SE 456 Final Project

Space Invaders

Design Document

Jilei Hao

Mar 24, 2019

Part I – Overview

Space Invaders

This project is to write the classic game in C# using design patterns. The game started with a group of aliens advancing towards player's base. Player can move a ship to shoot missiles to destroy aliens. Aliens can drop bombs as well, and the bombs can destroy player's ship and the shields that protect the player. Successful shooting aliens and UFOs earns players scores. And highest score will be recorded. There can be at maximum 2 players. Each player has 3 ships to use. The game is over when all players have their 3 ships destroyed.

Design

Along with the time increment, code will be executed iteratively. The looper in Azul.Game will loop through 3 major components of the game:

- Load(): Load and initialize objects and structures
- Update(): Being executed in high frequency to modify and refresh objects like graphics and sounds
- Draw(): Being executed in high frequency to render graphic object to the screen

The game content is divided into another 3 major components:

- Start Screen: Displaying welcome information, points table, and options
- Game Screen: Can be further divided into player1 and player2 screens. It is the major game content that allowing players to shoot aliens. Between each play, there are short displays of transition screens, showing which player is playing and what level they are playing
- Game over Screen: Showing the player's life has run out, and the current game cycle is over. Will return to Start Screen after short display.

This project is emphasis on using design patterns to reduce the repeating and redundant code of the game, to make it clean, efficient and scalable. The following sections will illustrate the patterns in detail.

Part II – Components Design

Object Pooling

Problem

Space Invaders game need creating and deleting many objects on the fly. For example, when an alien is hit by a missile, the alien needs to be deleted. When a player finishes a game, all the game objects related to this game need to be deleted, to clean the screen for next round of game. So, there will be large amount of creation and deletion during the running of the game, which can be infinitely long until the program gets terminated. A design is needed to efficiently manage these dynamically created objects, to make the system runnable in a long time with managed memory usage.

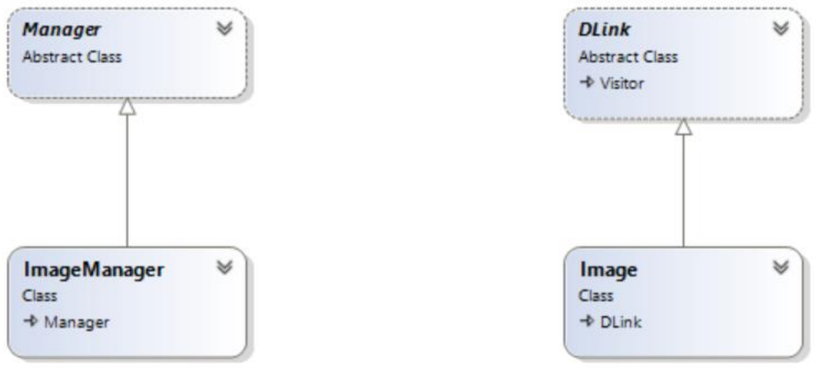
Object Polling Pattern

The purpose of the object pooling pattern is to instantiate objects in advance and recycle and reuse objects instead of deleting and create new objects. Creating and deleting objects is expensive both in cycles and memory space. It is especially true in real-time systems, as this type of systems are constantly running without stop or restart.

Object Pooling creates most of objects in advance and can be incremented in real-time when needed. When an object is needed, it pulls one object from the reserving pool, configures it and adds it to the active pool. When an object is no longer in use, it can wash and recycle it into the reserving pool for future use. Different derivative of object pooling managers can manage different types of objects. So, it is also used as a collection manager of a certain type of objects.

Example 1: ImageManager and Image

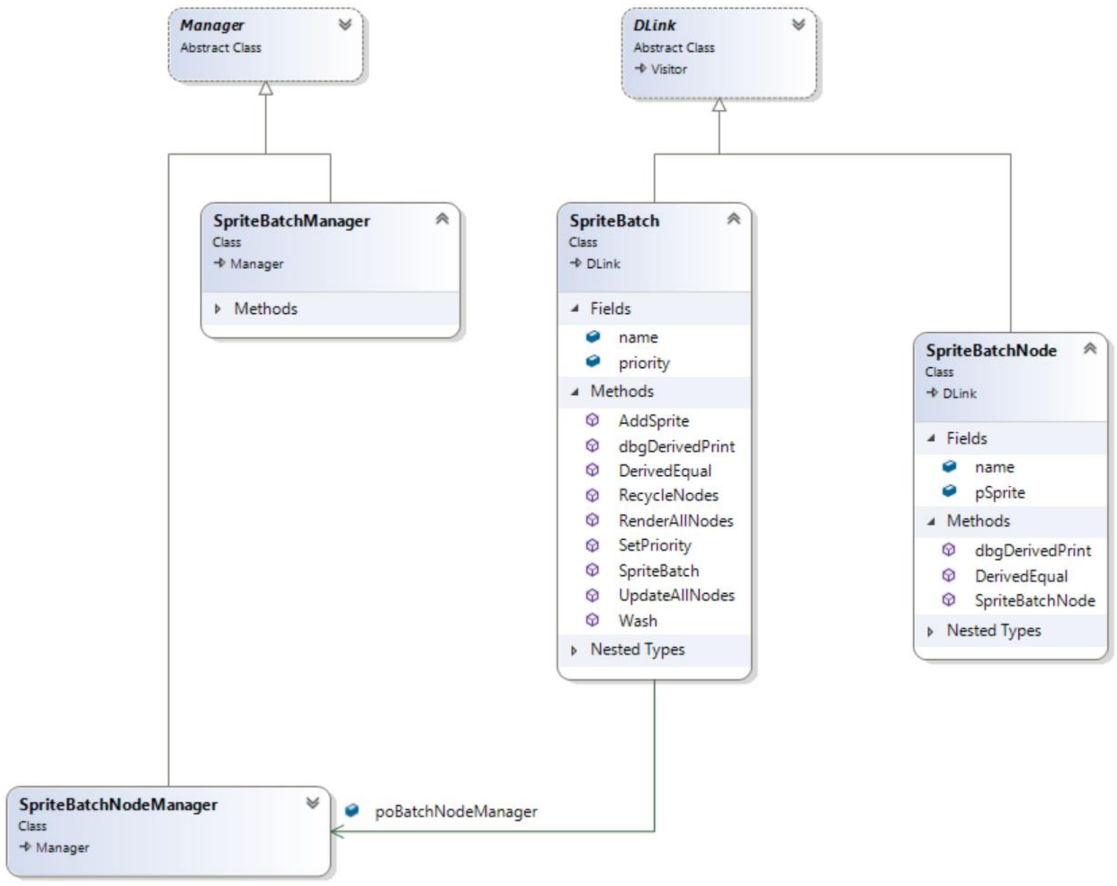
Image Manager can manage image nodes. Combined with singleton pattern it can also be used as a global collection of image objects to share images through out the entire program.



- Manager: Abstract Class. To provide an abstraction and default method implementations of an object pooling manager
- DLink: Abstract Class. To provide an abstraction and default method implementation of a data node object that can be managed by the object pooling manager
- ImageManager: Concrete Manager Class. Extended from Manager class. Can manage nodes with Image type, which is extended from DLink class
- Image: Concrete Image Class. Extended from DLink class. Holding the information about an image that can be used to create sprites in the game. Can be managed by ImageManager

Example 2: SpriteBatchManager and SpriteBatch

SpriteBatchManger manages SpriteBatch, a node also includes a manager in itself to manage the SpriteBatchNode objects. The SpriteBatchNode class has reference to a SpriteBase type, which can be either GameSprite or BoxSprite.



- SpriteBatchManager: Manages SpriteBatch

- SpriteBatch: A DLink extension that includes a Manager in itself.
- SpriteBatchNodeManger: owned by an instance of SpriteBatch, manages SpriteBatchNodes.
- SprietBatchNode: Has a reference to an object extends SpriteBase class

Singleton

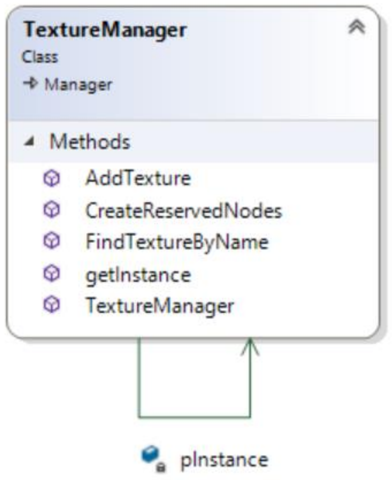
Problem

Although managers are needed in a lot of places in the game, most of them only needs one instance throughout the entire game. Using regular instantiation of class, the one instance of the manager will need to be passed around between different components of the game. Considering the game scale and complexity, this type of action is creating large chunk of repeated code and is error-prone.

Singleton Pattern

Using static member and getInstance() method, the singleton pattern creates only one instance no matter how many times or occasions the object is needed. There's also no need to pass references around since getInstance will always return the exact only instance. It is the perfect pattern to use for most of the object-pooling managers, since most of them need only one instance, such as ImageManager and TextureManager. It is also useful in switching game states (scenes), because only one instance of state is needed for each type of the four game screens.

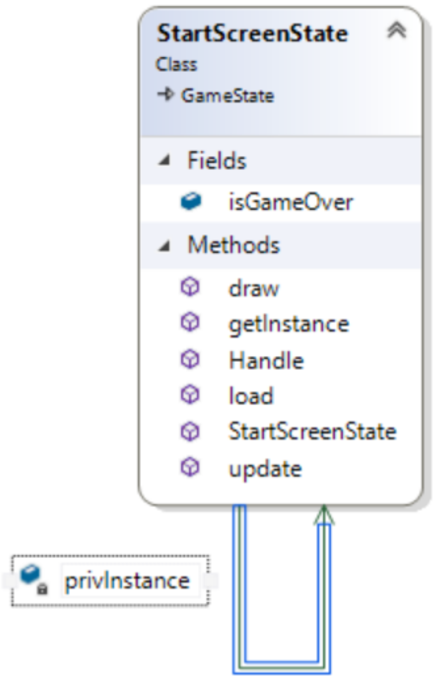
Example 1: Texture Manager



- TextureManager: Create only one instance and manage all textures in it. When a texture is needed, the program just call getInstance() to access the stored resources

Example 2: StartScreenState

As the beginning state of the game, it shows game title and instructions. A regular state pattern will need constantly creating states but that is not needed for the game state here. Singleton is used to make sure only one instance is created for a certain game state.



- StartScreenState: Manages elements to display in the start screen. Using singleton to maintain only one state at all times

<u>NULL Objects</u>

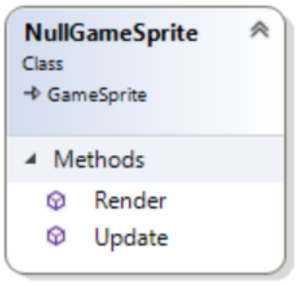
The Problem

While creating objects like boxes or composite game objects, a placeholder object is needed to make sure upper logic will call it as all other same type objects but do nothing.

Null Object Pattern

A placeholder object of certain type that inherits all methods from the abstraction but implement many of the important methods as doing nothing.

Example: NullGameSprites



- NullGameSprite: Created for GameObjects that don't actually drawing sprites on the screen. Those objects usually just managing other objects, such as alien grid and columns. They should be iterated indistinguishably but should not perform any actions.

<u>Command</u>

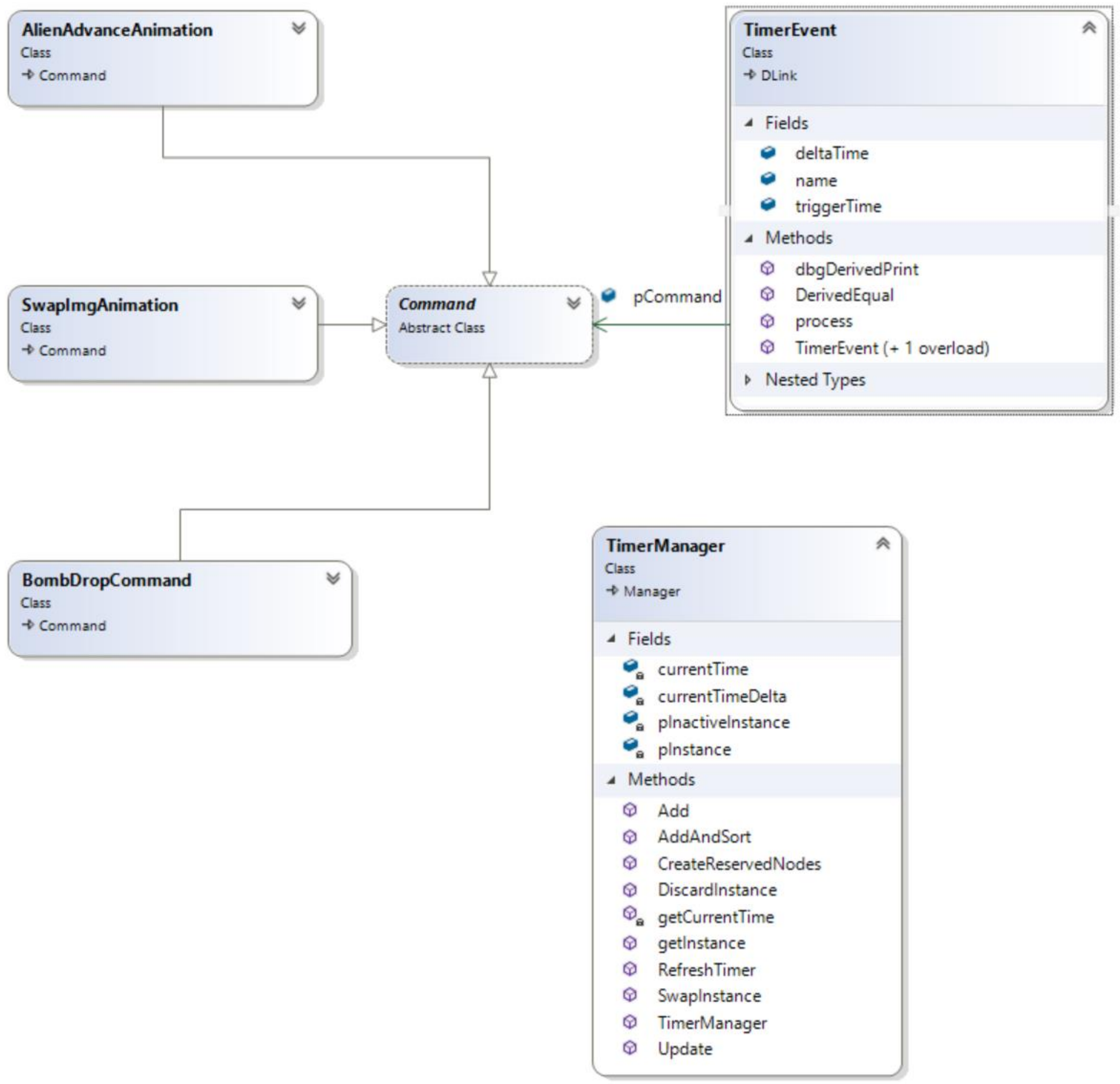
Problem

Sometime the game needs to execute something on the run. An abstraction of executable object that can be execute and discarded at will is needed. For example, to perform swap-image animation constantly, the regular approach will be creating and deleting numerous of objects carrying the method since blocking everything else and iterating does not fit the game.

Command Pattern

Execute once and forget or recycle for future use. A command is small, portable and can be efficiently managed by a manager. In the game it is used to create animation events to be executed frequently by the timer manager.

Example: SwapImageAnimation, AlienAdvanceAnimation, BombDropCommand



- AlienAdvanceAnimation: Move in specified time-interval. Commands are created and managed by timer manager through TimeEvents to be executed at given time.
- SwapImgAnimation: Swap Image constantly. The Command will be executed, removed and re-added to the manager to be executed repeatedly
- BombDropCommand: Can drop bombs from the bottom of columns. Created and recycled by timermanager to drop bomb at given time interval

State

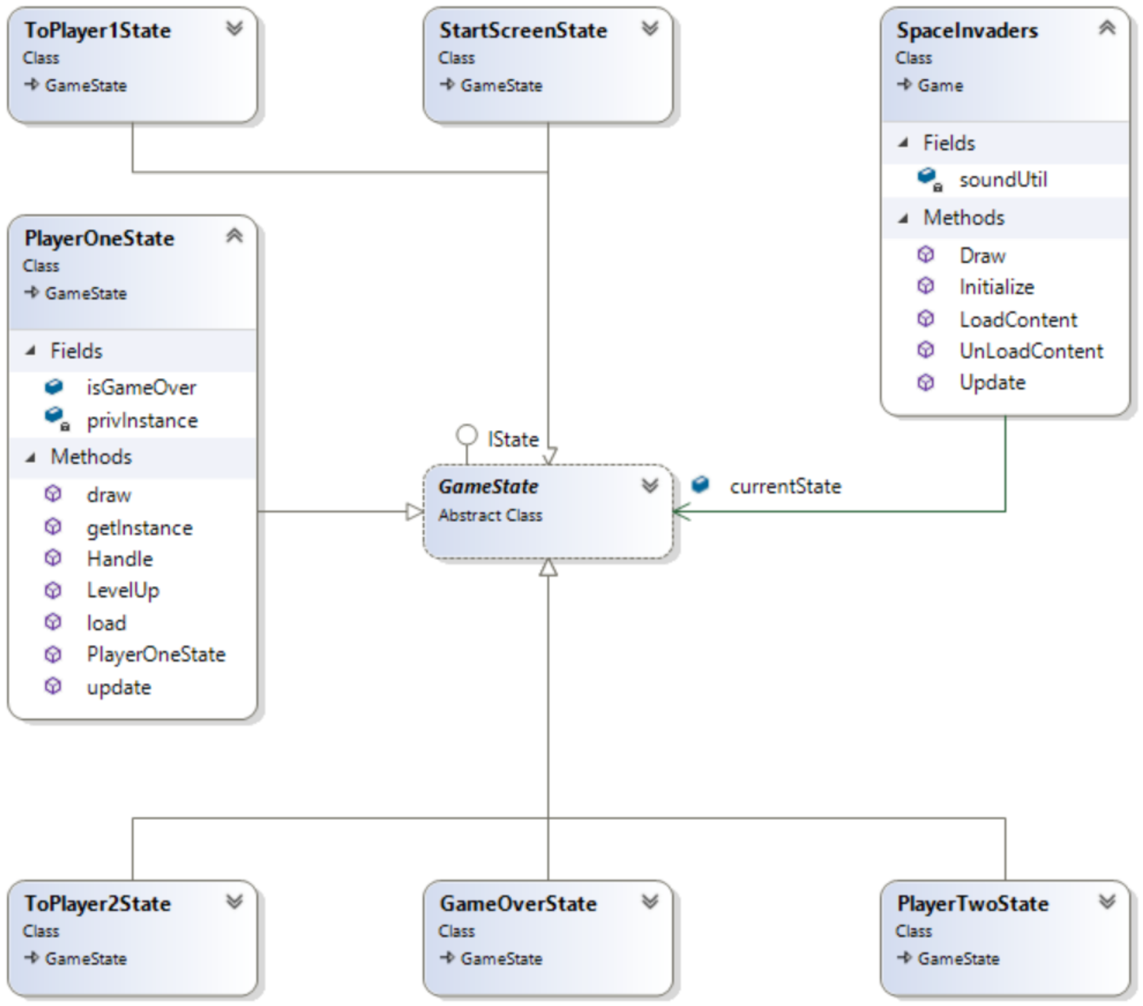
Problem

The SpaceInvaders game has three major screens: the start screen, the player screen and the game over screen. The program needs a clean way to manage all three screens without creating big chunk of switch or if-else statements.

State Pattern

State pattern provides encapsulation of logic between different states and provides a unified interface in switching between them. It reduces conditionals in the code significantly to ensure a cleaner code

Example: GameState



- StartScreenState: Game Title, Points table and player selection
- PlayerStates: PlayeOneState and PlayerTwoState contains major game elements for player 1 and player 2 respectively

- ToPlayerStates: Contains elements for transition screens. One for each player. Showing level and current controlling player
- GameOverState: Shows game over to players. Will display shortly before switching to StartScreen
- SpaceInvader: the context. Contains the reference to the current active state. State handle will be executed to switch the reference between state.

<u>Adaptor</u>

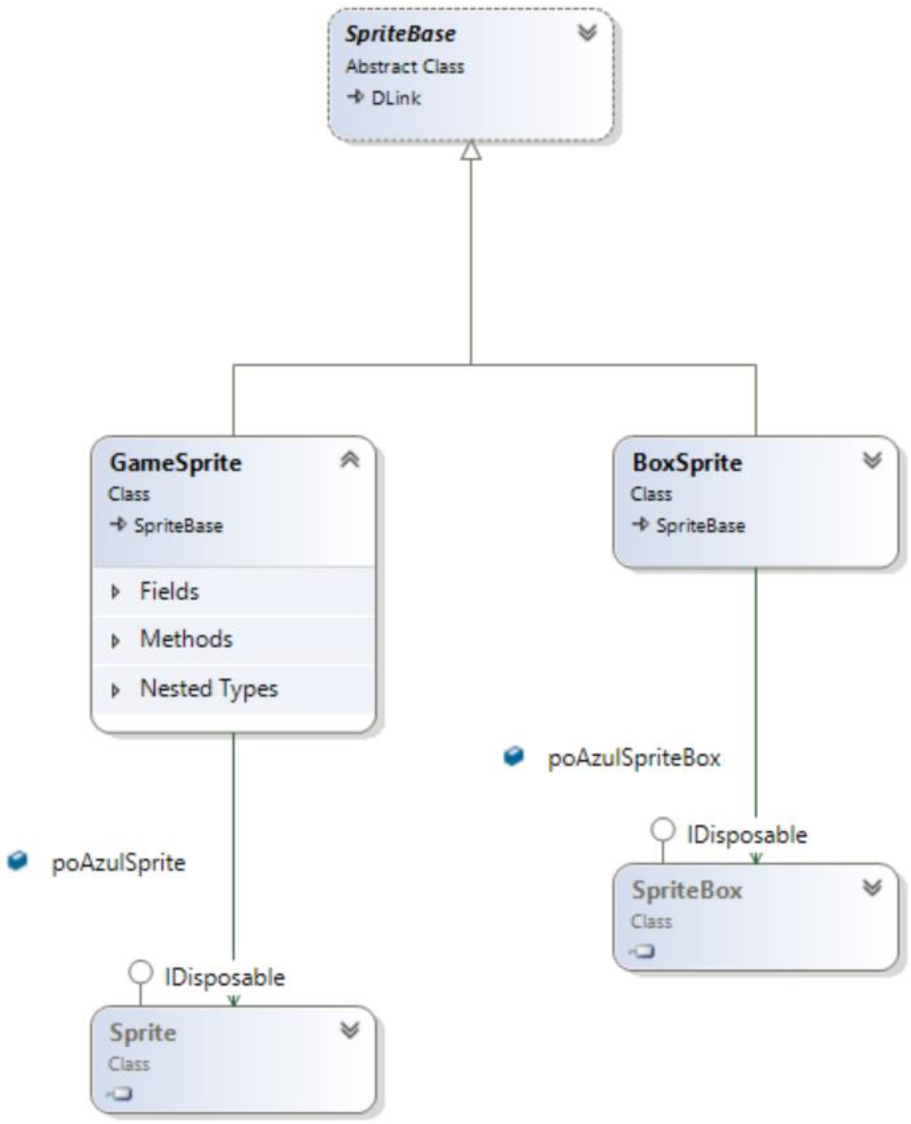
Problem

The game is using Azul library as its graphic library. The library is sufficient for the game, but the game needs a unified and simpler interface to use it efficiently.

The Adapter Pattern

Use an adaptor class to wrap the original program interface of the graphic component to provide a simpler and unified interface for other parts of the game.

Example 1: GameSprite



- SpriteBase: Abstraction of all wrapper classes of sprites
- GameSprite: Contains an instance of Azul.Sprite, drawing game elements to the screen
- BoxSprites: Contains an instance of Azul.SpriteBox, drawing rectangle to the screen, primarily for displaying object outline for debugging purpose

Factory

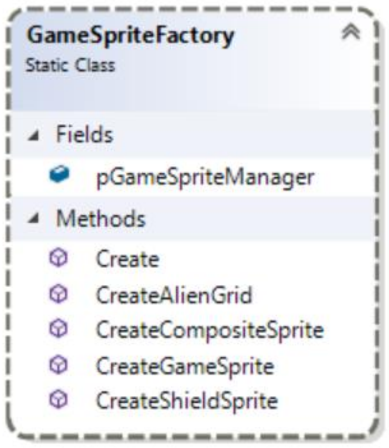
Problem

There are 55 aliens in each player's screen plus all their outline rectangles. There are also more than 100 shields elements created for every game play. Creating this large number of objects using their original constructor is not efficient. A unified and simplified interface is needed to create such objects.

The Factory Pattern

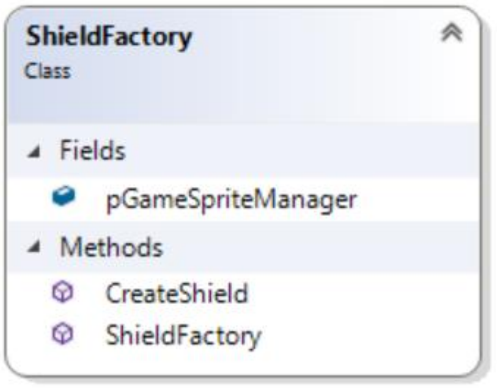
Encapsulates object creation logics and provide a simple and unified interface of creation. The factories can be implemented as singleton to provide flexibility to use. It can also encapsulate creation of large number of similar objects to provide cleaner code for other part of the game.

Example 1: GameSpriteFactory



- GameSpriteFactory:
  - Create alien grid in batch
  - Create GameSprites and corresponding GameObjects, add animations, and add to given SpriteBatch all in one
  - Create different types of Sprite using different methods

Example 2: ShieldFactory



- ShieldFactory: Create shield bricks and assemble them into columns and grids.

<u>Visitor</u>

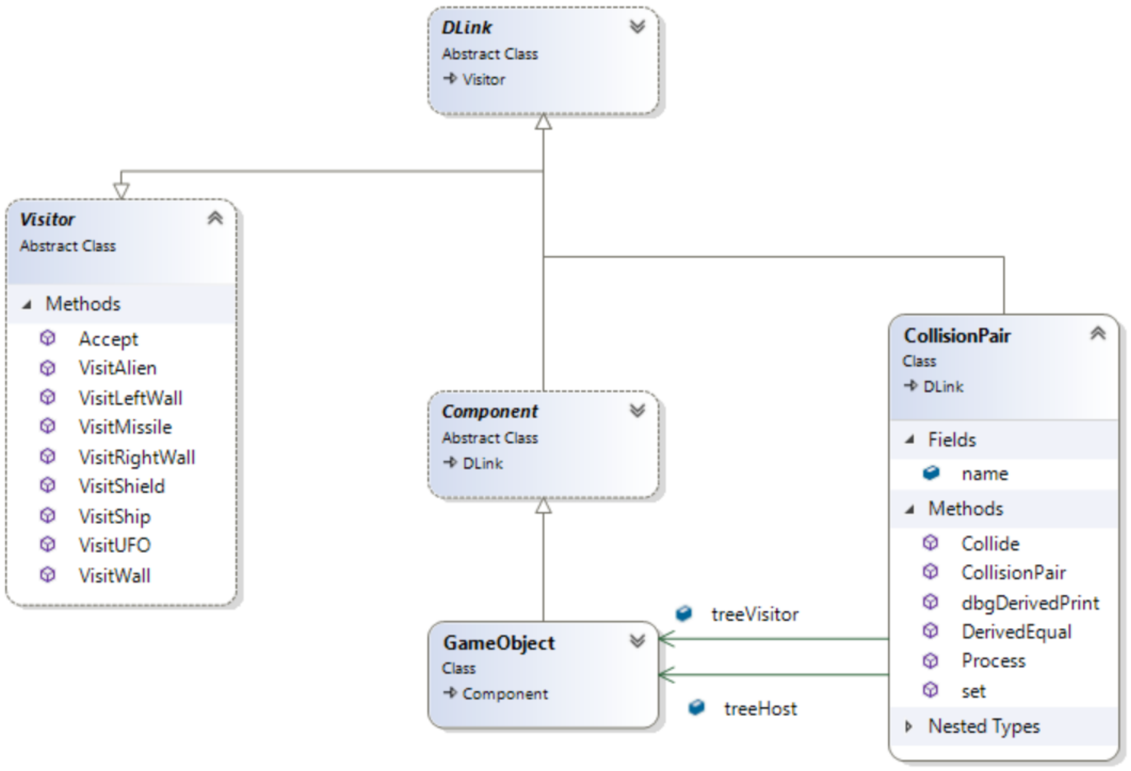
Problem

One of the major actions in the game is the collisions, as the primary goal of player in this game is to shoot out the aliens, which is colliding them with missile. Aliens also need collision with walls to stay inside of the screen. Missile need collisions with walls to be reloaded for next shoot. All collisions are involving with two objects and hence actions are needed for both objects. A mechanism is needed to efficiently dispatch the task into two objects without creating lengthy conditionals.

The Visitor Pattern

A class designed using visitor pattern can either be a visitor or accept a visitor. When a host accepts a visitor, it will call the visitor's visiting method in addition to its own accepting logic. This provides a elegant and efficient way to split the task into the two classes.

Example: GameObject



- GameObject: as the concrete object of visitor is the only visitor object in the game. But it almost covering all types of game elements.
- CollisionPair: Has references of both the host and the visitor and will call accept method once a collision happens

<u>Observer</u>

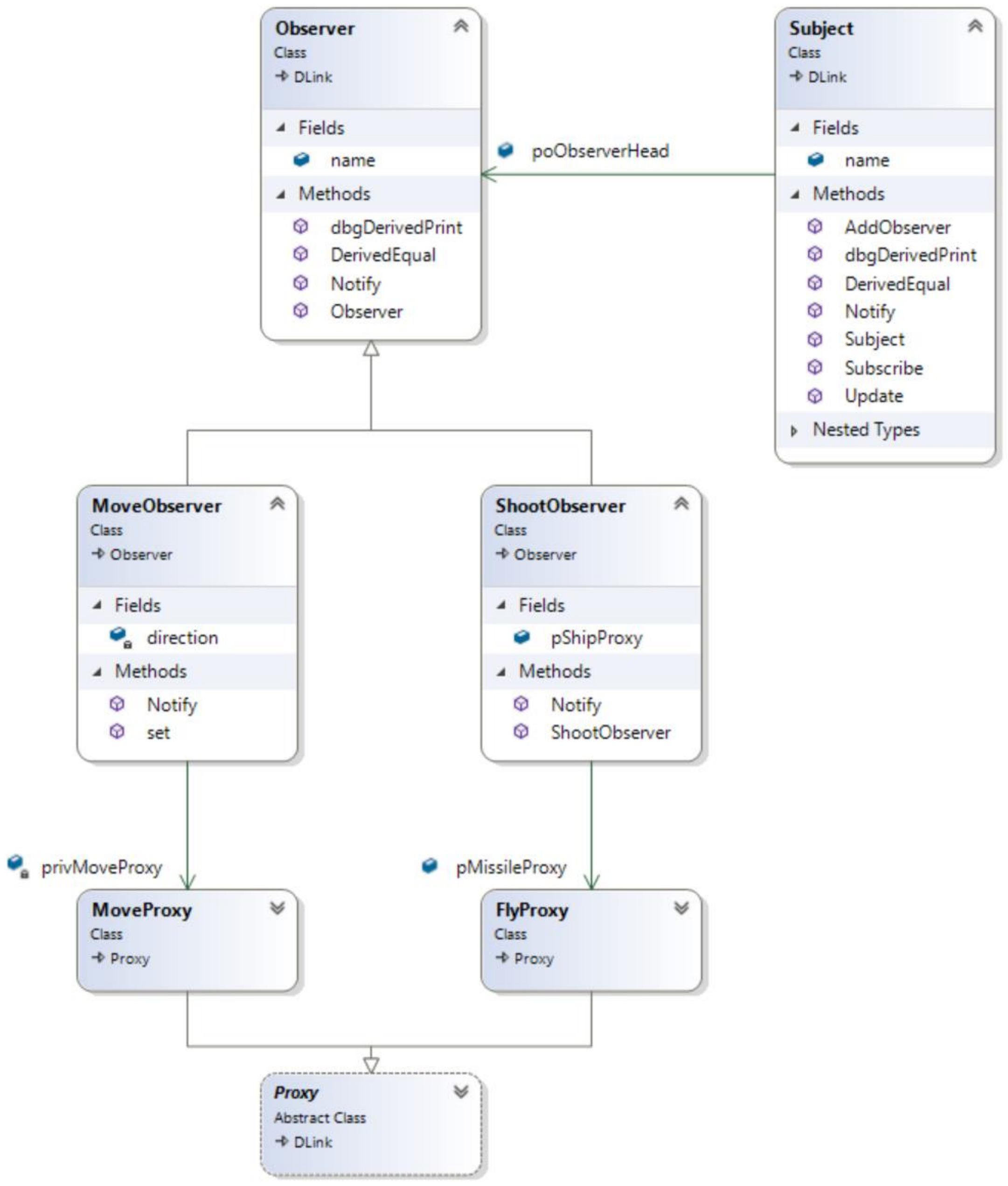
Problem

The Game is constantly running and as a single thread program it cannot block and listen to user input and then take actions. An event handling logic is needed to capture user input to control the game elements.

The Observer Pattern

Subject surrounded by observers as each observer is waiting for notifications about the events happened. Subjects subscribe observers and notifying subscribed observer about the events happened by using the notify() method, and it will subsequently call the notify() in all of its subscribers.

Example: Move Observer and Shoot Observer



- Subject: Processing Key press event and sending notifications to interested observers to complete some actions
- MoveObserver: When subject process arrow key events and calls its notify() method. The move observer gets the method and perform actions respectively
- ShootObserver: When subject process space key event and calls its notify() method. The shoot observer will shoot the missile out by using the flyProxy referenced.

Proxy

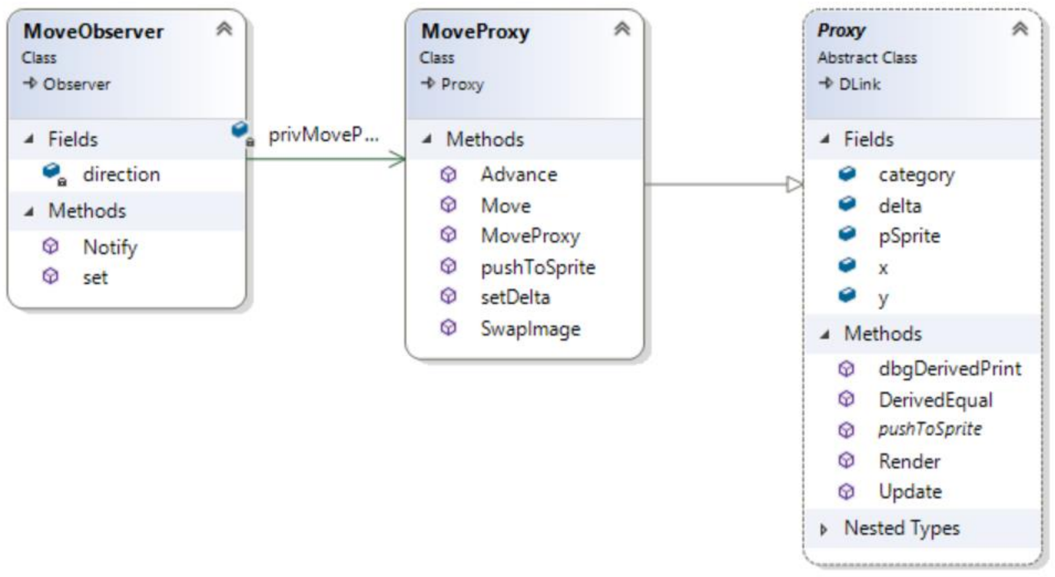
Problem

The actions of objects are complicated. Some objects need certain type of actions while some need others. And objects' interfaces for actions can also be complicated, and it is easy to mix up the methods to complete some actions.

The Proxy Pattern

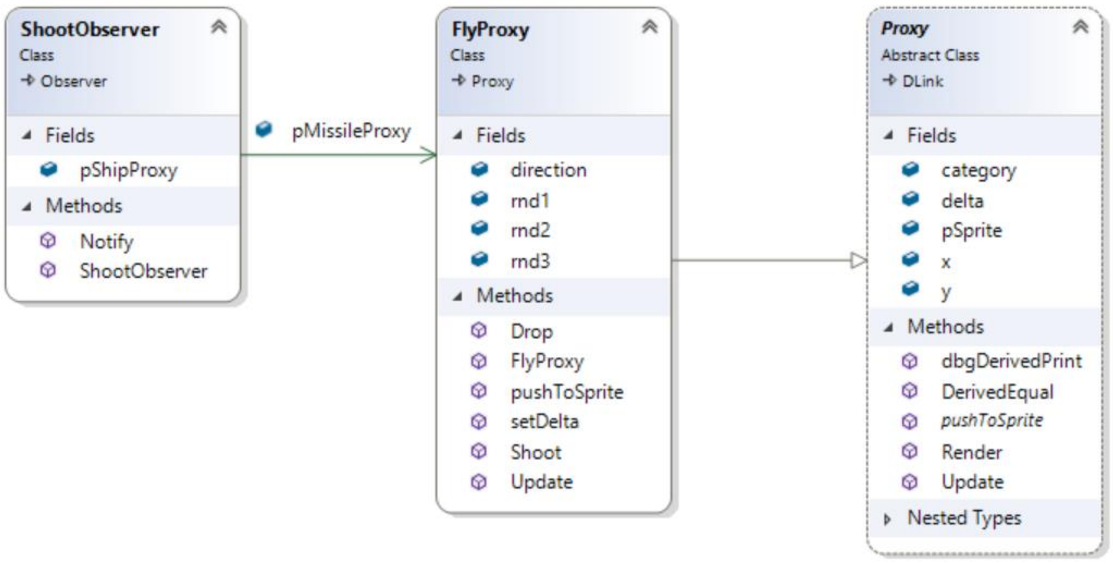
Proxies provides simplified interface for completing actions. It only provides needed interface for completing specific type of actions such as move in one direction, move horizontally, move vertically, move up, move down etc. Code can reference proxy as the representation of the full object behind it, without knowing unnecessary details.

Example 1: Move Proxy



- Proxy: Abstract class. Has reference to GameSprite.
- MoveProxy: Controls the 2D movement (x, y) of the GameSprite.

Example 2: FlyProxy



- FlyProxy: Controls the movement of the projectiles, such as missile and different bombs.

<u>**Composite**</u>

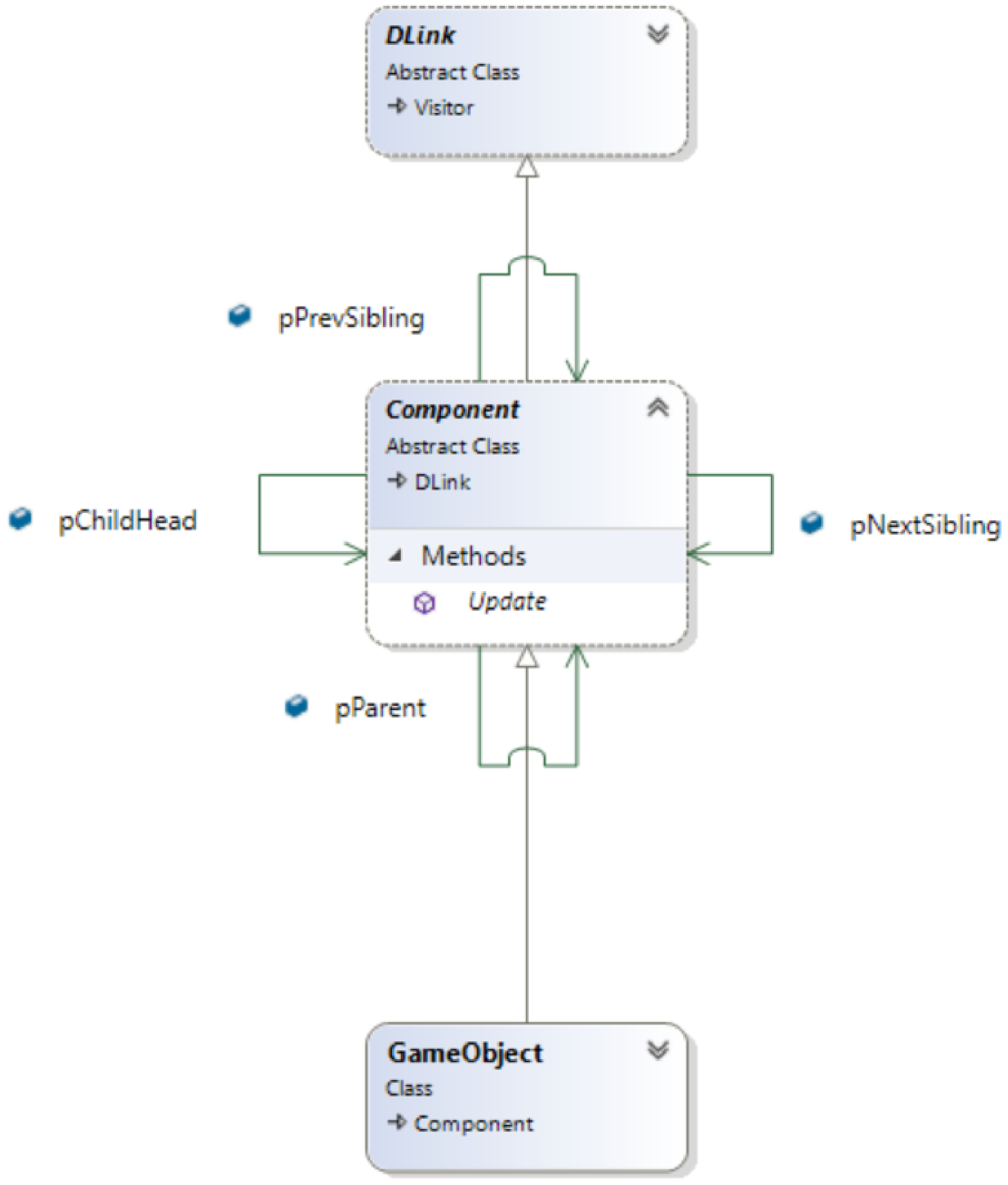
Problem

Aliens created need to be moved collectively instead of individually. Without a effective grouping logic, it can be a challenging task to keep the aliens in order considering they are constantly changing, moving and disappearing. When identifying collisions, an efficient method also needed instead of iterating every element in a collection.

The Composite Pattern

An elegant way to group objects into collections. Using recursive definition, an object itself can be the collection of the same type of objects.

Example: Component



- Component: Contains links to child, siblings and parent. To group same type of elements together.

<u>Flyweight</u>

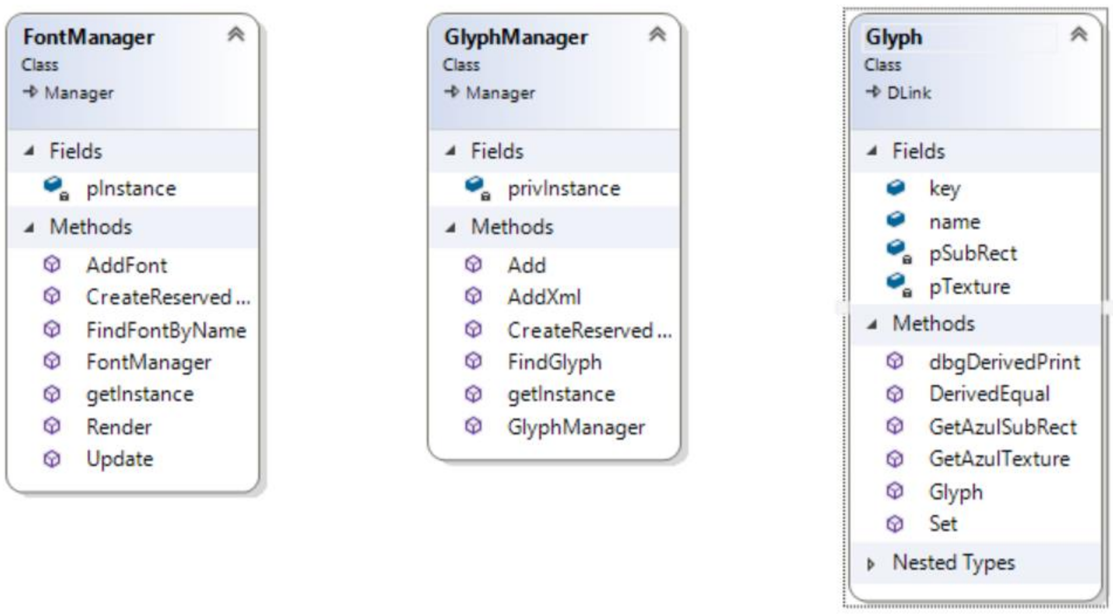
Problem

Many objects in the game are repeatedly used but remain immutable during their lifetime, especially for textures, images and glyphs. Creating one instance for every use is not efficient. A design of object is needed to efficiently share one instance of such object to multiple references.

The Flyweight Pattern

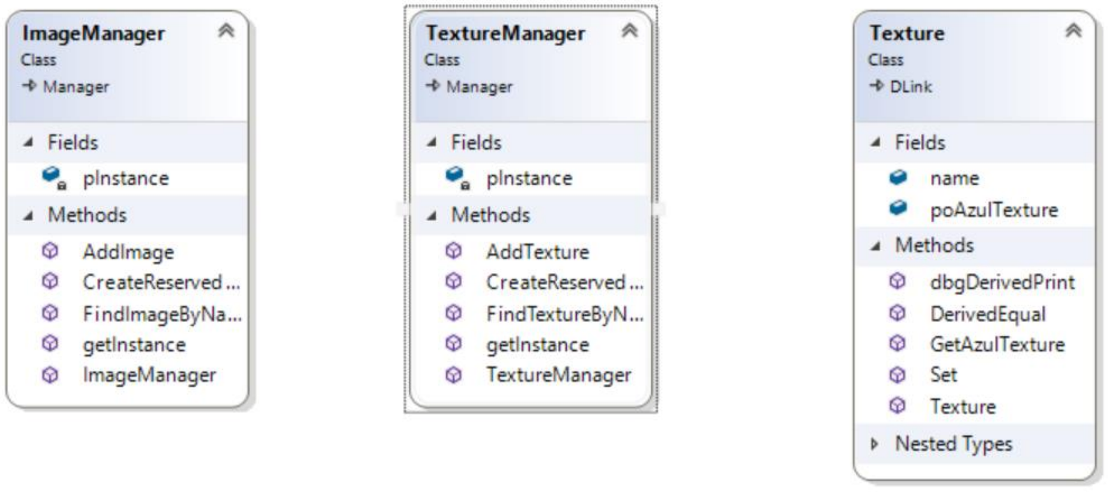
Create one instance for each immutable object and provide references whenever it is used. It minimizes the memory usage and simplifies the use of shared objects.

Example 1: Font Manager and Glyph



- GlyphManager: Create Glyphs using external files and store them into the object pool. Whenever a glyph is needed, the FindGlyph method will be called to get a reference of the specific glyph
- Glyph: Created by GlyphManager. Maintains only one instance for each distinct glyph. Only reference is provided when requested
- FontManager: An example using flyweight pattern to build strings to display on the screen by referencing every character from the Glyph Pool.

Example 2: Textures



- Texture Manager: Create and manages textures. Provide method to get reference of certain texture.
- Texture: Created by Texture Manager. Maintains only one instance for each distinct texture. Only reference is provided when requested.
- ImageManager: An example using the flyweight pattern to create images based on certain textures. Images created will keep the reference to the textures

PART III – Comments

<u>Design patterns make huge difference</u>

Those differences include using factories creating groups of objects, creating observers to handle events, dispatch actions into different objects to complete collisions, and so on. It is immense pleasure to remove big chunk of code and replace it with elegant and manageable code. It also brings clarity at later stage of the development when structures of the game got more and more complicated. I can easily find where to check to fix the ZigZagBomb hits ship bug, because of the visitor pattern. I can also add extra functionality to more than hundreds of objects by just adding one line of code in the factory method. And it is just for SpaceInvader, a game that can be developed within weeks by a single developer. Not even thinking about how important patterns can be in a system with much larger scale and are maintained by hundreds of people.

<u>Real-time architecture</u>

It took some time for me to fully understand the mechanism of the game. The most amazing part of the game is that it uses single thread simulated multi-threaded effects. All aliens, missiles, bombs, ship, collisions, sounds, are acting like they are acting in parallel but actually not. It creates very difficult situations for adding event-handling like behaviors without using multithreading or blocking. Besides patterns, this type of architecture is certainly another big take away from the class.

Works to be done

There are still plenty to be done because of limited time. And the code certainly could be written in a much better way:

- I want was thinking about using strategy pattern in the visitor pattern to encapsulate different visit methods into strategy instead of multiple methods in the GameObject. It could make the GameObject code much cleaner
- Using flyweight pattern for GameSprites. Since Aliens are all same, we probably don't need to create aliens every time when we add an alien into the grid. It will be nice to find a way to use the flyweight pattern to reference same type of alien sprites to only one instance
- Many objects are not following the single responsibility principle, such as the MoveProxy and FlyProxy. It brings convenience in short term development but not good for maintainability. GameObject is also too fat in my code. There should be a way to unload some of its responsibilities to other objects