SPACE INVADERS DESIGN DOC

By: Ashish Batra

Pattern Name	**Page Start**
ITERATOR	2
SINGELTON	5
OBJECT POOL	8
PROXY	11
COMPOSITE	14
FACTORY	17
COMMAND	19
OBSERVERS	22
STATE	25
STRATEGY	28

Pattern at hand: - Iterator

General use

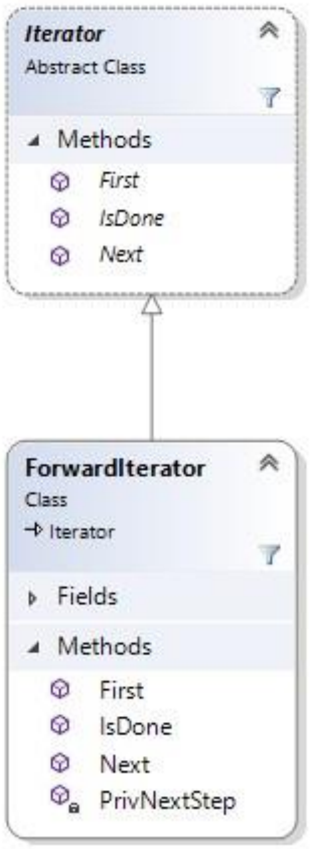
- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- Provides an abstraction that makes it possible to decouple collection classes and algorithms

Problem in Space Invaders

- In the game how to get a particular node in the tree of composite's like AlienGrid, ShieldGrid etc. in other words how to access the elements of aforementioned aggregate object sequentially

Solution in Space Invaders

- In the game if we want to get a particular node in the tree of composite(AlienGird, ShieldGrid etc.) forward iterator provides a guarantee to go through the entire tree top to bottom(bottom to top with reverse iterator) to find and return that particular node.



How the pattern works in game

- The main three methods that are at play are First() isDone() and Next(), First() and Next() return the composite(aggregate object).

- First():- this method is used to access the root(top node) of the composite(aggregate object)

- Next():- this method is used to access the data structure which in our case are doubly linked lists sequentially. The implementation of Next() would depend on the aggregate object we are trying to traverse. We call PrivNextStep() within Next() which holds the traversal logic.

- isDone():-this method helps us determine if we are at the end of the data structure we were traversing

Use example

```
public void MoveGrid()
    {
         ForwardIterator pFor = new ForwardIterator(this);
         Component pNode = pFor.First();
         while (!pFor.IsDone())
         {
             GameObject pGameObj = (GameObject)pNode;
             pGameObj.x += this.deltaX;
             pNode = pFor.Next();
         }
    }
```

- In the above code forward iterator is being used to traverse the AlienGrid to set movement of each Invader. The implementation would not have been this clean and succinct if we would have not used the iterator.

Final thoughts

It is not a necessity to use an iterator to traverse through aggregate objects but using one immensely improves the readability of the code while at the same time and provides abstraction to the traversal of the collection

Pattern at hand: - Singelton

General use

- Ensures a class has only one instance, and provide a global point of access to it.
- Encapsulated "just-in-time initialization" or "initialization on first use".

Problem in Space Invaders

- Need to have Managers for various components of the game for instance Texture Managers, Image Managers, CollisionPair Managers.
- Need to have global access to these Managers throughout the application

Solution in Space Invaders

- Singelton helps provide the much-needed support for globally accessing the various Managers and their member function like Add(),Create(),Find(),Remove() etc.

How the pattern works in game

- The methods that implements the singleton in the game’s Managers is Create it performs a check, if(pInstance == null) only then create a New pInstance.

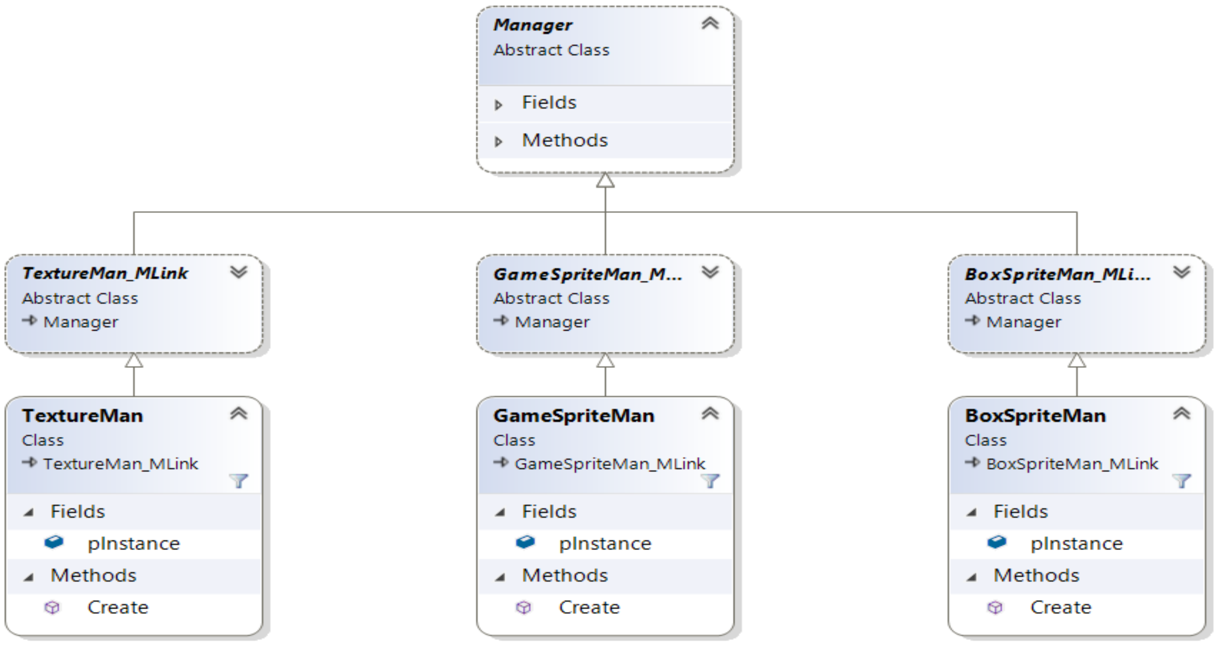
Use example

```
public static void Create(int reserveNum = 1, int
reserveGrow = 1)
   {
       // make sure values are ressonable
       Debug.Assert(reserveNum > 0);
       Debug.Assert(reserveGrow > 0);

       // initialize the singleton here
       Debug.Assert(pInstance == null);

       // Do the initialization
if (pInstance == null)
{
pInstance = new TextureMan(reserveNum, reserveGrow);
// NullObject texture

Texture pTexture =  TextureMan.Add(Texture.Name.NullObject,
                                        "HotPink.tga");
Debug.Assert(pTexture != null);
          // Default texture
pTexture TextureMan.Add(Texture.Name.Default,
                              "HotPink.tga");
       Debug.Assert(pTexture != null);
}}
```



- In the above code singleton pattern is being used to check if pInstance is not null and only then create the new Object Pool

Final thoughts

Singelton is a very easy and useful pattern to not only provide global access to a class but it's member functions as well. It is like having a global variable in a way but it is very well encapsulated.

Pattern at hand: - Object Pools

General use

- Ensures to Not recreate (new) many same objects by reusing them in a clean way.
- Best to use this pattern when cost of initializing a class instance is high, the rate of instantiation of a class is high, and the number of instantiations is finite.

Problem in Space Invaders

- Don't want to pay the cost of creation and destruction because they are all initialized in the same state.

Solution in Space Invaders

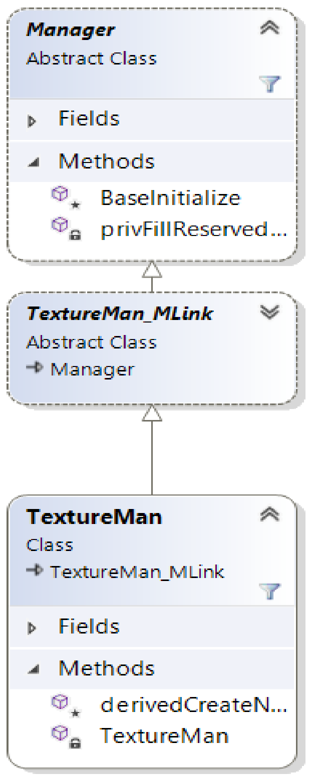
- Using object pooling concept to create the required objects deepening upon Initial Reserved and pool growth parameters.

- Do not delete the object upon removal add it to reserve list instead. When in need of more objects just pluck it out from the reserve and add it to the active list and grow the size of the list when I need

How the pattern works in game

- A bunch of methods are required to set up pooling few major ones are

- BaseInitialize(), privFillReserved() :- these are base class methods TextureMan(),derivedCreateNode() :- these are specific to derived class.
- parameters provided when creating using TextureMan() are passed onto BaseInitialize() which in turn calls privFillReserved() passing on Initial Reserved(number of reserve nodes to start with) BaseInitialize() then handles the creation of appropriate object in this case texture and loops InitialReserved times to create the appropriate amount.



Use example(Recycling)

```
protected void baseRemove(DLink pNode)
 {
  Debug.Assert(pNode != null);

  // Don't do the work here... delegate it
  Manager.privRemoveNode(ref this.poActive, pNode);

  // wash it before returning to reserve list
  this.derivedWash(pNode);

  // add it to the return list
  Manager.privAddToFront(ref this.poReserve, pNode);

  // stats update
  this.mNumActive--;
  this.mNumReserved++;
 }
```

- upon removing we just add it back to the reserve list and decrees the number of active nodes and increase reserved

Final thoughts

Object pools provides us with the Much needed performance in a Realtime system. Although they are bit of work to setup if the application really needs the performance the trade off is pretty good

Pattern at hand: - Proxy

General use

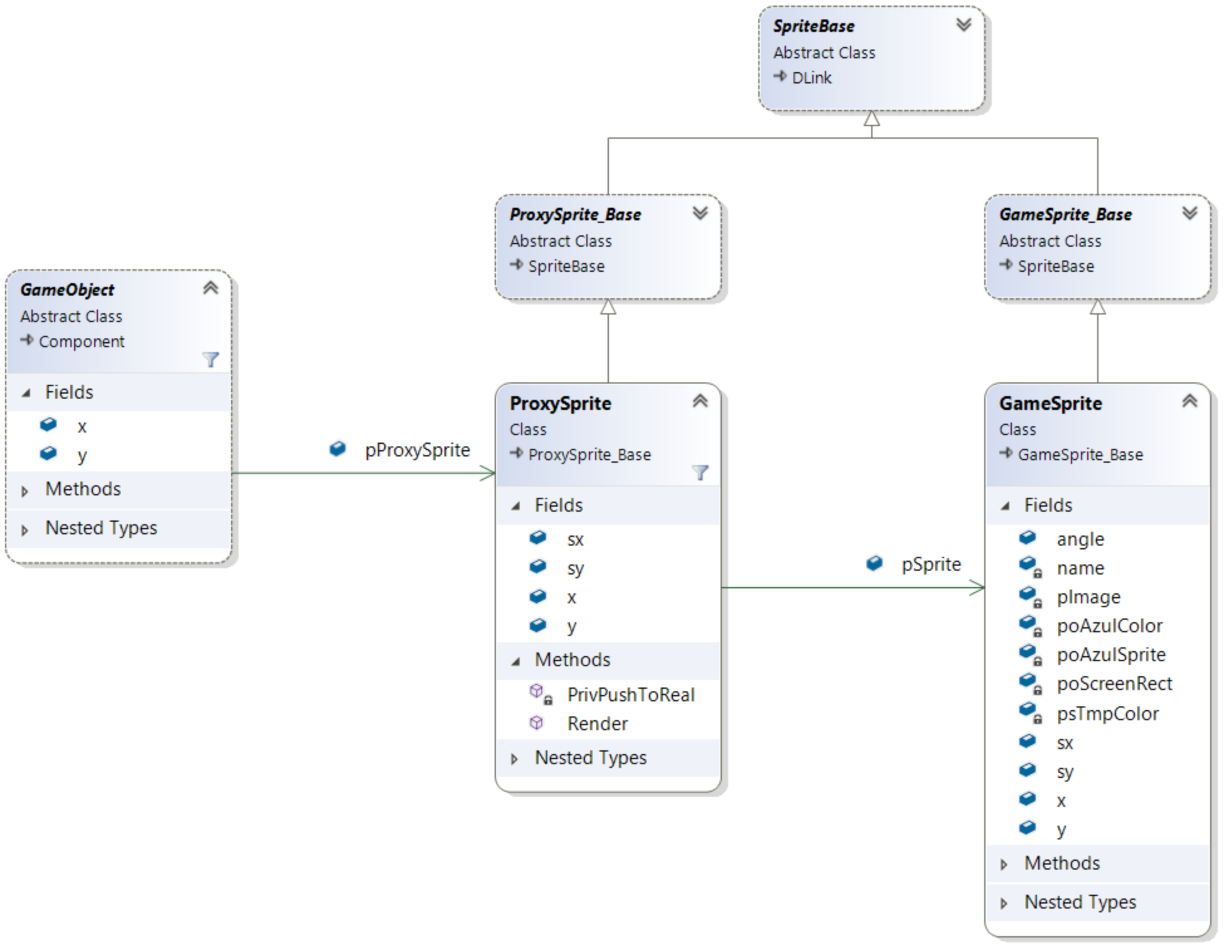
- Provide a surrogate or placeholder for another object to control access to it .
- Use an extra level of indirection to support distributed, controlled, or intelligent access.
- Add a wrapper and delegation to protect the real component from undue complexity .
- Add a wrapper and delegation to protect the real component from undue complexity .

Problem in Space Invaders

- We Want to have animated sprites Crab, Squid, Octopus and Reuse them in the Alien Grid
- We want to Set them up once and through an Automated process update their Unique instance which is position

Solution in Space Invaders

- Proxy objects Reuse the existing object which is Only one Animated GameSprite
- The only data that is different is the Instance data Position so we Overwrite the differing data



How the pattern works in game

- Instead of doing all the work we need to do in order to render a game sprite and have a long list of variables attached to our object we just push the data that needs to change in the Gamesprite

Use example

```
  private void PrivPushToReal()
  {
// push the data from proxy to Real GameSprite
Debug.Assert(this.pSprite != null);
this.pSprite.x = this.x;
this.pSprite.y = this.y;
this.pSprite.sx = this.sx;
this.pSprite.sy = this.sy;
   }
```

- proxySprite has a function PrivPushToReal() this method just updates the necessary values like the position and scale(not really needed) within the Game sprite to Render just calls PrivPushToReal()
- We Can have someone else animate the Sprite (Animation Sprite class)is flipping the Images
- All the proxy sprites on screen benefit form it and we save ourselves a lot of work otherwise

Final thoughts

Proxies are very beneficial when we have to same task over and over with a slight change. Our aliens in the grid fit the aforementioned vague description perfectly and when you have 5*11 objects you want to animate and you can get away with animating just 3 it’s a big win.

Pattern at hand: - Composite

General use

- Compose objects into tree structures to represent part-whole hierarchies.
- Composite lets clients treat individual objects and compositions of objects uniformly

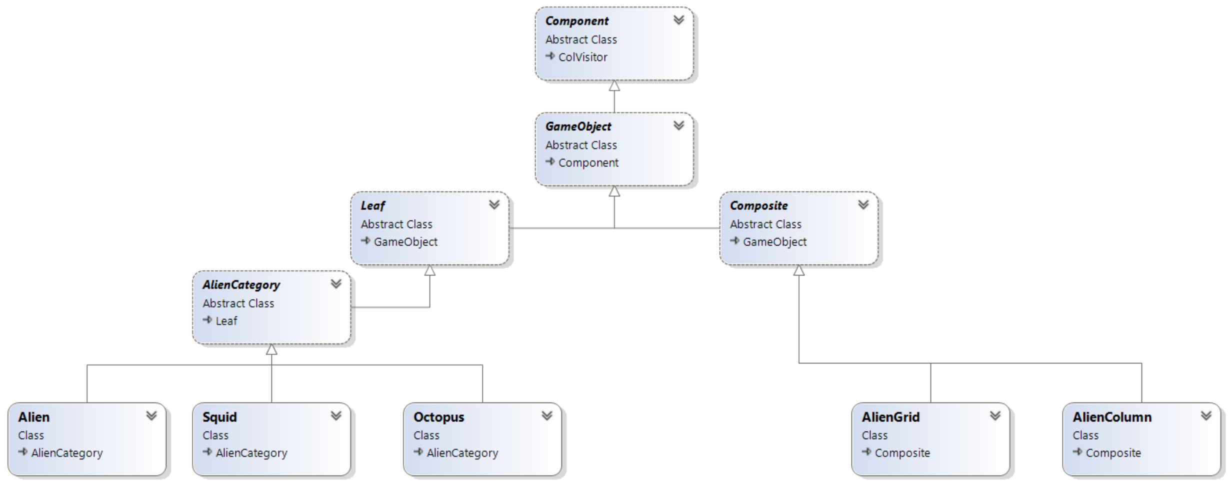
Problem in Space Invaders

- How To organize data in a Hierarchy & Allow data / behavior / collisions to interact
- How To Move data into structural relationships
- How To create Shield system as a grid with rows and columns of bricks , How to ensure Movement of Aliens as a group

Solution in Space Invaders

- using the composite pattern allows us to store various objects in an hierarchical manner this in turn helps us do operations on them as a whole or at the individual level(using iterators)
- For instance our alien grid is arranged in a composite pattern with a root that holds a number of columns in our case 15 then each of those columns hold 5 aliens (Squid,Crab,Octopus)

- Another example are the shields which consist of the same pattern as the aliens except the column now holds different kinds of shield bricks
- Other miscellaneous examples include the missile and bomb etc.



How the pattern works in game

- If we want to operate on the whole gird we go down the composite tree and then to apply the functionality to individual objects we traverse the leaf side of the tree and find the object using an iterator
- Everything below one node is a leaf of the given node for example columns are leaf for the grid and aliens are leaf for the columns
- Below is one of the use examples in which using a factory I am filling the alien grid with aliens by first creating an alien and attaching it to the column reaping this for all 5 rows and then do this again for each column and attach the columns to the grid

Use example

```
GameObject pGameObj = null;

// create the factory - needs reworking
AlienFactory AF = new AlienFactory(SpriteBatch.Name.SpaceInvaders,
SpriteBatch.Name.Boxes);

GameObject pGrid = AF.Create(GameObject.Name.AlienGrid, AlienCategory.Type.Grid);


for (int i = 0; i < 11; i++)
{
GameObject pCol = AF.Create(GameObject.Name.Column_0 + i, AlienCategory.Type.Column);
pCol.ActivateCollisionSprite(pSB_Aliens);

pGameObj = AF.Create(GameObject.Name.Squid, AlienCategory.Type.Squid, 50.0f + 66 * i,
900.0f);
pCol.Add(pGameObj);

pGameObj = AF.Create(GameObject.Name.Alien, AlienCategory.Type.Alien, 50.0f + 66 * i,
834.0f);
pCol.Add(pGameObj);

pGameObj = AF.Create(GameObject.Name.Alien, AlienCategory.Type.Alien, 50.0f + 66 * i,
768.0f);
pCol.Add(pGameObj);

pGameObj = AF.Create(GameObject.Name.Octopus, AlienCategory.Type.Octopus, 50.0f + 66 *
i, 702.0f);
pCol.Add(pGameObj);

pGameObj = AF.Create(GameObject.Name.Octopus, AlienCategory.Type.Octopus, 50.0f + 66 *
i, 636.0f);
pCol.Add(pGameObj);

pGrid.Add(pCol);
}
   GameObjectMan.Attach(pGrid);
```

Final thoughts

Using a composite is a good trick to works on data that makes sense to be arranged in an hierarchy. In retrospect I do not see any other way to have an alien grid without using the composite.

Pattern at hand: - Factory

General use

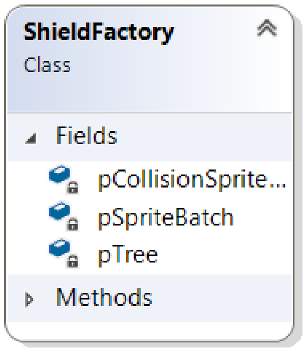
- Creates objects without exposing the instantiation logic to the client.
- Refers to the newly created object through a common interface

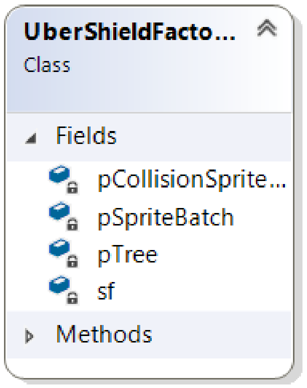
Problem in Space Invaders

- How To do repetitive work of creating individual aliens in the alien grid
- How To do even more repetitive work of creating individual alien bricks
- How to do all this and not make the code look overwhelming
- How to do all this and hide the underlying creation mechanism from the user

Solution in Space Invaders

- using Factories for aliens and Shields become more and necessary when we a further along the development process and the prototyping is at a point where we have something resembling the actual game one the screen
- A very common theme with factories emerge which goes like. “Make the factories suite the data you want to create” which makes sense because doing factories too early would lead to changing the factory a lot to suit our data needs


ShieldFactory
Class
Fields
pCollisionSprite...
pSpriteBatch
pTree
Methods


UberShieldFacto...
Class
Fields
pCollisionSprite...
pSpriteBatch
pTree
sf
Methods

How the pattern works in game

- We initialize the shield factory with the shields sprite batch the collision spritebatch and the pointer to the composite shield root
- The uber shield factory then takes that root and initializes all the column and induvial bricks under the hood using the shield factory hiding the initialization logic from the user

Use example

```
// Create the factory ... prototype
            Composite pShieldRoot = (Composite)new ShieldRoot(GameObject.Name.ShieldRoot,
GameSprite.Name.NullObject, 0.0f, 0.0f);
            GameObjectMan.Attach(pShieldRoot);

            ShieldFactory    SF     =     new     ShieldFactory(SpriteBatch.Name.Shields,
SpriteBatch.Name.Boxes, pShieldRoot);

            UberShieldFactory USF = new UberShieldFactory(SF, pShieldRoot, 100, 200);
            UberShieldFactory USF2 = new UberShieldFactory(SF, pShieldRoot, 300, 200);
            UberShieldFactory USF3 = new UberShieldFactory(SF, pShieldRoot, 500, 200);
                UberShieldFactory USF4 = new UberShieldFactory(SF, pShieldRoot, 700, 200);
```

Final thoughts

Using Factories we can do the initialization of the shields just using the above lines instead of 140*4 lines of code, this is still not considering the benefits of aliens factories

Pattern at hand: - Command

General use

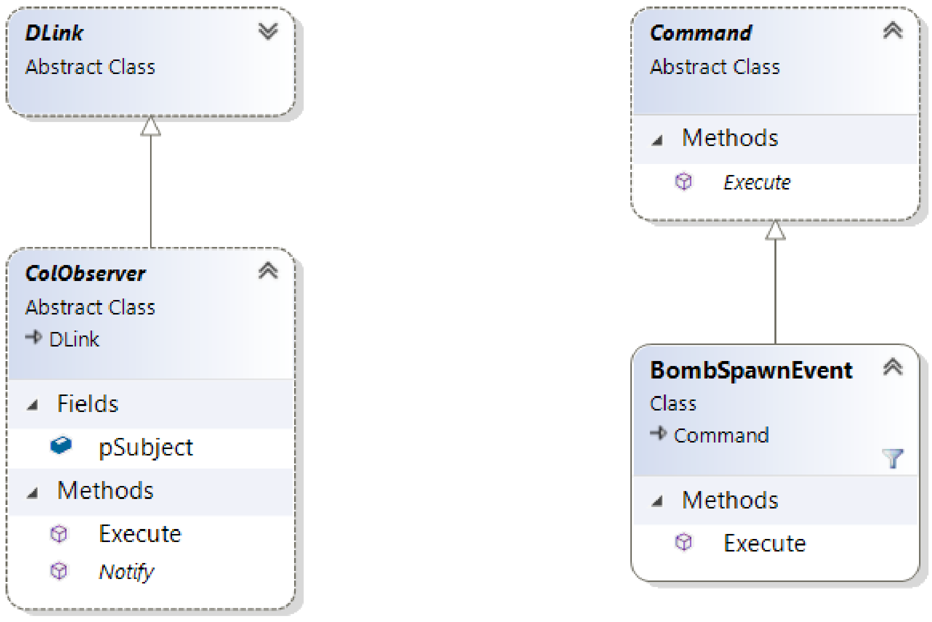
- Encapsulate a request as an object, thereby letting you parametrize clients with different requests,
- Helps queue or log requests, and support undo operations.
- Promote "invocation of a method on an object" to full object status

Problem in Space Invaders

- How to trigger various time events like dropping of bombs animation of the aliens sound of alien grid
- How to remove objects using a delayed object manager so we do not end up deleting the linked list we are walking

Solution in Space Invaders

- command pattern helps us trigger event depending upon time that we can pass in the execute. For example trigger time for dropping the bombs
- we can specialize the execute as per our need therefore I am not passing anything in the execute of Collision observer so when the delayed object manager comes around for removing the markedfordeath objects it just triggers various execute() in a list of (markedfordeath==true) object’s



How the pattern works in game

- just to point out that various patterns can be combined Collison observer is both a command and an observer
- both ColObserver and Command have an execute in them which indicates the presence of command pattern the abstract method of execute differs in signature hinting at the difference in application of both the commands
- the pattern used for ColObserver is as described above for the workings of delayed object managers and on the other hand
- BombSpawnEvent is a timer event which works by adding itself back to the timer because the way our Timmer class works is it triggers an event based on the timestamp of the even then removes the timer event. So if we want more than one bombs to drop we need to add back the timer event to itself

Use example

```
abstract public class ColObserver : DLink
    {
        public abstract void Notify();

        // WHY not add a state pattern into our Observer!
        public virtual void Execute()
        {
            // default implementation
        }
        public ColSubject pSubject;


       }


public abstract class Command
    {
        // define this in concrete
        abstract public void Execute(float deltaTime);
       }
```

Final thoughts

- The two methods above indicate the different ways in which command pattern can be implement and within the same application it can take care of different kinds of task
- One big thing to keep in mind and understand is, when to use it one example is whenever we want to make some behavior to happen continually. Another way to think of command pattern is to think of it as a trigger

Pattern at hand: - Observers

General use

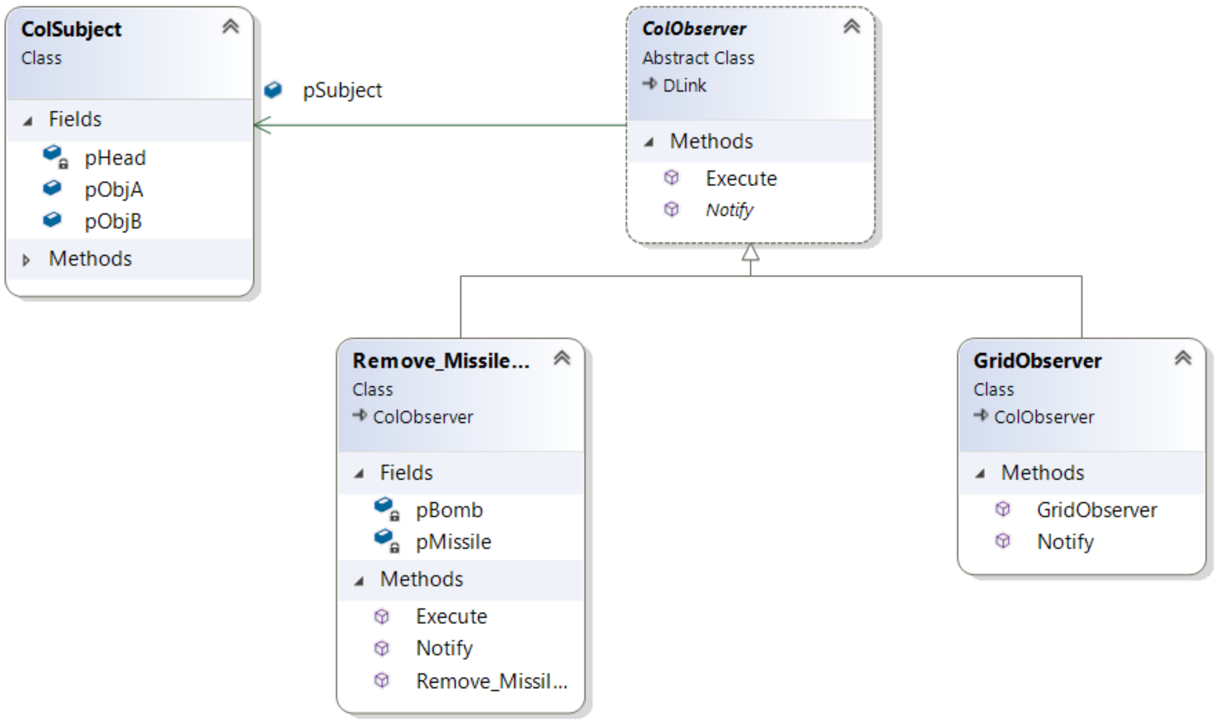
- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Observer is used when the Observable needs to notify other objects about certain events, but doesn't know *which* objects to notify.

Problem in Space Invaders

- How to trigger various actions like deletion of aliens, deletion of bombs etc.
- How to play sound on collisions for example player destruction sound

Solution in Space Invaders

- Using observer pattern in combination with visitor pattern we can detect the in game collisions using the visitor and upon their detection trigger the appropriate event
- As an example I am using various observers in game namely
    - alien remove observer:- to remove the alien
    - missile bomb observer:- to remove missile and bomb upon collsion etc.



How the pattern works in game

- the way the pattern works is there are N number of observers attached to a subject and depending upon the state of the subject the appropriate observer is called
- when the collision subject is grid and when grid hits the wall, with the help of the appropriate visitor collision is detected and the grid observer is called
- On the other hand when a bomb hits a missile same process is repeated but the subject now is missile bomb col observer there remove missile bomb is called

Use example

```
//bomb vs missile

            pColPair = ColPairMan.Add(ColPair.Name.Missile_Bomb, pMissileGroup, 
pBombRoot);
            pColPair.Attach(new Remove_Missile_bomb());
            pColPair.Attach(new ExplosionObserver(sndEngine));
                                pColPair.Attach(new ShipReadyObserver());


// grid vs wall
            ColPair pColPair = ColPairMan.Add(ColPair.Name.Alien_Wall, pGrid, 
pWallGroup);
            Debug.Assert(pColPair != null);
                                  pColPair.Attach(new GridObserver());
```

Final thoughts

The two examples above makes it clear that observers are things that need to happen upon the change in subject. When the subject it bomb and missile we might need to do more things compared to the collision between wall and grid which only requires us to change the moving direction of the grid.

Pattern at hand: - State

General use

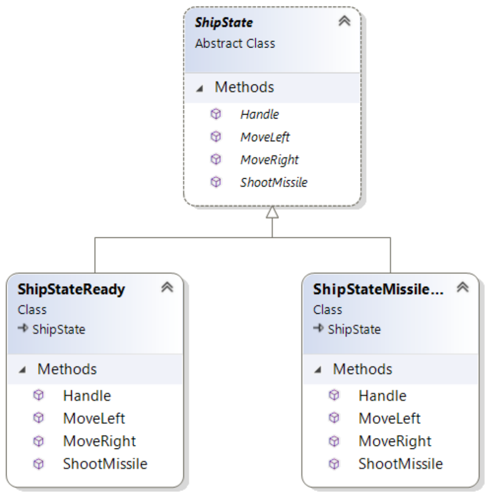
- Allow an object to alter its behavior when its internal state changes. The object will appear to change its class .
- Unlike a procedural state machine, the State Pattern represents state as a full-blown class

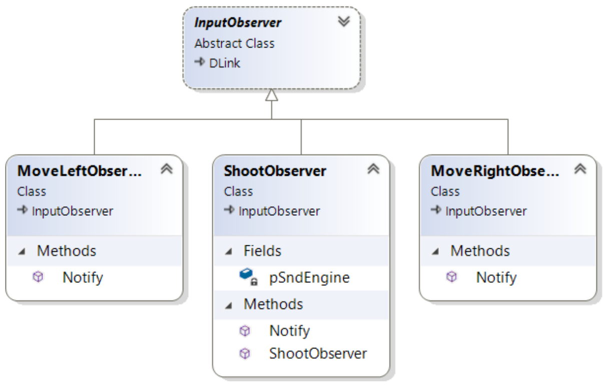
Problem in Space Invaders

- How to stop ship from shooting misses when a missile is already in the air or when the ship is dead
- How to change scenes from splash->Game->Game Over

Solution in Space Invaders

- Using a combination of state and observer pattern we are able to control the movement and shooting of the ship by setting the state to ready when the missile is not in the air and the ship it ready to shoot, setting the state to missile flying when the missile is in the air and in turn disabling shoot functionality until the missile collides with something(this is done by having an abstract shoot method and when the state is flying shoot does nothing)
- Another big task which is essentially making the whole game work, is switching between the game scenes and state pattern helps out with that





How the pattern works in game

- the way the pattern works is depending on the input move left, shoot and move right observers are triggered and they call the ship's method.
- The ship's method in turn calls this.state.MoveLeft() for instance, and the respective implementation is processed

Use example

```
class ShipStateMissileFlying : ShipState
{
public override void Handle(Ship pShip)
{
    pShip.SetState(ShipMan.State.Ready);
}


public override void MoveRight(Ship pShip)
{
    pShip.x += pShip.shipSpeed;
}

public override void MoveLeft(Ship pShip)
{
    pShip.x -= pShip.shipSpeed;
}

public override void ShootMissile(Ship pShip, IrrKlang.ISoundEngine pSndEngine)
{
}}
```

Final thoughts

Looking at the code on the previous page it becomes clear that using a state pattern can help in behavior change without if else conditionals which produce the most bugs in an application. Using this pattern to change the state of the scene in the game proved to be essential

Pattern at hand: - Strategy

General use

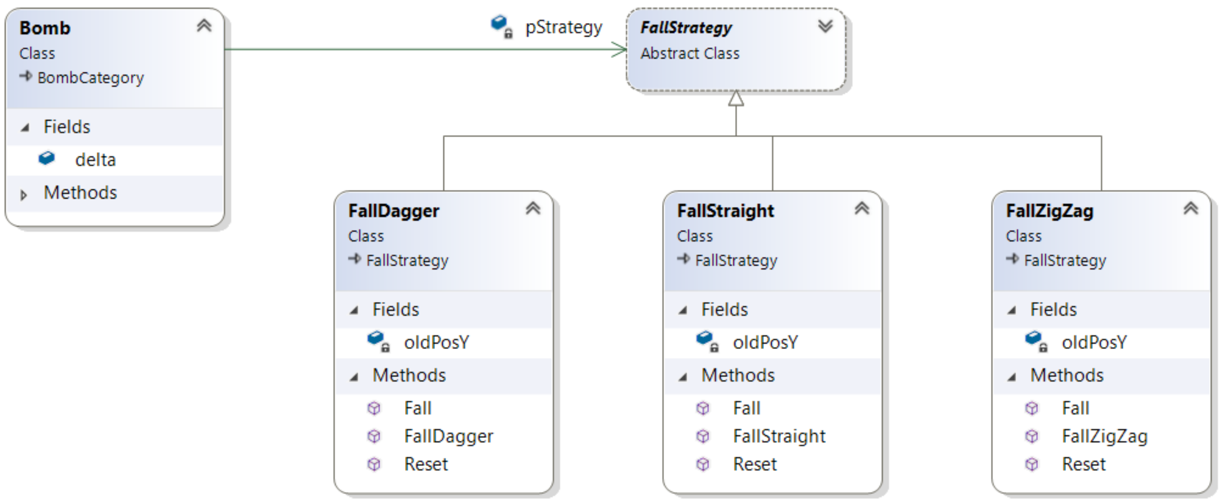
- Define a family of algorithms, encapsulate each one, and make them interchangeable.
- Strategy lets the algorithm vary independently from the clients that use it .
- Capture the abstraction in an interface, bury implementation details in derived classes.

Problem in Space Invaders

- How to make the falling bombs animate. We can use the command pattern to animate the bombs like the aliens but how to do it with lesser work.

Solution in Space Invaders

- Using 3 kinds of fall strategies we can switch how the bomb animate while falling without having to crate sperate class for each bomb
- The algorithm of how the animation works is hidden inside the bomb implementation which switches between the animation algorithm depending upon the strategy we ask for



How the pattern works in game

- The BombSpawnEvent spawns a bomb every second and when the new is called it randomly switches between the fall strategy
- In the bomb constructor depending upon the strategy parameter received from the BombSpawnEvent the Bomb picks its strategy through a reference to the collection of strategies.

Final thoughts

Strategy pattern has much in common with the state pattern, but it differs in its intent. Strategy Pattern is a flexible alternative to sub-classing while State Pattern is an alternative to putting lots of conditionals in your context

Use example

```
        private Bomb BombSelector(ColObject GridColumsBox)
        {
            float val = pRandom.Next(0, 3);
            Bomb pBomb = new Bomb(GameObject.Name.Bomb, GameSprite.Name.NullObject,new
FallStraight(), 0,0);
            if (val == 0)
            {
                Bomb Bomb = new Bomb(GameObject.Name.Bomb, GameSprite.Name.PlungerShotA,
                new FallDagger(), GridColumsBox.poColRect.x, GridColumsBox.poColRect.y -
GridColumsBox.poColRect.height / 2);
                pBomb = Bomb;
            }
            if (val == 1)
            {
                Bomb Bomb = new Bomb(GameObject.Name.Bomb, GameSprite.Name.RollingShotA,
                new FallStraight(), GridColumsBox.poColRect.x, GridColumsBox.poColRect.y
- GridColumsBox.poColRect.height / 2);
                pBomb = Bomb;
            }
            if (val == 2)
            {
                Bomb Bomb = new Bomb(GameObject.Name.Bomb,
GameSprite.Name.SquigglyShotA,
                new FallZigZag(), GridColumsBox.poColRect.x, GridColumsBox.poColRect.y -
GridColumsBox.poColRect.height / 2);
                pBomb = Bomb;
            }

            return pBomb;
           }
```