Design Document

“Space Invaders Arcade Game”

SE 456

Ed Masterson

March 17, 2015

High Level View

To recreate the arcade video game classic “Space Invaders” , I began by building the managers of the basic building blocks of the game including images, sprites, and textures. Later, I included managers for batch groups and dead objects. The basic architecture of the game is that there is game time that increases as the game goes on. There is a Priority Queue of TimeEvents set in sorted order. I put the events on the list in the order I need them to occur. These events can be anything from drawing an alien sprite onto the screen, changing the alien image to make it animate, changing the player’s score, or moving the grid of aliens, or firing missiles from the player’s ship. Once these events are processed, they are removed from the Priority Queue.

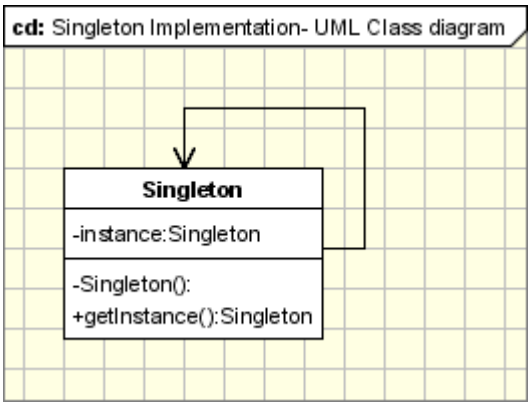
Ed Masterson Space Invaders Game

Component Systems

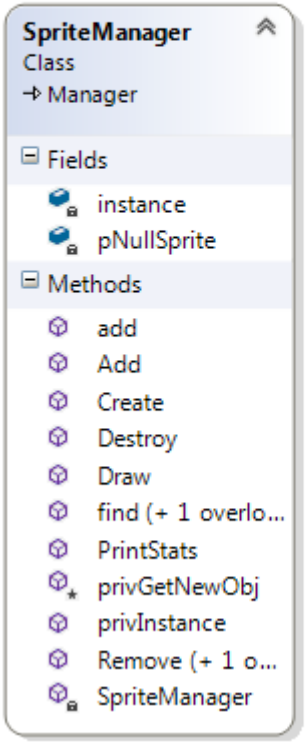
Created Managers with Singleton and Manager Patterns

The Singleton Pattern is a way to ensure only one instance of a class. It also allows access to that instance from anywhere in the program.

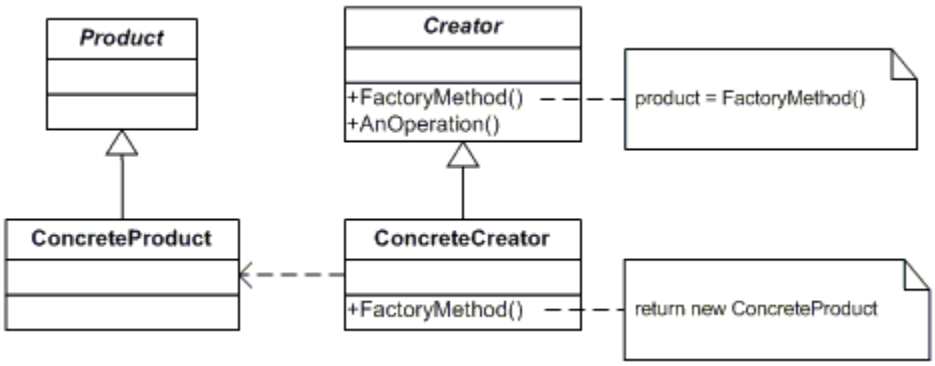
Using the Singleton pattern allowed me to create one and only one instance of each individual manager that would hold, store and allocate objects to active and reserve lists.



The problem it solved was having too many instances of a particular manager when I only needed one. The way it was used in “Space Invaders” was this. The Create() method is called in the Initialize() method of the Game class. There is an if statement that if the instance equals null, set the instance to a new SpriteManager. Now the instance is not null, So if Create() is called again, another instance can’t be created.

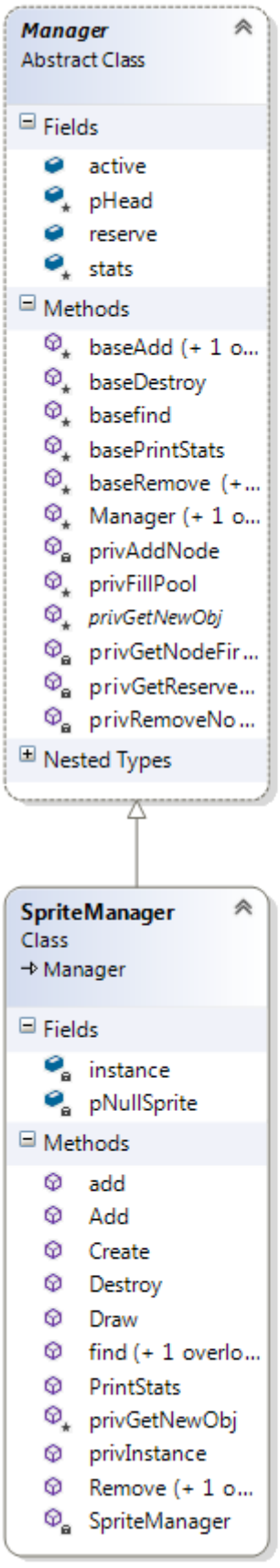


Within the Manager class is a Factory Method. The Factory Method is a way to create objects, but have the subclasses determine from which classes to create objects.



As illustrated above, whatever ConcreteCreator calls the base class Creator, that is the type of ConcreteProduct that is made.

The problem that it solves in our project is how to have one class with all the code on how to create objects instead of rewriting the code for each class of objects. The mechanism of how the problem is solved is using inheritance. The Manager class is an abstract class. All the different Managers like the SpriteManager, SpriteBatchManager, TimerManager, all inherit from the Manager class. When these classes are instantiated, in their constructor is a call to the base class with the number of instances that should be created. The constructor of the Manager class takes that number and uses it as a parameter in the privFillPool() method. The privFillPool() method assigns an object of type Object, which is the highest class in C# and every class inherits from it. This new object is assigned to privGetNewObj(). The privGetNewObj() method is in the concrete class that called the base constructor. This creates an object of the type of the concrete class and assigns it to the Object object. For example, if the SpriteManager called the Manager class, then the privGetNewObj() method would create new Sprite objects. This new object is put on the reserve list for Images, Textures, Sprites or whatever objects where just created. This is illustrated below.



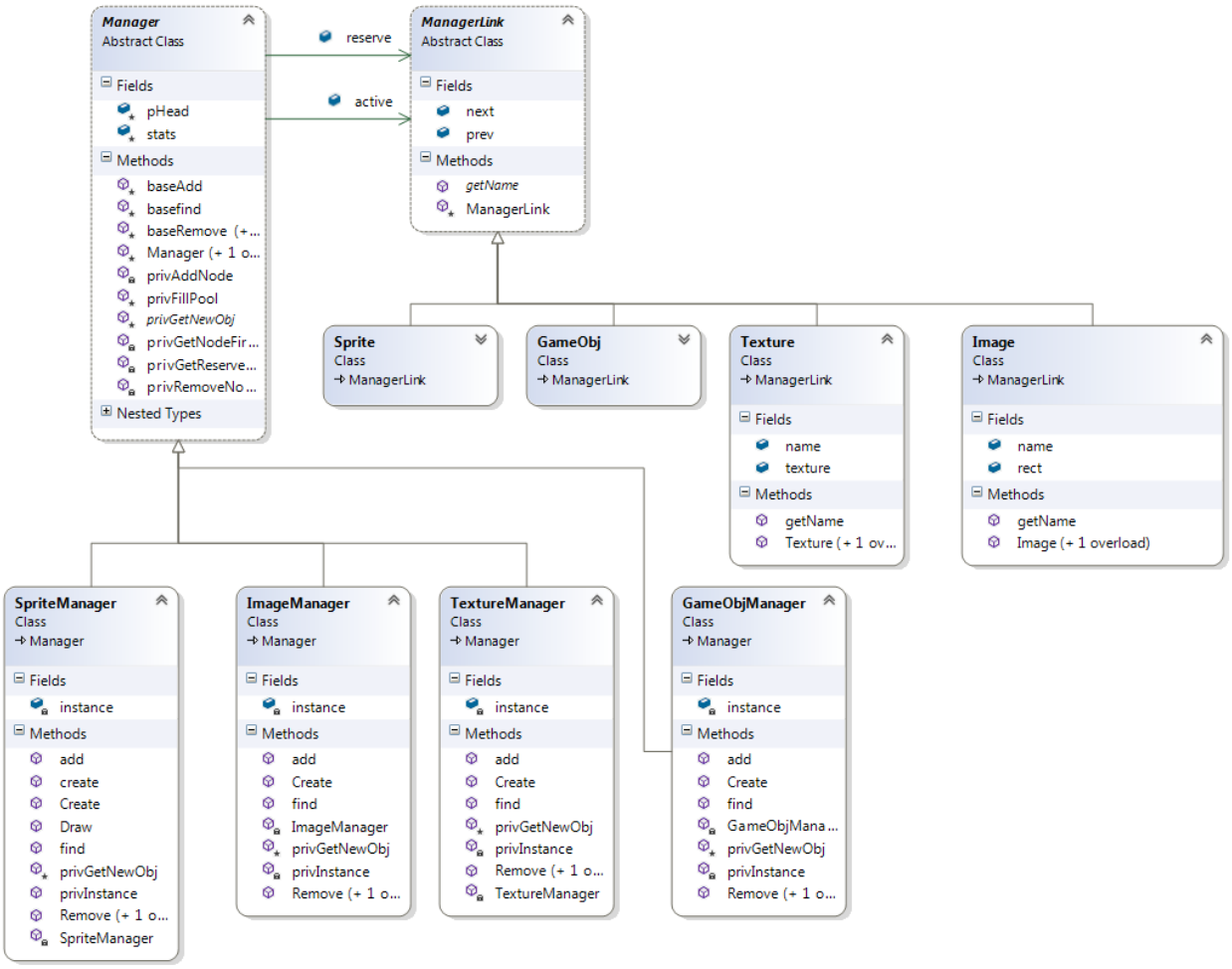
Manager With Factory Pattern

The Manager class is an abstract class that creates nodes we need in the reserve list. This is done in the beginning of the game to grab all the memory we need so we don't run out later. It sets up active and reserve lists for textures, sprites, images, game objects and sprite batch groups. The reserve lists are set up to grab the memory we need, then nodes are taken from the reserve and put on the active as we need them throughout the game.

The lists are LIFO(Last In First Out) for speed. We don't have to search, we just grab the first one in the list, when we add we just put in the front of the list.

The Manager class is also responsible for keeping the stats of how many nodes become active over the course of the run of the program. This allows us to see how many we need to set our default number to for efficiency. If we can create all we need up front, we don't need to take any time while the game is playing to create new objects.

All of this functionality is inherited by the individual managers: SpriteManager, ImageManager, TextureManager, and GameObjManager. These classes are static which allows us to ensure only one instance of each manager is created.



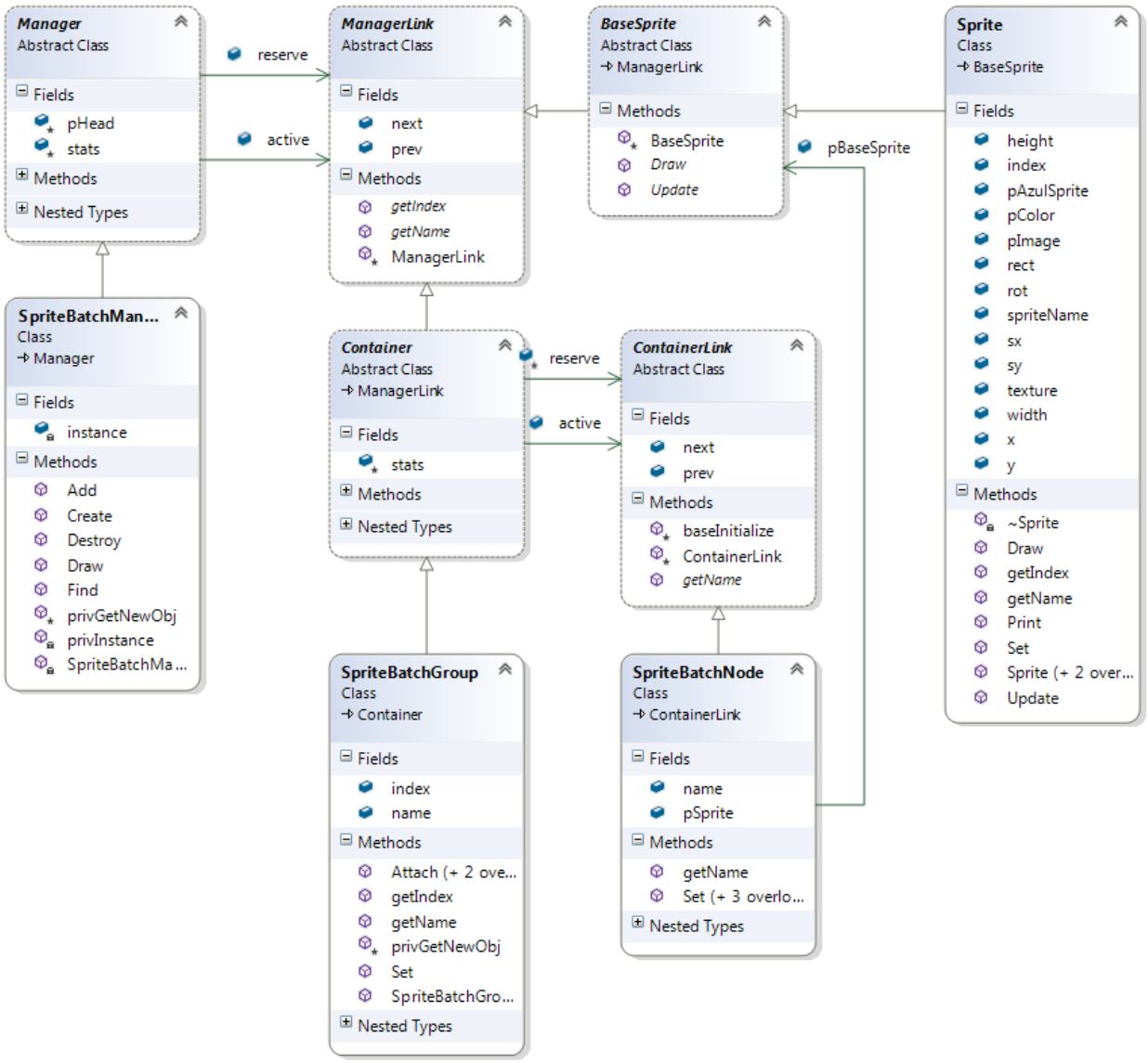
The ManagerLink class has the next and prev data fields. It is an abstract class as well and allows the specific objects to inherit data along with getName() and getIndex() methods. The specific objects like Texture, Image only need to contain data that pertain to them like the individual Azul.Texture textures, Azul.Rect rectangles, names and indexes.

The Game class inherits from the Azul Game Engine. It consists of five funtions that are overridden: Initialize(), LoadContent(), Update(), Draw(), and UnLoadContent(). Initialize() is where the individual managers create their reserve lists. LoadContent() is where the user adds individual textures, images , sprites, etc. to the managers and the managers move those on to the active lists. Update() allows the individual pieces on the screen to be added, removed, moved and the scores to be changed. Draw() is what draws those individual pieces on the screen. UnLoadContent() destroys all the objects.

Sprite System

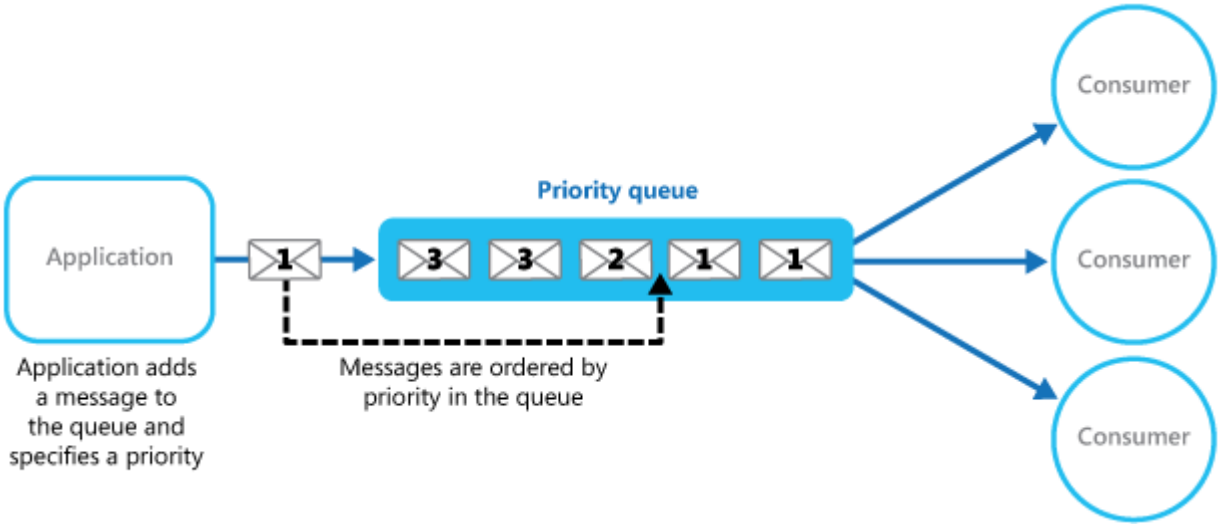
The Sprite System has a SpriteManager that is similar to the TextureManager and the ImageManager. The SpriteBatchManager, however, is different because it has a Draw() method. The Sprite Class also has a Draw() method which distinguishes it from the Image and Texture classes.

The SpriteBatchGroup class is basically a list that tells the game what to draw on the screen. It provides a way for the programmer to decide what order things are drawn in. Sprites that need to be on top can be made to draw last. When Draw() is called in the SpriteBatchManager, it iterates through each SpriteBatchGroup in order and for every individual group, the Draw() method goes to each individual SpriteBatchNode and calls the Update() method on the Sprite located in that node. All of the data is transferred from the Sprite to the Azul Sprite. Next, the Draw() method is called on the Sprite which calls the Azul Engine Draw() and the Sprite is drawn on the screen.



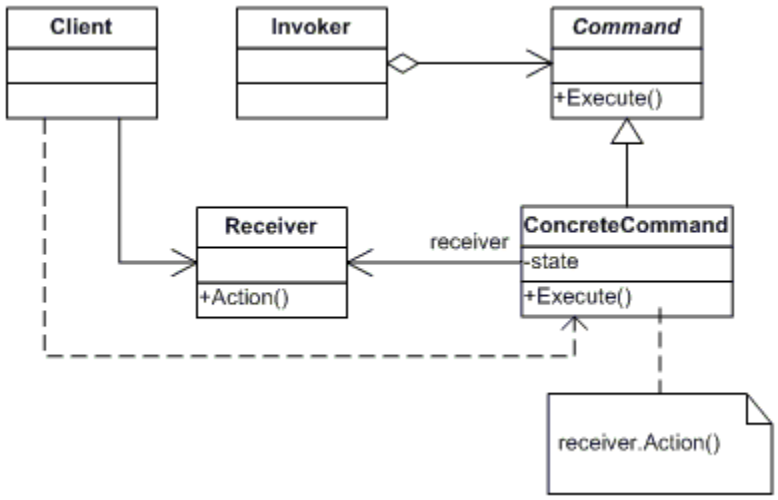
Timer System with Command Pattern and Priority Queue

The Priority Queue pattern is a way to create a list that is ordered in a certain way. It gives a ranking of each item on the list according certain criteria as shown below.



The problem it solved in the “Space Invaders” game was how do we order events within the game. How do the Aliens know when to march, and how long until the next UFO comes out.

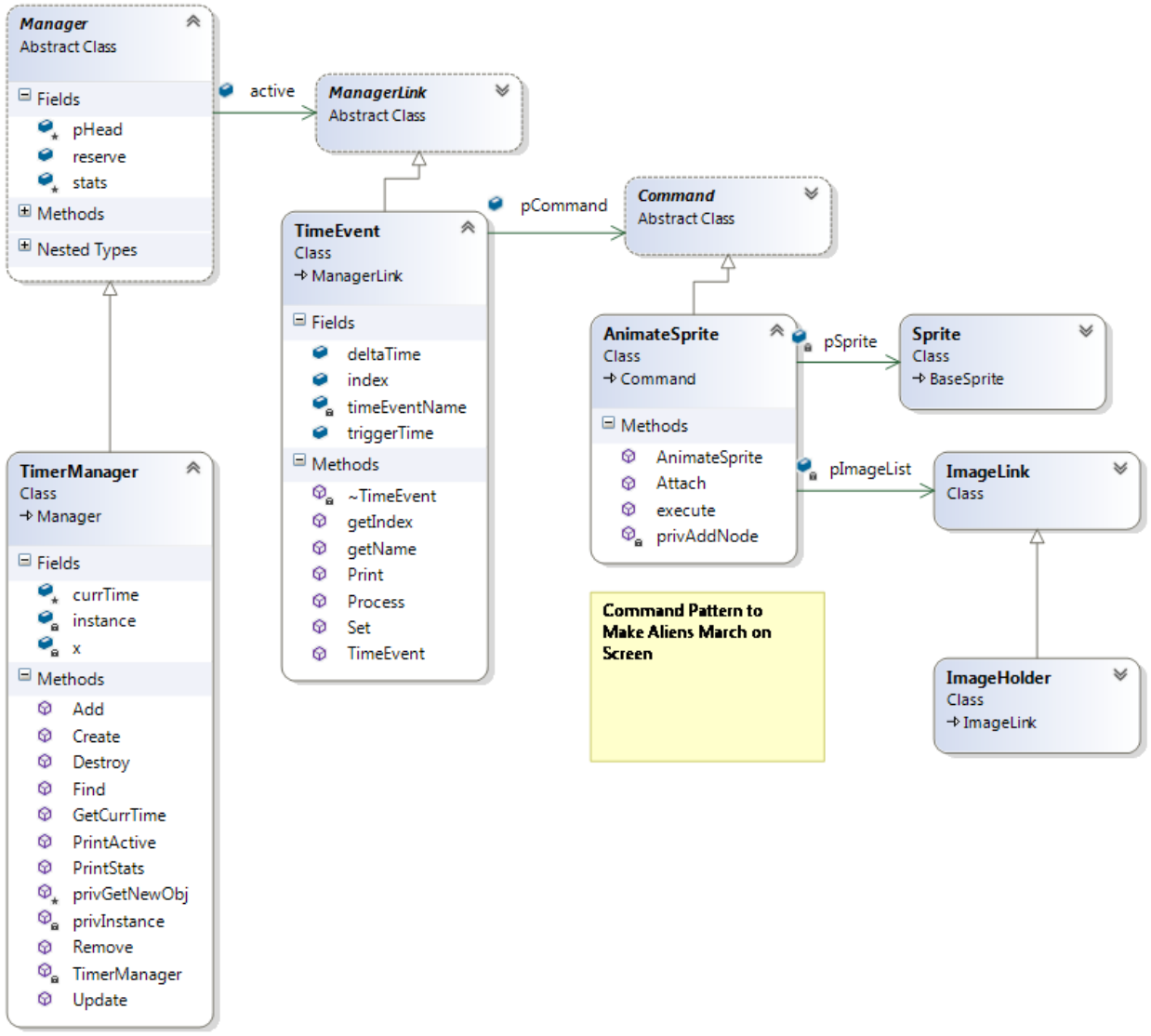
The Command Pattern works as follows. The client creates a command. The invoker basically rents the command because it has an aggregation relationship. The Invoker uses the Command, it does not own it. First the Invoker sets the command. Later, the Invoker calls execute on the Command. In the execute() method in the ConcreteCommand, the Reciever.Action() method is called that performs the action. This is illustrated down below.



The problem the Command pattern solves in the game is how to fire of the events in the priority queue. Timer Manager allows a way to have time events laid out in a Priority Que. When the game time is greater or equal to when those events are scheduled to go off, we can use Command Pattern to fire off those events.

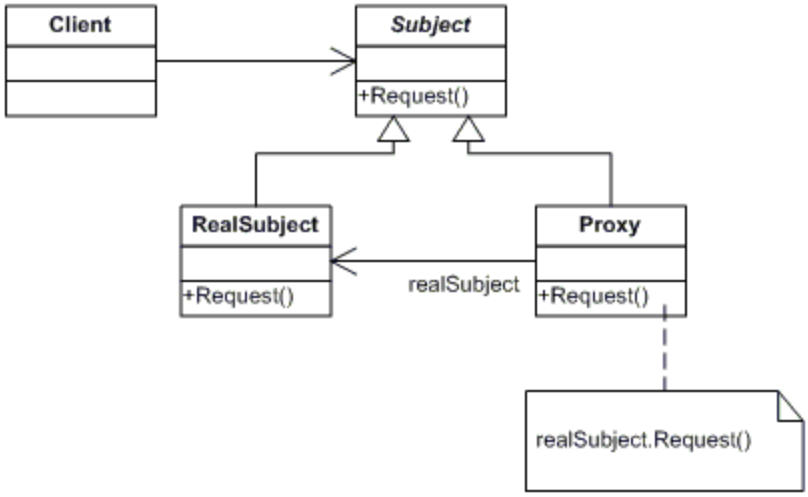
The TimerManager puts TimeEvents onto a list that is sorted by time, lowest to highest. This makes it a Priority Queue. The events that need to be fired off first in order of time are at the front of the list. Once the game time is greater than or equal to the time on the TimeEvent, the TimerManager calls the Process() method. This calls the execute() method in the concrete Command object. In this case the concrete Command object is AnimateSprite. This allows us to do many dynamic things within the game. We can change the image in the sprite to allow us to give animation to our game. We can change the score, or move the sprites around on the screen to different coordinates.

The different images for our AnimateSprite are held in the ImageHolder. Depending on which image is in the ImageHolder, that is the image that will be drawn on the screen. To change the image, we put a TimeEvent in the Priority Queue along with the time we want the image to change. When that TimeEvent comes up, it will get processed, the execute command will be called in AnimateSprite, and then the image that ImageHolder points to will be changed to the next image in the list.

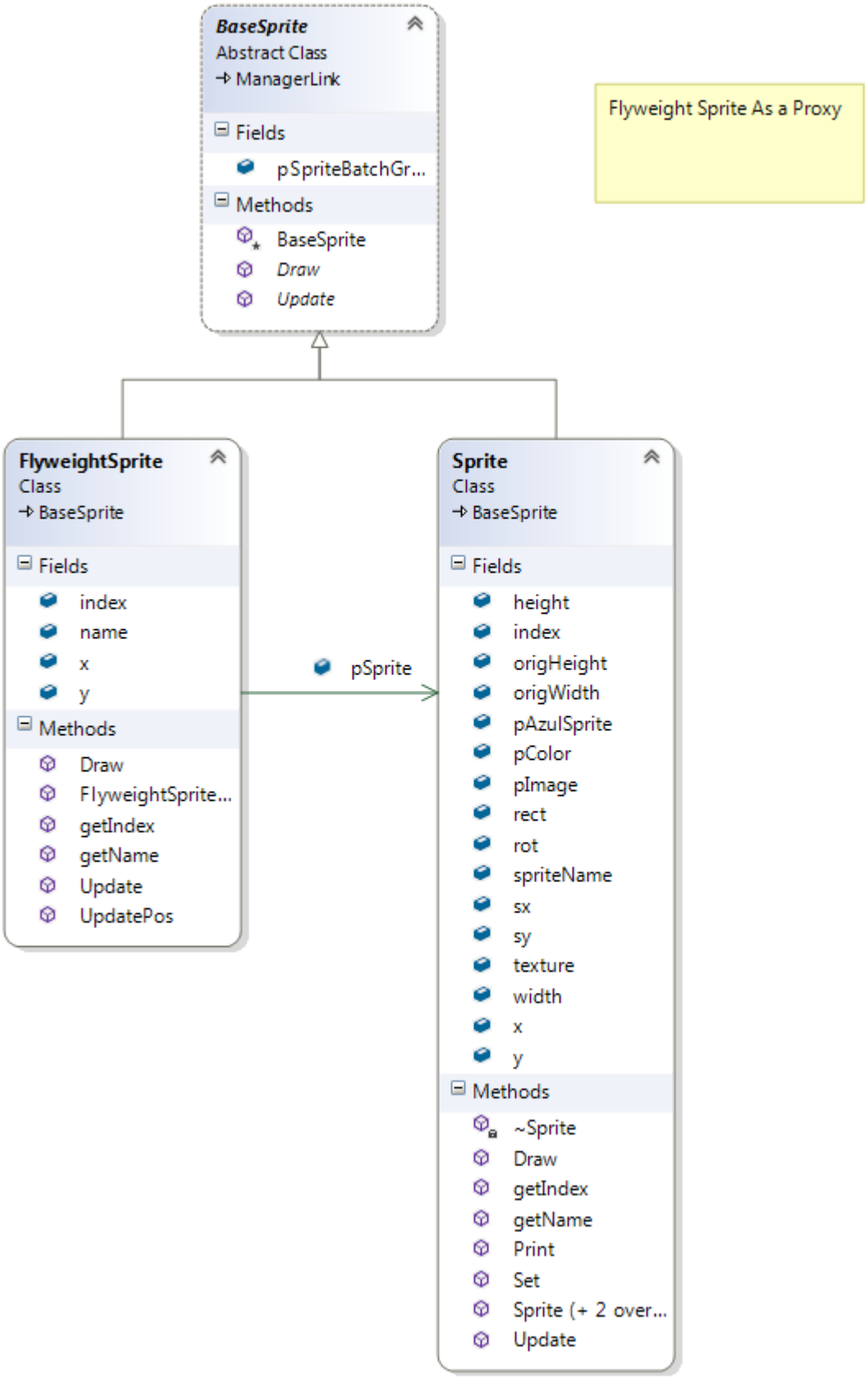


FlyweightSprites as Proxy Objects

The Proxy pattern is a way to have a placeholder for an object that might be memory intensive to create many times over. Often you can create a proxy that has a portion of the information needed for the real subject. The proxy also has a pointer to the real subject so it can pass on its information to the real subject as shown below.

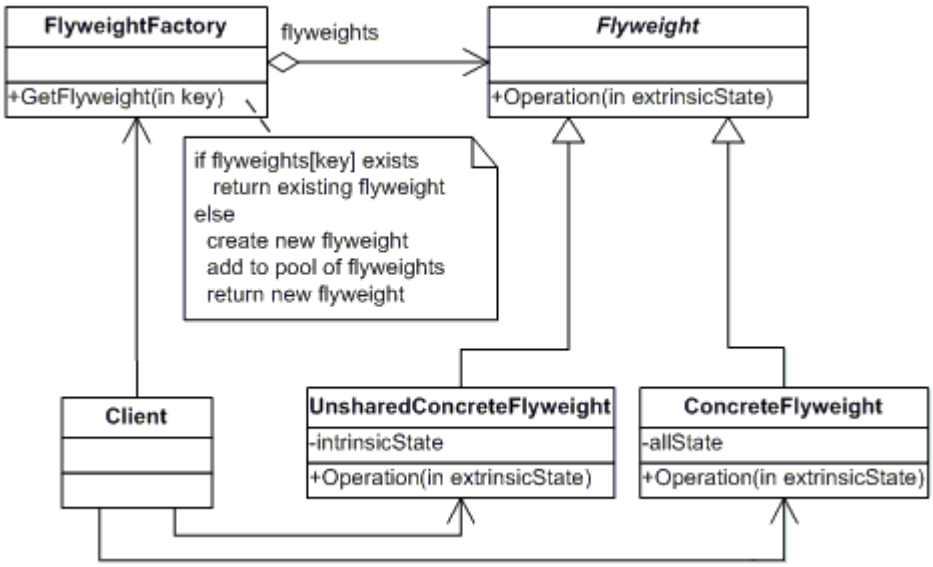


The problem that the proxy solves in “Space Invaders” is allocating too much memory. We want to conserve the amount of memory we use. The mechanism to solve this is to have FlyweightSprites act as proxy objects for regular sprites because they can hold just the x and y coordinates on the screen without having to hold all the other data a sprite needs. If it needs to be drawn it can then just point to a sprite and call its draw method. This is a great way to conserve memory if we are drawing multiple sprites that are the same image with a different location on the screen. This is shown below:


BaseSprite
Abstract Class
ManagerLink
Fields
pSpriteBatchGr...
Methods
BaseSprite
Draw
Update
Flyweight Sprite As a Proxy
FlyweightSprite
Class
BaseSprite
Fields
index
name
x
y
Methods
Draw
FlyweightSprite...
getIndex
getName
Update
UpdatePos
pSprite
Sprite
Class
BaseSprite
Fields
height
index
origHeight
origWidth
pAzulSprite
pColor
pImage
rect
rot
spriteName
sx
sy
texture
width
x
y
Methods
~Sprite
Draw
getIndex
getName
Print
Set
Sprite (+ 2 over...
Update

GameObject System with Flyweight and Alien Factory Pattern

The Flyweight pattern allows us to create a large number of objects that share some of the same internal data. This allows us to save on memory resources.



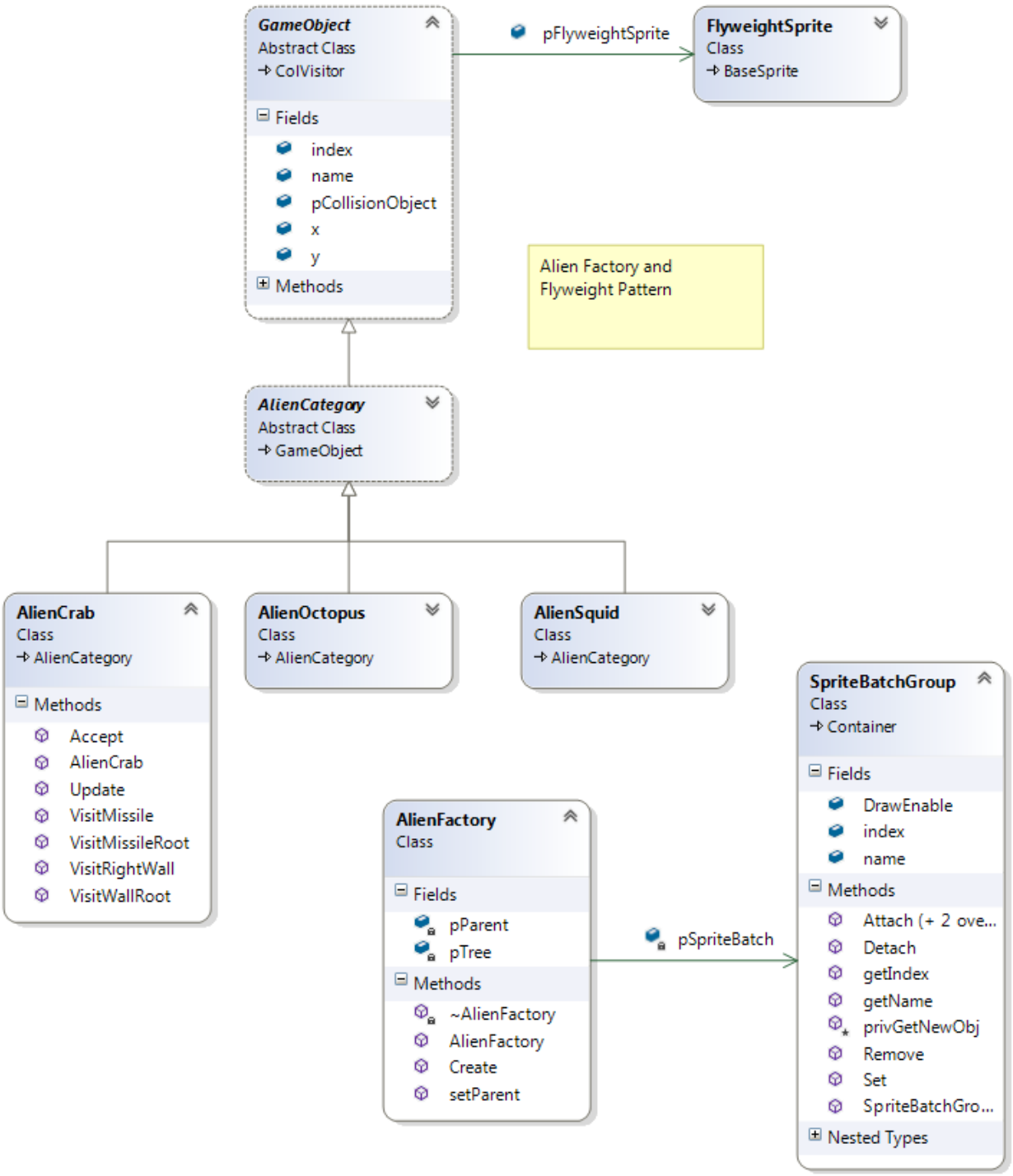
The problem that the Flyweight Factory solves is limited resources. This is the mechanism through which it is solved.

GameObjects are basically the building blocks of the game. They have a one to one relationship with Flyweight Sprites and all the Flyweight Sprites are owned by the GameObjectManager. Incorporated in the GameObject System are the Flyweight pattern and the Factory Pattern to create Aliens.

The AlienFactory class is where all the aliens are made. In the Create() method, the AlienCategory type that is passed into the method determines what type of alien(Crab, Squid, or Octopus) is instantiated in the switch statement. The new object is then assigned to the AlienCategory class pointer. This is the where the Factory Pattern comes into play. The AlienFactory 's Create method continues to crank out different types of Alien objects, depending on what type we ask for. This allows us to not use "new" in Game.cs file. It is good practice to keep our "new" and "delete" key words away from the client.

The Flyweight pattern is used to allow us to create just one instance of each particular alien Sprite. Each time a GameObject is created, we create a FlyweightSprite with the Sprite that is passed in to the GameObject constructor. Because many of the aliens are the same, we can create one sprite and

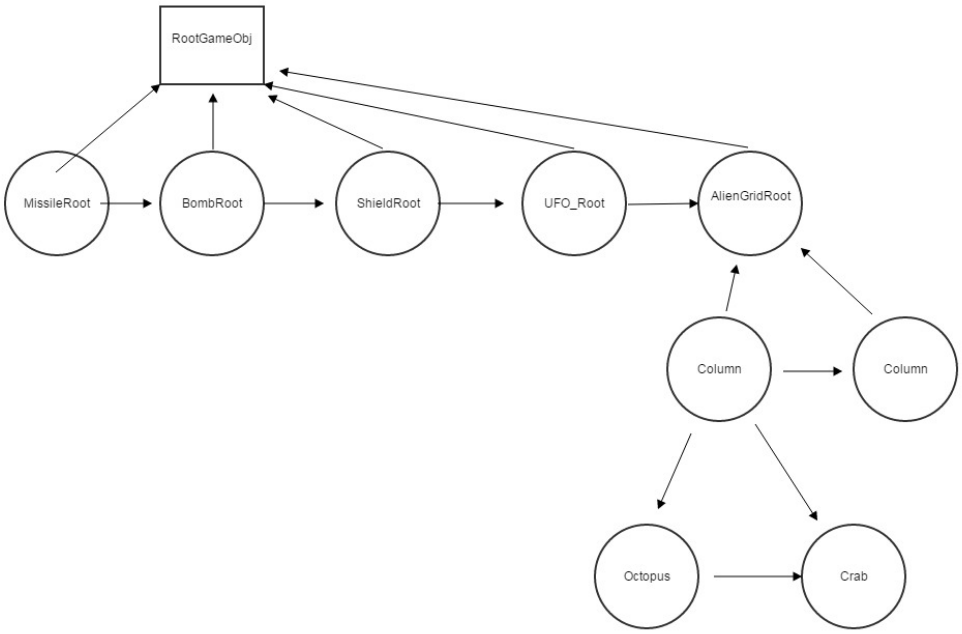
11 flyweight sprites. The flyweight sprites are lighter on our memory usage seeing as they only contain an x-coordinate and a y-coordinate and a pointer to a Sprite object. This is better than creating 11 sprites because they contain more data than FlyweightSprites. Since each Alien class inherits from a GameObject, we can use the GameObject's pointer to a FlyweightSprite to attach to a SpriteBatchGroup in the AlienFactory to have the FlyweightSprites drawn to the screen. This is illustrated below.


GameObject
Abstract Class
ColVisitor
Fields
index
name
pCollisionObject
x
y
Methods
pFlyweightSprite
FlyweightSprite
Class
BaseSprite
Alien Factory and
Flyweight Pattern
AlienCategory
Abstract Class
GameObject
AlienCrab
Class
AlienCategory
Methods
Accept
AlienCrab
Update
VisitMissile
VisitMissileRoot
VisitRightWall
VisitWallRoot
AlienOctopus
Class
AlienCategory
AlienSquid
Class
AlienCategory
AlienFactory
Class
Fields
pParent
pTree
Methods
~AlienFactory
AlienFactory
Create
setParent
pSpriteBatch
SpriteBatchGroup
Class
Container
Fields
DrawEnable
index
name
Methods
Attach (+ 2 ove...
Detach
getIndex
getName
privGetNewObj
Remove
Set
SpriteBatchGro...
Nested Types

Hierarchy System PCSTree Patttern

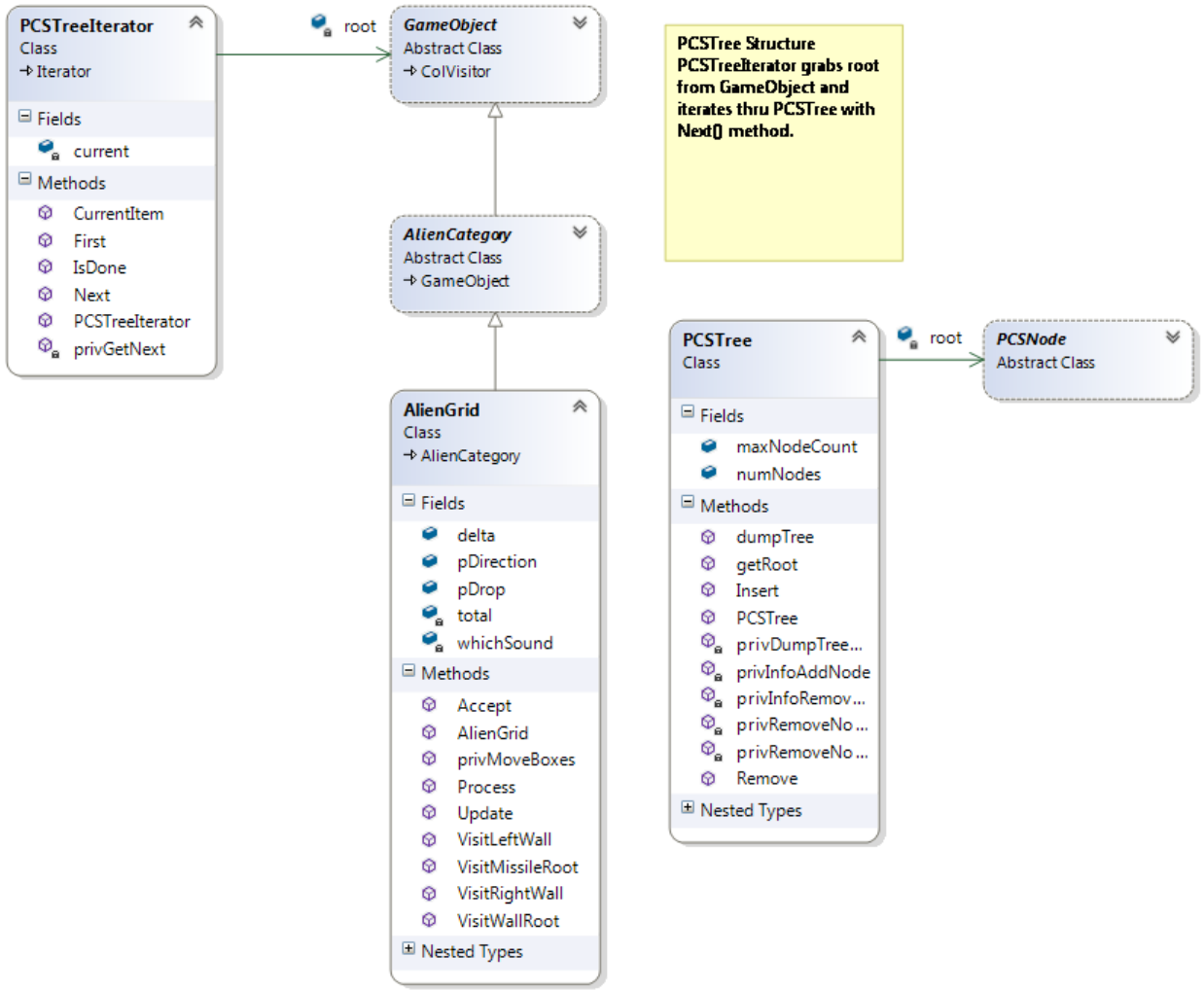
The PCSTree allows us to solve the problem of how to have the aliens march in step. The Hierarchy System uses the PCSTree pattern to move the entire grid of aliens in unison. The PCSTree class can add and remove nodes to the PCSTree. It can also locate the root. The individual nodes are PCSNodes which have pointers to a parent PCSNode, a child PCSNode, and a sibling PCSNode.

We can use this tree structure to iterate thru the tree and change the x and y coordinates of all the alien FlyweightSprites to allow us to move the grid together in a group. The PCSTree structure is shown below.



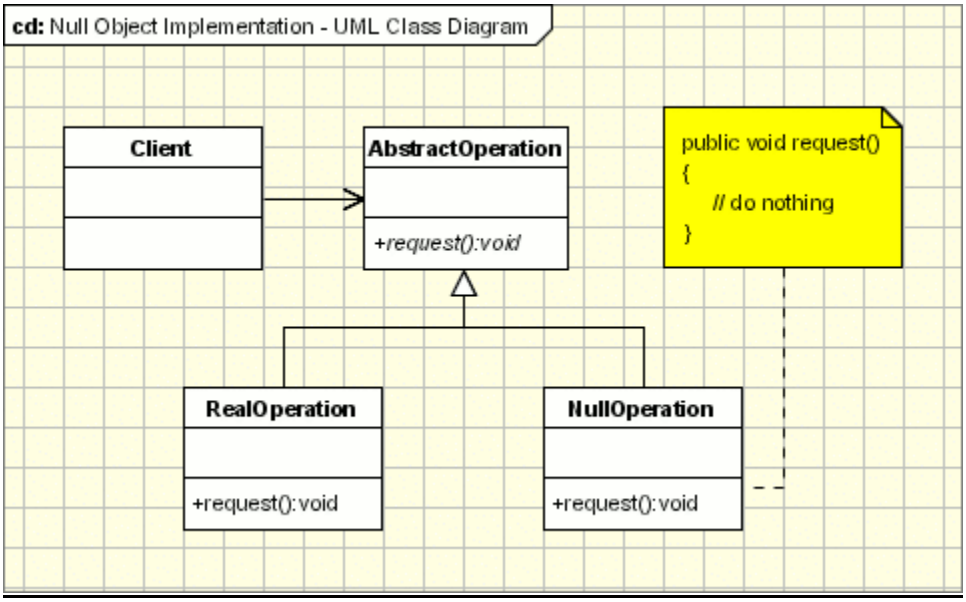
In the Update() method in the TimerManager, when the TimeEvent is triggered, the TimeEvent is processed, and an AlienGrid object called pGrid is assigned the GameObject called Grid. The privMoveBoxes() method is called which iterates thru the PCSTree and changes the x coordintes of all the aliens in the grid by the same amount so when the Draw() method is called on their Sprites the aliens move on the screen in unison.

Down below you can see the project has a PCSTree class with methods to insert and remove nodes. The PCSTreeIterator has methods that can grab the root of the tree and then next methods to move throughout the tree.

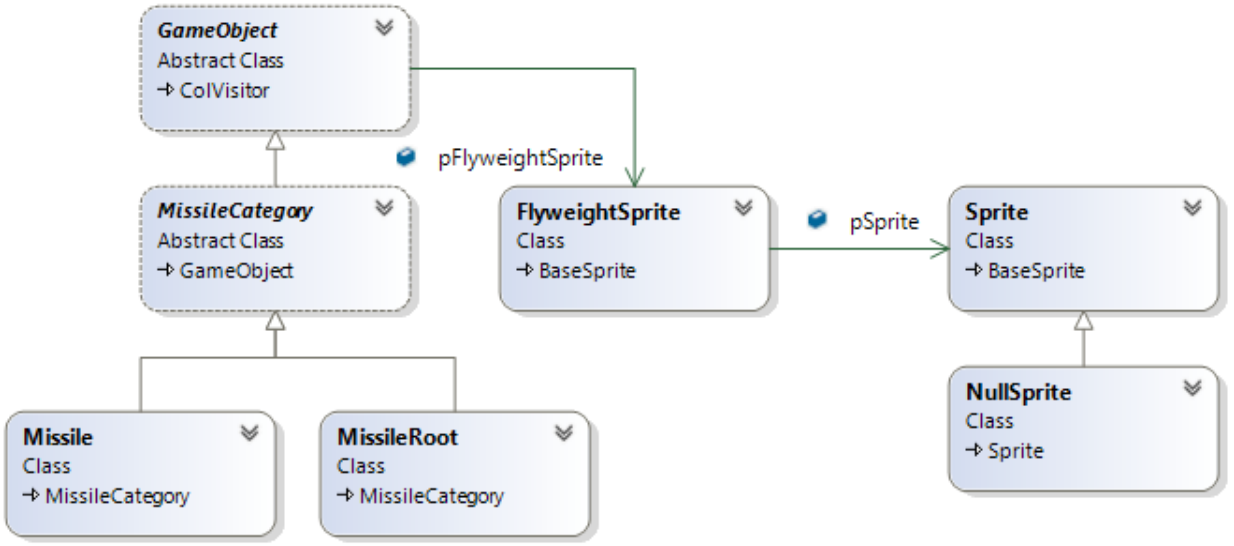


Null Object Used with PCSTree Roots

The Null Object design pattern is ideal for situations where the programmer wants to provide a substitute for an object. It is used on occasions when the programmer doesn't want the program to do anything. The problem it solves in "Space Invaders" is when you have objects like root objects that you do not need to draw to the screen.

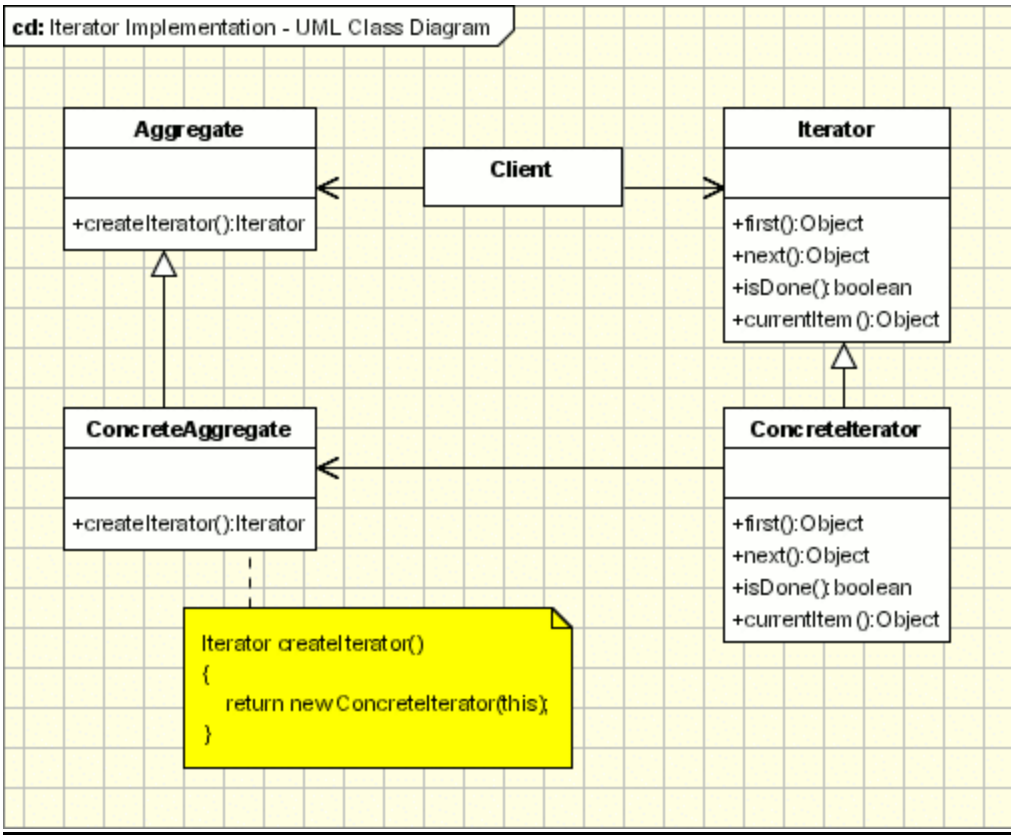


The root objects that the GameObjects attach to is a great example. Here is the mechanism. The MissileRoot attaches the main root of the overall game called RootGameObject, there is no need to actually draw that object on the screen. So when the constructor asks for the sprite associated to the MissileRoot, we can pass in a Null Object that has no image associated with it . This is shown in the diagram below. The MissileRoot, which inherits from the GameObject class, can use the pointer to the FlyweightSprite. The FlyweightSprite has a pointer to the Sprite class. It can use the Sprite class which has an image, x and a y or the Null Sprite which has no image and no x and y.

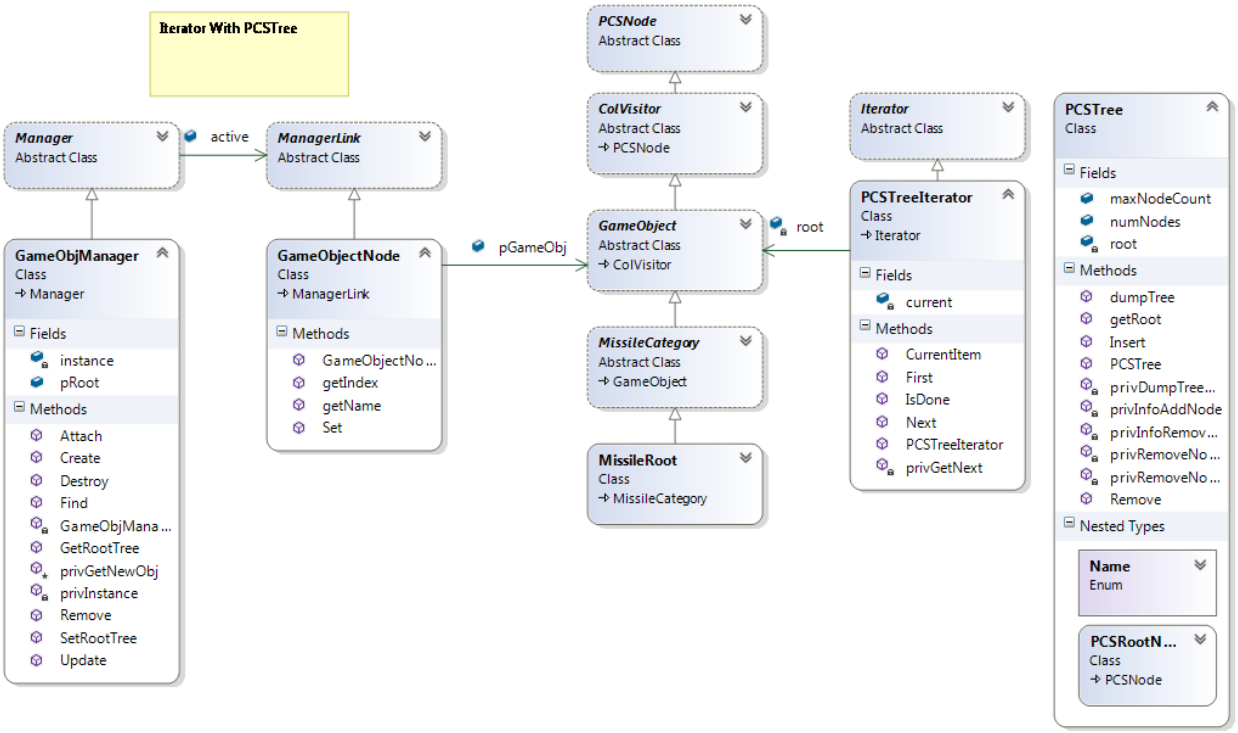


Iterator with PCS Tree

The Iterator pattern provides a way to access elements in a collection without the client having to know the inner working details of the structure. The iterator has access to the individual items in the collection and know how to get from one item to the next. Down below, the ConcreteIterator has a pointer to the ConcreteAggregate. The Iterator has methods for getting the first object, moving to the next node, and letting us know whether it is done iterating.



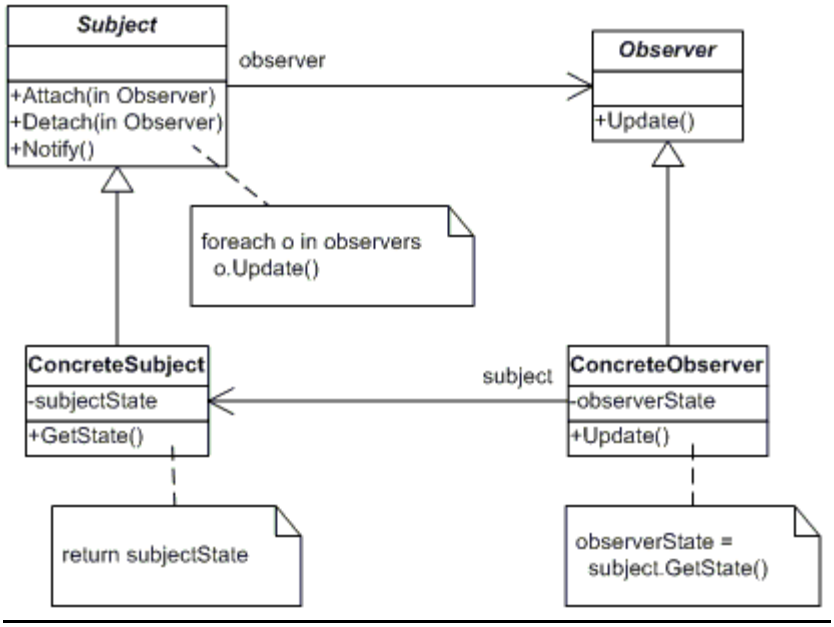
In our GameObjectManager, the problem the iterator solves is we need a way to traverse through our PCSTree and either locate a GameObjectNode, or process the nodes to find the dimensions of our collision boxes. This is done with the PCSTreeIterator, and the PCSTreeContraryIterator. As shown below, if we need to find the MissileRoot, we can use the Find() method in the GameObjectManager. This uses the forward iterator to locate specified nodes. The PCSTreeIterator grabs the root node of the tree with the First() method and then uses the Next() method to go from node to node. Once we find the right GameObject we can return it.



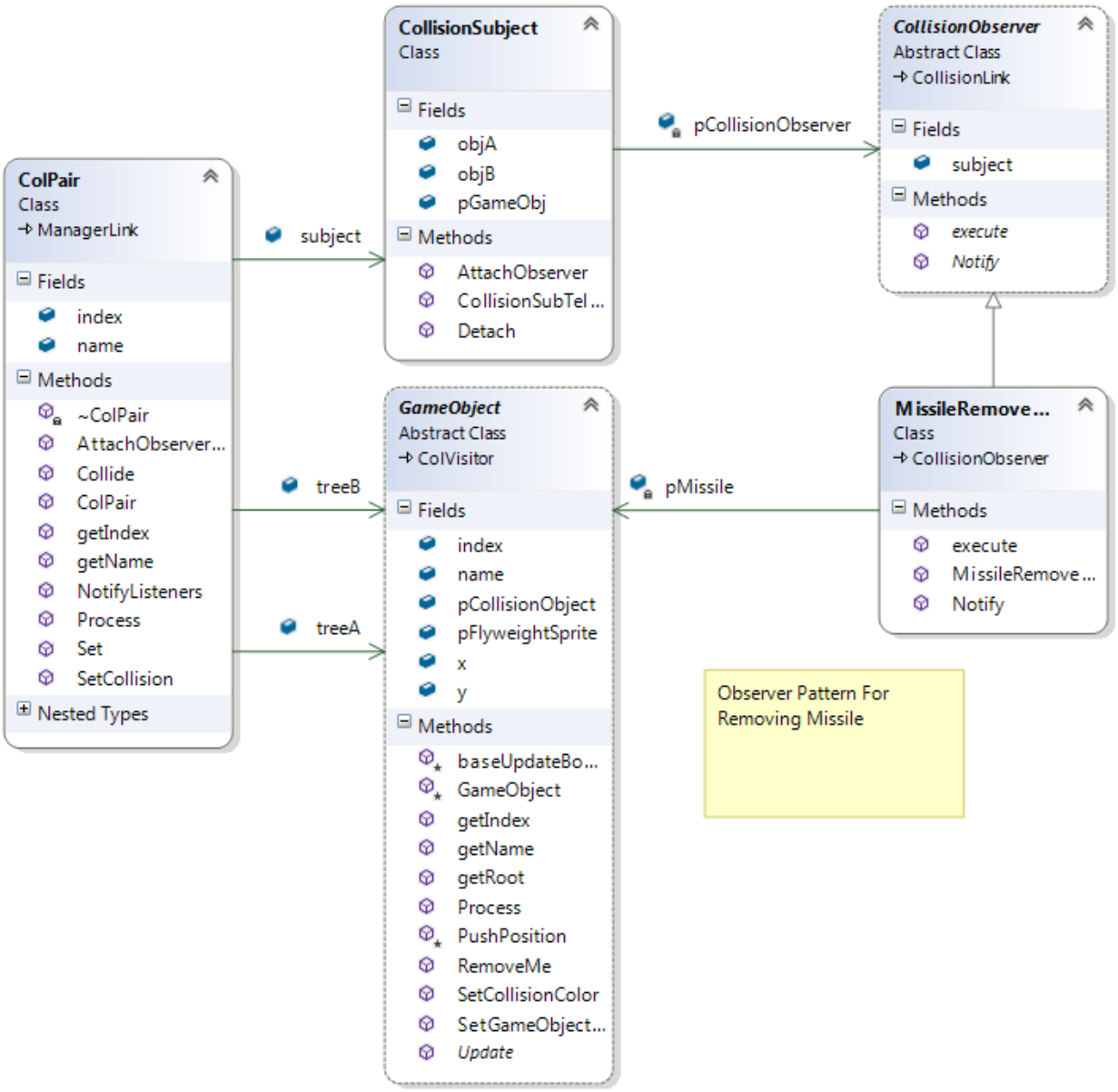
Observer for Making Missiles and Bombs Disappear

The Observer pattern allows many objects to keep track of the state of a single object. When the single object changes its state, the other objects who have an interest in the state of that object, or the observers, can be notified.

In general the Observer works as illustrated in the diagram down below. There is a single subject. In our case it will be the collision pair. The subject has methods to attach and detach observers, and another method to notify them if the state has changed. When the state of the subject has changed, the Notify() method is called. That in turn calls the Update() method on all the attached observers, refreshing their information on the state of the subject.

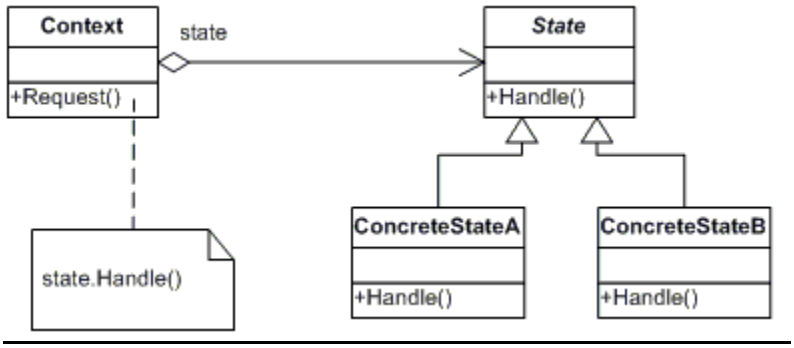


This is how the Observer pattern is used in “Space Invaders”. The problem the pattern solves is how do we know when to remove a missile from the screen? We can set up an observer to watch for when a MissileRoot and the grid of Aliens collide. When this happens the collision pair, lets say it is an octopus and a missile, will notify the CollisionSubject. The CollisionSubject will in turn tell the CollisionObservser. When the CollisionObserver calls Notify() it will check to see if Missile is one of the subjects in the collision pair. Once that is verified, the pMissile in the MissileRemoveObserver is assigned the missile from the collision pair. Later, the execute() method is called. In execute() the RemoveMe() method is called to take the missile of the list of sprite batches so its will not be drawn, its collision object is removed so it can’t collide, and is it removed from the tree of GameObjects. This is shown on the illustration below:



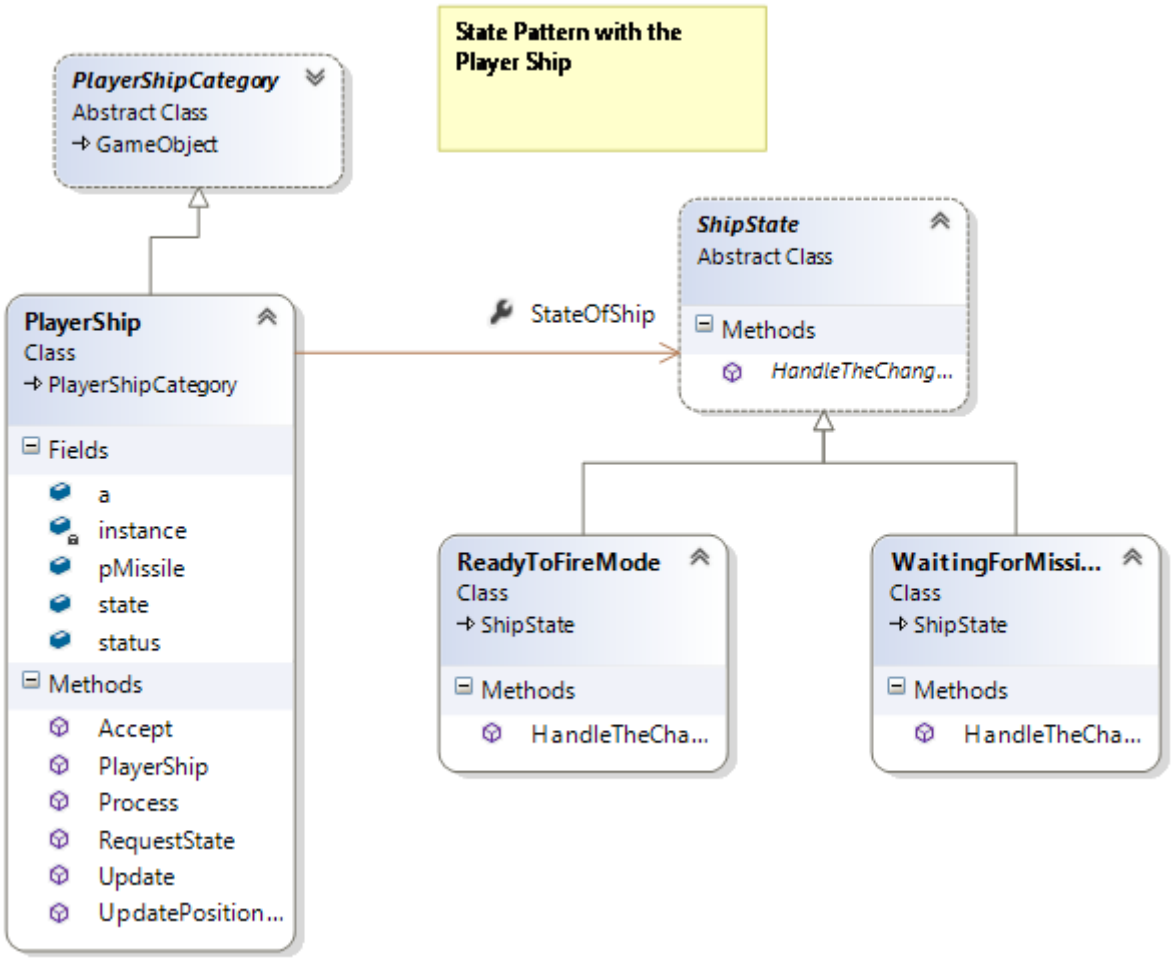
State for Firing Missiles from Player Ship

The purpose of the State pattern is to allow an object to change its behavior based on its internal state. It is a way for the object to act in different ways during runtime without using a lot of if-then conditional statements. The general pattern is shown below.

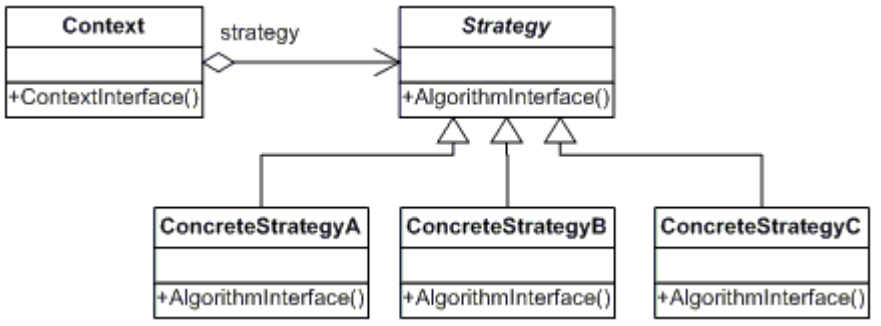


The context is the object which can change its state. In our case it is the PlayerShip. The context has an aggregation relationship with the state. This means it rents the state. It does not own the state. When the request() method is called on the context, this in turn calls the Handle() method. It is in the Handle() method that the states are made to change. The Handle() method in ConcreteStateA changes the state to ConcreteStateB, and the Handle() method in ConcreteStateB changes the state to ConcreteStateA.

How is this used in “Space Invaders”? The problem that it solves is how to we make it so the ship can only fire one missile at a time. The ship is originally put in the ReadyToFireMode state. Once it is determined that the space bar has been pressed on the keyboard, the RequestState() method is fired off. This calls the HandleTheChangeState() method on the ShipState. Since the ship was in the ReadyToFireMode state, it is handles by the HandleTheChangeState() method in that class. That method assigns a new WaitingForMissileToExplodeMode() object to the ShipState. Until the state is changed backed to ReadyToFireMode, the ship cannot fire a missile. The UML is shown below:



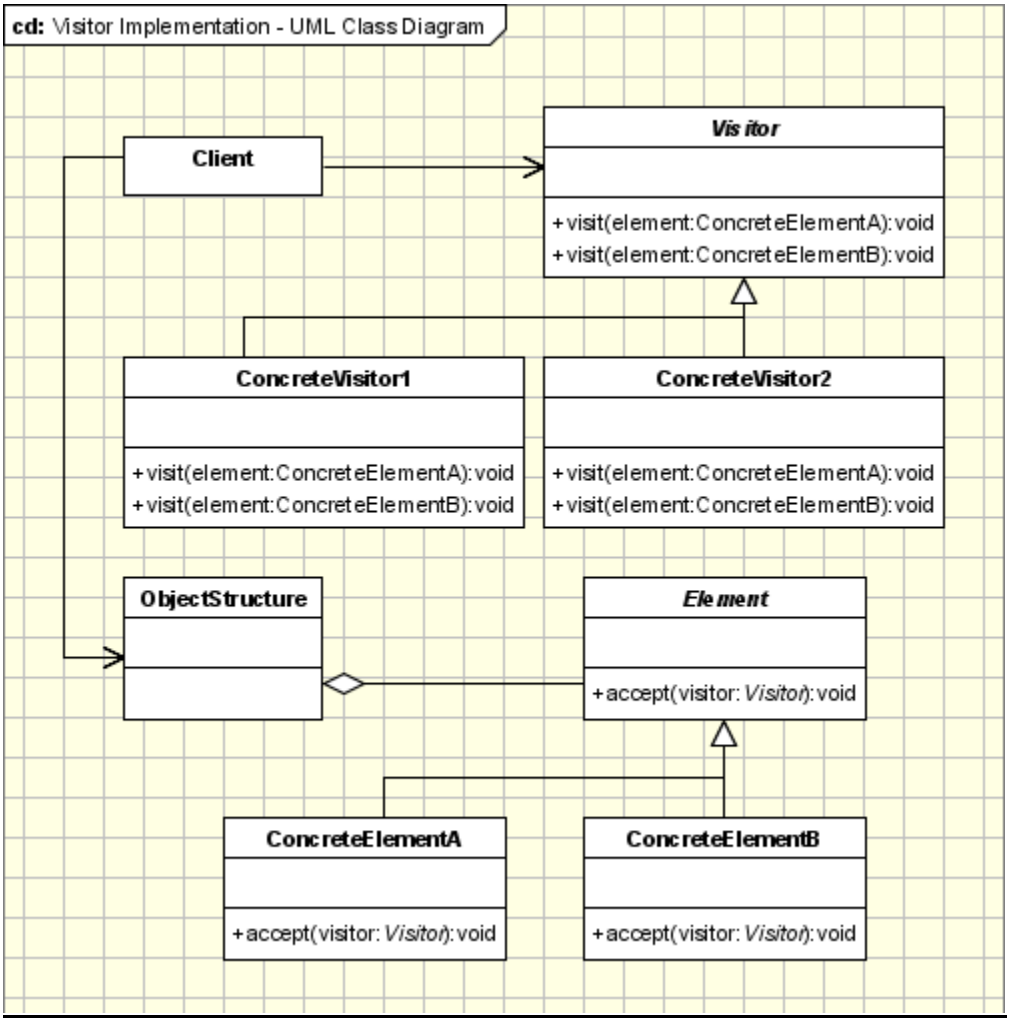
Strategy For Choosing an Alien?



Visitor Pattern to test for Collisions

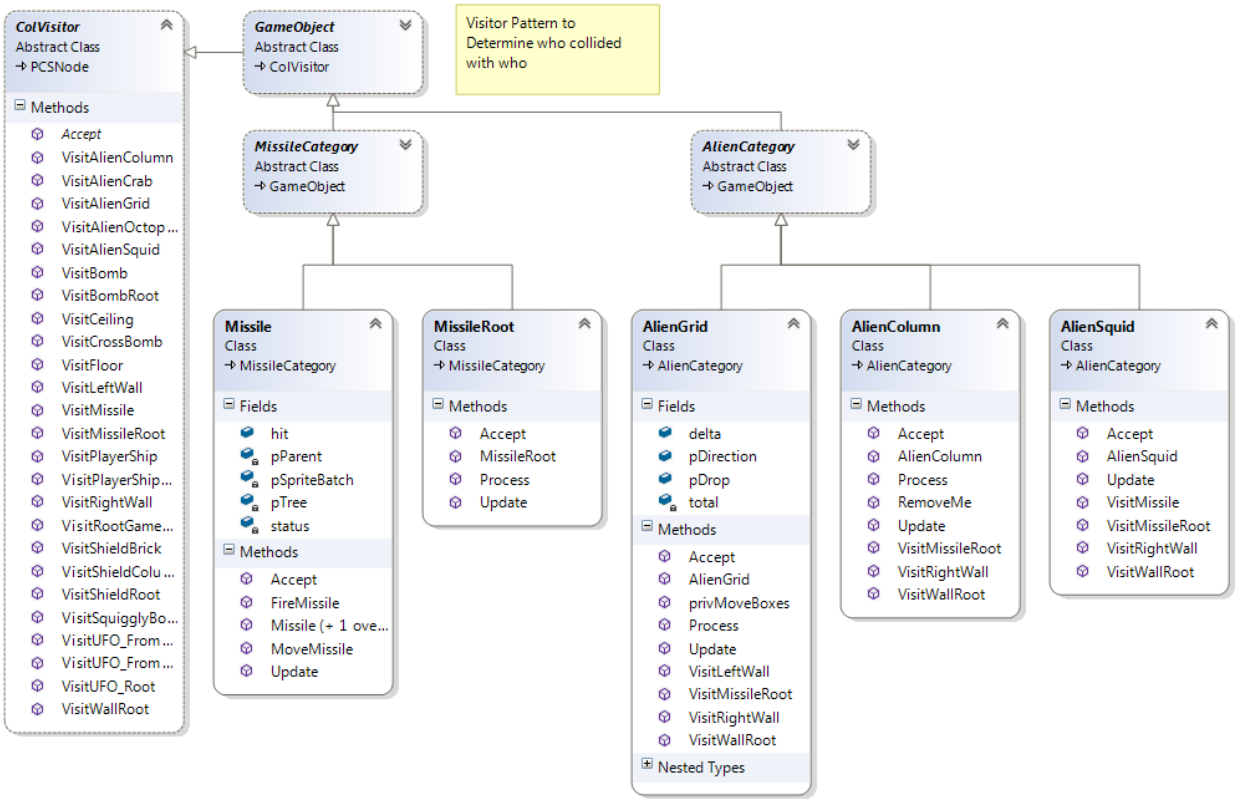
The purpose of the Visitor pattern is to add functionality to a group of objects and you are not worried about encapsulation. That means you don't care if objects know about the internal representation of other objects. The Visitor pattern uses double dispatch which means different function calls will be made depending on which types of objects are making the calls.

As illustrated below, ConcreteElementA will call Accept(ConcreteElementB). In ConcreteElementA's Accept() method, ConcreteElementB will call its own VisitConcreteElementA(ConcreteElementB) method. This allows the program to jump from element to element. The next element that the visitor goes to is determined by which Element accepts which element as determined by the algorithm.



In “Space Invaders” the Visitor pattern is used to determine who collided with who. That is the problem that it solves. The mechanism of how it is solved is this. If a MissileRoot and the AlienGrid collide, then in the ColPair class, MissileRoot will call Accept() with AlienGrid as the parameter. Since GameObject inherits from ColVisitor all the GameObjects must implement the abstract method Accept(). They also have the option to implement the virtual methods. This optionality is what allows different objects in the game to collide or not collide with other objects in the game. It allows the visitor pattern to follow different algorithmic paths depending on what objects collided.

In MissileRoot’s Accept() method, AlienGrid will call its own VisitMissileRoot() method with MissileRoot as the parameter. The VisitMissileRoot(MissileRoot m) will call Colpair.Collide() with the MissileRoot and the child of the AlienGrid which is the AlienColumn. This process will continue until AlienColumn calls Colpair.Collide with the MissileRoot and the AlienSquid. Then AlienSquid will call VisitMissileRoot() which will in turn call the Colpair.Collide() between the actual Missile and the AlienSquid. There are no more children at this point so it is determined that the AlienSquid and the Missile have collided. Now it is time to notify the observers to have some action take place. This is shown down below:



POST-MORTEM

Improvements

There are many systems that need to be improved in my game. To start the Shield system is not proportional. They need to be smaller. I was unable to make my bomb dropping system very efficient. I need bombs to drop from more columns in the Grid. I was unable to get the splash screen going, nor was I able to get Player2 working.

Commentary

My first thought that comes to mind is how to I improve my understanding of the class material so I can spend more time on the software architecture side of this class. I had so much trouble getting the demos to work and understanding the internal structure of the game, TimerManager, Collision System, CollisionBoxes etc, I didn't have enough time for the fun part which is getting the missiles to fire, explosions to occur, UFO's to appear, players to change, high score and others. Did I not spend enough time on Perforce? That is probable. However, I see Perforce as a chicken and egg idea. The guys and girls who are ahead of the game have time to spend on Perforce helping others and showing off their cool games. Students who struggle are spending time trying to get the demos to work. Maybe I had too much faith in the demos. Maybe I just lack enough self confidence in myself and I need a Lot more practice.