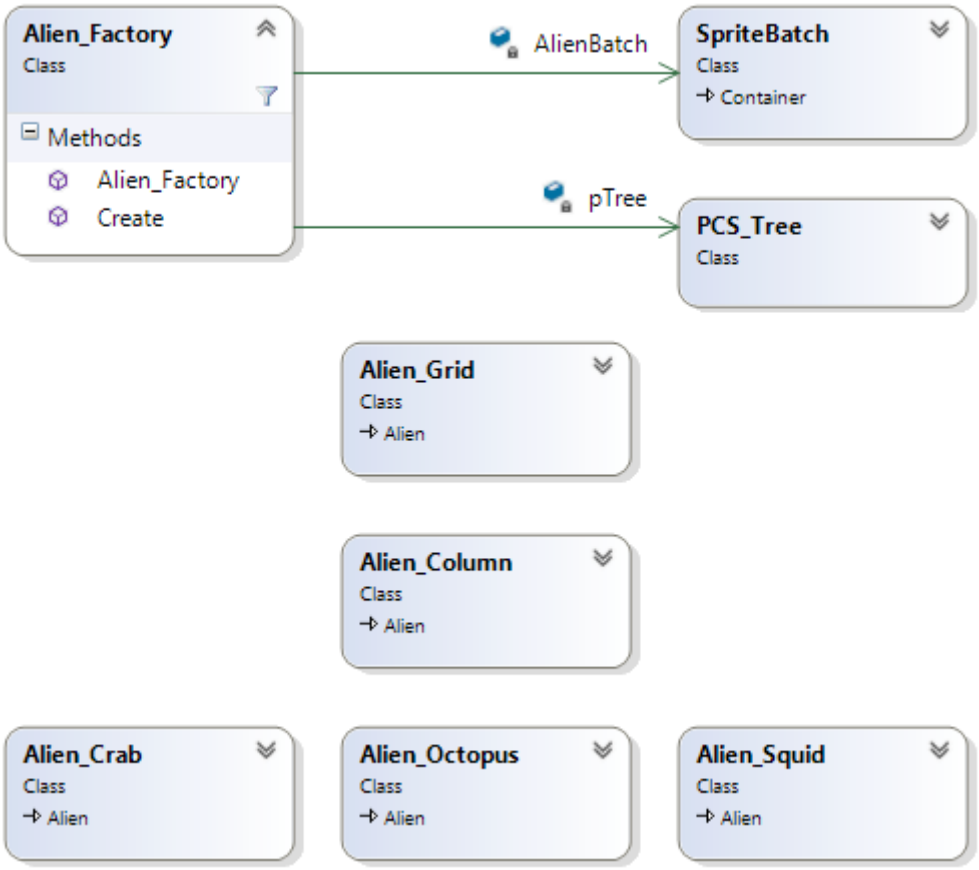
We saw before that the Manager is a singleton and you may have noticed that the instance is created using the numReserve and reserveGrow ints that are passed into the Manager's Create() function.

These ints are used to create two double-linked lists of nodes inside the Manager -- active and reserve. NumReserve is the number of nodes created and added to the reserve list initially. Then whenever you Add() to the Manager, it pulls a node from the reserve list and pushes it to the active list. If the reserve list gets emptied, then it is replenished with a reserveGrow amount of nodes. Whenever you Remove() from the Manager the node is pulled from the active list and put back onto the reserve list in order to recycle the nodes.

```
protected void privFillPool(int count)
{
    //**************************************
    //* This method fills the reserve list
    //**************************************

    // allocate Objects
    for (int i = 0; i < count; i++)
    {
        // create a new Object
        ManLink pObj = privGetNewObj();
        // move it to reserve list
        privAddNode(pObj, ref this.reserve);
    }
}

private ManLink privGetReserveObject()
{
    //*****************************************************
    //* This method fills reserve if == null
    //* then pops the head off the list and returns it
    //*****************************************************

    if (this.reserve == null)
    {
        // fill pool with more nodes
        this.privFillPool(this.stats.reserveGrow);
    }

    // Detach 1 from the reserve pool
    ManLink pManLink = this.privGetNodeFirstNode(ref this.reserve);

    return pManLink;
}
```

The Manager pattern does this to reduce the number of dynamic allocations during run time. If you set you numReserve high enough, you will buy all the memory you need up front and just continue to recycle those nodes.

Factory



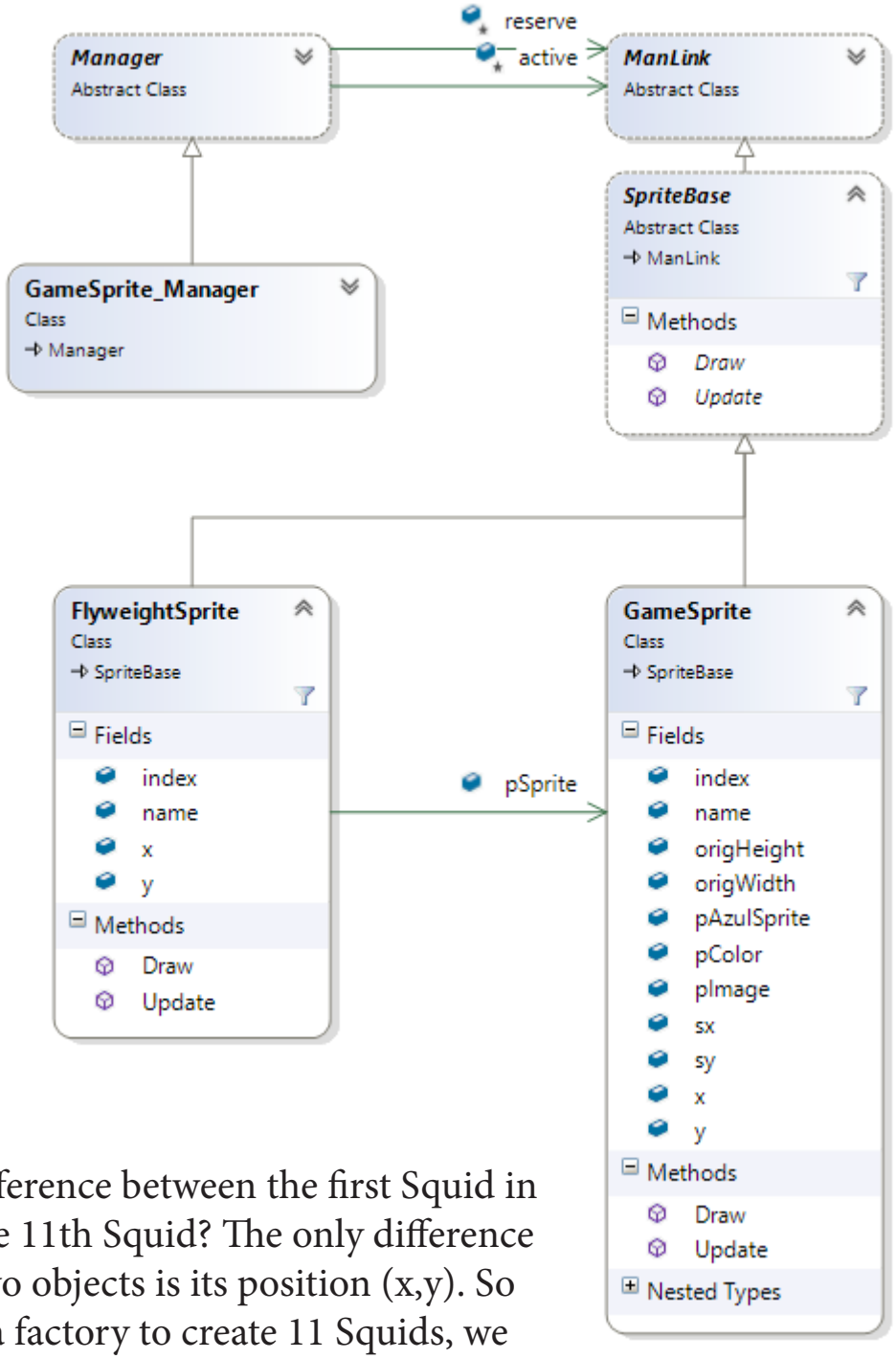
Alien_Factory creates an alien based on a passed in type, adds that alien to a PCS_Tree, and puts the alien on a SpriteBatch to be drawn. We defer the creation of the specific type of alien to the Create() method of the factory by passing it a type. By putting this Factory in a for loop, we create the entirety of our alien grid with hierarchy.

```
// Create the tree
PCS_Tree pAlienTree = new PCS_Tree();

// Create the factory and give it a tree to use
Alien_Factory factory = new Alien_Factory(pAlienTree);

// Create the grid container
Alien pGrid = factory.Create(Alien.Type.Grid, Index._0, 0.0f, 0.0f, 8.0f);

// Set the tree's root as the grid container
pAlienTree.setRoot(pGrid);

for (int i = 0; i < 11; i++)
{
    // Create a column whose parent is the grid container
    factory.setParent(pGrid);
    Alien pCol = factory.Create(Alien.Type.Column, Index._0 + i);

    // Create aliens whose parent is the column at a specific coordinate
    factory.setParent(pCol);
    factory.Create(Alien.Type.Squid, Index._0 + i, x * i, y);
    factory.Create(Alien.Type.Crab, Index._0 + i, x * i, y * 2);
    factory.Create(Alien.Type.Crab, Index._11 + i, x * i, y * 3);
    factory.Create(Alien.Type.Octopus, Index._0 + i, x * i, y * 4);
    factory.Create(Alien.Type.Octopus, Index._11 + i, x * i, y * 5);
}
```

The Create() method of the factory uses a Switch/Case statement to decide what kind of alien to create and then does all of the necesaary actions to get that object set up, drawing, and updating in the PCS_Tree.

```
case Alien.Type.Crab:
        //Create the alien
        pAlien = new Alien_Crab(GameObject.Name.Crab, _index, GameSprite.Name.Crab, _x, _y);

        //Set the collision box
        pAlien.pCollisionObject.pColSprite.Set(CollisionSprite.Name.Crab, _index, color, rect);

        // Attach the alien to the sprite batch to be drawn
        AlienBatch.Attach(pAlien.pFlyweight);

        // Insert the alien into the tree
        this.pTree.Insert(pAlien, this.pParent);
    break;
```

Flyweight and Proxy

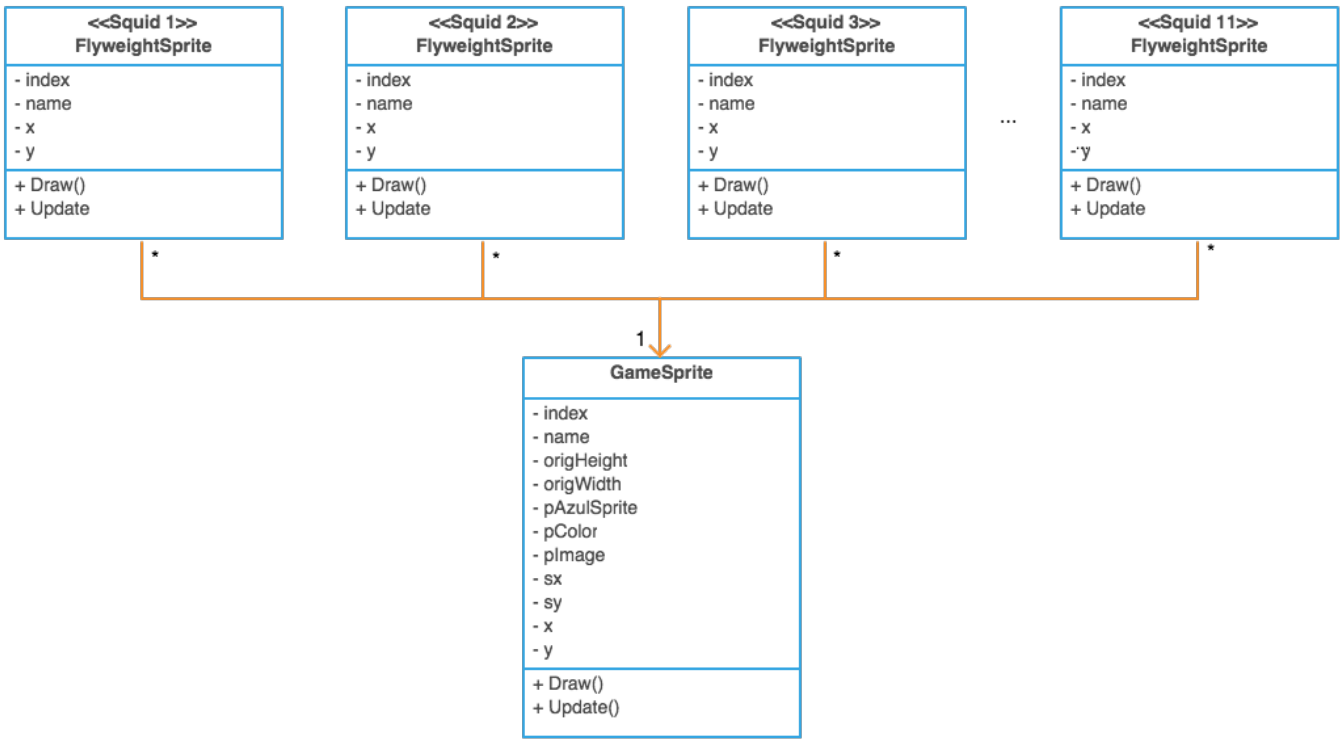


What is the difference between the first Squid in the row and the 11th Squid? The only difference between the two objects is its position (x,y). So when you use a factory to create 11 Squids, we don't need to use 11 full GameSprites. Instead we create 11 FlyweightSprites.

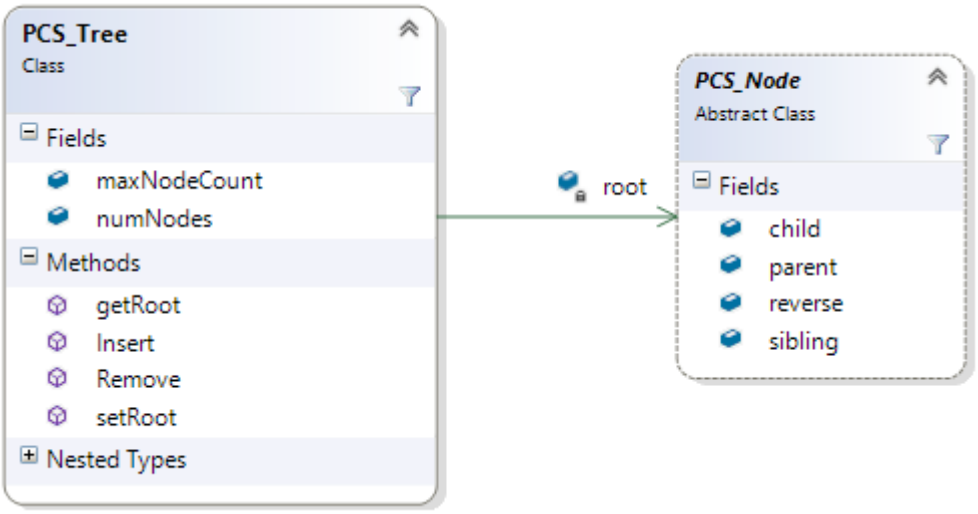
The FlyweightSprite is just a light version of the GameSprite that takes advantage of the objects only uniqueness: posiition. The FlyweightSprite, by contract of inheriting from SpriteBase, has its own Update() and Draw() that simply push data to and call the GameSprite. This way when the SpriteBase's Draw() and Update() are called, it handles the FlyweightSprite and the GameSprite the same way, except that that FlyweightSprite calls some GameSprite functions.

```
//*****************************************
//*** FlyweightSprite's Update() and Draw() ***
//*****************************************
public override void Update()
{
    pSprite.x = this.x;
    pSprite.y = this.y;
    pSprite.Update();
}

public override void Draw()
{
    pSprite.x = this.x;
    pSprite.y = this.y;
    pSprite.Draw();
}
```

The FlyweightSprite holds a reference to a GameSprite. This is the Proxy that allows the FlyweightSprite to call the full GameSprite's functions -- which has the remaining data needed to be drawn on screen. The FlyweightSprite pattern has a many-to-one relationship with the concrete GameSprite and allows for the reuse of common data.

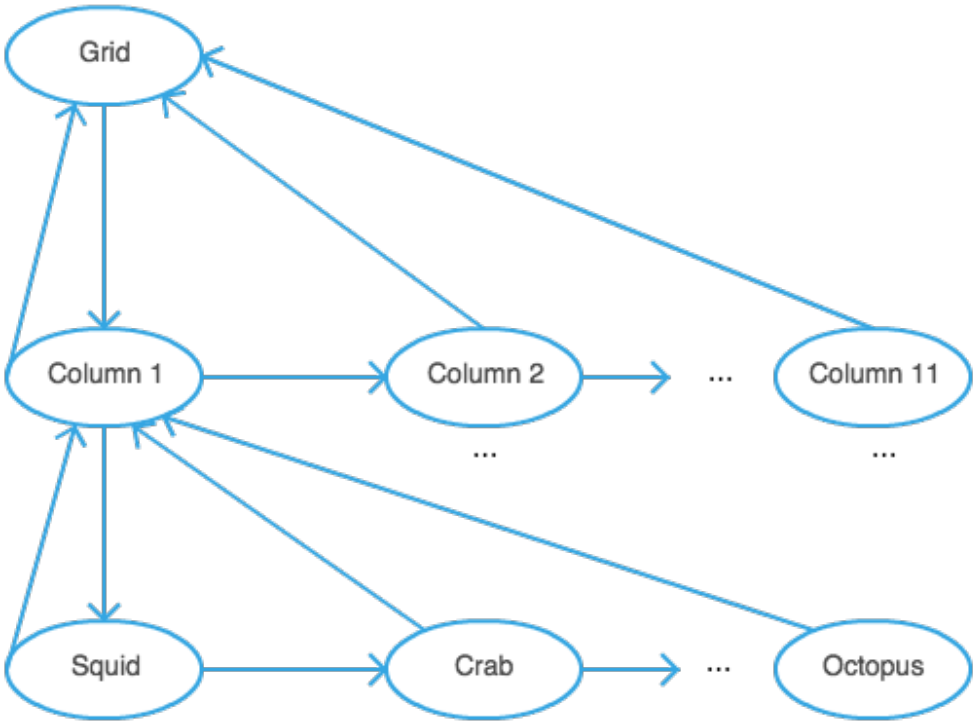


Composite



The Composite pattern is essential in Space Invaders. The way we update all of the aliens positions and collision boxes requires hierarchy, which is why we use the PCS_Tree as the Composite pattern. The PCS_Tree stands for Parent-Child-Sibling Tree, which is otherwise known as a LCRS Tree (Left Child, Right Sibling).

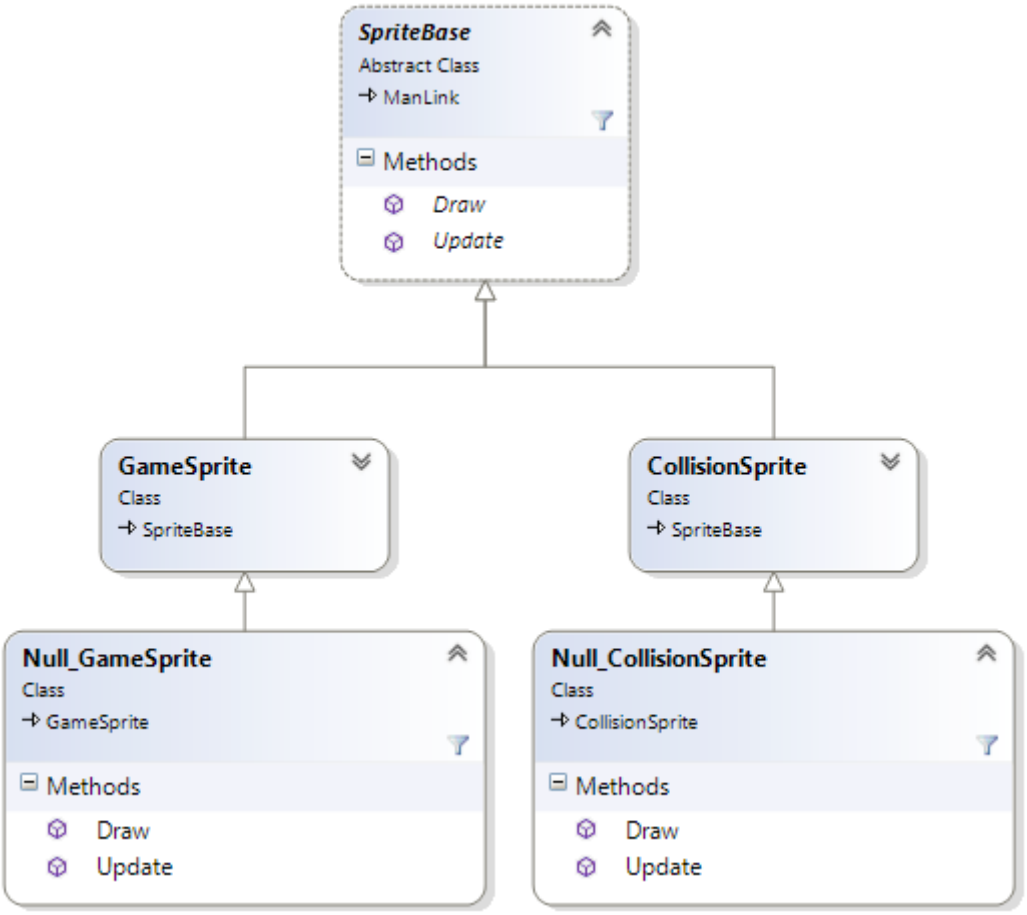
The PCS_Tree is structured like this: Each Parent can have one Child, each Child can have one Parent and one Sibling. When we want all objects of a single hierarchy we simply traverse the siblings.



This works really well for us as the Grid and Column collision boxes are dynamically calculated based on their Children. Each Column calculates all of its Children (the aliens) and creates a vertical bounding volume containing all of the aliens. The Grid then calculates all of its Children (the columns) and creats a bounding volume containing all of the columns (which contain the aliens).

The Grid's bounding volume is red.
The Column's bounding volumn is yellow.
And the Alien's bounding volume is green.

Null Object



Sometimes we want a GameSprite, but we don't want it to have a CollisionSprite. And sometimes we want the opposite: a CollisionSprite, but not a GameSprite. We want all objects to be treated the same and not have to have special cases when drawing.

For example; Our Grid has a CollisionSprite that has the volume of all of the Columns and Aliens, but this Grid doesn't have a GameSprite. So we create a Null_GameSprite that inherits from GameSprite whose Draw() simply does nothing because we don't want it to draw a Game-Sprite. Now the Grid is treated just like every other object and calls Draw() when it's required to do so, but it's Draw() simply does nothing when trying to draw a NullSprite.

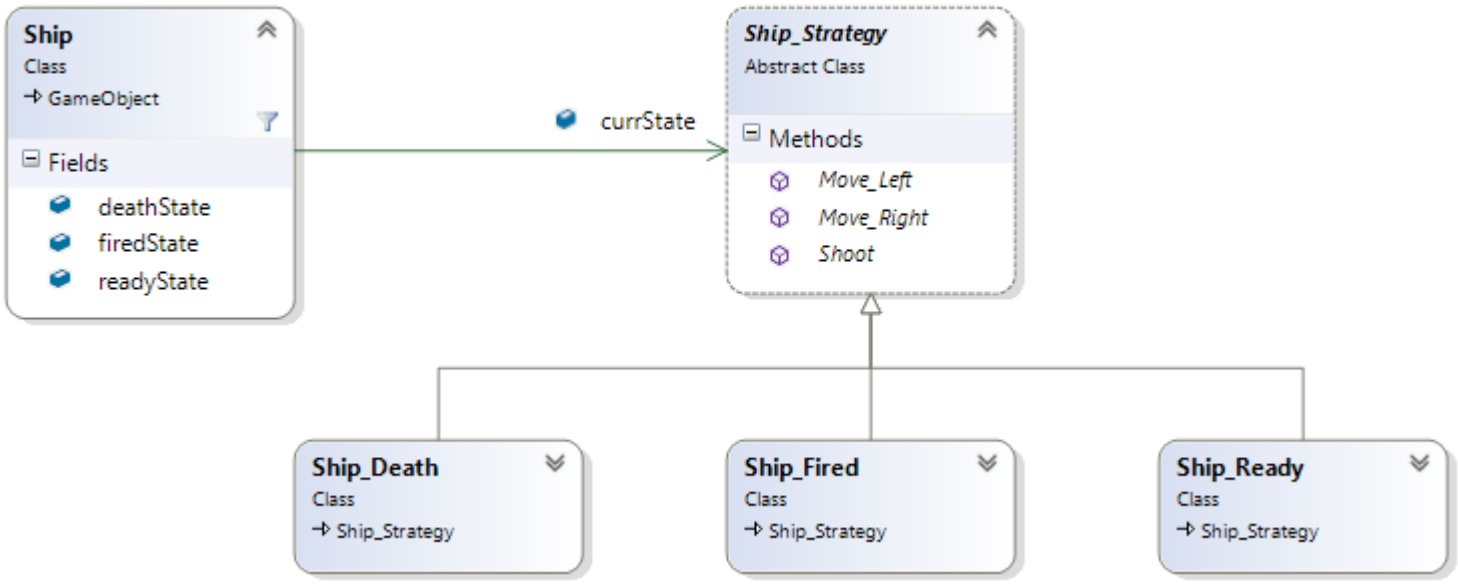
```
override public void Draw()
{
    // Do nothing - this is a nullSprite
}
```

To attach the NullSprite, we call Find() on the GameSprite_Manager with GameSprite.Name.NullObject as the passed in name.

```
public static GameSprite Find(GameSprite.Name name, Index index = Index._None)
{
    // Get the singleton
    GameSprite_Manager pGSManager = GameSprite_Manager.privInstance();

    GameSprite pGameSprite = null;

    // Special case for Null Object
    if (name == GameSprite.Name.NullObject)
    {
        // Return the stored NullSprite instead
        pGameSprite = (GameSprite)pGSManager.pNullSprite;
    }
    else
    {
        pGameSprite = (GameSprite)pGSManager.baseFind(name, index);
    }

    return pGameSprite;
}
```

This same pattern can be used when we don't want to draw a collision box by simply using the Null_CollisionSprite.

State



Our Ship can only fire one missile at a time. When the Ship gets hit by a bomb or an alien, it can't move. When a missile hasn't been fired, the ship can move and shoot.

These are all special case scenarios that we wan't to avoid and we do that by using the State pattern.

The Ship holds a reference to the current state and always performs its actions on that state. We swap the currState depending on a specific event within the game. All states implement the Move_Left(), Move_Right(), and Shoot() methods by contract of inheritance.

We store all of the states on the Ship itself because there are only four of them. If there were more, we'd want to implement some kind of data structure to access these.

When the Ship is created we set the currState to the readyState. The readyState is of class Ship_Ready whose Move_Left() and Move_Right() methods move left and right respectively, but the Shoot() method will fire a missile and swap the Ship's currState to firedState.

```
public override void Shoot()
{
    // Set the missile's location to the ship
    this.ship.missile.x = this.ship.x;
    this.ship.missile.y = this.ship.y;

    // Send the missile up
    this.ship.missile.fired = true;

    // Swap the state to fired
    this.ship.currState = this.ship.firedState;

    // Play sound!
    Sound sound = Sound_Manager.Find(Sound.Name.Missile_Fire);
    sound.Play();
}
```

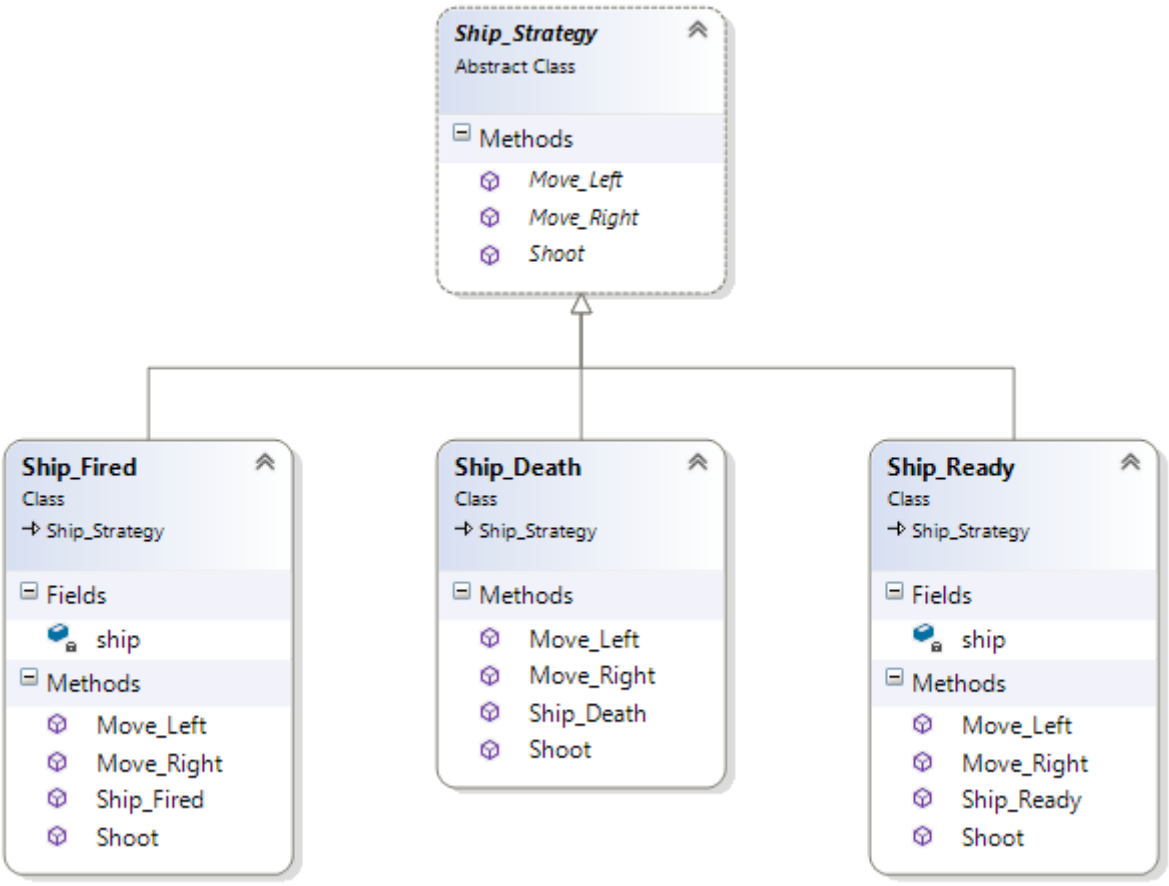
Now that the missile has been fired, we don't want the Ship to be able to shoot again until that missile has hit something (an alien, shield, wall, etc.). Since we swapped the currState from readyState to firedState, the Ship won't be able to fire another missile until the currState is swapped back to the readyState becase firedState's Shoot() function does nothing.

```
public override void Shoot()
{
    // Do nothing
    // State will be switched by observer on collision
}
```

The currState will get swapped back to readyState when the missile collides with something (an alien, shield, wall, etc.) by that Collision Observer, which we'll talk about soon.

When you press the spacebar we always call the currState.Shoot() method, but depending on the state that the Ship is in at that particular moment in the game, it may shoot or it may not. It's the same thing for Move_Left() and Move_Right().

Strategy



You may have noticed that our State pattern is used very closely with the Strategy pattern because the states are built using the Strategy pattern.

By contract of inheritance, each Strategy must implement the Move_Left(), Move_Right(), and Shoot() functions as they are abstract methods in the base class Ship_Strategy.

The Strategy pattern allows for each Strategy's overloaded methods to be of different shapes and sizes to perform different actions based on what is appropriate for that Strategy.

```
// Ship_Death's Move_Left()
public override void Move_Left()
{
    // Do nothing
}
```

```
// Ship_Ready's Move_Left()
public override void Move_Left()
{
    this.ship.x -= ship.move;
}
```

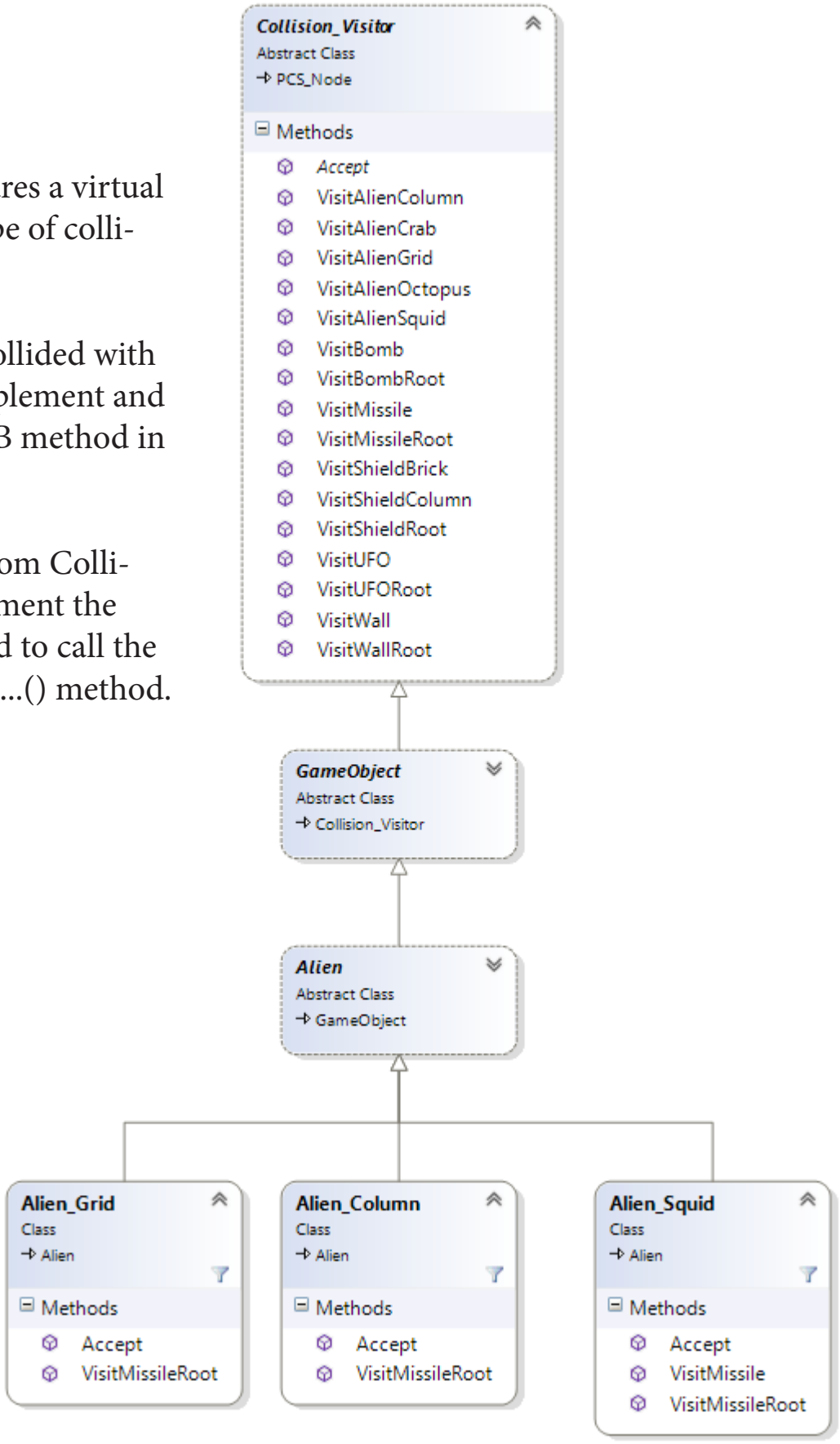
When the Ship is hit by a bomb, it enters a death animation. During that time we don't want the user to be able to move the Ship around the screen. That would look weird if the Ship is exploding and you're able to change its position. So the Move_Left() function of Ship_Death does nothing. Though if the Ship isn't dying, then the Move_Left subtracts a distance from the Ship's x position in the game space.

Visitor

The Visitor pattern declares a virtual visit method for each type of collidable object available.

If objectA is able to be collided with objectB, objectA will implement and override the VisitObjectB method in its class.

All classes that inherit from Collision_Visitor must implement the abstract Accept() method to call the appropriate object's Visit...() method.



Upon collision of Missile_Root and Alien_Grid, Missile_Root's Accept() method will be called with Alien_Grid being passed in as the Collision_Visitor visitor.

```
public override void Accept(Collision_Visitor _other)
{
    // We're in Missile_Root so we know it has collided with "_other"
    // Call the appropriate collision reaction
    _other.VisitMissileRoot(this);
}
```

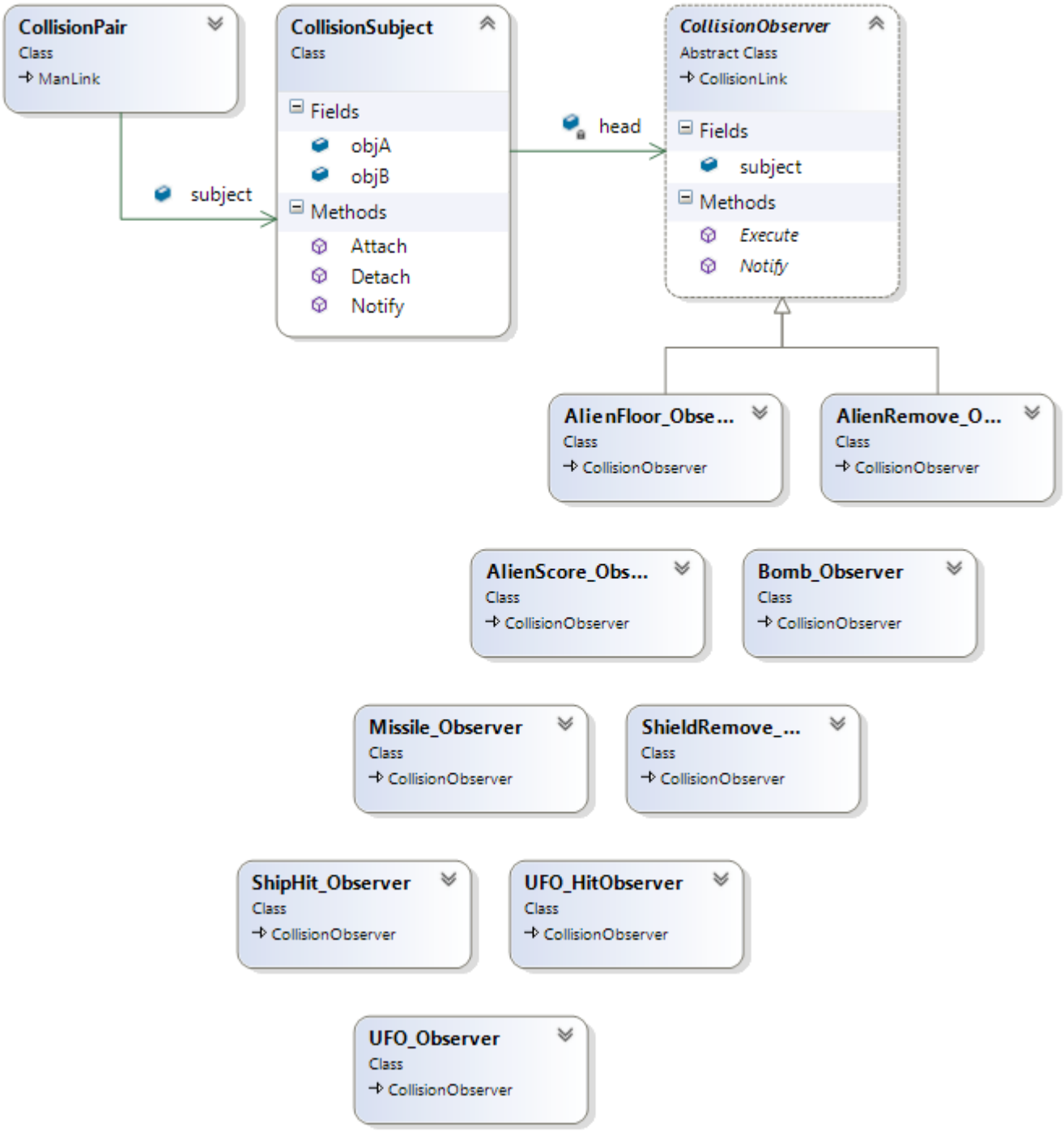
We then call the VisitMissileRoot() function on the Alien_Grid, while passing in the Missile_Root. The VisitMissileRoot() function inside of Alien_Grid will then check for a collision with Alien_Grid's child (Alien_Column).

```
public override void VisitMissileRoot(Missile_Root _m)
{
    // Call collide on MissileRoot vs Columns
    CollisionPair.Collide(_m, (GameObject)this.child);
}
```

If the Alien_Column and the Missile_Root collide then the VisitMissileRoot() function in Alien_Column will be called. It then does collision checks all the way down the hierarchy until we have a Missile colliding with an Alien_Squid. Alien_Squid's VisitMissile() function will then be called, which calls Notify() on the active collision pair's Subject, which ultimately fires off all of the attached Observers (it's coming, I swear) that perform actions when a tangible collision occurs.

```
public override void VisitMissile(Missile _m)
{
    // Get the active CollisionPair
    CollisionPair pColPair = CollisionPair_Manager.GetActiveCollisionPair();

    // Set the pair's collision
    pColPair.SetCollision(_m, this);

    // Notify all Observers (Listeners)
    pColPair.NotifyListeners();
}
```

Observer



Each CollisionPair that is set up has a Subject. The Subject holds a list of Observers and Subject.Notify() will call every attached Observer's Notify() and Execute(). These Observer's Execute() function is where we actually perform actions based on the collision (add score, trigger death animation, remove an alien, etc.)

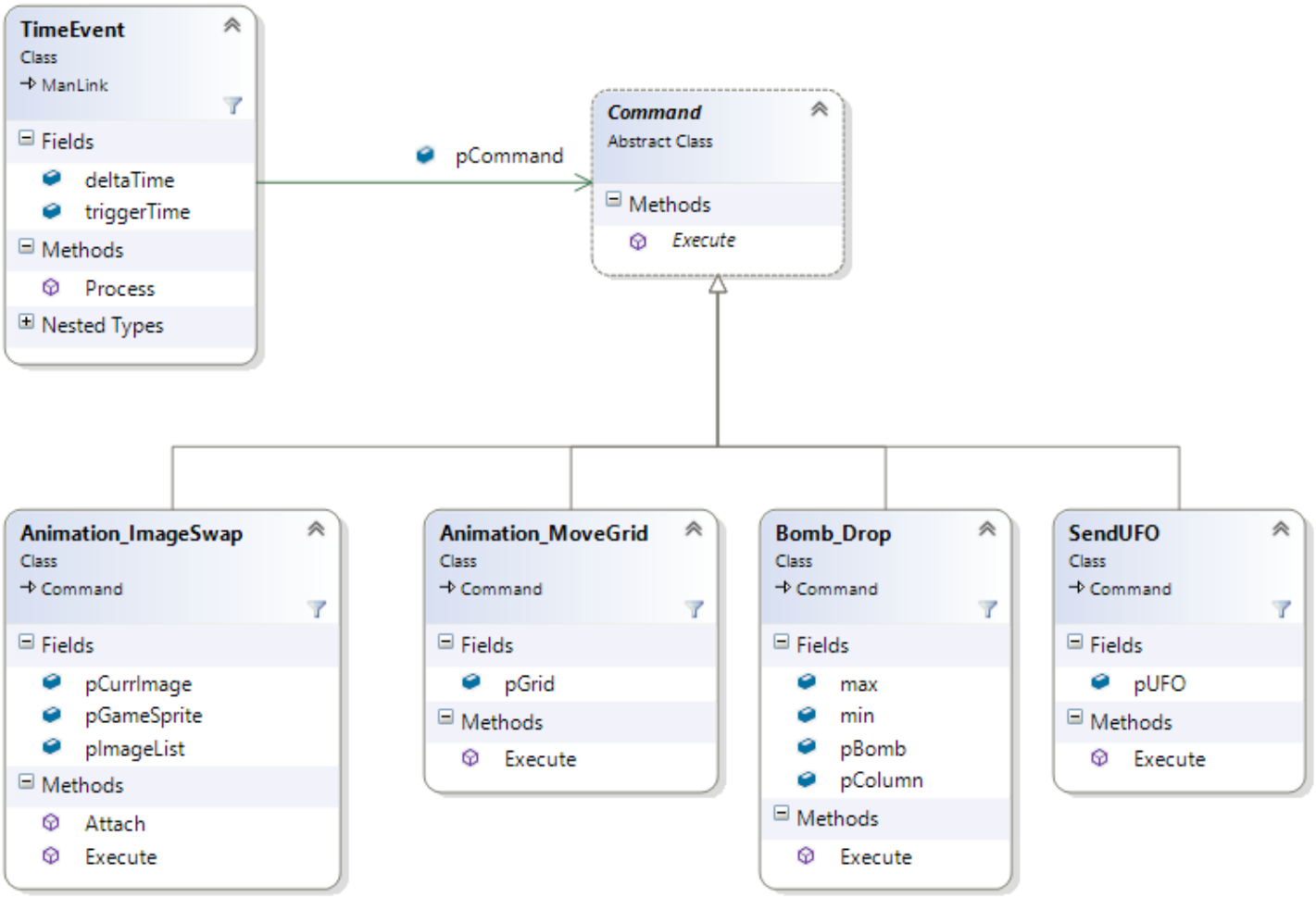
When we create a CollisionPair, we attach two trees that we check against each other. In our previous example, which I'll continue with, those trees were the Missile tree and the Alien (Grid) tree. We use the Visitor pattern to check the collisions all the way down in hierarchy to a specific Alien (Alien_Squid) colliding with the Missile. Inside of AlienSquid's VisitMissile() the active CollisionPair's Subject's Notify() method is called and that calls all of the attached Observer's Notify() and Execute().

```
public override void Execute()
{
    // Gets the Global variables
    Aliens_Remaining aliensNum = (Aliens_Remaining)Global_Manager.Find(Score.Name.Aliens_Remaining);
    Score score = (Score)Global_Manager.Find(Global.Name.Player_Score);

    // Decrease the number of Aliens by 1
    aliensNum.Decrease();

    // Add the Alien's score to the score's data
    score.AddToData(pAlien.getScore());
}
```

There are many different Observers that do different things. We only attach the Observers that are relevant to that particular collision pair. For example, we don't want to add score when a Missile hits a Shield, but we will want to remove the Shield's brick that was hit and reset the Missile so we'll attach the ShieldRemove_Observer and the Missile_Observer. Now when that collision happens those Observers will Execute().

```
// Create the CollisionPair with a name, the Missile tree and the Grid tree ------------------------------
CollisionPair pColPair = CollisionPair_Manager.Add(CollisionPair.Name.Missile_Alien, missileRoot, gridRoot);

// Attach Observers
pColPair.Attach(new Missile_Observer());
pColPair.Attach(new AlienRemove_Observer());
pColPair.Attach(new AlienScore_Observer());
```

Command



Throughout Space Invaders we have some events that are based on a particular time rather than an action: TimeEvents. TimeEvents are put into Priority Queue, but more on that in a bit. Each TimeEvent has a particular Command whose Execute() method is fired off when the time has come. All classes inherit from Command and must implement the Execute() method, but have no constraints outside of that. Each class can be a different size and contain different amounts of data.

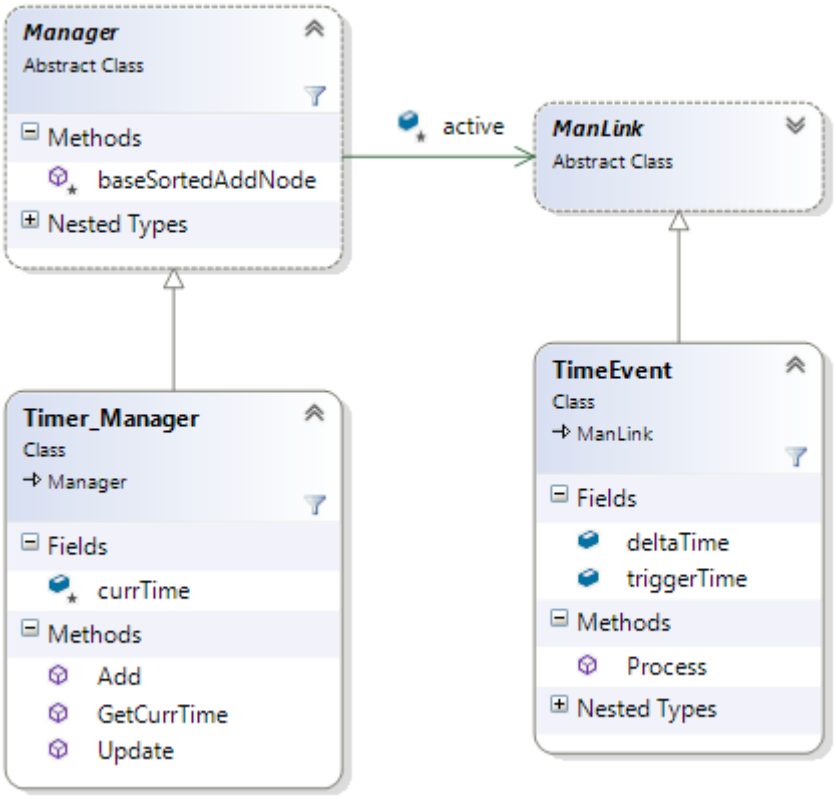
Starting out moving every second and reduced by a small amount every time an Alien is killed, we animate the Alien_Grid and all of its children based on the Execute() of the Animation_MoveGrid.

Animation_MoveGrid.Execute() traverses the grid tree a adds a set distance to each Alien's position. We then add the event back to the Timer_Manager so this event can happen again based on the amount of Aliens_Remaining.

```
override public void Execute(float _deltaTime)
{
    // Move the grid
    this.pGrid.MoveGrid();

    // Add myself back to timer to trigger at the Alien's speed
    Alien_Speed speed = (Alien_Speed)Global_Manager.Find(Global.Name.Alien_Speed);
    Timer_Manager.Add(TimeEvent.Name.GridAnimation_1, this, speed.amount);
}
```

A Command is essentially an instruction for an object(s) in a very self-explanatory kind of way. Animation_MoveGrid moves the Grid of Aliens, Bomb_Drop sends a Bomb from an Alien, SendUFO sends the UFO flying across the screen, etc.

Priority Queue



Time_Events that execute Commands are stored in the Timer_Manager's active list, which is a Sorted Priority Queue. Based on the current time in the game, if the Time_Event's triggerTime has passed then the Time_Event is fired off. Once we hit a Time_Event whose triggerTime has not passed, we exit the list traversal as an early-out.

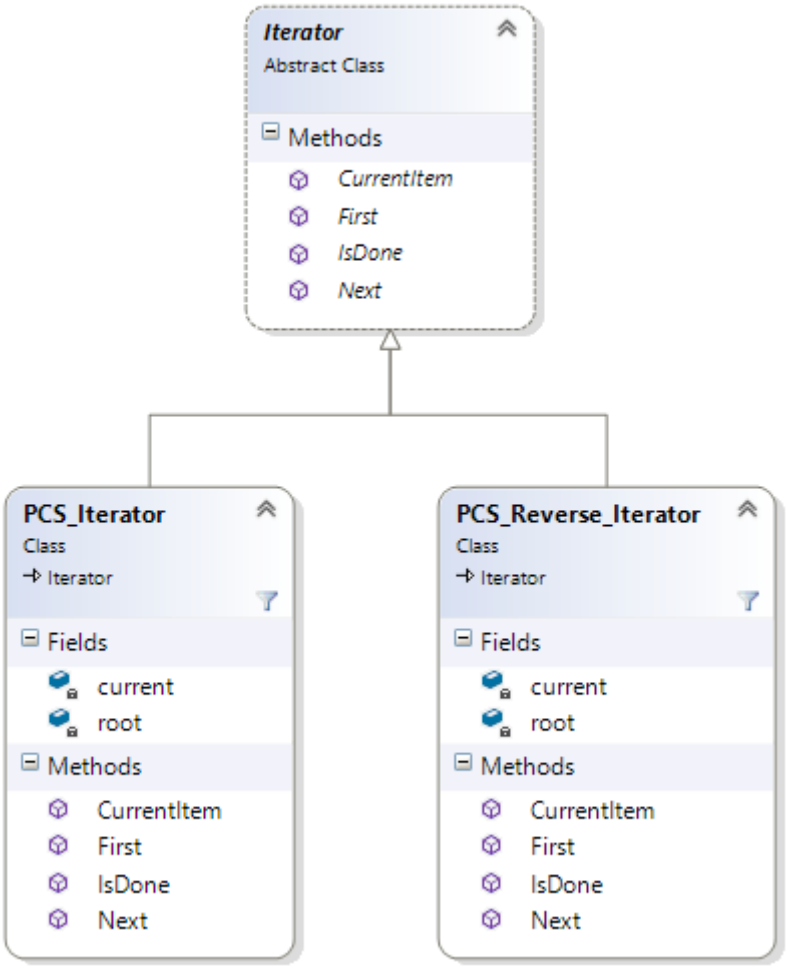
When you add a Time_Event into the Timer_Manager, Timer_Manager.Add() calls baseSortedAddNode() on that Time_Event to put it into the active list. This method makes sure that all the Time_Events are sorted by their deltaTime with the smallest deltaTime first.

```
protected void baseSortedAddNode(ManLink node, ref ManLink head)
{
    //*********************************************************
    //* This method adds the passed in node in ascending order
    //* of the passed in reference head
    //*********************************************************

    ManLink current;

    /* Special case for the head end */
    if (head == null || head.getData() >= node.getData())
    {
        node.next = head;
        head = node;
    }
    else
    {
        /* Locate the node before the point of insertion */
        current = head;

        // getData() return's the TimeEvent's triggerTime
        while (current.next != null && current.next.getData() < node.getData())
        {
            current = current.next;
        }

        // Set the new pointers and insert the node
        node.next = current.next;
        current.next = node;
    }
}
```

When we walk the Priority Queue of Time_Events, we add the Time_Event's deltaTime to the game's current time and store it in the Time_Event's triggerTime. If the game's current time is greater than the Time_Event's triggerTime, then the event is fired off.

```
TimeEvent pEvent = (TimeEvent)pTimerMan.active;

// Walk the list until there is no more list OR currTime is greater than triggerTime
while (pEvent != null && (pTimerMan.currTime >= pEvent.triggerTime) && (pEvent.triggerTime > 1e-9))
{
    // can't walk and remove at the same time so set up a pointer
    nextEvent = (TimeEvent)pEvent.next;

    if (pTimerMan.currTime >= pEvent.triggerTime)
    {
        // call it
        pEvent.Process();

        // remove from list
        pTimerMan.baseRemove(pEvent.name, pEvent.index);
    }
```

Iterator



An Iterator is a way to access all elements of a data structure regardless of its hierarchy or orientation without exposing its representation.

We have two iterators in Space Invaders, PCS_Iterator and PCS_Reverse_Iterator. Both of these traverse the PCS_Tree, accessing every node until isDone().

Using the PCS_Iterator on the Alien_Grid tree you will access every node by calling Next() starting with the root of the tree and continuing to the very last child.

```
public override GameObject Next()
{
    this.current = privGetNext(this.current);

    return this.current;
}

private GameObject privGetNext(GameObject node, bool UseChild = true)
{
    GameObject tmp = null;

    if ((node.child != null) && UseChild)
    {
        tmp = (GameObject)node.child;
    }
    else if (node.sibling != null)
    {
        tmp = (GameObject)node.sibling;
    }
    else if (node.parent != this.root && node.parent != null)
    {
        // Recurse here
        tmp = this.privGetNext((GameObject)node.parent, false);
    }
    else
    {
        tmp = null;
    }
    return tmp;
}
```

The PCS_Reverse_Iterator will do the exact same thing, but it will access the nodes starting with the very last child and finishing with the root of the tree.