Edmund Wright

SE 456 Winter 2018

DePaul University

Milestone 2: Space Invaders

Contents

Introduction

The system was implemented almost entirely from scratch in C# with all data structures and algorithms being built from the ground up. No premade containers where use. There are only 2 components of the project that I didn't build myself. They were a bare bones game engine developed by DePaul University called Azul which was used for hardware rendering of sprites to the screen and an audio library called irrKlang. This document is meant to supplement my implementation of the Atari game Space Invaders by highlighting a few of the object-oriented design patterns used extensively throughout the project. These design patterns allowed me to build a highly reusable framework for producing simple 2D games in a relatively short period of time.

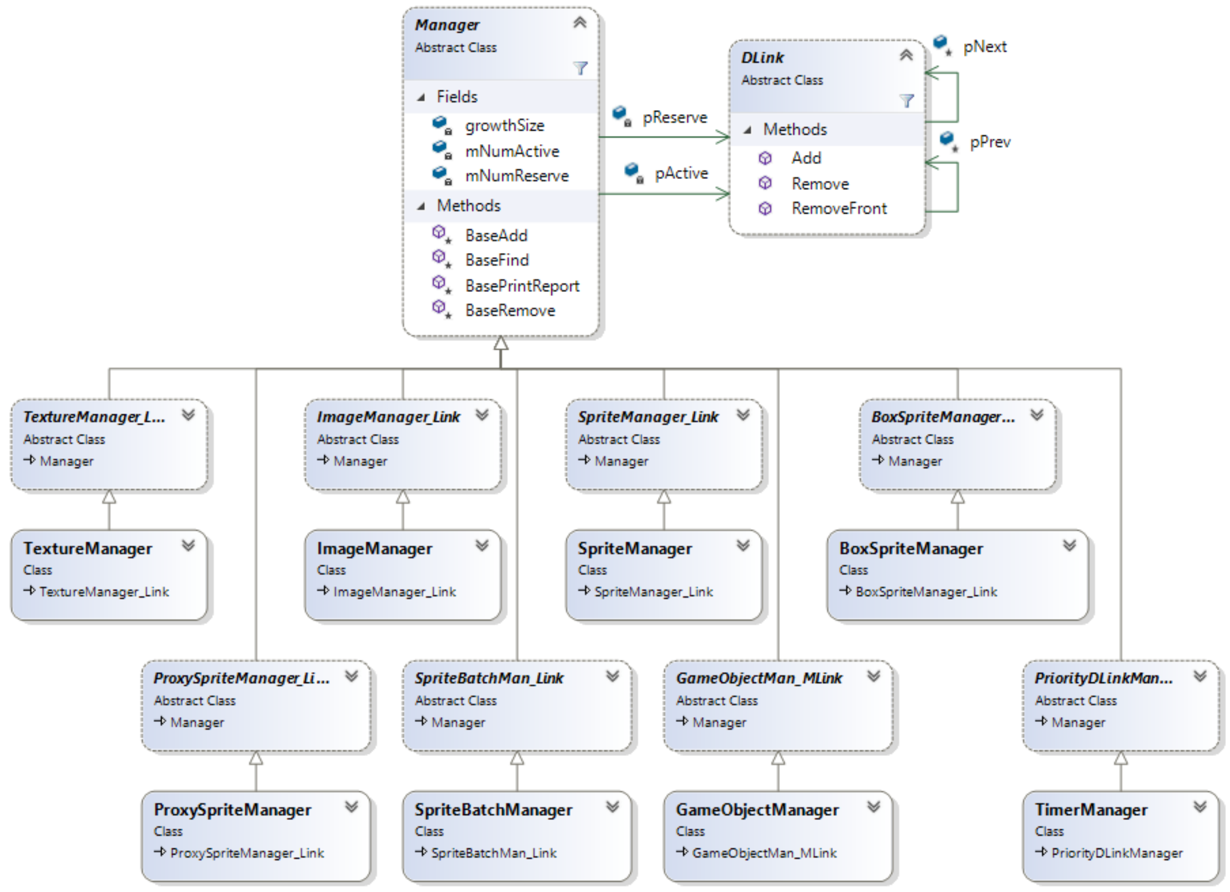
Design Patterns

Object Pool Pattern

Problem Statement
When designing real-time systems considerations should be made for optimizing performance. The frequent allocation and destruction of resources are time sinks that cannot be afforded at runtime. This is especially true when the creation and instantiation of objects are particularly expensive due to the marshalling of extremely slow resources like the file system. In our case, many game sprites which rely on hardware textures must frequently be created, updated, and destroyed.

Explanation of Pattern
The object pool pattern allows resources to be used, reinitialized, and reused without being created, instantiated, or destroyed. It does this by creating and instantiating a "pool" of objects which it manages for the duration of object pool's existence. At runtime instead of creating a new object, an initialized object can be retrieved from the object pool and initialized. When an object is requested, the object pool will search for the object within the pool it manages and return it to the requestor. When the requestor is done using the object, instead of destroying the object, it will simply be returned to the object pool to be recycled and reused.

UML



Pattern and Object Mechanics

When the object pool is created, it immediately creates a pool of uninitialized objects which it tracks as a double linked list of objects called the “reserve” list. The reserve list is inaccessible to all outside of the object pool and from the outside the pool looks empty. When objects are “added” to the pool, an uninitialized object is simply initialized and moved from the reserve list to the “active” list. If the reserve list is empty and a request is made to add an additional object to the pool, the reserve list will be repopulated with a number of objects equal to the pool’s growth rate. The number of objects initially created in the pool, as well as the pool’s growth rate are set when the object pool constructor is called.

Use in Space Invaders Game

The object pool design pattern is used prolifically throughout the project. Nearly every major resource and object manager in the game is implemented as a child class of the Manager object pool. As you can see from the UML diagram above, the Manager class has at least 8 children which manages everything from texture resources and sprites to timed events.

Singleton Pattern

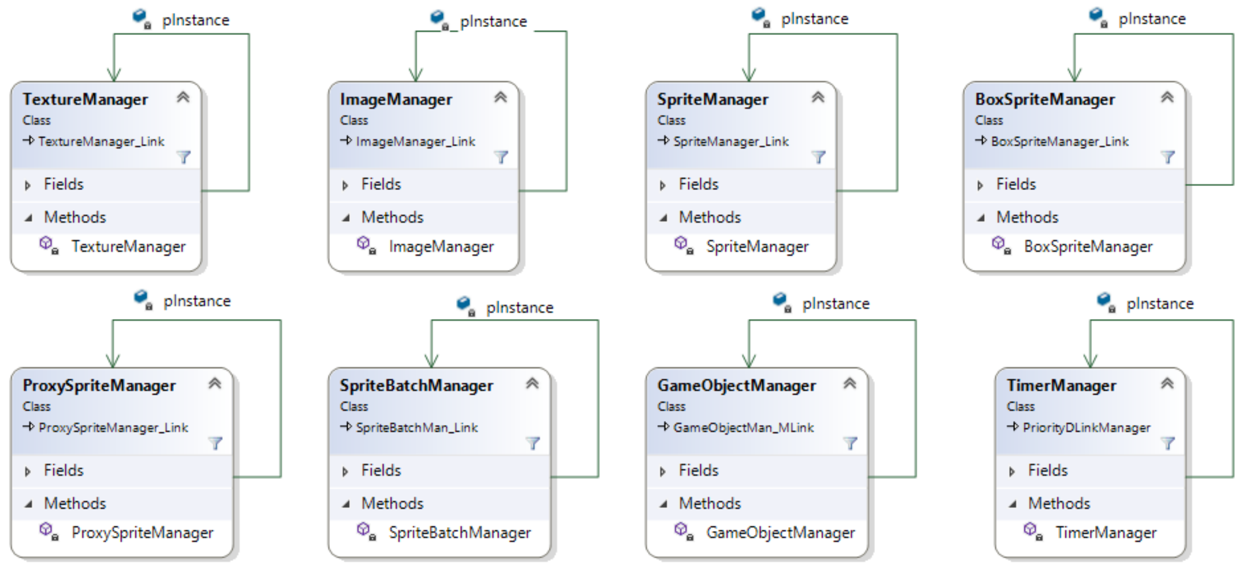
Problem Statement

Our system is so large and complex that it is unsustainable for the programmer to manually keep track of and actively control access to all the objects and resources involved in the system. Additionally, accessing objects and resources from different parts of the system requires that the accessor keep references to the objects it needs to use. In a system with over 100 classes, this would quickly turn the system into an unorganized web of dependencies. We need a way to pass all object dependencies between subsystems without creating crazy constructors and methods that take dozens of parameters.

Explanation of Pattern

The singleton pattern helps solve these problems by doing 2 things: ensuring that only one instance of an object is ever created and providing global access to that object. This object can then act as a single point of dependency for accessing resources and it can facilitate global access to resources so that object references don’t need to be passed around as parameters in methods. The singleton pattern is a good choice when building object managers that need to be accessed globally and there should only ever exist a single instance of.

UML



Pattern and Object Mechanics

The singleton pattern is one the simplest design patterns. It works by just hiding the class constructor and holding a static object reference to itself called an instance. This ensures that the class constructor can only ever be called from within the class and so it can control object creation and ensure that only one instance of the object is ever created. The class may only publicly expose static methods. These static methods typically operate on the single static instance that the class holds.

Use in Space Invaders Game

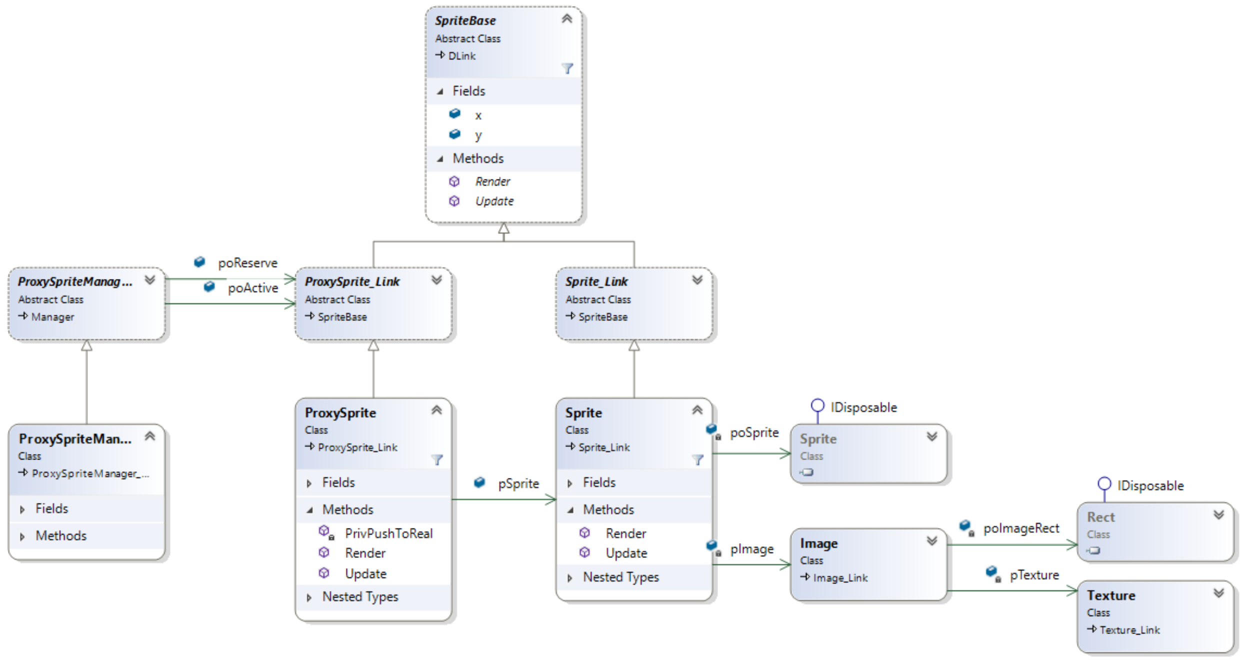
The singleton pattern is the second most commonly used pattern in the project after object pool. In fact, those two patterns are frequently used together such that most resource managers are both object pools and singletons. This combination allows for resource managers to provide global access to the objects in their pools while ensuring a single point of access is enforced. Most of these singletons are created at application initialization and are then dynamically accessed during the lifetime of the application to provide resources like textures and sprites.

Proxy Pattern

Problem Statement
In our game we must display a large number of sprites to the player that look exactly alike and share common textures, animations and behaviors. The only real difference between them is their location on the screen. Sprites are both costly to create and manage. We want to be able to create lightweight copies of sprites that allow us to manipulate just the simple x, y screen position data that differentiates them from each other.

Explanation of Pattern
The proxy pattern allows for the creation of light proxies of much more expensive objects. These proxy objects have similar interfaces as their real counterparts and can be utilized to deal with only the portion of the real object that is being manipulated. A single real object and the rest of its more expensive faculties can be reused by several proxy objects. The proxy pattern also controls access to the real object to some extent by pushing updated data to the real object when needed.

UML



Pattern and Object Mechanics
The proxy object and the real object share a common parent class or interface. The application deals with only the common interface allowing proxy objects to be treated as real objects. The proxy object has a reference to the real object which allows it to access the real objects methods and data. The proxy object also has a method that allows it to push updated data into the real object.

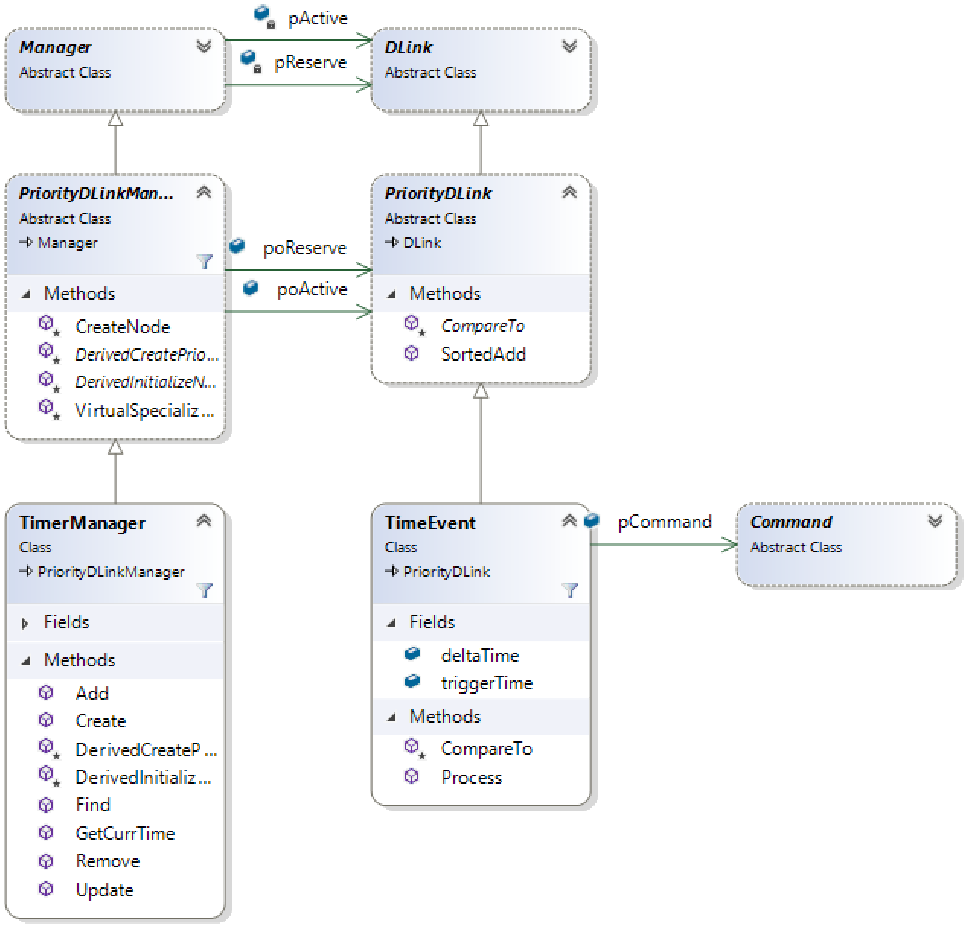
Use in Space Invaders Game

In this application, the proxy pattern allows us to render many sprites without paying the creation, instantiation, and memory costs of managing multiple real sprite objects. The real sprite objects hold the references to an image and a hardware sprite resource. Several proxy sprites will have references to the same real sprite. Each proxy sprite will have its own distinct x, y position values. The application deals with proxy sprites as real sprite objects when updating their positions. The proxy sprites have both an update and render method that when called, push the proxy’s position data into the real sprite and then call the real sprite’s update and render methods. This essentially turns the real sprite into a static resource that is no longer treated as a sprite but instead as a vessel for accessing a hardware sprite’s rendering functionality.

Priority Queue

Problem Statement
Many of the actions and events that occur in the game must happen sequentially and periodically. In addition, we want to be able to schedule actions and events for some time in the future. This way various aspects of the game can be synchronized, such as the movements of the aliens, coinciding the animations of the sprites, and the marching sound effects.

Explanation of Pattern
The priority queue allows us to have a queue of TimeEvent nodes which can execute an action or event with each tick of the game clock. Each event node has a certain level of priority associated with it. This allows the queue to self-organize such that the node with the highest priority is at the front to be executed first, with each subsequent node having less priority than the one before it.



Pattern and Object Mechanics
The nodes of the priority queue have a derived CompareTo method which allows them to be compared with one another using their priority to determine which node is greater. Nodes are added in sorted order using the Compare method of the node along with the SortedAdd method of the PriorityDLink abstract class. In the case of TimeEvent priority queue, the priority of each

node is determined by the trigger time associated with each node. Nodes with trigger times that are closest to game clock's current time are sorted to the front of the queue while items with long trigger times are paced further back.

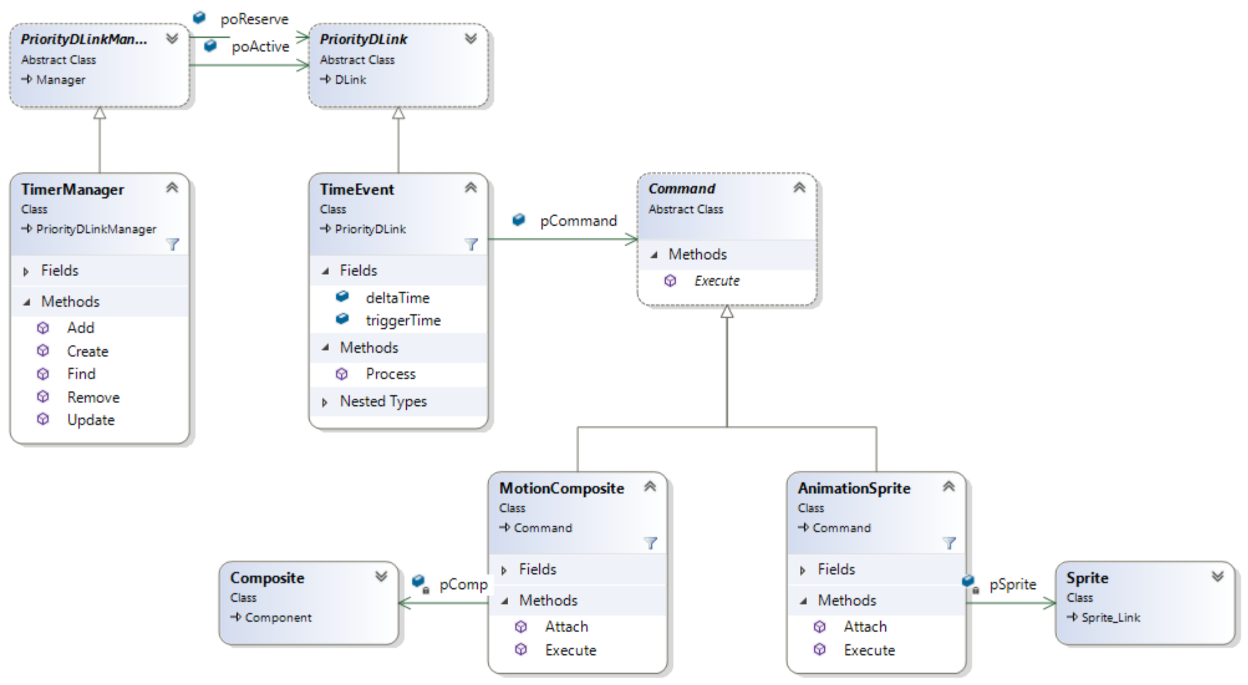
Use in Space Invaders Game

In the game TimeEvents are primarily used for synchronizing the motion, animation, and sounds for the game sprites. There were also used to handling the random spawning times of the UFOs and bombs from the alien grid.

Command Pattern

Problem Statement
The TimeEvent priority queue needs to be able to handle the execution of wide range of the actions or events when the trigger time arrives. As a matter of fact we want the priority queue to be able to handle any arbitrary type of even that can occur. Ultimately the sequential timing of events needs to become disassociated from the actual execution of the event such that the queue need no knowledge of the specifics of the event.

Explanation of Pattern
The command pattern allows for the creation of an object that will encapsulate an event or action into an object, along with any information or parameters that would be needed to execute that action in the future.



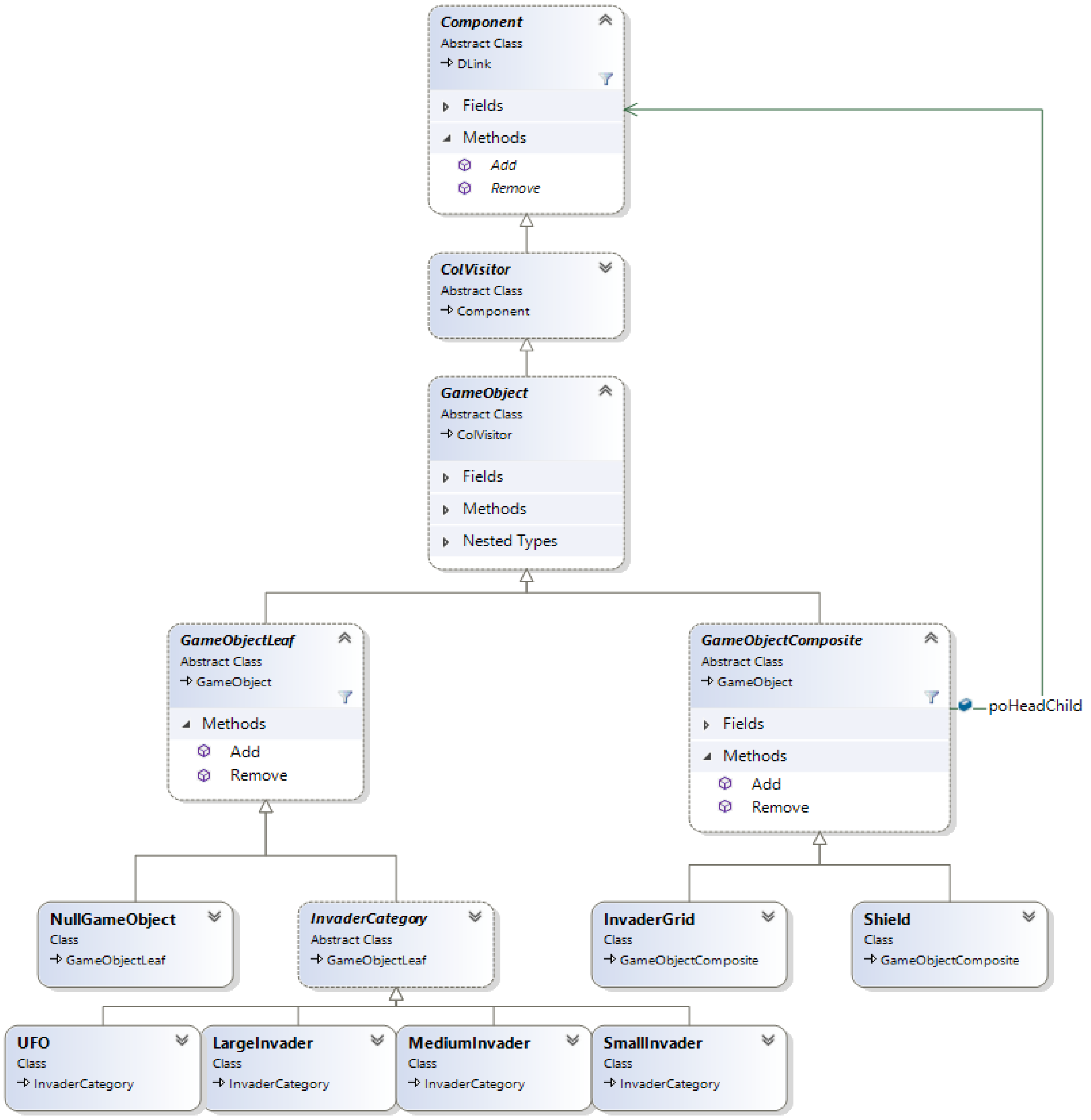
Pattern and Object Mechanics
This pattern is very simple. In order for a class to become a TimeEvent it only has to inherit from the Command abstract class and implement a single method called Execute. The details of what the Execute method can do are not bounded so the class is free to perform any actions inside of the execute method. Each TimedEvent keeps a reference to a command object. When a TimeEvent is processed it simply calls the Execute method of its Command object.

Use in Space Invaders Game
The fact that the TimeEvent is decoupled from the event it executes allows the priority queue to handle a wide range of actions such as playing sound files, randomly spawning GameObjects, coordinating synchronized motions, and animating sprite images.

Composite Pattern

Problem Statement
There are objects in the game that are held in tree-type data structures where the collections of these objects behave the same way as their constituents and often time must be treated indistinguishably from their constituents.

Explanation of Pattern
The composite pattern allows for hierarchies of collections and individual node to be organized, managed, and manipulated using a uniform interface that does not discriminate between the collection types and the individual types of nodes.



Pattern and Object Mechanics

The composite pattern is a hierarchical tree with 3 key features. Each node of the tree is either a leaf node or a composite node. Leaf nodes represent individual objects. Composites represent a collection of nodes that may be made up of any combination of leaf and composite nodes. Both types of nodes inherit from an abstract Component class. The collection of nodes held by Composites are of type Component. This is how it is possible for the Composite to be unaware if the nodes in its collection are leaves or composites. The Component abstract class typically requires the derived classes to implement an interface of abstract methods. This makes it possible for collections and leaf nodes to be manipulated uniformly using this common interface.

Use in Space Invaders Game

In the game several of the GameObjects are implemented as collections of other GameObjects. Some examples are the alien grid and shields. The alien grid is made up of a hierarchy of individual aliens where these individual aliens are organized into columns and eleven columns make up the grid. Similarly, the shields are made up of tiny individual blocks that are also organized into columns and finally into the overall shield shape. This pattern allows me to do things like updating all of the aliens in the grid by simply updating the position of the overall grid.

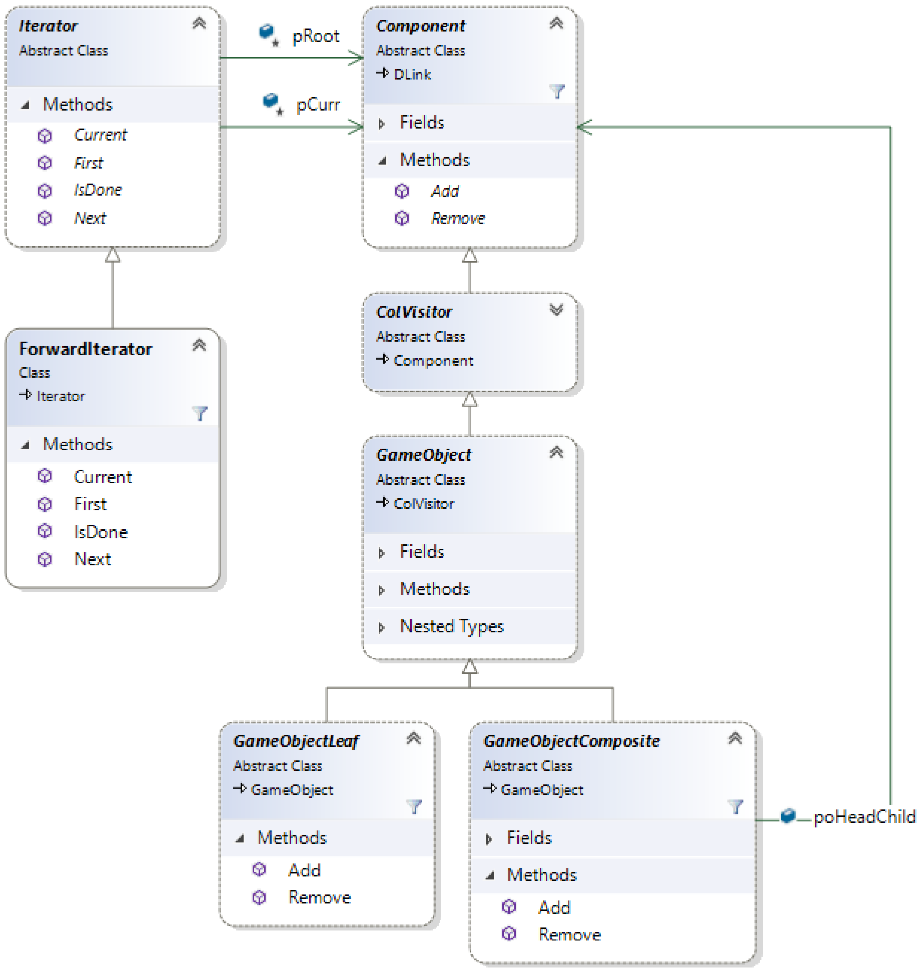
Iterator Pattern

Problem Statement

Organizing GameObjects into trees brought a lot of ease to manipulating collections however the shape and complexity of a hierarchical tree can make traversal of the nodes quite challenging and result in several lines of tricky boilerplate code just to iterate between nodes. Implementation of this boilerplate code requires that the developer has intimate knowledge of the inner workings of the GameObject tree collection.

Explanation of Pattern

The iterator pattern provides a simple way for us to traverse any ordered collection of objects in a uniform way that abstracts away the type of the collection being traversed and makes it unnecessary for the developer to know anything about underlying implementation of the collection itself.



Pattern and Object Mechanics

The iterator pattern consists of an abstract iterator class which defines an interface of abstract methods for accessing the elements in a collection in a sequential manner. These interface methods can be used in while-loops to greatly reduce the boilerplate code required to traverse a complex collection. Typically the loop just involves calling IsDone in the test expression of the loop then calling the Next method somewhere within the loop until the IsDone method finally returns a value of true.

Use in Space Invaders Game

When we want to manipulate all nodes of a tree composite of GameObjects, the most natural way would be for use to use the Component interface to make a recursive call to a method implemented by all nodes. This can be dangerous depending on the size of the tree since it can lead to a stack overflow scenario.

In the game we use the Composite pattern for its data organization properties however we use an iterator to handle traversal of the tree. The ForwardIterator concrete class holds all of the logic and complexity for proper traversal of the tree without the use of recursion. From then on we can use the iterator’s simple interface whenever we want to iterate through the collection in order to manipulate each individual node.

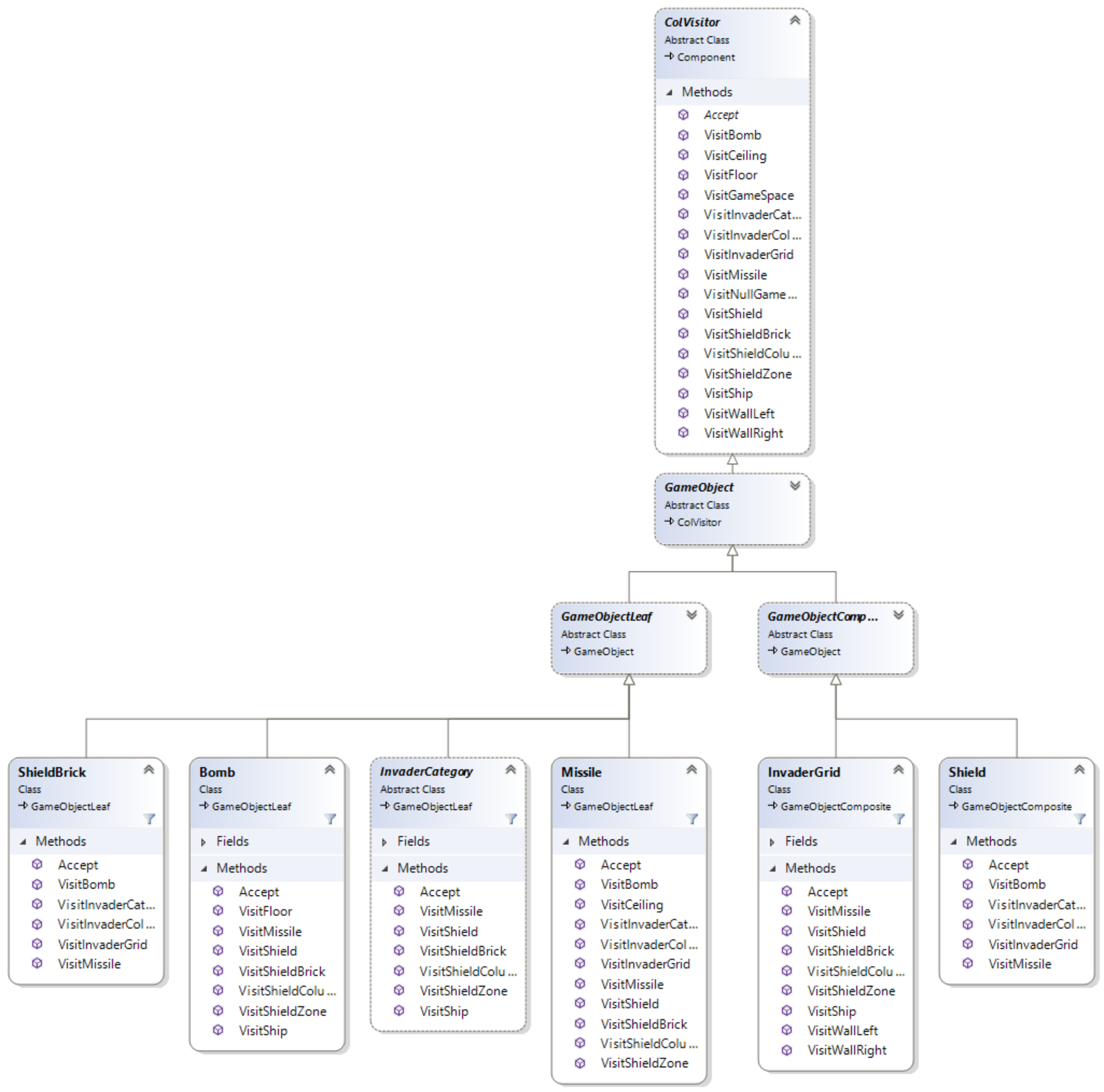
Visitor Pattern

Problem Statement

Many of the GameObjects in Space Invaders have the ability to collide with other objects. When this occurs, we need to know which two objects collided so that we can execute any actions or updates that should occur as a result of the collision. The naive approach to this problem would be to simply perform the collision calculations on every single unique pair of collidable objects in the game. The shear number of collisions that would need to be calculated each clock cycle make this method prohibitively expensive.

Explanation of Pattern

The visitor allows us to reduce the number of collision calculations by providing an interface that helps us to identify the types of objects involved in a collision.



Pattern and Object Mechanics

The visitor pattern involves an abstract Visitor class which defines abstract “visit” methods for every type of concrete object that might be involved in an operation. The concrete children of Visitor then must implement the “visit” methods for the types that they may perform operations with. In our case, the Visitor class also defines an abstract Accept method which takes a parameter of type Visitor. When the Accept method is called within an object, it then calls the “visit” method of its own type on the parameter object it was passed. After that point knowledge of both object types involved in an interaction are known.

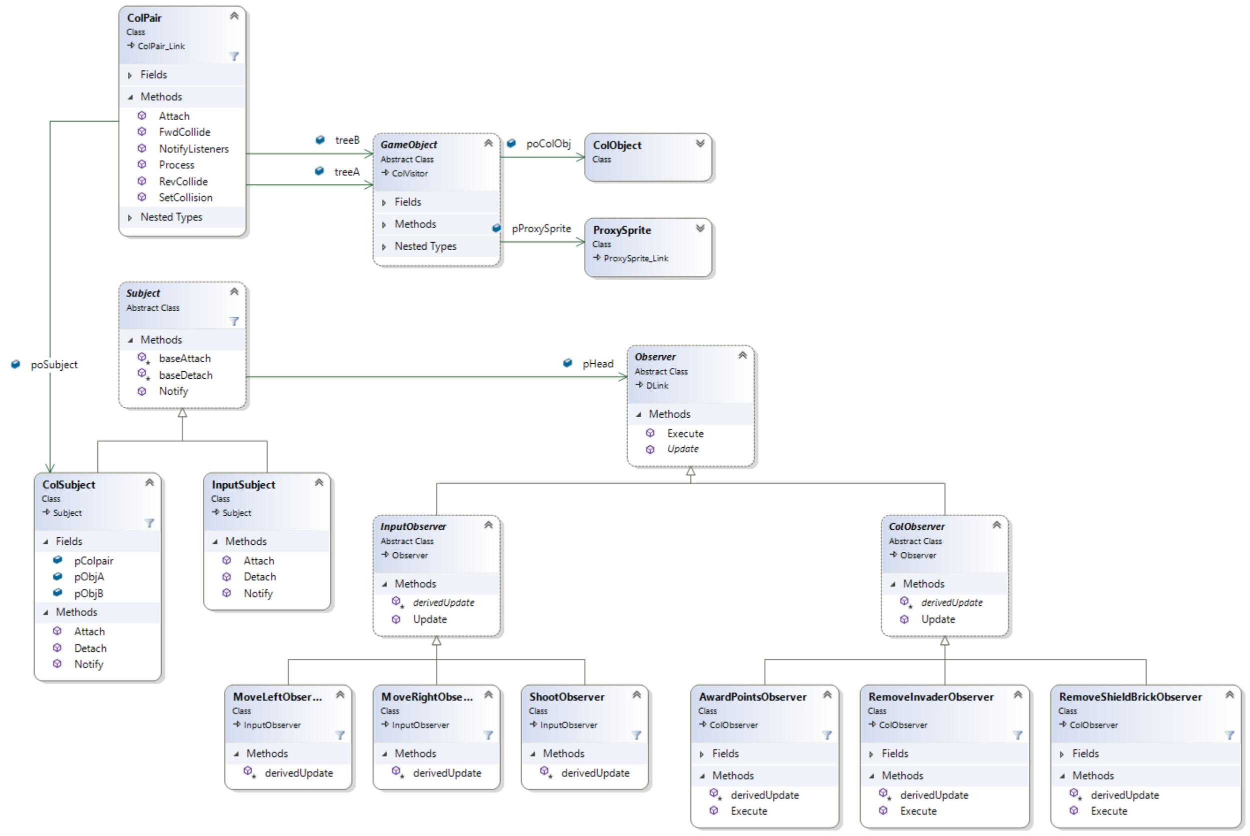
Use in Space Invaders Game

The visitor pattern is used to handle collision interactions between objects in Space Invaders. When two objects collide, such as a ship missile and an alien, the visitor pattern allows us to not care which object collided hit which object. Whether the missile accepts the alien, or the alien accepts the missile, we will know that the collision occurred between the alien and the missile and we can trigger the same update for both cases.

Observer Pattern

Problem Statement
When specific events occur in the game we often times want to trigger a series of additional action to be taken as a result. Several parts of a system may be dependent on the state of a single object or system. When the state of that object changes we need a way to notify the dependent entities about the change in state,

Explanation of Pattern
The observer pattern provides a way to decouple a single object from the entities that depend on it's state. Additionally, it facilitates the many-to-one relationship between the object and its dependents.



Pattern and Object Mechanics
The single object is called an Observable or a Subject. The subject typically keeps a collection of Observers that it will notify when an event occurs. The Subject abstract class requires the concrete subjects to implement the methods Attach/Detach for manipulating its collection of Obeservers as well as a Notify method for notifying each observer in the case of a state change event. The Observer class is an abstract type that requires it’s children to implement an Update method. This Update method is what is called by the subject’s notify method when state changes.
.

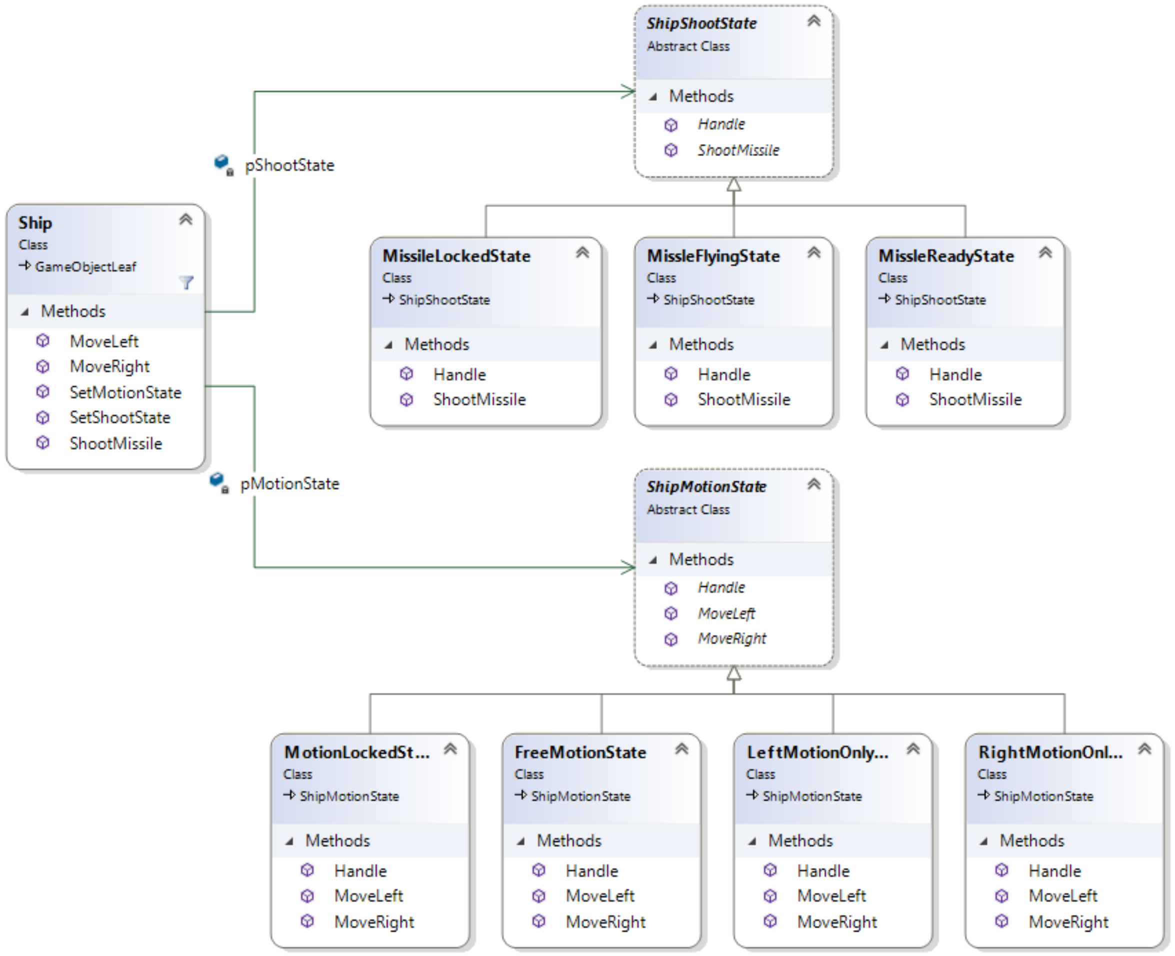
Use in Space Invaders Game

In the game we use the observer pattern prolifically, especially when dealing with GameObject collision events. The collision event is encapsulated as a CollisionPair (ColPair). Each ColPair has a ColSubject associated with that collision event and each ColSubject keeps a collection of Observers that it will notify when the collision event occurs. Take the example of when a missile collides with an alien. There is a collision subject associated that will notify Observers that handle everything from removing the alien sprite from the screen to awarding additional points to the player who killed the alien.

State Pattern

Problem Statement
There are Objects in the game whose entire behavior must be able to change at runtime. We would prefer to be able to delegate state-specific behavior rather than having large methods with chains of if-statements. Additionally we'd like to encapsulate all behaviors related to a specific internal state and keep them discrete with respect to behaviors associated with other states.

Explanation of Pattern
The State pattern allows for an object to change the behaviors and characteristics of a class so drastically that it appears to change its class completely without changing its interface.



Pattern and Object Mechanics
Similarly to the Strategy pattern a context class provides a method whose implementation algorithm is not defined by the class itself. Instead it delegates the task to a class that it holds internally. However, this goes a bit further than the basic strategy design pattern. The state pattern has the context object hold a reference to a state object which encapsulates the behavior associated with that specific state. There can be several types of states but they all derive from a

State class which defines abstract interface methods along with an abstract Handle method. The handle method can be used switch out one state object for another.

Use in Space Invaders Game
The state pattern is used for managing the behaviors of several key objects in the game. Some of them include the alien grid, the player's ship, and the state of the overall game itself. An example is of the two types of states processed by the player ship. The player ship has references to a MotionState and a ShootState. The ShootState determines which algorithm will be called when the player presses the spacebar button to shoot a missle. The MotionState determines which algorithm will be used when the player pressed the arrow buttons to move the ship. As an example, when the ship collides with the right side of the screen it enters a specific motion state that doesn't allow the ship to move right any further. In the same vain, when the player fires a missile the ship enters a ShootState that prevents the use from firing another missile until the previously fired missile has died.