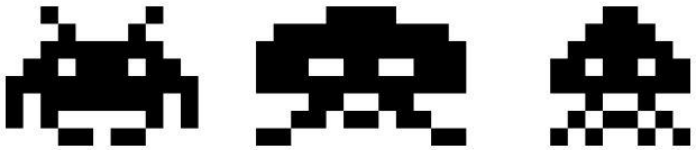
Design Patterns in Space Invaders



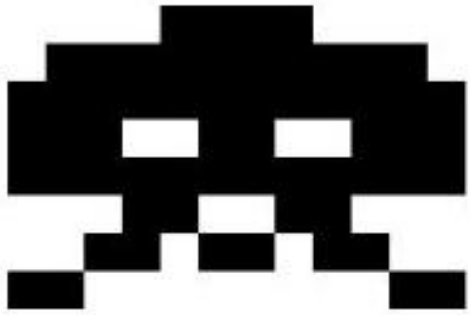
Anna-Lisa Suarez (Vu)

SE 456

Winter Quarter 2024

Contents

Creational Patterns



Factory Pattern

Problem

In Space Invaders, I was tasked to create the same type of object multiple times. Sometimes the creation of objects can be complex. Furthermore, there could be different ways to create the object. Therefore, there is a need to hide the complexity behind the creation of objects to ensure that objects are created in a consistent and safe way.

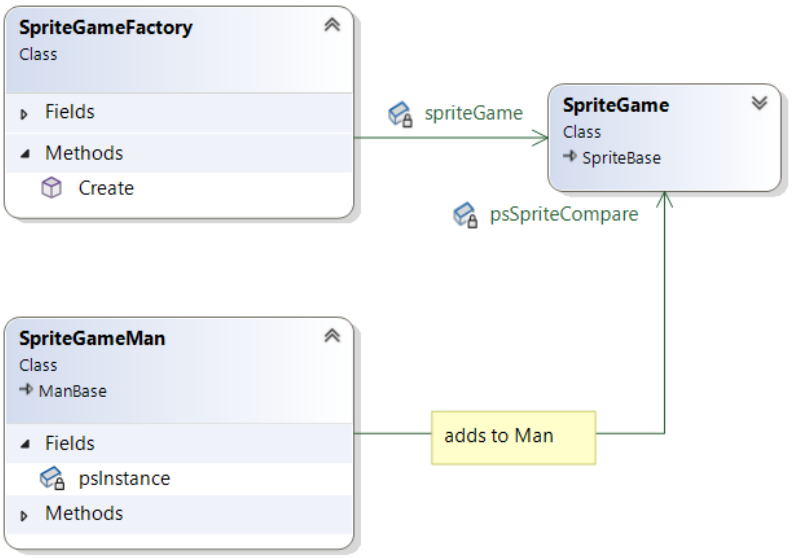
Solution

The intent of the **factory pattern** is to create objects without exposing the complexities behind creating it. It allows for a consistent way to create objects through a common interface.

Factory Pattern Mechanics

Once it is decided to create a factory for a specific type of object, a special factory class must be created. This class is the only class that calls ***new*** on the object to be created. A public ***Create()*** method must be created. This method can take in parameters that is needed to create the object. This ***Create()*** method returns an instance of the newly created object.

UML Diagram of the Sprite Factory Class in Space Invaders



Factory Pattern in Space Invaders

I use the factory pattern to create aliens, sprites, shields, and other objects in space invaders. This provided me with an easy and consistent way to create these objects.

Object Pool Pattern

Problem

I needed to create and re-create a lot of objects to facilitate the game play. These objects included anything from aliens, ships, text, boxes and more. The creation, destruction, and re-creation of these objects can slow down our program because each time we create and destroy an object we are calling ***malloc()*** and ***free()***. To keep the program running in the most optimal manner, I needed to find a way to "recycle" these objects so that I didn't have to create new instances every time.

Solution

The intent of the **object pool pattern** is re-use objects instead of creating and destroying objects each time. It is best to use this pattern on objects which are created and destroyed many times.

Object Pool Pattern Principles

It is important to adhere to the following principles when using the object pool pattern.

- Classes (a.k.a clients) using the objects should not "own" the object. The object must always be returned to the object pool for others to use.
- Clients should not delete or clone the objects. This would defeat the purpose of using an object pool.
- Every time an object is requested from the pool, it must be returned to the pool. Otherwise, there would be memory leaks.

Object Pool Pattern Mechanics

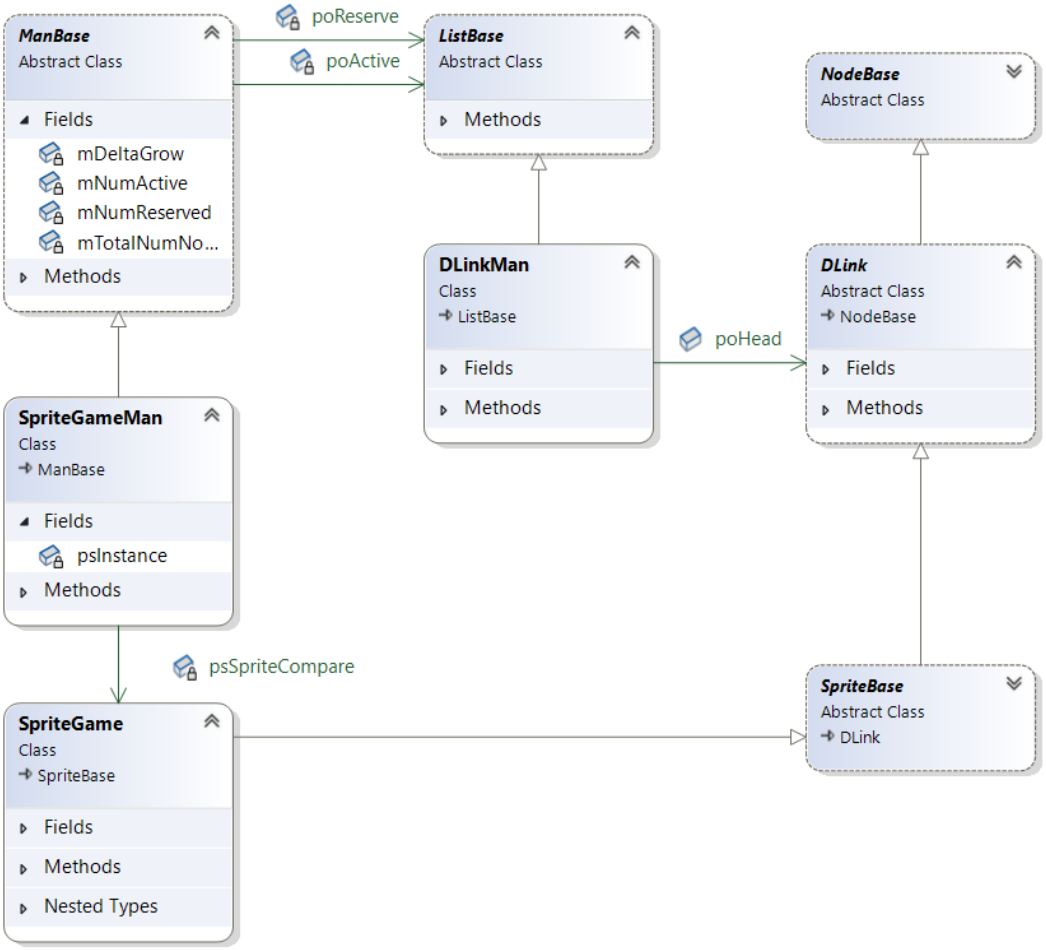
The basic flow of an object pool is as follows:

1) A client needs to create an object and requests as object from the pool.
2) If there are objects available in the pool, the pool returns it to the client.
3) If there are no objects available in the pool, a new object is created and returned to the client.

When a client is finished using the object, all member variables in the object must be reset before it is returned to the pool.

Clients are generally not aware that the object they are using is recycled. This information is not important in the context of what the object is being used for.

UML Diagram of the Sprite Game Class which uses Object Pool Pattern in Space Invaders



Object Pool Pattern in Space Invaders

I am using the object pool pattern for all the managers in Space Invaders. These managers keep track of the creation of key objects we need for the game.

To manage the creation and recycling of the objects, there are 2 linked lists – 1 to manage actively used objects and the other to manage reserve objects. When an object is created, it is first placed in the reserved list. Then, when there is a request to use an object, it is taken from the reserve list to the active list. When the client returns the object to the pool, it is taken from the active list, reset (set all member variables to defaults) and placed in the reserve list.

This type of mechanism ensures that the a minimal amount of objects are created at any given time. There is also a mechanism in the pool to control the initial number of reserve objects and the growth rate. This gives us more control on the number of times objects should be created.

Singleton Pattern

Problem

There are many different types of objects that interact with each other in our game. To have a fully functional game, I must be able to have access to any given object at any given time. Getting a hold of these objects can become problematic since that means we would have to be passing managers all over the place. Therefore, to implement our solution in an efficient way, we needed a global way to access any object that we need at a given time.

Solution

The intent of the **singleton pattern** is to ensure a class is only instantiated once and there is a global way to access it.

Ensuring a Single Instance

To ensure that only a single instance of a class is created, it is imperative that the singleton class contains an instance of itself. This instance is only initialized once and is accessed whenever there are methods that need to be run on the singleton object.

Global Access to Singleton

There are 2 ways to control the access of an instance of a singleton class. They are:

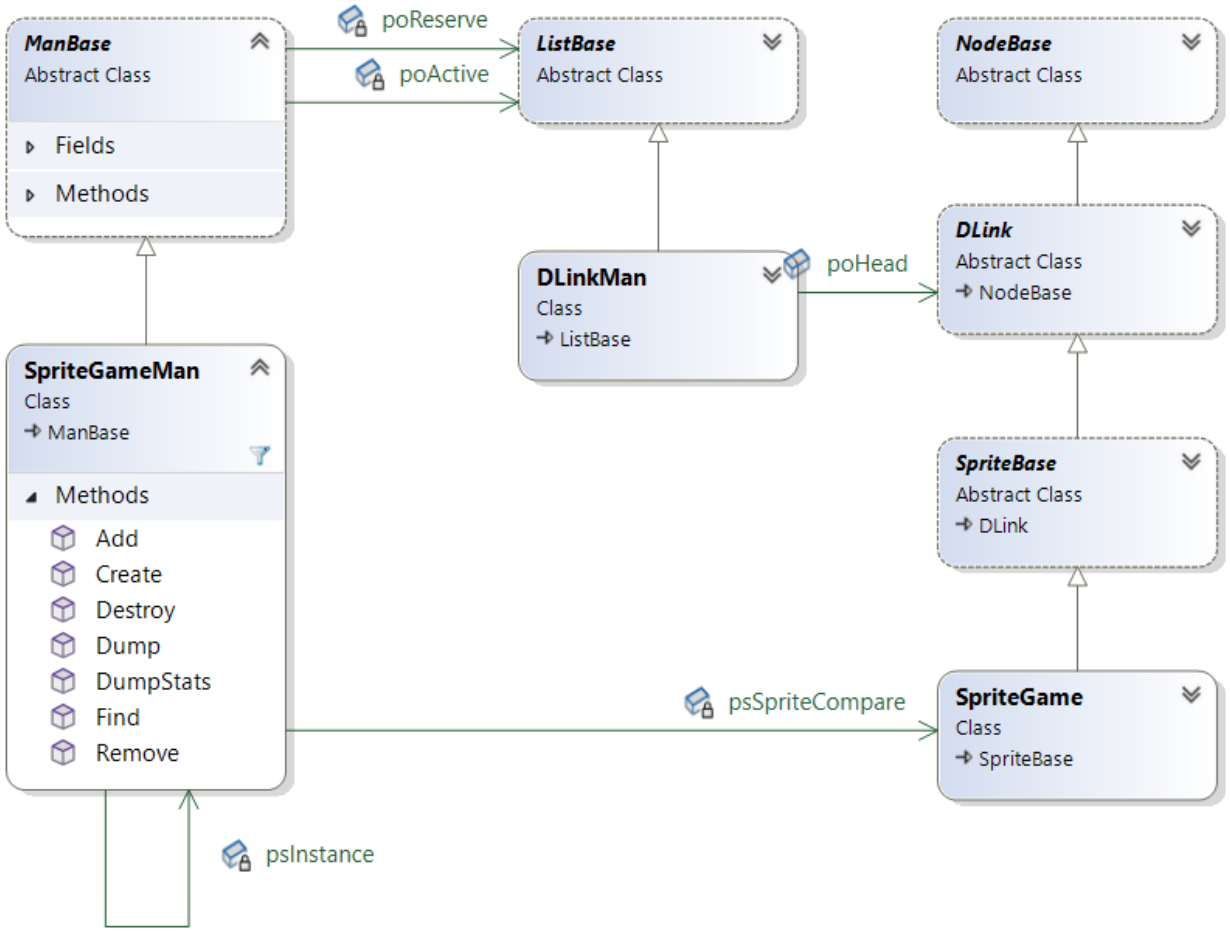
1) Provide a getter for the instance so other classes can access it and operate on it.
2) Provide controlled access for the instance by making it private. Public methods are provided in the singleton class so that the operations on the instance are only handled by the class itself.

The second option is much more secure because it controls access to the singleton instance. By controlling access, we can be sure that other classes aren't doing any harm to the singleton object.

Singleton Pattern Mechanics

To use the singleton pattern, one must create an instance of the singleton object inside the class itself. This instance should only be initialized once. This can occur either in the constructor or through a public ***Create()*** method. If there is any attempt to create the instance again, this will be handled by returning the existing instance. For cases where one might want to provide controlled access to the singleton instance, one must create public methods to operate on the instance. All public methods in the singleton class must always operate on the existing singleton instance.

UML Diagram of Singleton Classes in Space Invaders



Singleton Pattern in Space Invaders

The singleton pattern is used in all the managers in Space Invaders. This was extremely helpful because if we needed to access any sprite, sprite batch, sprite proxy, or box, we can just access it by calling the manager. To protect the usage of the singleton instances, I have made all instances private; therefore, to access an instance, one must call a public method. This allowed for extra safety so that no other class adversely changes the instances when not expected.

Structural Patterns

Proxy Pattern

Problem

In Space Invaders, I had to create 55 different aliens to move together in a grid-like object. The 55 aliens are represented by 3 different type of aliens – a Squid, a Crab, and an Octopus. Each alien is the same except for the location of it in the grid. Instead of creating 55 actual alien objects where the x and y are the only things different, could there be a more optimized way to represent the aliens?

Solution

The intent of the **proxy pattern** is to provide a wrapper around an existing class to allow for an extra level of indirection between the class and the rest of the program. This pattern can be used for the following cases:

1) Protecting a class's access from other classes. E.g. Allowing classes to access certain class members or methods from another class.
2) Creating a "lightweight" version of an object instead of a full copy of an object.

Protecting Class Access from Other Classes

This pattern can act as a "shield" for cases where we need to protect access to a particular class. This this type of use case can occur when we are developing with a 3rd party library. 3rd party libraries may not always contain the safest APIs for our system. By providing a wrapper (a.k.a proxy) class around access to a particular library class, we can ensure that developers will only use the 3rd party library in a safe manner.

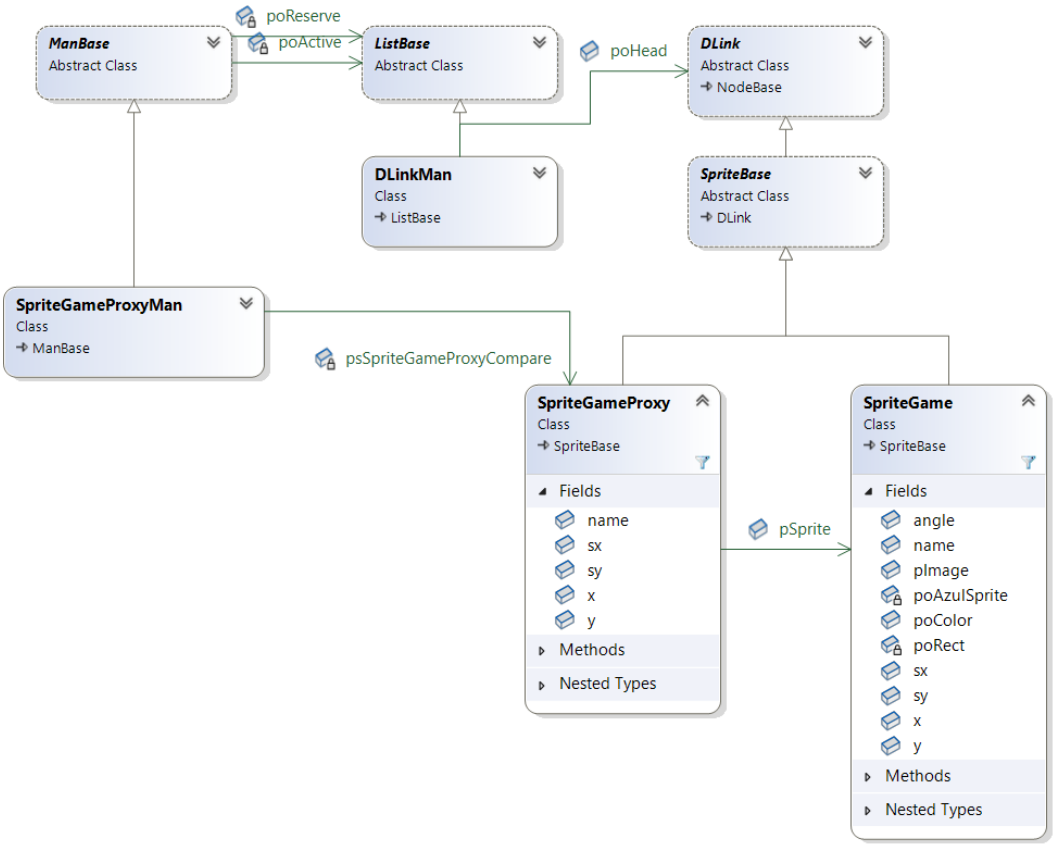
Creating a Lightweight Version of an Object

This pattern also allows us to create lightweight versions of certain objects. This is helpful in situations where we are working with large objects that need to pass around our program. Making objects as lightweight as possible helps with memory consumption and speed. For example, in a financial trading system, there is a concept of a tradeable contract. A contract object is bloated because it contains many fields to represent the specifications of it. Such an object can slow down a system if passed around different components through a network. Creating a lightweight proxy contract object is advantageous so that we can just pass around information that is critical for the system to do its job. A proxy contract class can contain things like symbol and price; while other information about the contract can stay in the contract object itself.

Proxy Pattern Mechanics

To use the proxy pattern, one just must create a “proxy” class with a reference back to the class that it is acting as a proxy for. The proxy class can have different variables which represent the data that just needs to change in reference to the class it is wrapping. The intention of a proxy class is to be lightweight; therefore, it should not contain more members than the class it is acting as a proxy to.

UML Diagram of Sprite Proxy Classes in Space Invaders



Proxy Pattern in Space Invaders

In Space Invaders, I had to create 55 aliens which included 11 squid, 22 crabs, and 22 octopuses. For each alien type, the only difference between them is the x and y coordinates. Otherwise, each squid, crab, and octopus are essentially the same. Therefore, instead of creating 55 different actual instances of the aliens, I was able to create a proxy around each alien (a.k.a. sprite) type. Each proxy class had a direct pointer back to a particular sprite. The only piece of information in each proxy that was different was the x and y coordinates of the alien. This allowed me to represent 55 aliens by just 3 different instances of aliens (sprites).

Composite Pattern

Problem

To facilitate the processing of collisions, I needed a way to group objects together so that we can test whether objects collided with each other. The type of grouping we would need for such a process to occur would have to allow for groups of objects to own other groups of objects. While storing data in existing C# collections would work, we would still need to create our own type of structures to allow for groups of objects to store other groups. Therefore, a more efficient pattern would be needed to satisfy the complexities of what we needed.

Solution

The intent of the **composite pattern** is to compose objects into tree structures to create hierarchies. There are 2 types of objects in this pattern:

1) A Composite class (a.k.a node) – a type of class that holds many other objects which can either be a COMPOSITE node or a LEAF node.
2) A Leaf class (a.k.a. node) – a type of class that cannot hold other composite nodes

Leaf objects are otherwise known as end-leaf objects of a tree; while composite objects are otherwise knows as mid-level or top-level leaf objects of a tree.

A composite tree can be constructed through recursion and there is a one-to-many relationship between the root node and leaf nodes. Clients can interact with composite and leaf nodes uniformly.

Composite Pattern Mechanics

To use the composite pattern, one must create objects that can be categorized as either a composite or leaf node.

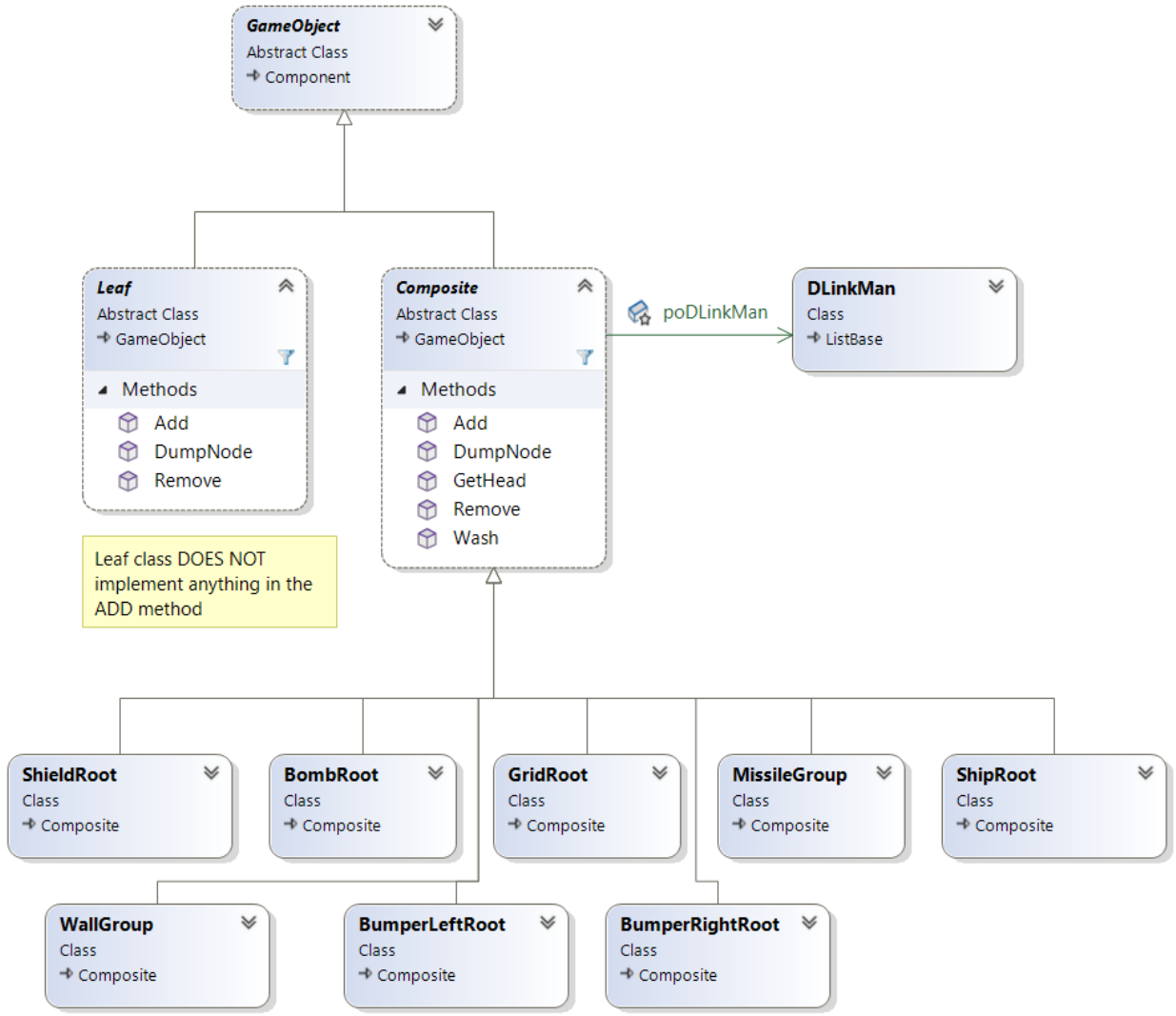
Composite Nodes

Clients can create composites nodes whenever needed. They just need to specify that a particular node is a composite. Composite nodes must allow clients to attach either other composite nodes or leave nodes to it. Therefore, an ***Attach()*** or ***Add()*** method is necessary in this type of node.

Leaf Nodes

Clients can create leave nodes whenever needed. They just need to specify that a particular node is a leaf. Clients cannot attach other leaf or composite nodes to leaf nodes. Leaf nodes are meant to be the end nodes of a tree.

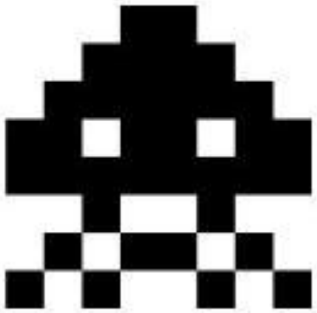
UML Diagram of Composite Pattern in Space Invaders



Composite Pattern in Space Invaders

To best organize all the objects used in the game, it was necessary to create a tree-like structure where each root node is a different type of object used in the game. I created composite roots (a.k.a) trees out of shields, bombs, alien grids (GridRoot above), Missiles, Ships, Walls, and Bumpers. Each type of object contains its own hierarchal tree. This type of organization makes things efficient when processing collisions between 2 different types of objects because all that is needed to do is compare 2 different trees.

Behavioral Patterns



Command Pattern

Problem

I needed a way to encapsulate an action in an object to allow actions to be executed in the game. In some cases, the object who would be executing the action may be unknown during run-time. Therefore, I needed a way to store and make these actions "passable" in the program.

Solution

The intent of the **command pattern** is to encapsulate an action into an object. The requestor and executor of the action do not have to be known during the creation of the command. The requestor and executor of the action also do not have to have a relationship with each other. Furthermore, the executor of the action doesn't need to be involved in the mechanics of the command being executed. This makes things simpler in terms of separation of responsibility and roles.

Command Pattern Mechanics

A series of classes below are needed to use the command pattern.

Command Interface

An interface for the command must be provided. Typically this interface has an ***execute()***and optionally an ***undo()*** method (for cases where an action needs to be un-done).

Receiver Class

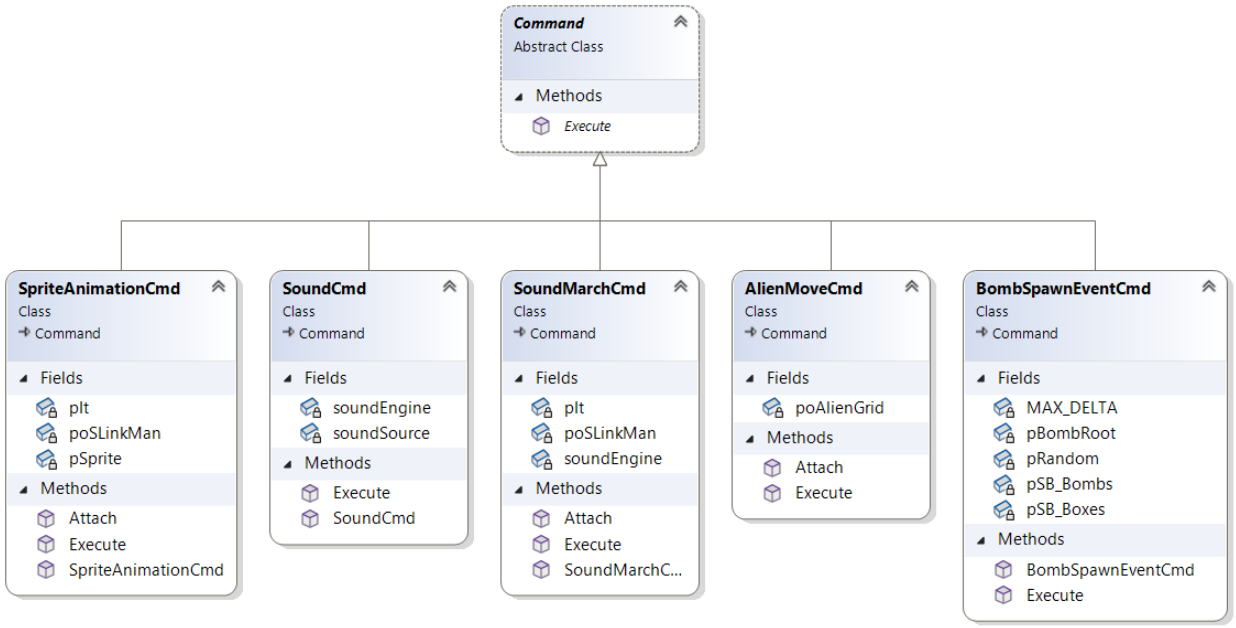
A receiver class is the class or object responsible for doing the work. This class is passed into a command object and it does the work needed when the execute is called on the command class.

Concrete Command Class

A concrete command class implements the Command interface. It must implement the method execute() and optionally the method undo() (if it is defined in the interface). When the execute() method is called on the command class, it sues the receiver object to do the work or action needed.

It is up to the client to determine when to run the command. This can be done by executing the execute() method of the command class.

UML Diagram of Command Pattern in Space Invaders



Command Pattern in Space Invaders

I used the command pattern to encapsulate the execution of the sprite animations, sounds, alien movement, and bomb dropping. All these commands are passed as in as part of a TimerEvent class object which is handled by a TimerEventManager class. The encapsulation of these commands made it easy to pass these into the TimerEventManager class which is responsible for executing events at a specified time.

The TimerEvent class or TimerEventManager did not need to know what was being executed in the commands. Instead, these classes just call the ***Execute()*** method so that commands can run as needed.

Observer Pattern

Problem

In the game there are many action-dependent things that need to be done on a conditional basis. For example, when a missile hits a wall, I need to remove the missile from the screen. When a missile hits a shield, I need to remove a missile and a shield from the screen. Things get a little more complicated when a missile hits an alien. For this action, we need to remove the missile, remove the alien, play a sound, display an alien splat image, update the score, etc.

The work that needs to be done to facilitate collisions between these objects are different depending on the scenario. Therefore, I needed a consistent way to handle what actions need to be done with a collision or state changes in any particular part of our game.

Solution

The intent of the **observer pattern** is to define a one-to-many relationship between objects and actions. When a certain object changes state or when circumstances arise that require a list of things that need to be done, the observer pattern can be used as a "notifier" to other objects about the event. The objects can then do the job it is meant to do. The number of objects to notify in reaction to the event can be different depending on the scenario. In fact, in some cases the observer itself doesn't even have to know the objects it is notifying. This makes it easy to create an observer for any type of scenario we need it for.

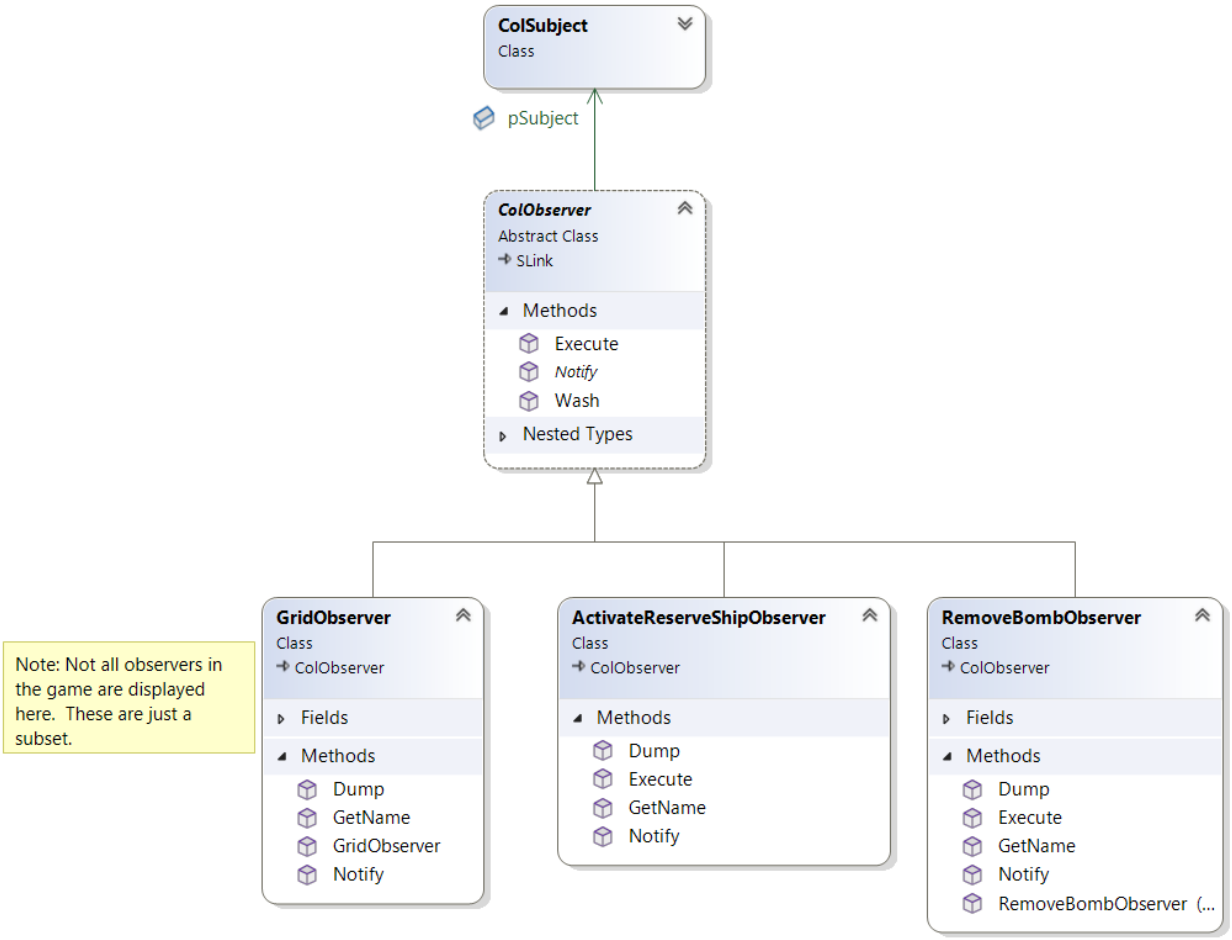
Observer Pattern Mechanics

There are 2 key parts to include to use the observer pattern.

1) Subject – A subject class is responsible for detecting a state change or action in which a list of observers need to be called to do something. The subject needs to call a specific method on a list of observers for notification.
2) Observer – Observer classes are classes which implement a common abstract class in which it must define an action for a specific notify() method. The idea is that when a subject class calls on this notify() method, the action is executed.

Such mechanics of this pattern is helpful for situations where there a *N* number of things that must be done when a change or reaction is detected.

UML Diagram of Observer Object Pattern in Space Invaders



Observer Object Pattern in Space Invaders

I used observers heavily in Space Invaders to deal with the actions that need to be done when collisions occur. When a collision occurs, *N* number of observers are called upon to do a certain set of actions. The subject classes used for this pattern are classes which are defined as part of the ColPair class which is an object that handles the possible collision between 2 GameObject objects. When a collision is detected, the ***Notify()*** method is called from the subject class and then propagated to *N* number of observer classes to do its job. Observers are attached to ColPair (collision pair) objects so it is rather easy to add a new action in response to a collision. This is what makes the observer class a powerful pattern.

Null Object Pattern

Problem

There are cases in the game where I need to pass in parameters for a certain object that is not needed. A prime example is when constructing composite objects like the alien grid, and all game object roots for shields, bombs, UFO, and missiles. For such objects, a sprite (image) is not needed because these types of objects are meant to be containers of other sprites. Therefore, it doesn't make sense to pass sprite games into the creation of these methods. In order to share common code between the handling of a composite (container) object and a leaf object, passing in a sprite (a.k.a image) is needed. Instead of having to explicitly specify null for these parameters, it would be better to pass in a special object so that we don't have to add conditional checks for null all over our code base.

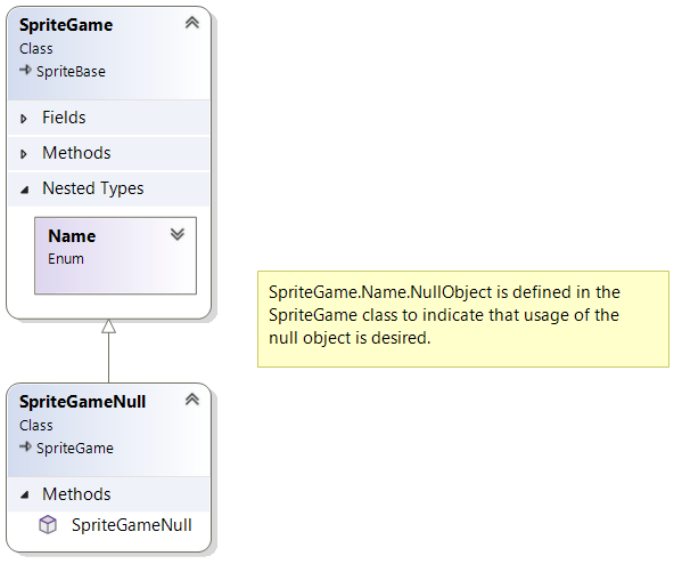
Solution

The intent of the **null object pattern** is to provide a special object that is specifically used for instances where we might otherwise have explicitly specified null in a list of parameters for object creation. The creation of a special null object is to not do anything when methods are called upon it. It allows one to pass in a parameter into a constructor or method without having to add null checks.

Null Object Mechanics

To use the null object pattern, one must create a special null object. The purpose of this object is to handle the absence of an object by implementing the same methods that would do something for non-null objects. The implementation of the methods in a null object are usually no-op operations. There is no implementation needed at all because null objects don't need to do anything.

UML Diagram of Null Object Pattern in Space Invaders



Null Object Pattern in Space Invaders

When creating composite objects in Space Invaders, I passed in a SpriteGame.Name.Null object in the constructor of these objects. Passing in a sprite game in the constructor was necessary for us to use common code when creating composite and leaf objects. By passing in a null object, we avoid the overhead of having to add null checks. When methods on a null object are called, these methods are just no-ops.

For example, in the SpriteGameNullObject class, here are the method definitions for Render() and Update():

```
public override void Render()
{
    // do nothing
}

public override void Update()
{
    // do nothing
}
```


Iterator Pattern

Problem

There are 2 types of data structures that we primarily use in Space Invaders – a Singly Linked List and a Doubly Linked List. This linked list is a custom linked list that was created from scratch. Therefore, it was necessary to define a common interface to iterate over the items in the list.

Solution

The intent of the **iterator pattern** is to provide an abstracted way to iterate through elements of a custom collection without having to expose the underlying implementation of it. The point of this pattern is to separate the responsibilities of the collection class from the code that is iterating over the collection.

Iterator Pattern Mechanics

Iterator Interface

To construct common iterator to use across custom made collections, it is necessary to create an interface which defines a set of methods that would be called on the iterator. Common methods to call are

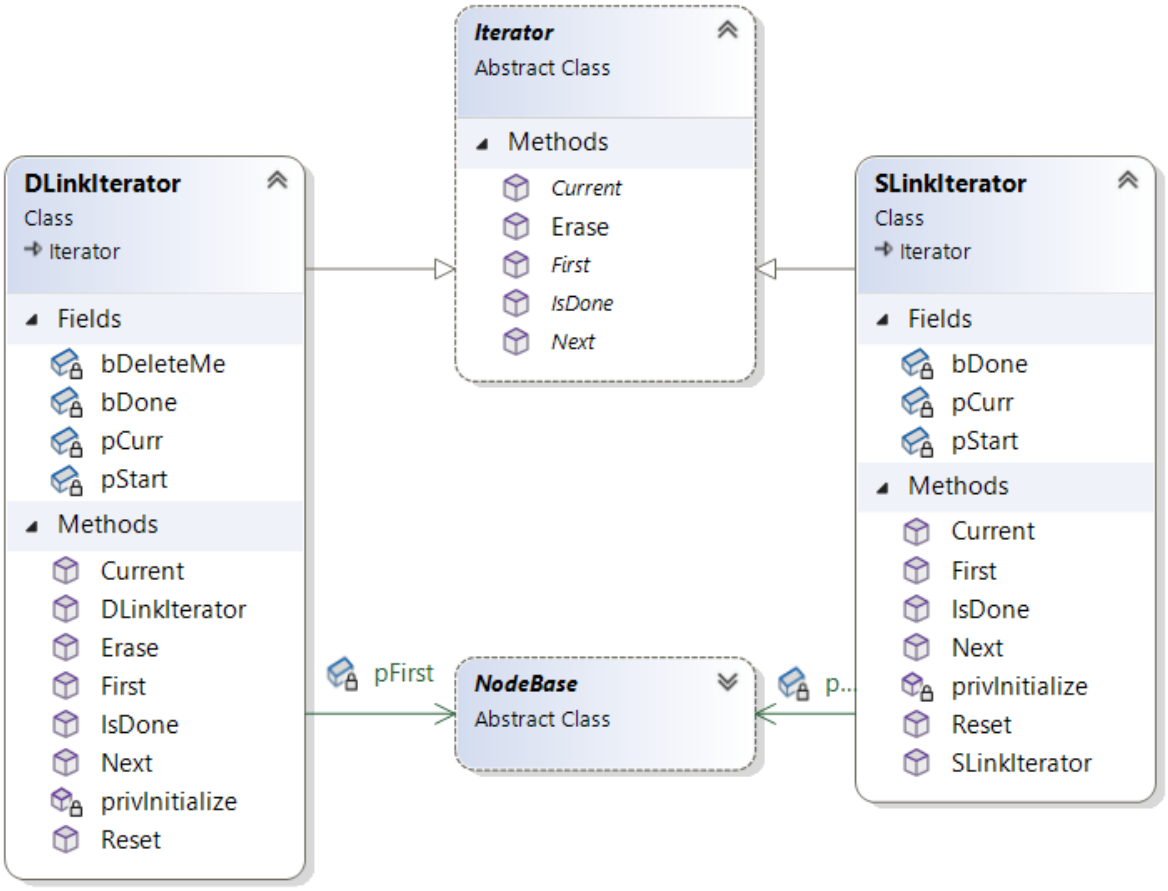
- ***Next()*** -> Retrieves the next item in the list
- ***HasNext()*** or ***IsDone()*** -> returns whether there are more items on the list or not
- ***Current()*** or ***Curr()*** -> Retrieves the current item that the iterator is pointed to

Additional methods apart from these can be added as needed.

Concrete Iterator Class

A concrete iterator class is created to implement the methods defined in the Iterator interface. This class implements the actions that must be taken when the methods in the interface are called. The collection to iterate over must be passed into this class so that this class can use it during the execution of the methods.

UML Diagram of Iterator Object Pattern in Space Invaders



Iterator Object Pattern in Space Invaders

Iterators are used to iterate through objects that are defined to be part of the DLinkMan (A custom double linked list manager) or SLinkMan (A custom singly linked list manager). The head of the collection is passed in so that implementation on the ***Next()***, ***Current()*** and ***IsDone()*** method can be achieved. Having a common way to access elements over the linked lists created for the program was beneficial for the sake of consistency. It also made accessing elements in the collection easy and straightforward.

Strategy Pattern

Problem

One of the challenges I faced when implementing this game was finding a way to drop bombs and move the UFO in different ways. For the bombs, I was required to have them drop in 3 different animated ways. For the UFO, I was required for it to move in 2 different directions. Allowing for behavior to randomly change in any given time was necessary. Therefore, finding optimal way to handle these different behaviors was key in making this program run efficiently.

Solution

The intent of the **strategy pattern** is to encapsulate a family of implementations (a.k.a strategies) to make them interchangeable. The idea is that each strategy can be used passed in or dropped into a method or class and the clients can use it. The implementation of the strategy is independent of the clients that are using it. In other words, the details of how the strategy is implemented is abstracted from the way a client would use it.

One can use this pattern as an alternative to inheritance. The one downside to inheritance is that behavior is defined during compile-time. On the contrary, when using the strategy pattern, it is possible to change behaviors during run-time; giving us more flexibility on behavior modification.

Strategy Pattern Mechanics

To use the strategy pattern, the following types of classes need to be created.

Strategy Interface

A strategy interface contains a single method that executes a particular strategy.

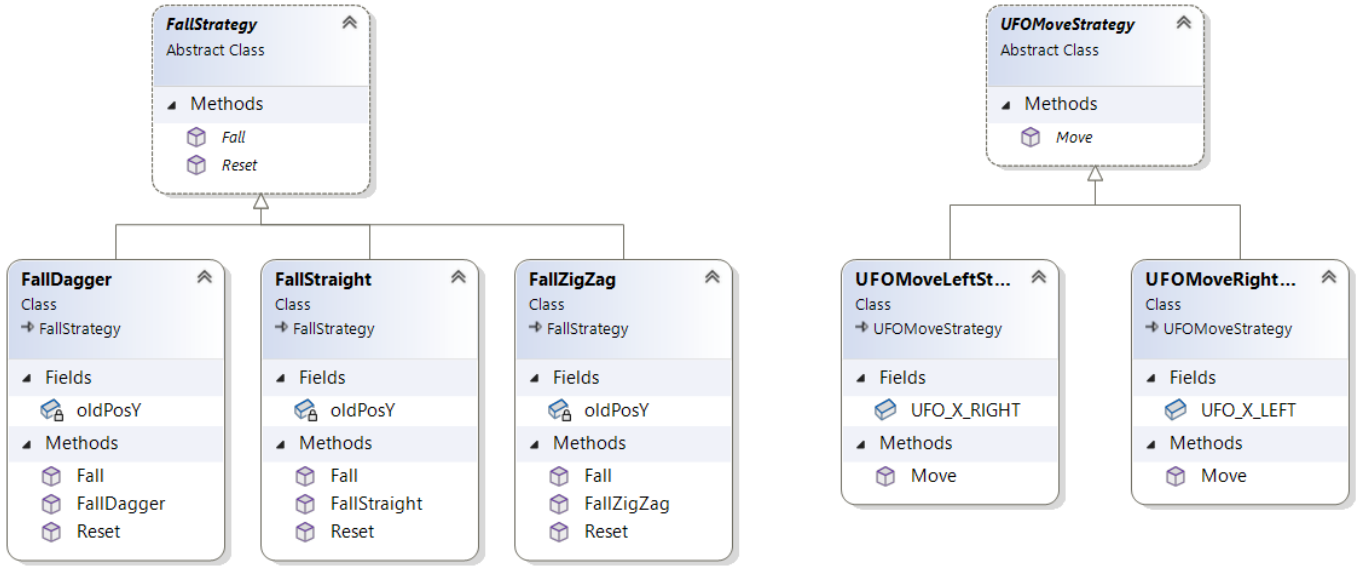
Concrete Strategy Class

A concrete strategy class implements the strategy interface and defines the execution for the method defined in it.

Context Class

A context class is a class that holds a reference to the strategy. Since multiple concrete strategies implement the interface, it is possible for the context class to switch strategies in real-time. To execute the strategy, the context class just needs to call the method defined in the strategy interface.

UML Diagram of Strategy Pattern in Space Invaders



Strategy Pattern in Space Invaders

I used the strategy pattern to tackle the challenges I had with dropping the bomb in 3 different ways. Each concrete strategy class implemented the ***Fall()*** method in a different way. After randomly selecting way to drop the bomb, I passed in the selected strategy to use when I created the timer event to drop the bomb. Similarly, I also used the strategy pattern when executing the UFO logic. I had both concrete strategy classes implement the ***Move()*** method in a different way. When deciding from which side the UFO should enter the game space, I randomly selected either left or right. Depending on what was selected, I passed in the corresponding strategy to the UFO timer event for execution.

State Pattern

Problem

In Space Invaders, there are many instances where objects go through a lifecycle. One example is the lifecycle of the game. There is a select screen, a game play screen, and a game over screen. This cycle represents the game life cycle. Another example is the life cycle of a ship. A ship can be in ready mode, shooting mode, and dead (end) mode. To best represent the states of these objects, there was a need to effectively track and execute game object states to fulfill the requirements of its lifecycle.

Solution

The intent of the **state pattern** is to allow objects to change its behavior in real-time when its state changes. Like the strategy pattern, it allows objects to execute a method differently during run-time. One key difference between the state pattern and the strategy pattern is that the state pattern focuses on changing the actual concrete state object when behavior must change. In the strategy pattern, the object (or context class) that uses the strategy does not change. Only the behaviors of that class change.

Adding additional state behaviors should not alter the behavior of existing ones. In fact, this is one of the most powerful aspects of this pattern. One object may contain *N*-different types of state classes. This allows us to implement different permutations of states that an object may undergo. Otherwise, multiple conditional statements would be required to attain such behavior. This makes the state pattern open to extension but closed for modification.

State Pattern Mechanics

To use the state pattern the following classes must be defined.

Abstract State Class

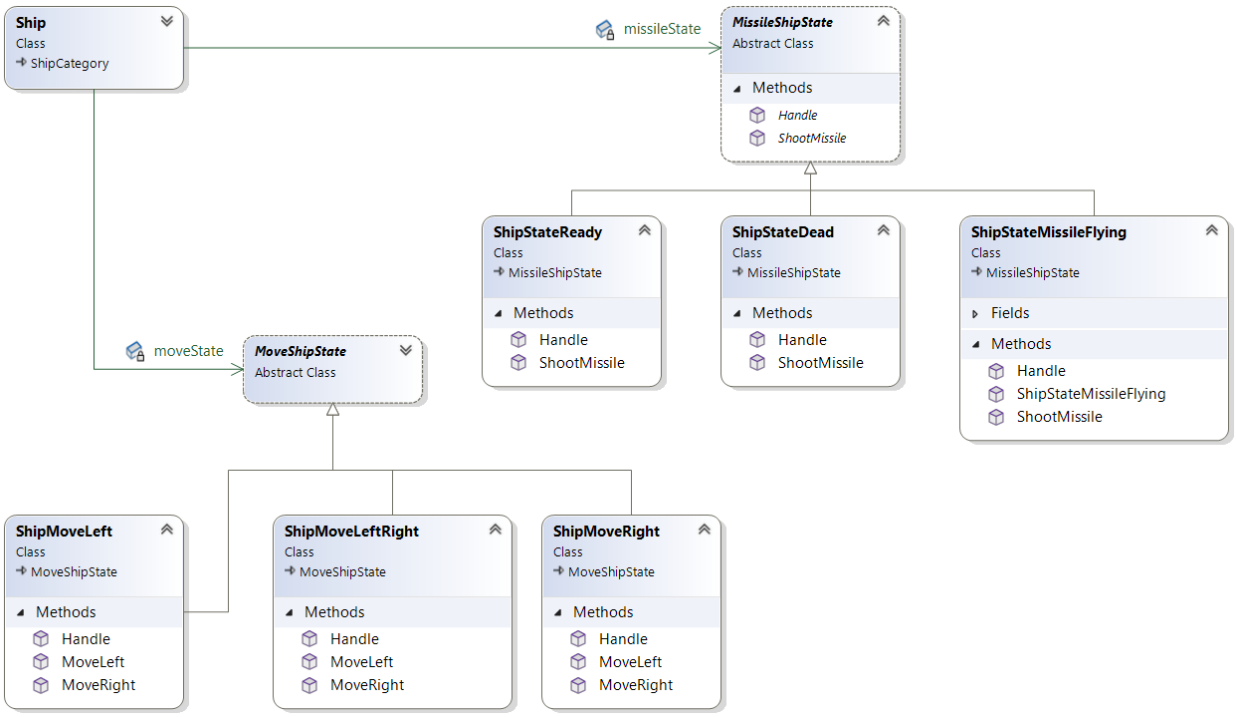
The abstract state class must contain a ***Handle()*** method in addition to other methods that behave differently depending on what the state of the object is. The ***Handle()*** method is a way that an object can change its own state.

It is possible that one object may have more than 1 abstract state class to represent different types of states an object may under-go. For example, if there is a “Person” object. One might implement a “*PersonEat*” abstract state class to represent different behaviors for eating in addition to a “*PersonSleep*” state class to represent different behaviors for sleeping.

Concrete State Class

The concrete state class extends to the abstract state class. Each concrete state class represents a single state of an object. The concrete state class overrides the implementation of all the methods in the abstract state class. Implementation of the methods may be no-ops if an object does not need to implement that method depending on its state.

UML Diagram of State Pattern in Space Invaders



State Pattern in Space Invaders

The state pattern was incredibly useful to use when implementing the different types of states a ship may undergo. For the ship, there are 2 different types of states to consider – 1) Ship Movement and 2) Ship Missile State.

For the ship movement state, I needed to consider the ship's position to determine whether it is allowed to move both ways, only right, or only left. When the ship is touching a bumper, I needed to know which bumper it was touching so that we can prevent the ship from moving beyond the bumpers. Hence, there are cases in the ShipMoveLeft and ShipMoveRight classes where methods are no-ops because we only wanted the ship to move one way.

For the ship missile state, we needed to allow the ship to only shoot 1 missile at a time. Therefore, the ship had to be put in ready, shooting, and end (in the event the ship is bombed) state.

Accounting for the different permutations between the move and shoot state would have been difficult if we did not have a pattern to follow. The code would have been full of conditional statements. By using the state pattern, we found a clean and elegant way to handle different behaviors depending on the state the ship is in.

Visitor Pattern

Problem

Collisions between 2 objects in Space Invaders is a necessity. In fact, without having the ability to process and deal with collisions there would be no game. For that matter, finding a way for each object to interact with each other was necessary.

Solution

The intent of the **visitor pattern** is to represent an action that needs to be performed on *N* different objects. The operational logic of what needs to be performed is taken away from an object and handled by a different class.

Visitor Pattern Mechanics

To implement the visitor pattern, the following types of classes must be implemented.

Element (a.ka. Visitiable) Interface

The element interface contains an ***Accept(Visitor visitor)*** method.

Concrete Element (a.k.a Visitable) Class

Concrete element classes implement the element interface and must provide an implementation for the ***Accept(Visitor visitor)*** method. The idea of this method is for the class to "accept" a visitor for execution on its class.

Visitor Interface

The visitor interface contains a ***Visit(Concrete Element)*** method.

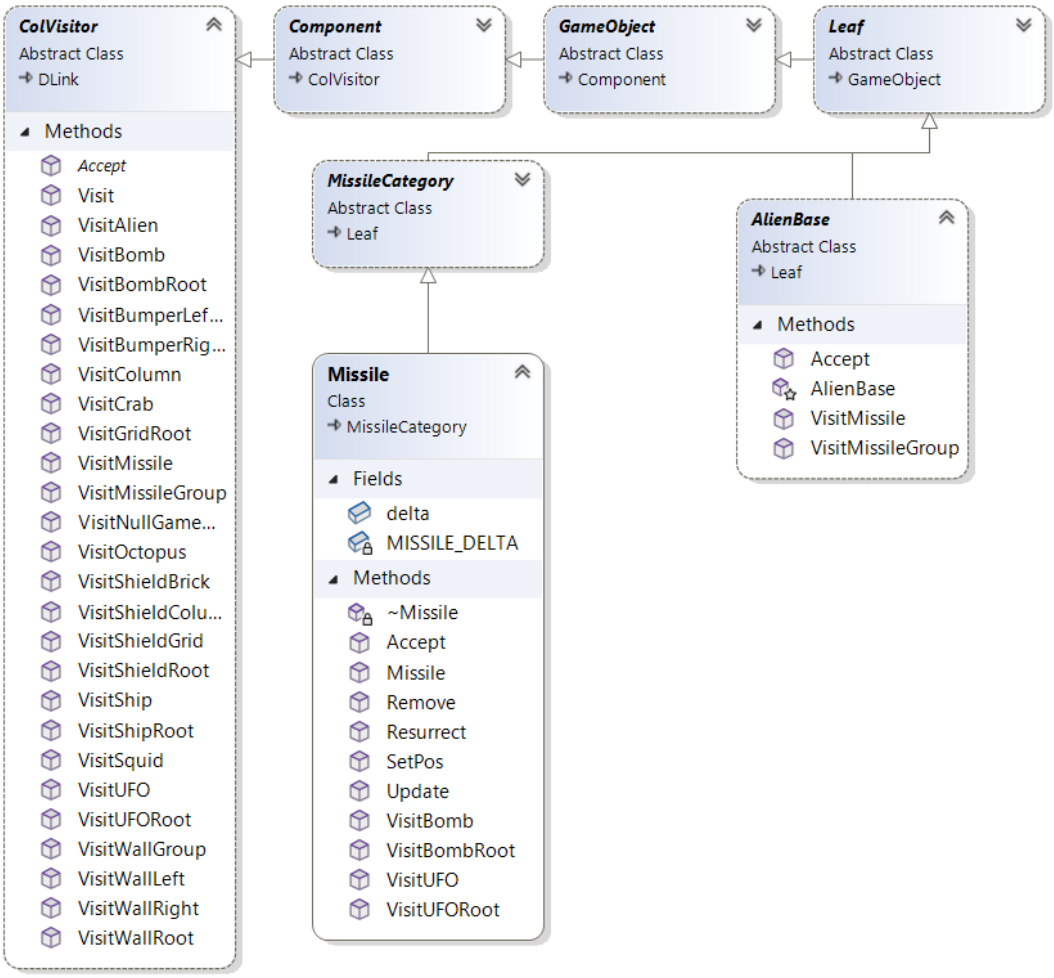
Concrete Visitor Class

The concrete visitor class contains the implementation for the ***Visit(Concrete Element)*** method. This method handles what the visitor must do when it is accepted by the class.

When an element class "accepts" a visitor, it immediately calls the visit function of the visitor. This type of calling back and forth can otherwise be known as "double dispatch".

UML Diagram of Visitor Pattern in Space Invaders

(For simplicity, I am only showing the visitor pattern between Alien and Missile class)



Visitor Pattern in Space Invaders

The visitor pattern is used to process all collisions between game objects in the game. When two objects collide, one object must “visit” the other object. The other object then executes its own ***Visit(Element)*** method on the object that just accepted it. Depending on which object is visiting each other, the collision method is either called again on the children of the object or, if the object is visiting a leaf node, then actions to process the collisions are done. In the case of Space Invaders, we go ahead and call the observers of the collided objects to process the collision.

Using the visitor pattern allows us to easily add additional behavior to two classes interacting with each other. The alternative to implement this would have been to add a lot of conditionals or switch statements which would have rendered our program not scalable or extensible.