Kerwin Ghigliotty Rivera

Design Document for Final Fantasy Demo

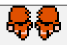
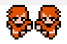
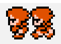
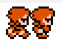
Contents

High Level Overview

The goal of this project is to recreate a simple town from Final Fantasy®

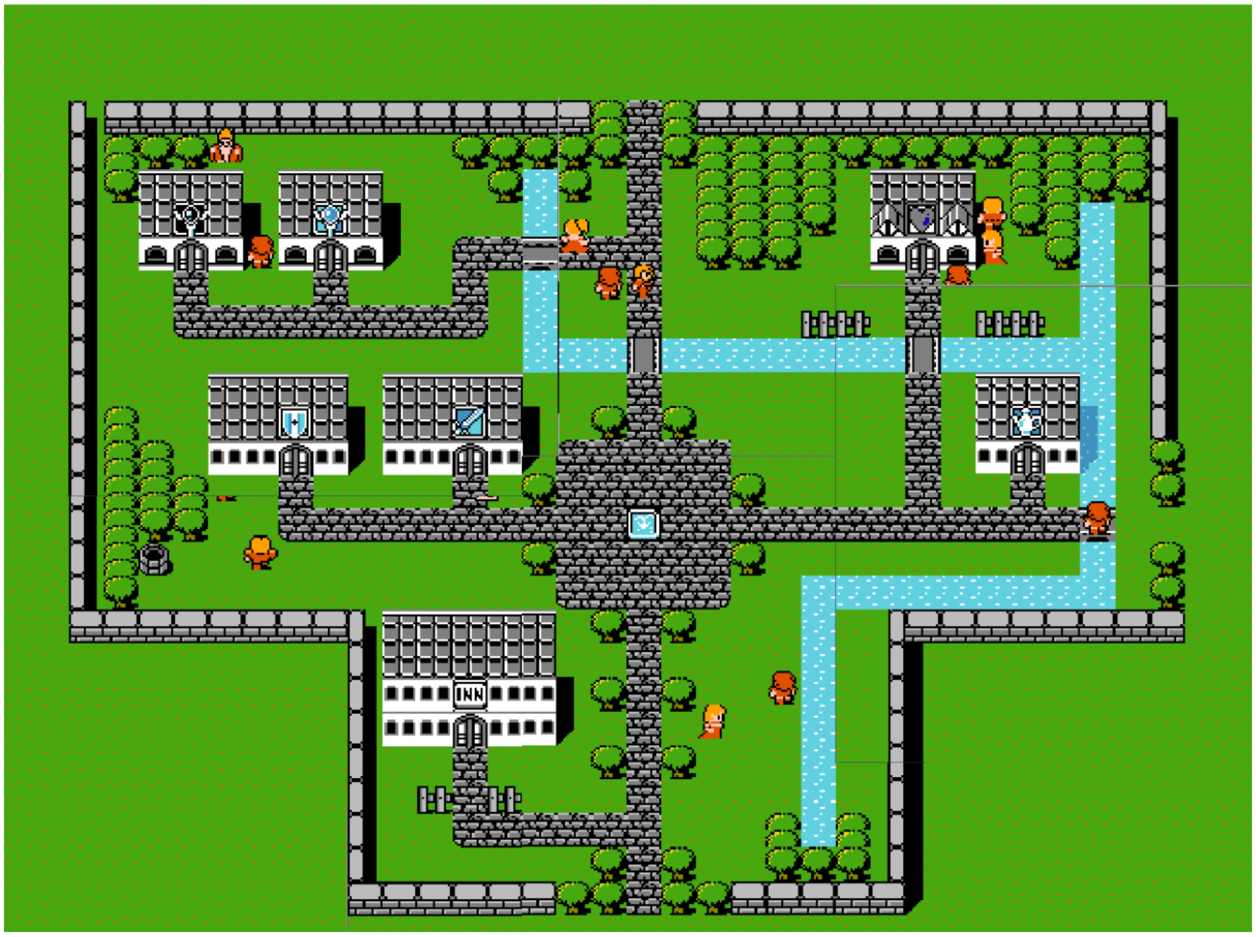
The player will be in control of the main character (The Fighter in this case) and will be able to move around the town, interact with other characters as well as enter buildings and lastly enter a battle sequence. As for the NPCs, until I can get their appropriate sprites, I will be using the sprites of the other jobs to represent them.

Movement options and their associated sprite

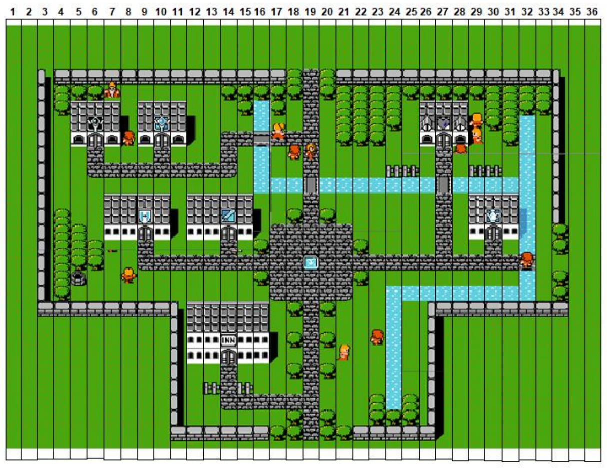
Movement	Sprite (1 per step)
UP	
DOWN	
LEFT	
RIGHT	

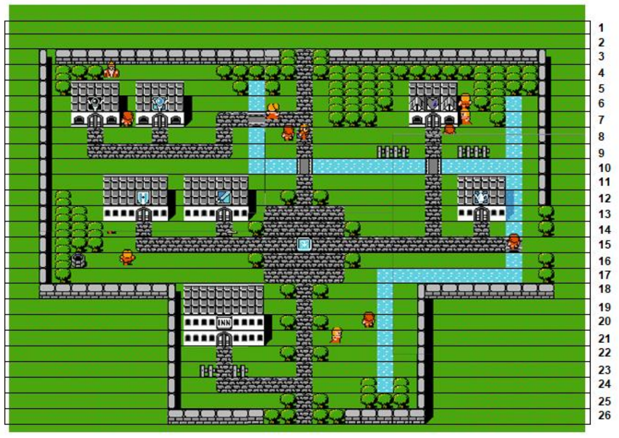
The Town (Full image was taken from the main game)

This is the objective, to have the full town created.



Composed of 36 Columns and 26 Rows





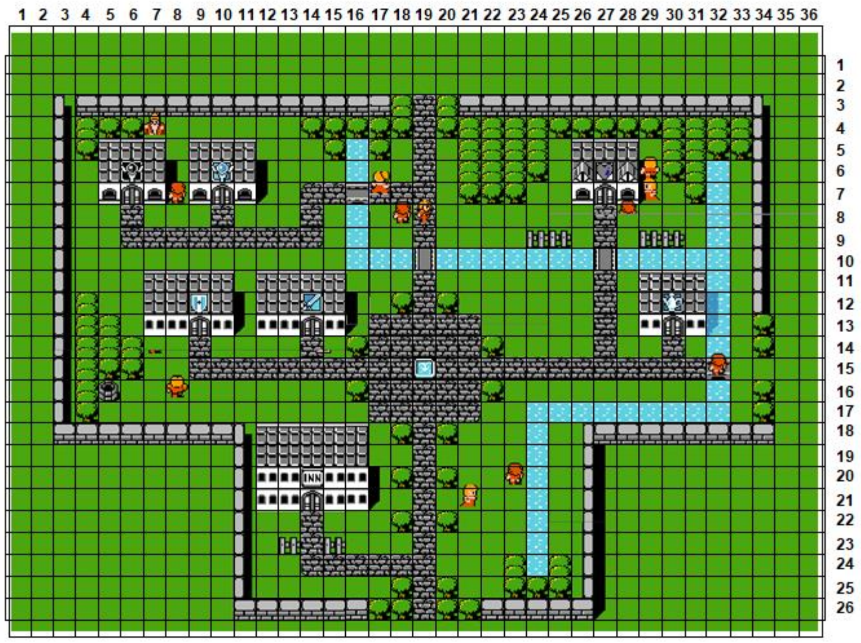
We are working with a 36x26 (X, Y) grid.

In which:

Small Houses = 3x3

Medium Houses = 4x3

Large Houses = 5x4



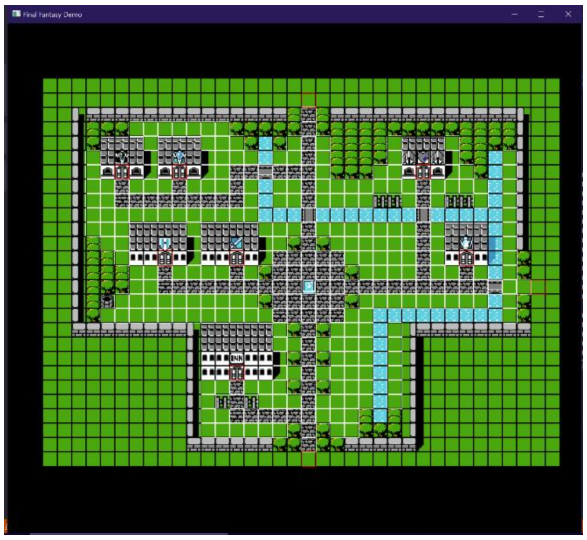
Output of the MapFactory::CreateFullTown Method

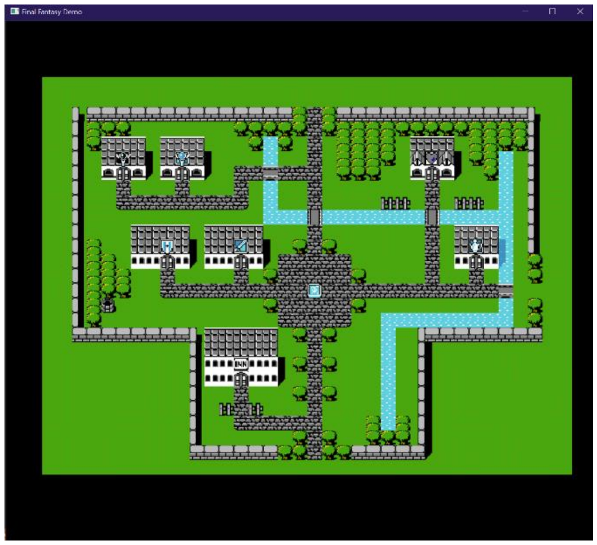
Collision Boxes set

Red = Entrances

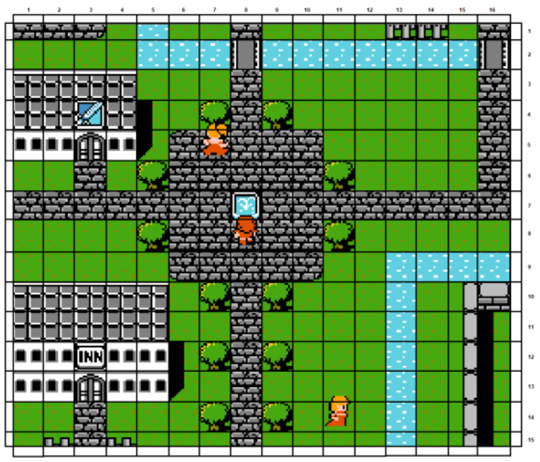
White = Walkable Ground

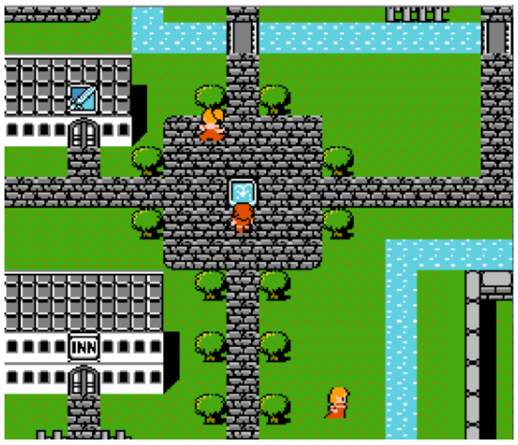
Black = Walls





The problem now is to get it to only render the squares in a 16x15 pattern (7 rows above/below the player character and 7 columns to the left 8 columns to the right of the player character) anything outside of this grid should not be rendered.

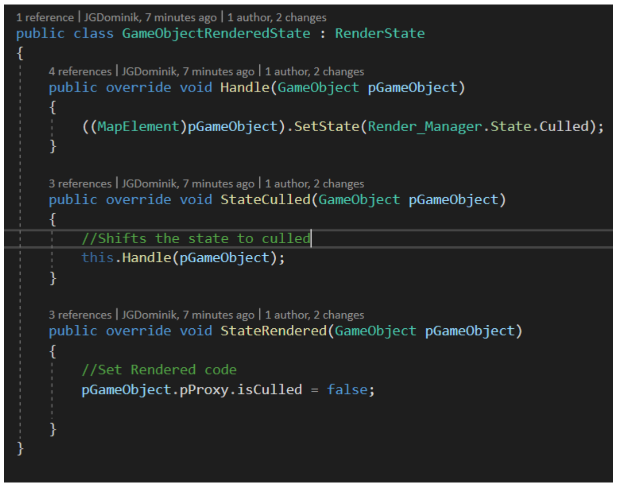




Note:
This grid includes half of the topmost and bottommost pixels (they should be rendered but blocked by screen dimensions)
NPCs and Player Character are Drawn a bit higher than normal pixels (explains the overlap of characters on multiple squares)

Output of the SpriteBatch_Manager::RenderCamera method when combined with the RenderState class for each element, once they are within the camera range (shown in blue square on the right) the state changes to StateRendered, once the elements are out of the Camera range the state shifts to StateCulled

[Showcased in this video - https://youtu.be/rO8HL4sU4qU](https://youtu.be/rO8HL4sU4qU)

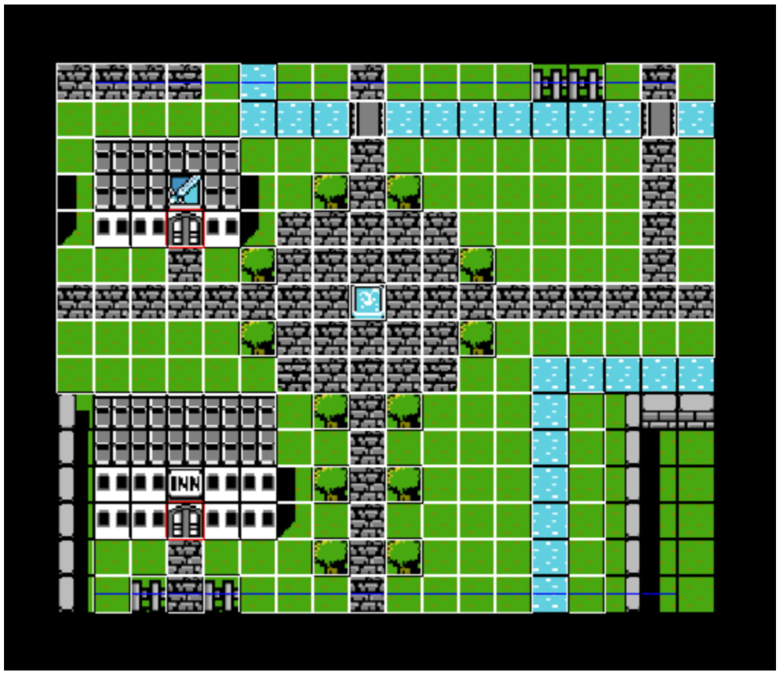


```
1 reference | JGDominik, 7 minutes ago | 1 author, 2 changes
public class GameObjectRenderedState : RenderState
{
    4 references | JGDominik, 7 minutes ago | 1 author, 2 changes
    public override void Handle(GameObject pGameObject)
    {
        ((MapElement)pGameObject).SetState(Render_Manager.State.Culled);
    }

    3 references | JGDominik, 7 minutes ago | 1 author, 2 changes
    public override void StateCulled(GameObject pGameObject)
    {
        //Shifts the state to culled
        this.Handle(pGameObject);
    }

    3 references | JGDominik, 7 minutes ago | 1 author, 2 changes
    public override void StateRendered(GameObject pGameObject)
    {
        //Set Rendered code
        pGameObject.pProxy.isCulled = false;

    }
}
```



Design Patterns Needed (Tracker)

Object Pooling – (Managers and Double Linked Lists) **- Done**

Manager for Texture Files **- Done**

Manager for Image Sources**- Done**

Manager for Sprites **– Done**

SpriteBatch and SpriteNodes as well as their managers **– Done**

SpriteBoxes and its manager **– Done**

SpriteProxy and SpriteBoxProxy – Done

Managers for Proxies **– Done**

ListBase for our Double Linked Lists **- Done**

Iterator base and DLink Iterator for our Double Linked Lists **– Done**

GameObject, GameObjectNode and their managers **– Done**

Composite, Component and Leaf **– Done**

Iterators for Composite **– Done**

Ghost and Delayed Object Manager for Removing/Recycling **– Done**

Collision Object/Rect **– Done**

ColVisitor and **ColObserver – Done**

InputManager/InputSubject/InputObserver **– Done**

State for Player and NPC movement**– Done**

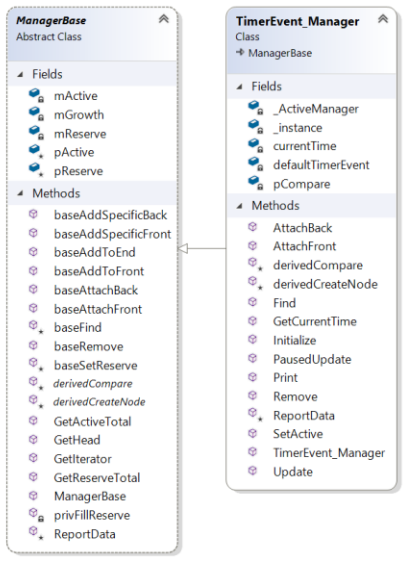
TimerEvent and its Manager **– Done**

Command– Done

Design Patterns Used in this Project

Singleton – (Managers are singleton instances)

UML Diagram



```
public static void Initialize(int _reserve = 3, int _growth = 1)
{
    //LTN THIS IS OWNED BY THIS TimerEvent_Manager CLASS
    if (_instance == null)
    {
        _instance = new TimerEvent_Manager(_reserve, _growth);
        _ActiveManager = _instance;
    }
}
```

```
public static void SetActive(TimerEvent_Manager pTM)
{
    Initialize();
    TimerEvent_Manager._ActiveManager = pTM;
}
```

General

The basic idea of the Singleton pattern is to streamline the use of a class to a single instance, this is used with all the Managers in this project.

The main thing to note is for singletons their constructors are private so if we want to use this pattern the user needs to call the Initialize method, which will create an instance of the class and store it within itself, if the user tries to call the Initialize method again then we check if there is an instance already created, if there is, we ignore this request (or show error if we needed to), however by making sure that all crucial calls to the manager call the initialize function we guarantee that the user will not use the Manager without creating an instance.

The Problem

The problem that we face with this pattern is that what if we had multiple scenes and each scene required an independent version of this singleton? Well in that case we would have to make some adjustments to our singleton, we have to make our constructor not private and then we add another instance of itself. Whenever we need to use this for a particular scene, we just create a separate instance of the class and set it as active while we are in that scene, if we switch scenes, we switch the active instance.

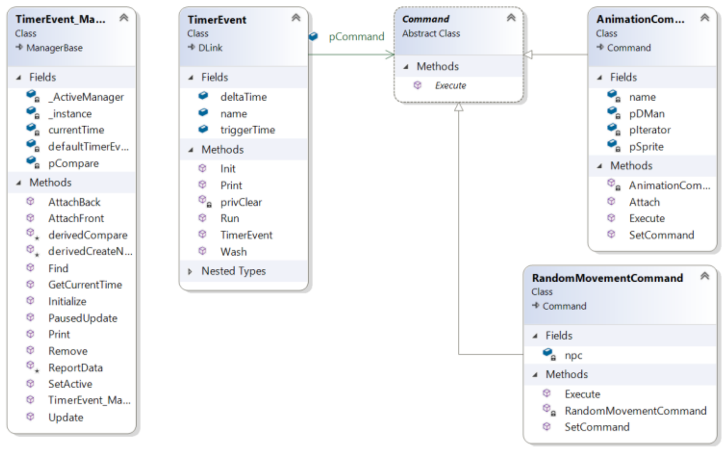
The benefit of this pattern

It is handy to keep track of only one instance of the class rather than having to create a variable for one and using that as a global variable, everything we need from the class is already static, and so all the methods are available to us.

This makes it so that everyone has access to the same instance rather than everyone creating their own.

Command with Timer Event Manager

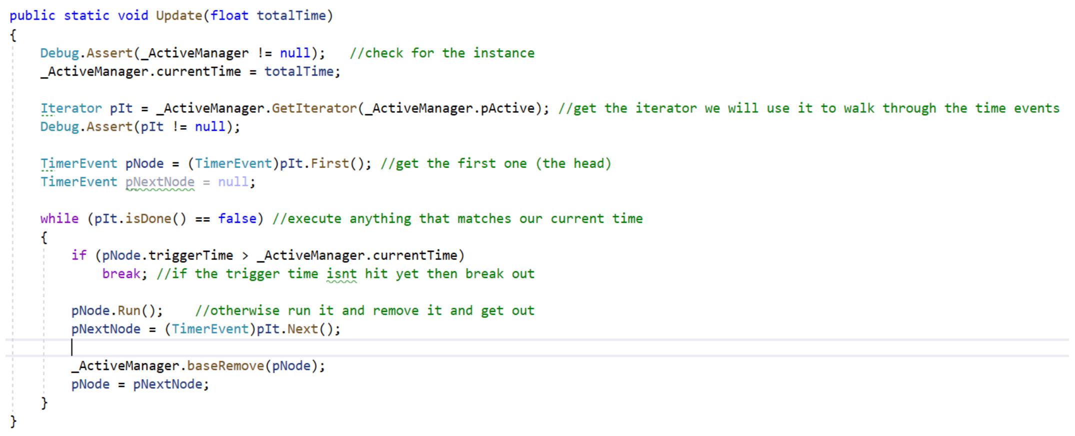
UML DIAGRAM



General

Commands are objects that execute their functions at a given time, we use them in conjunction with our Timer Events.

The design consists of an abstract Command class with an Execute method, then all our different Commands can use it to execute their functions as part of our Timer Events.



```
public static void Update(float totalTime)
{
    Debug.Assert(_ActiveManager != null);   //check for the instance
    _ActiveManager.currentTime = totalTime;

    Iterator pIt = _ActiveManager.GetIterator(_ActiveManager.pActive); //get the iterator we will use it to walk through the time events
    Debug.Assert(pIt != null);

    TimerEvent pNode = (TimerEvent)pIt.First(); //get the first one (the head)
    TimerEvent pNextNode = null;

    while (pIt.isDone() == false) //execute anything that matches our current time
    {
        if (pNode.triggerTime > _ActiveManager.currentTime)
            break; //if the trigger time isnt hit yet then break out

        pNode.Run();    //otherwise run it and remove it and get out
        pNextNode = (TimerEvent)pIt.Next();

        _ActiveManager.baseRemove(pNode);
        pNode = pNextNode;
    }
}
```

The Problem

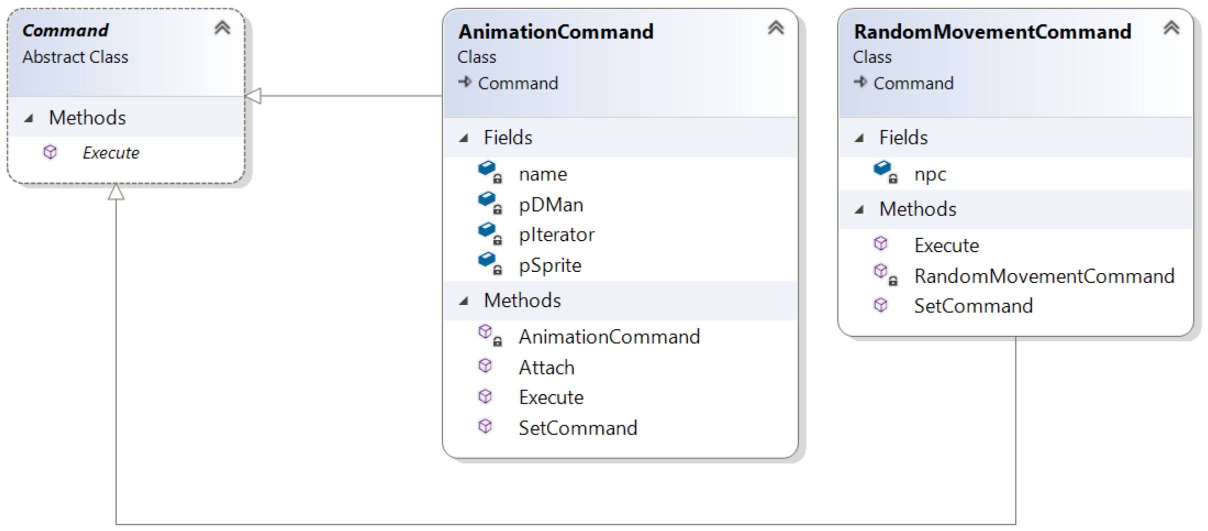
The main problem with these types of behavioral patterns is that if we don't necessarily know what gets executed as all the command types have an Execute method.

The benefits of this pattern

This is the working force behind our game, everything is set on timers and commands, Animation commands used in this project are extremely useful, as they control the animation of our objects while the Timer Event Manager controls the timing of when they animate. Movement for the NPCs are also controlled in the same way, a command is set and at a random interval it will trigger and reinsert itself then recalculate the timing for the movement.

Strategy – (Commands)

UML Diagram



General

The main idea is to define algorithms related by type (in this case our Commands fit that pattern as well) and we just extrapolate all the main functions into abstract classes in the main class, and all it is all taken care of by its subclasses.

This pattern is very similar to the state pattern but in the case of state pattern we just switch the state objects, and the functionality is taken care of, but here we are pretty much subclassing with different takes on the abstract methods depending on the use.

The Problem

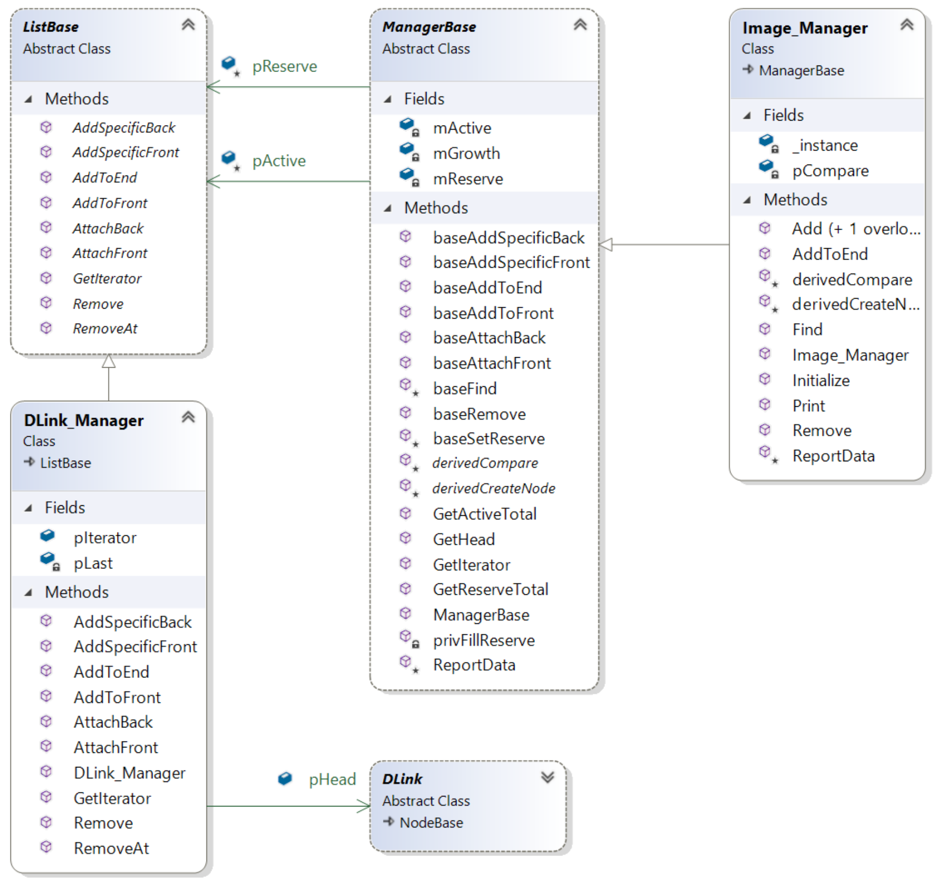
To change the behavior of a class you would need to create a different object.

The benefits of this pattern

Our commands implement this pattern to some degree, they are all children of a base class and the way its abstract methods are created means that all the subclasses can use these methods with their own twist.

Object Pooling – (Managers and Double Linked Lists)

UML Diagram



General

This pattern consists of creating two separate pools, one active and one reserve, these pools work together tightly, whenever we create any objects we just take from the reserve and add it to active then we can initialize that element as we see fit, whenever we are done with the elements we can remove them by adding them back to the reserve list and washing them of any data, then they get recycled again when new objects are needed.

It will also grow in size depending on how many elements are needed, if there are not enough elements in the reserve list then it will simply add more and use those.

The Problem

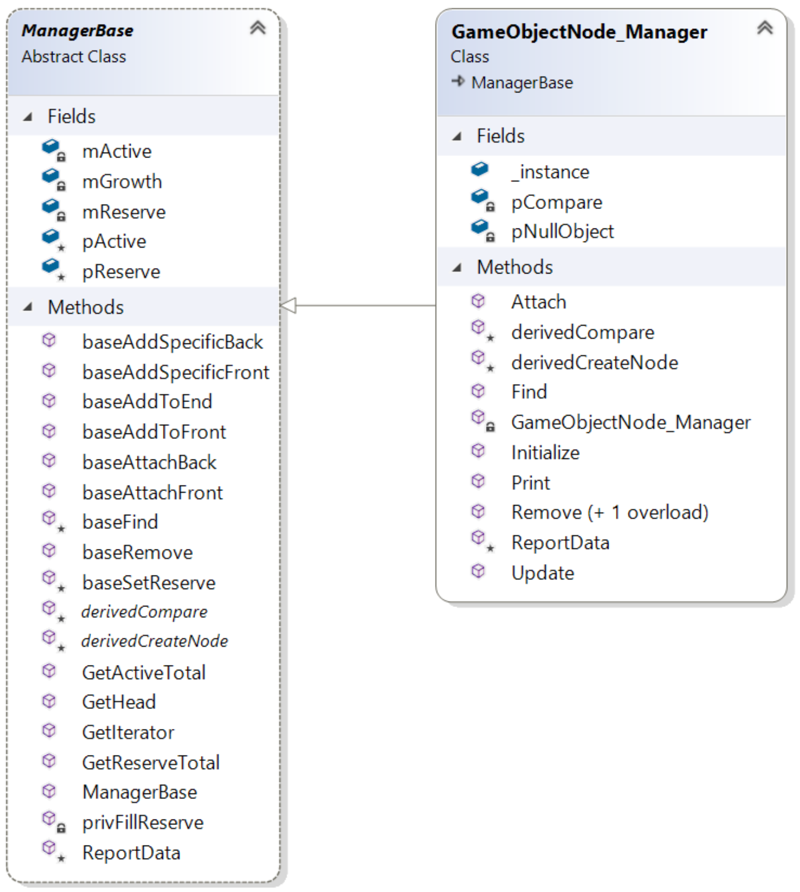
This pattern brings one problem, and it is that no object ever gets destroyed so no memory is freed, it's all kept in reserve pools.

The Benefit in this case

The textures, images, sprites, and the game objects type themselves are always kept in the manager whenever the user needs them, no need to re add them.

Template Pattern – (For Managers and their derived implementations)

UML Diagram



General

The Template design pattern allows for a parent class to have some methods implemented and the child classes could use these derived methods and implement their own version based on their needs.

The Problem

The need for this pattern arises at the fact that if we had multiple classes with the same functions except one or two steps then we could just have a base class have the basic skeleton of the process and the child classes can use them in their own way.

The benefits of this pattern

Once we turn all our methods into steps then we can extrapolate each and use what we need then the rest could be on the child classes.