SPACE INVADERS

DESIGN DOCUMENT

DEPAUL UNIVERSITY

ARCHITECTURE OF REALTIME SYSTEM

HO TIN NGHIA LE

03/20/2021

LE – SPACE INVADERS DESIGN DOCUMENT

PART 1

INTRODUCTION

1.1 Scope

The Space Invaders Design Document (SIDD) describes the detailed structure of the game Space Invaders designed by Le including numbers of design patterns at medium level which satisfies the basic requirements of the classic game. This document will go through mostly every details of the structure as well as the desired behaviors of the software based of the given requirements. It is recommended that the reader should have basic knowledge of an object oriented system before revising or commenting the document.

1.2 Aim

The document present multiple purposes including as below:

- Software Introduction
- Description of functional structure, data and design patterns implemented
- System requirements
- Requirements Verification
- Maintenance Requirements

1.3 Intended Audience

The SIDD is written for software engineer professionals and designers as well as people with related knowledge, who are:

- Supervisor: My professor Ed Keenan is the one who instructed and provided me with essential knowledge, and guided me through the software build-up. His contact email is upon request.
- Auditors: The later version of the game is intended for general market so that publishers may have an auditor to examine the distinct qualifications of the program.
- Reviewers: People who is interested in commenting or advising may contact me at the following email: louis9286@icloud.com.
- Interviewers: The SIDD serves as a proof of production project and will be presented to interviewers in future job interviews.

PART 2

MODULE DESIGN

This part will detail description of objects defined in the SPDD, including implemented design patterns and class diagrams used to achieve the coding process.

2.1 Design Patterns

2.1.1 What is Design Patterns?

Design Patterns are solutions for common software design problems, which have been reused and optimized overtime. They perform as a set of thought-out solutions that have resolved common issues in software architecture designs.

2.1.2 The essentials of Design Patterns

Software engineers utilize Design Patterns as a common language when they discuss about Software Architecture. Even though it is not specifically a language for communication, Design Patterns can be implemented in most programming languages, providing with sufficient solutions in object oriented.

A software using a right pattern is easier to use and understand. It usually performs with high quality while remains low cost in maintaining. In reality, people usually implements many but one design pattern in a software so that they can maximize the efficiency. There are three common types of patterns that are used in common: Creational Patterns, Structural Patterns and Behavioral Patterns.

2.2 Implemented Design Patterns

2.2.1 Singleton and Object Pool

Every time users add a node from the reserved list to the front or the end of active list, the manager has to check if the reserved list is empty (refill it with Delta Grow if it is empty), then remove the front node in the reserved list and set new value for it (Name and Data in parameter) before adding it to the front or the end of the active list. The program reverses all the steps when removing a node from active list and adding it back to reserved list as the node is no longer used.

In order to remove a node from the active list, we need to find that node using its name. The function Find return the node that users want to find, which can be uses as a parameter for the function that removes that node from active list and add it back to reserved list.

In DLink, Next and Prev keep track and move the pointer around in order to remove or add a node. The adding and removing node described above are main functions of the Object Pool pattern.

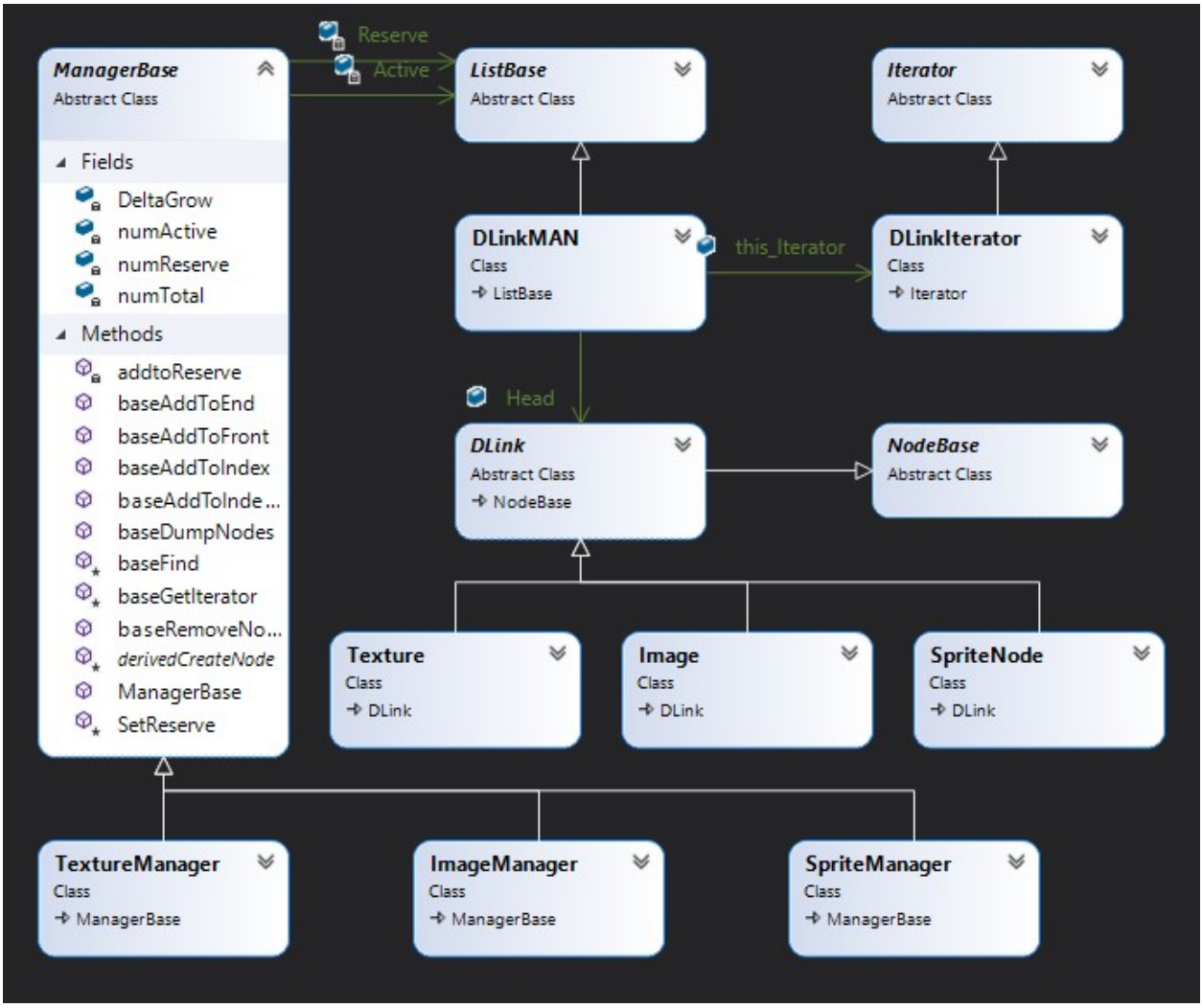

Diagram 1.1 Manager Base

Every time users add a node from the reserved list to the front or the end of active list, the manager has to check if the reserved list is empty (refill it with Delta Grow if it is empty), then remove the front node in the reserved list and set new value for it (Name and Data in parameter)

before adding it to the front or the end of the active list. The program reverses all the steps when removing a node from active list and adding it back to reserved list as the node is no longer used.

In order to remove a node from the active list, we need to find that node using its name. The function Find return the node that users want to find, which can be uses as a parameter for the function that removes that node from active list and add it back to reserved list.

In DLink, Next and Prev keep track and move the pointer around in order to remove or add a node. The adding and removing node described above are main functions of the Object Pool pattern.

By generalize the type of node from DLink, the software creates different types of nodes such as Texture, Image and Sprite.

- Texture, Image and Sprite work exactly as a regular node do. They also have all the functions in original Node Base class.
- These nodes are controlled by their own managers which use the abstract class Manager Base.
- Image Manager to cut those objects out of the texture using x, y, width and height.
- Sprite is used to render the objects onto the screen with a fixed dimension.
- In the Update() function, the locations of those objects are updated constantly to make them move.

2.2.2 Iterator Pattern

In order to avoid using loops among the complex architectural structure of the program, an Iterator is created inside the DLink manager to serve as a moving pointer for tracking from the first node of a list and reset when needed.

Each manager typed DLink will have their own iterators implemented from the Iterator base class.

2.2.3 Adapter Pattern

The adapter pattern wraps an existing interface of an outside object and use it as another interface. The system does not control this outside object. The whole Texture class is an adapter for Azul.Texture (See Diagram 1.2)

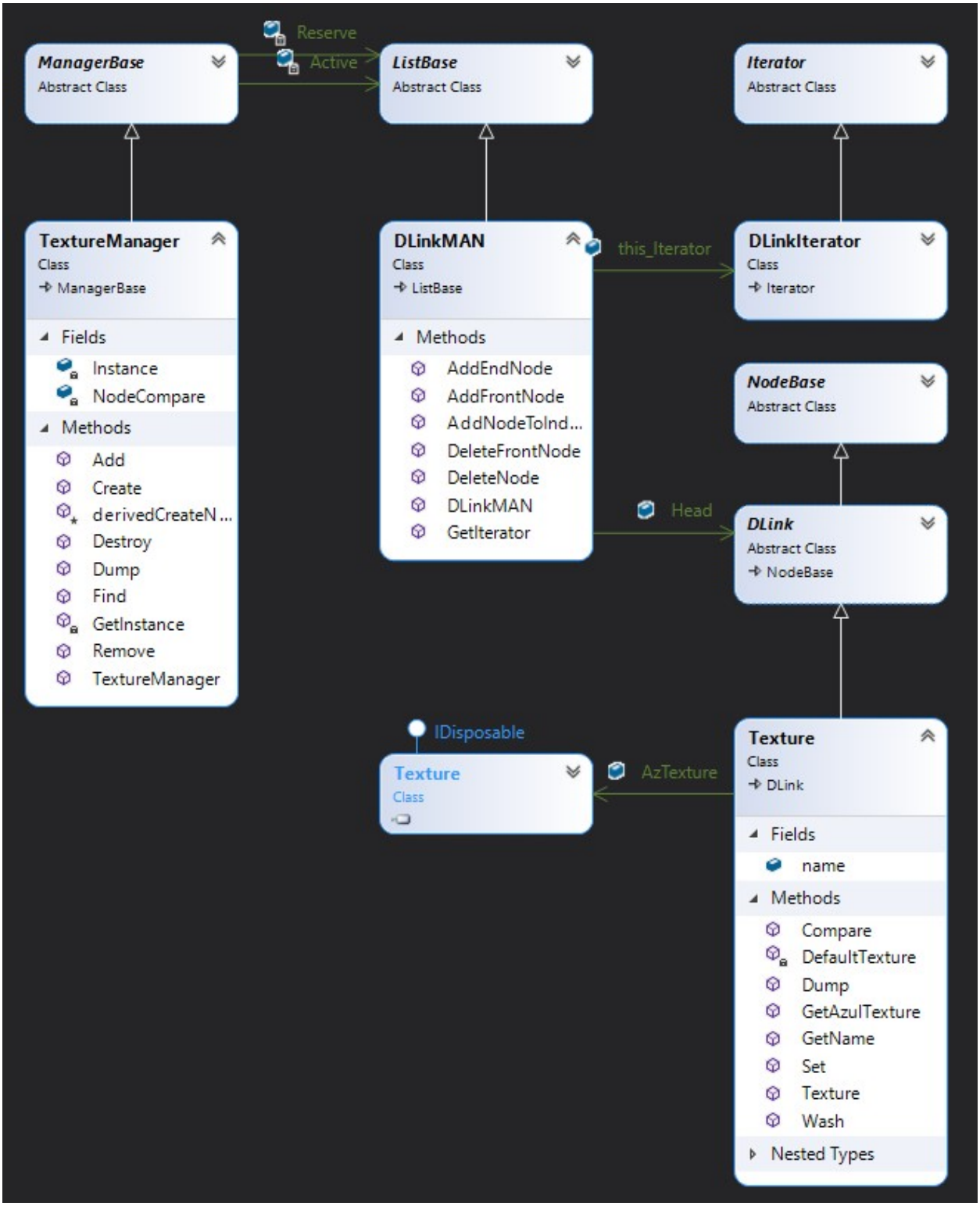

Diagram 1.2 Texture Manager and Azul.Texture

2.2.4 Proxy and Factory Pattern

Proxy is a Sprite Base class that carries a sprite along with all its information such as X and Y. The purpose of this is that the object sprite can be reused over and over.

The Proxy Pattern is used to push new data to the real sprite, which is a Game Object. This object holds a sprite and the data it holds can be updated later.

Factory Pattern allows the software to reuse a Proxy. Thus, by changing the data inside the Proxy, the Factory can create multiple sprite only by change those position data X and Y then use the Sprite Batch Manager to render those minions onto the screen.

For example, the Factory uses the bricks proxy to create multiple copies of their game objects and draw them together to make the shield. The Alien grid is also created the same way.

Every time a new sprite is created with new position data, it is then attached to Sprite Batch for drawing and Game Object Node Manager for managing.

After all the sprites are updated with new position data, the Sprite Batch Manager renders them onto the screen using iterator.

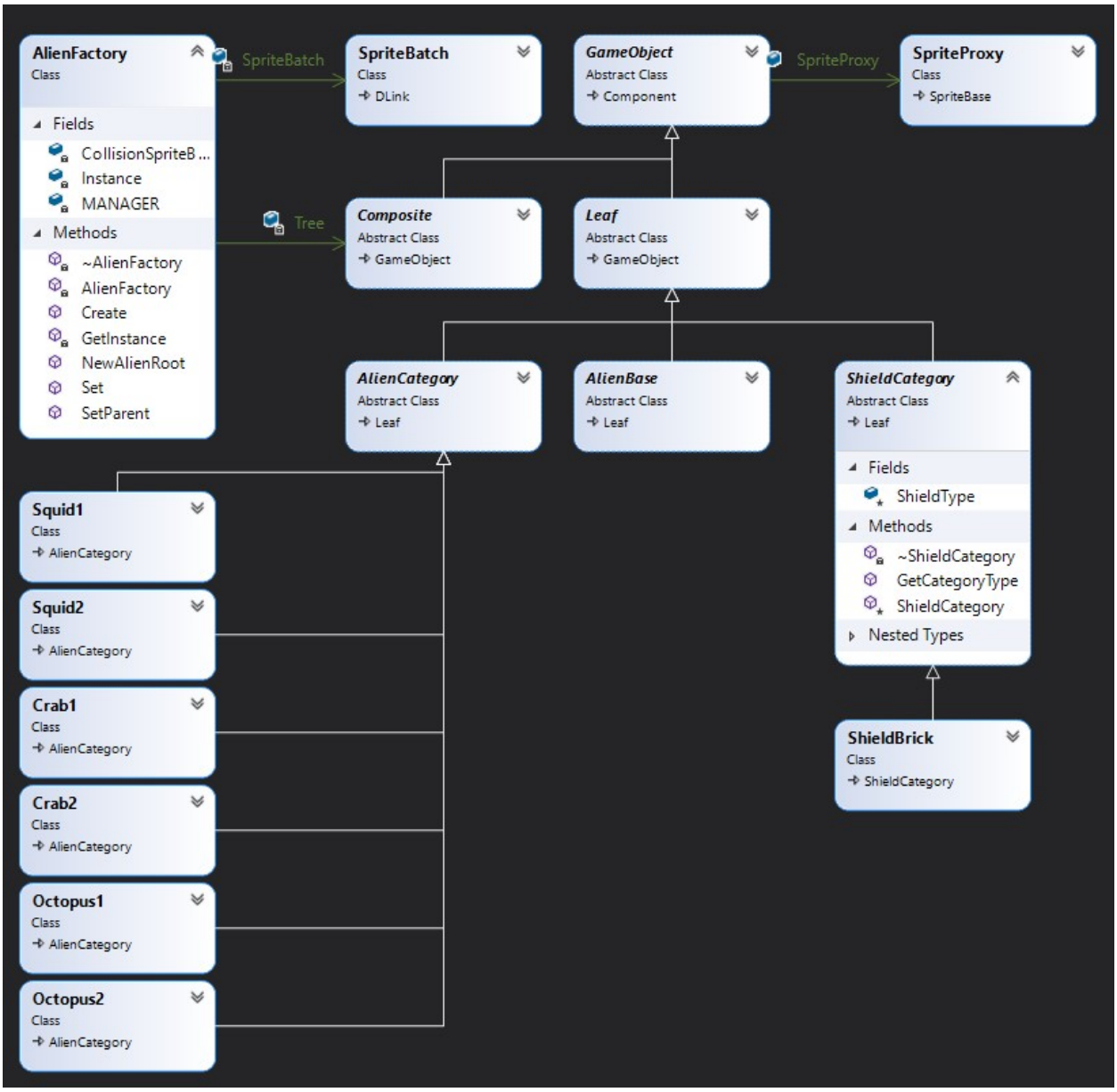

Diagram 1.3 Factory and Proxy

2.2.5 Command Pattern

This pattern serves to control an object that carries all the information needed to perform an action at a later time. The software uses a Timer Event Manager to add all the commands that need to be executed later at a specific trigger time.

In this software, there are three types of command that need to be executed, which are Animation Command, Movement Command, Bomb Command and Bomb UFO Command. Every time the

Timer Event Manager is updated, it walks through the list of the commands added earlier and execute them one by one as they were added in a priority queue.

As the Animation Command helps swap the image in a holder that a Sprite carries at a certain time, the image change perform the animation of that sprite making the alien like walking.

When the Movement Command is executed, it tracks to the Alien Grid and walk through each of the child to change the X & Y of that child so that it can move the whole grid at a certain time. This happens the same way with the alien bombs and UFO bomb.

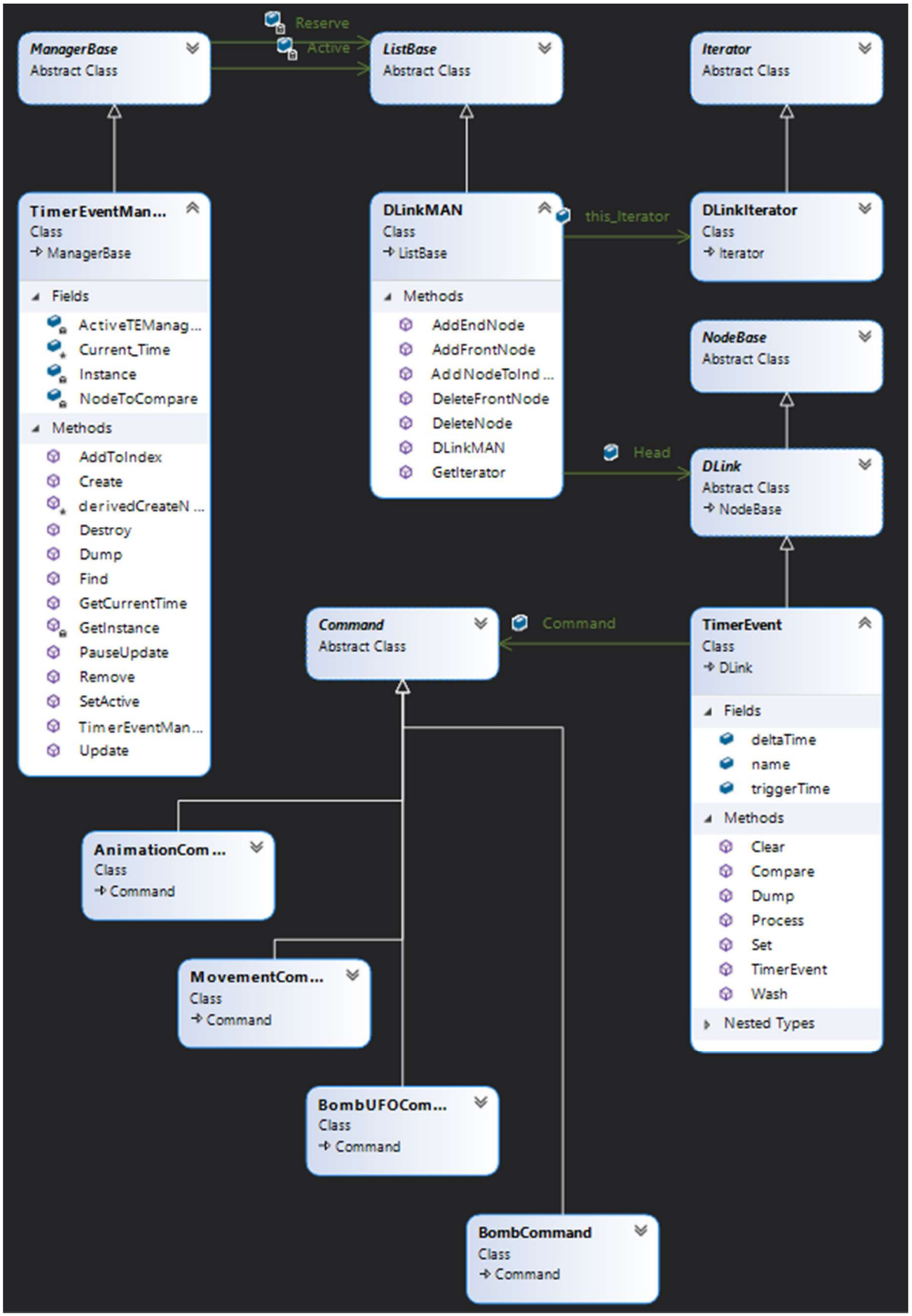
Reserve
Active
ManagerBase
Abstract Class
ListBase
Abstract Class
Iterator
Abstract Class
TimerEventMan...
Class
ManagerBase
Fields
ActiveTEManag...
Current_Time
Instance
NodeToCompare
Methods
AddToIndex
Create
derivedCreateN...
Destroy
Dump
Find
GetCurrentTime
GetInstance
PauseUpdate
Remove
SetActive
TimerEventMan...
Update
DLinkMAN
Class
ListBase
Methods
AddEndNode
AddFrontNode
AddNodeToInd...
DeleteFrontNode
DeleteNode
DLinkMAN
GetIterator
this_Iterator
DLinkIterator
Class
Iterator
NodeBase
Abstract Class
Head
DLink
Abstract Class
NodeBase
Command
Abstract Class
Command
TimerEvent
Class
DLink
Fields
deltaTime
name
triggerTime
Methods
Clear
Compare
Dump
Process
Set
TimerEvent
Wash
Nested Types
AnimationCom...
Class
Command
MovementCom...
Class
Command
BombUFOCom...
Class
Command
BombCommand
Class
Command

Diagram 1.4 Command Pattern and Timer Event Manager

2.2.6 Composite Pattern

The Composite pattern allows me to treat all the similar components as a big object. Inside this object, a hierarchy is created to assign which are the parents and which are the children.

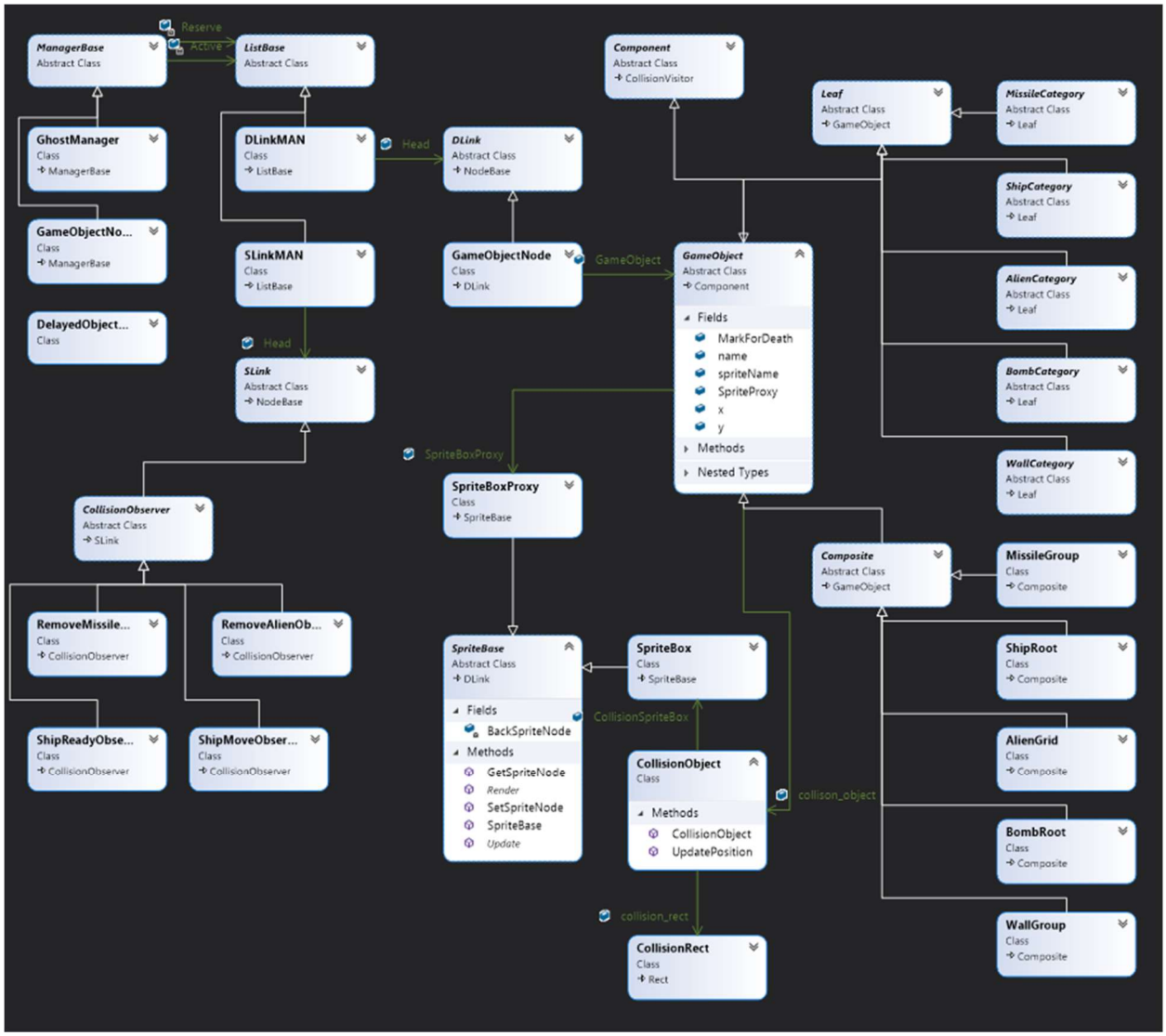

Diagram 1.5 Composite Pattern

Diagram 1.5 shows the relationship between Composite objects and Leaf objects. For example, the Alien Grid and Alien Column are objects typed Composite. The Game Object class carries and Add() method that set parent and child for the object A that adds the object B.

Diagram 1.6 shows a deeper level of the hierarchy as each Alien like Squid or Crab or Octopus typed Leaf are added to the Alien Column. Therefore we have the relationship as follow:

- Grid is the parent of Column
- Column is the parent of each Alien such as Squid

This Composite pattern is also applied to Wall and Wall Root, Bomb and Bomb Root, Bumper and Bumper Root, etc.

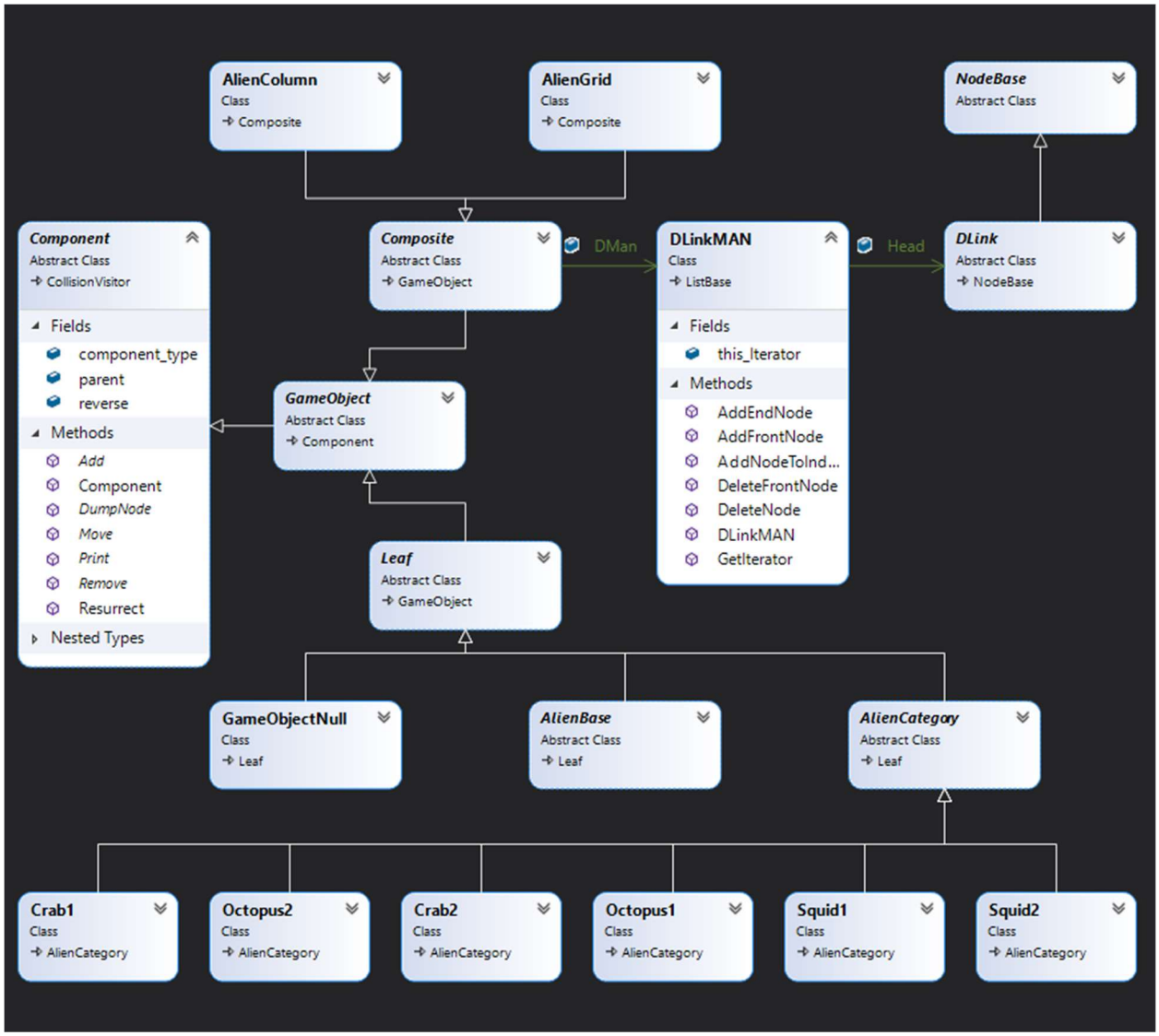

Diagram 1.6 Composite Pattern – Inside Category

2.2.7 Visitor, Observer and Flyweight Pattern

The Visitor pattern sets up and takes control the communication between bricks, column, root versus the missile group using Collision Pair Manager (Diagram 1.7). Every time the Collision Pair Manager is processed, it goes through its list of collision pairs and check up every pair that was added to the manager at the beginning to see if they collide.

To check if two game objects collide, a rectangle around each object is created using Sprite Box Collision and the function finds the intersection of these rectangles.

If they find a collision pair that is valid, the procedure Accept – Visit is now processed. To be simple, if object A accepts object B, object B then has to visit object A and setup a collision pair.

In case of a group visit, the visit method has to track to the child node of the parent. For example, a missile collide with a shield has to track down to a child node to find out which column and brick that it collides with. As the leaf brick that collides with the missile is found and the missile accepts, the brick has to visit the missile. At this moment, the collision pair node has to notify the listeners, which means the collision subject the node holds will have to notify. What the subject do is to go through a SLink list of the Observers it holds and notify those observers.

The Observer pattern will have an observer do something when they get the notification from the Subject of the Collision Pair.

For the sound, IrrKlang system is used to create an instance as a sound manager, and attach it to a collision pair. The Sound Observer will play the sound when it is notified by the subject of collision pair.

To cut multiples letters and characters out of a texture and print them onto screen, the best way is to use Flyweight pattern. More specifically, Glyph Manager captures the position and the size of each letter and character in a Glyph file. Sprite Font then sets those data to itself, then Font Manager can use that Sprite Font to draw the letter or character onto the screen.

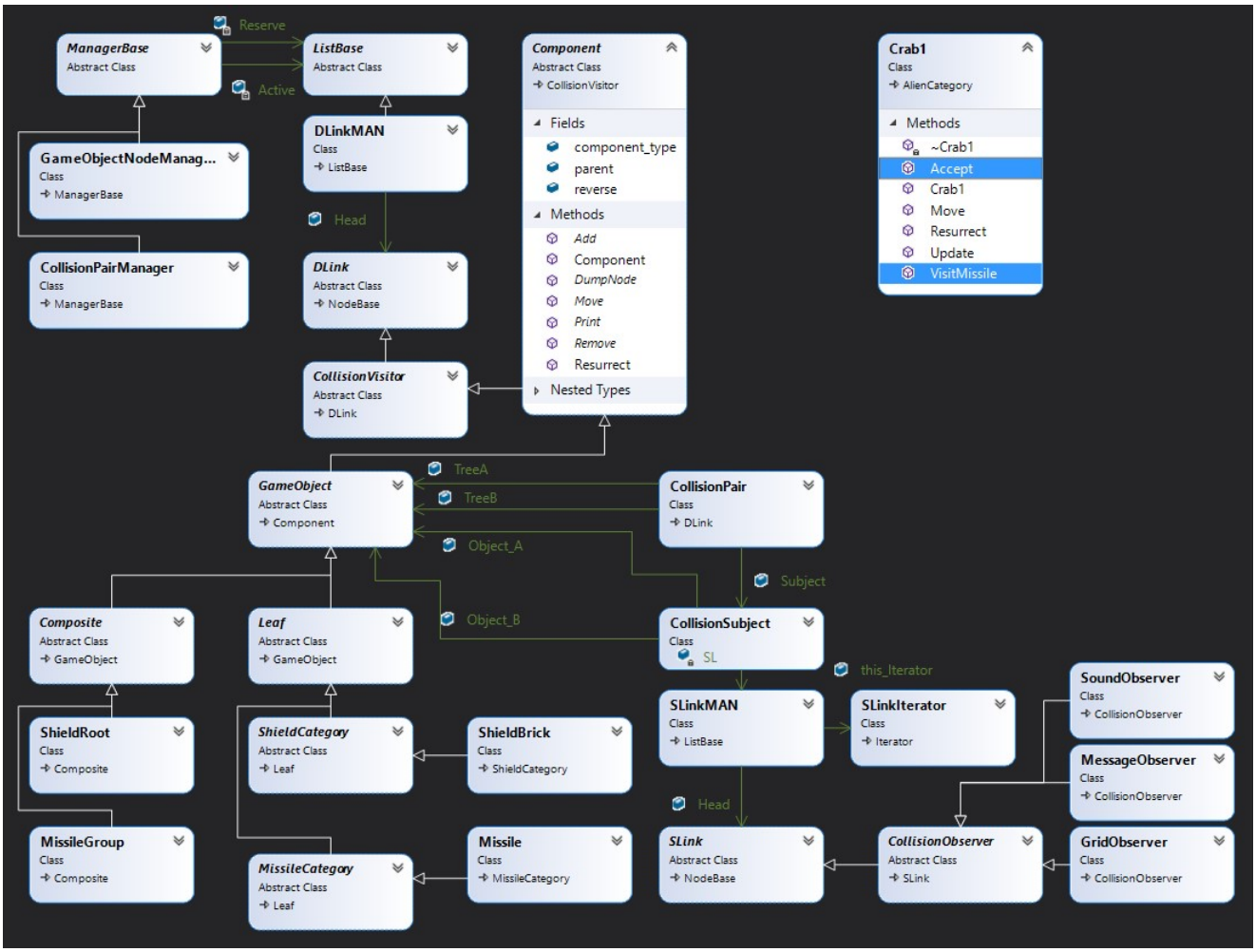

Diagram 1.7 Visitor and Observer Pattern

2.2.8 Strategy Pattern

This is the pattern that enables a strategy to be perform at a specific time. So as long as the Bomb has three different strategies, the code will instruct which Bomb Strategy should be performed at a specific moment. (Diagram 1.8)

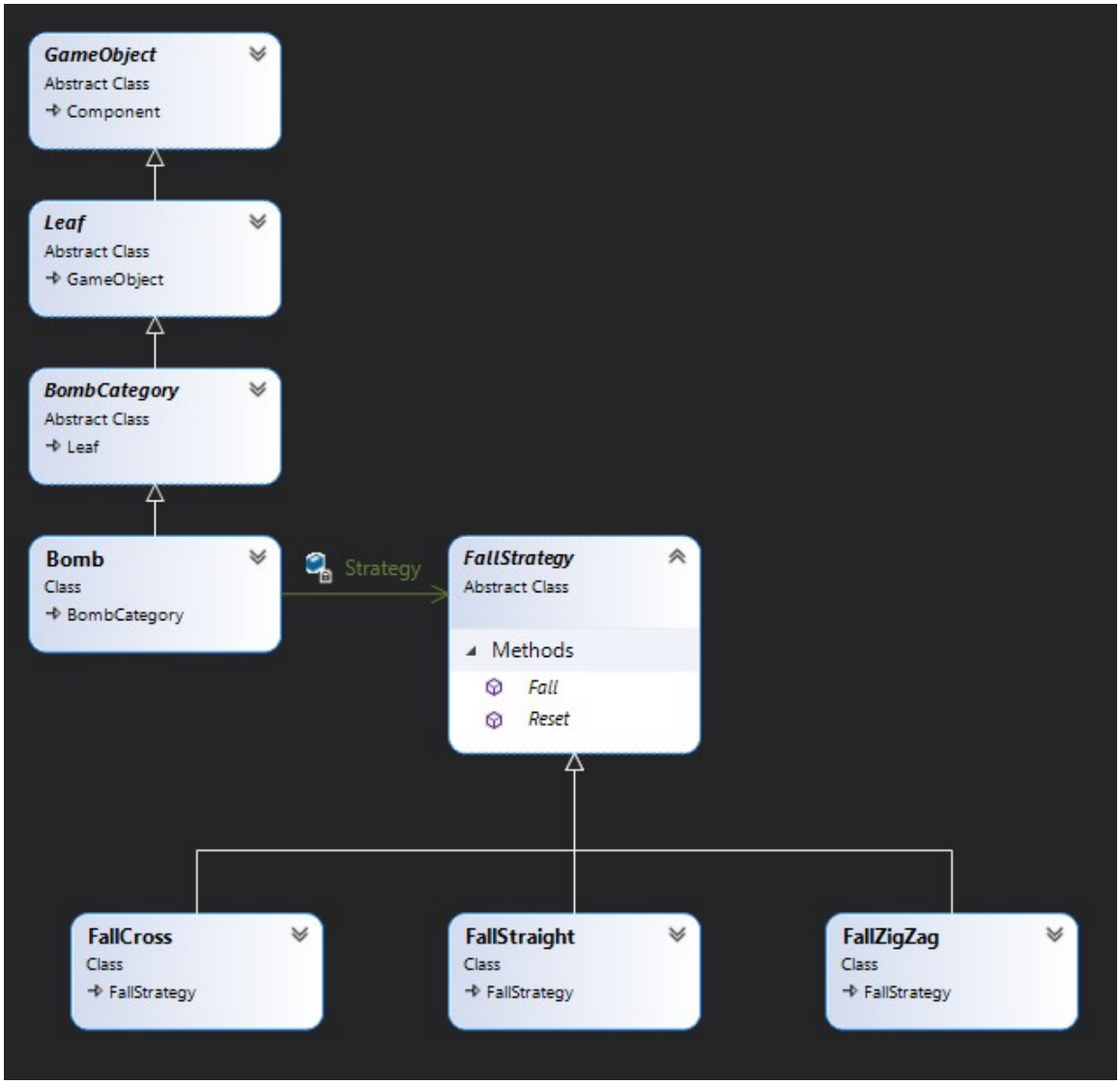

Diagram 1.8 Strategy Pattern

2.2.9 State Pattern

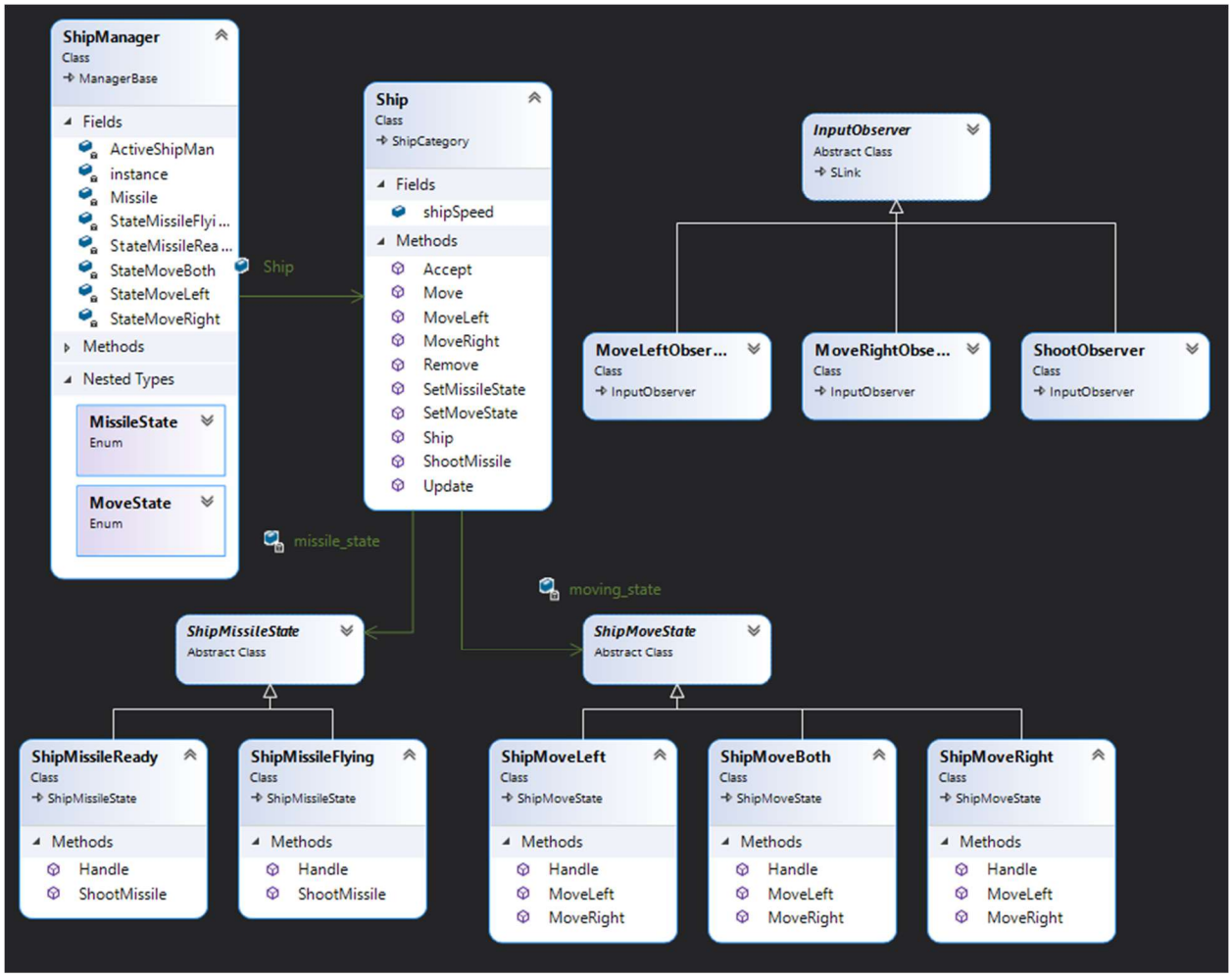

Diagram 1.9 State Pattern

An example of this is that the programmer has to perform different situations of the ship. For example, the ship cannot shoot two missiles at the same time or moving across the screen without crossing the lines where they are supposed to be around.

The problem sounds simple but there are many situations that can be listed:

- When the ship is ready, it can fire the missile at any time.
- When the missile is flying, the ship holds on until the missile explodes.
- The ship can move left or right across the screen.
- As it reaches the limit edge, he can only chooses the other way.

Imagine that if the programmer does not set a condition where the ship is all the way on the left of the screen as it can keep moving left. The ship will disappear which seems ridiculous.

The Diagram 1.9 specifies how the State Pattern solves the problem. At the beginning, a Singleton is to create an instance of a Ship Manager which can control all the movements of the ship. The manager has five fields which are five different stages the ship has to deal with. The ship carry two fields that depict two types of states: shooting and moving. In each types, the ship may have two or three different states when it can only perform a limit of actions at the specific time.

These states are added to a Subject which is an Slink. An iterator walks through them and process a list of action that each state carries. For example, at state Ready, the ship can shoot the missile and then set to state Missile Flying.

The observers tell the ship what to do when a state is activated. When a collision happens, it notifies the listeners which then set a limit of actions for the ship. For example, when the ship collides with the wall, the listener got the notification and tell the ship that it can only go back the other way. All the observers are attached to a SLink where an iterator can walk through it and notify any observer that is on the link.

In case there are many notifications happen at the same time and a set of actions has to be performed. By attaching them into Delay Object Manager, these observer nodes will be processed one by one when they are notified.

Simple but not very simple, the State Pattern in the current situation allow the machine to control multiple states of the ship by switching around when a specific observer is notified. This is a simple enough example to show that State Pattern is absolutely a good choice in dealing with many cases at the same time.

The State Pattern is implemented to create multiple scenes by setting a scene state for the Scene Context as well. From the Game, a new Scene Context initializes four different scenes. All these four scenes use a scene base which is called Scene State.

The Scene Context also holds a field called Scene State so that the beginning state for the context by running the Enter() of that state can be set. In the game, user enters the Scene Select first.

In each scene, a unique Sprite Batch Manager, Font Manager, Timer Event Manager and many others are created so that each players can have their own unique assets.

To do this, each scene must have a data field for each manager so that it can set them Active when a specific scene uses it. These managers are unique since they are fields that are owned by that specific scene. In each of the manager now, instead of using an instance to control the 2 lists of nodes, a data field with the same type of that instance is used. The SetActive() will set the manager data field of a specific scene to that data field of the manager.