SPACE INVADERS

Design Document

Submitted by,
Ann Joy

INDEX

DESIGN PATTERNS

SINGLETON PATTERN

Challenge:

On implementing the LifeMan, a class that maintains the lives of the player in Space Invaders, the functionalities in the class were required at various locations in the program, and they all needed to access a single instance of the class. If multiple instances of the class were to be created then the multiple LifeMans will exist which could then manipulate the data. For example, it could update the remaining lives in different instances and would not be reflected in the instance that is being checked for remaining lives and hence manipulate the data.

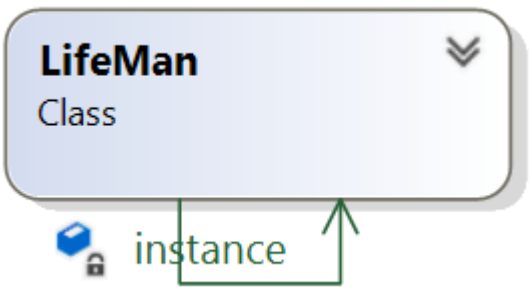
Solution:

A class that can only hold one instance of itself with static functions that can be accessed from various points of the program.

Pattern Description:

The singleton pattern is a creational pattern. This pattern makes sure that only one instance of a class is created. This is done by adding a static variable of itself in the class. The created static variable provides a global access point to its instance. With the global access in place, all functionalities of the class access the single instance.

UML Diagram:



Key Object-Oriented Mechanics:

In the project, the LifeMan class was created with a static private instance of itself. Also, all the variables associated with it are set to private so that only the object of the class can access them in the instance that is stored in the class. Similarly, all the functions that are externally accessed are static and the internal functions are all set to private for protection.

In a scenario, if the life variable of the LifeMan is to be updated, the client must invoke a static function of the LifeMan. This static function has to then fetch the private instance of the class. Then the required change is done to that instance.

In the project, to get the instance inside the singleton class a private function, privInstance() is used. Inside the function, the instance is checked before returning it. If the instance is null, the instance is set to a new object of the class, else the existing instance is returned.
This makes sure that the instance need not be created at one particular point in the program and will be set the first time the singleton is invoked.

ITERATOR PATTERN

Challenge:

In the project, to keep track of various types of objects by various managers, the objects were inherited from a DLink or SLink node. This provided the object with pointers to either both the previous and next node or just the next node. With this structure in place, to traverse through a list of nodes, double or single link list, the internal pointers inherited from DLink or SLink had to be accessed. This broke the protection of internal values of variables in the top hierarchy.

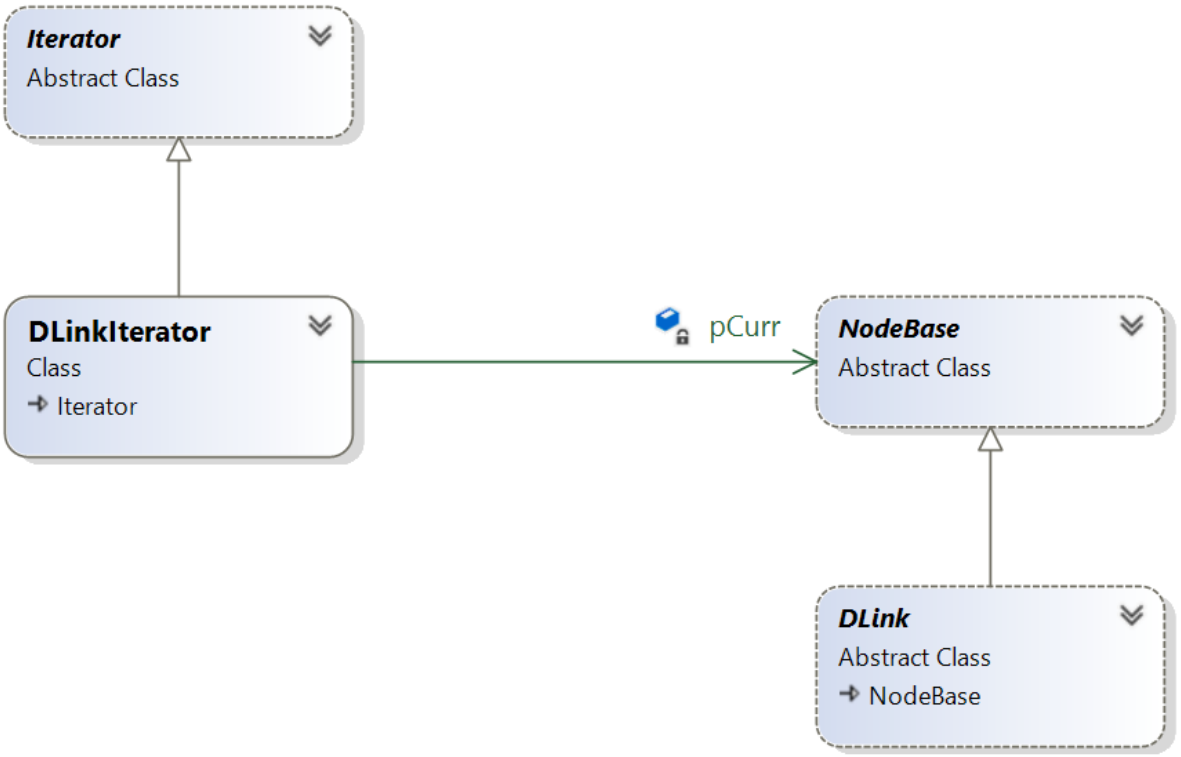
Solution:

A component is responsible for accessing the internal or underlying values of a node to traverse through the collection of nodes while not exposing the node's internal structure and providing a uniform interface.

Pattern Description:

The iterator pattern is a behavioral pattern. It extracts the traversal behavior of a collection into a separate object called an iterator. Along with it, the pattern also encapsulates internal details.
Structurally, the pattern has an iterator interface, a collection interface, a concrete iterator, and a concrete collection. The iterator interface declares the required functions for traversal, the collection interface declares the getting iterators compatible with the collection, and the concrete classes implement them.

UML Diagram:



Key Object-Oriented Mechanics:

In the project, an abstract iterator class was implemented to declare the required functions such as Current, Erase, First, IsDone, and Next. These functions are implemented in the concrete class, DLinkIterator.

The class DLink stored the variables pointing to the next and previous elements of the list. Now with this implementation, the client can use DLinkIterator to iterate through the list without exposing the internal structure of DLink.

NULL OBJECT PATTERN

Challenge:

When creating various GameObjects, some do not require a SpriteGame attached to them. This leads to adding conditionals in various parts of the project to handle when the SpriteGame is null.

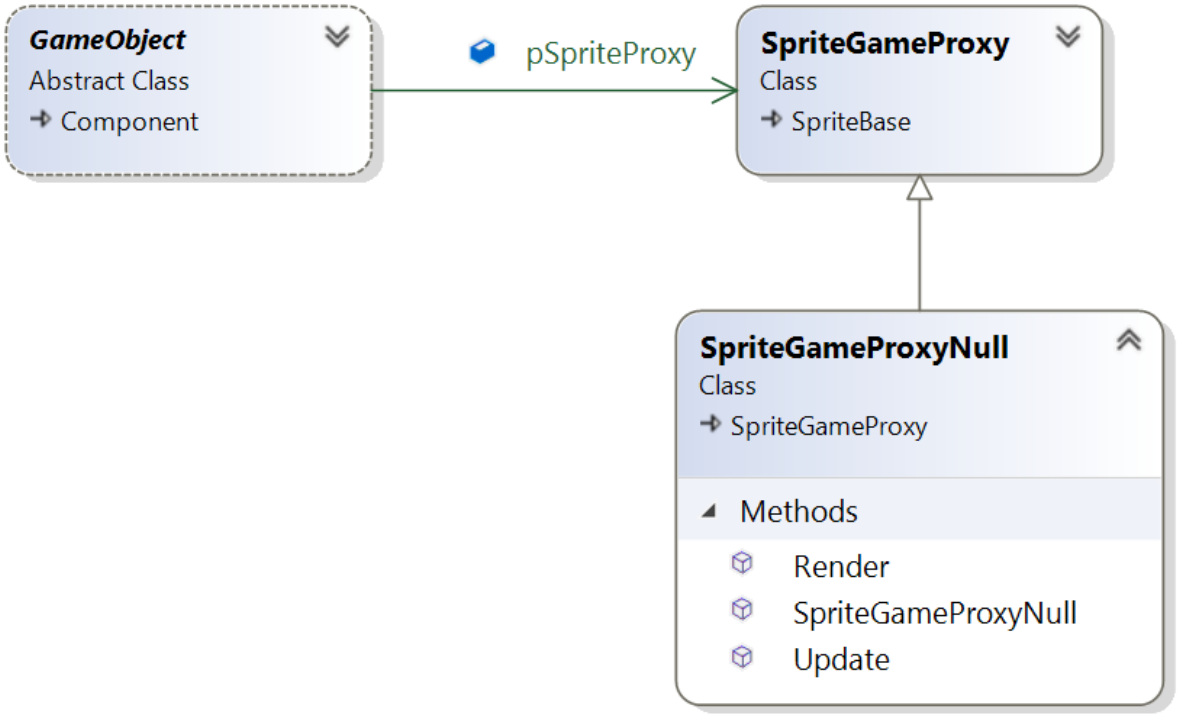
Solution:

A concrete class that uses the interface required by the actual concrete class that holds a SpriteGame but has a do-nothing implementation of the class and hence can be used seamlessly without a null check.

Pattern Description:

To implement the null object design pattern, an abstract class is required that declares all the behaviors of the real object required. The real object and the null object are then made to inherit from this abstract class so that the same behaviors are to be defined in both. Then in the null object, all the behaviors do nothing, thus providing the same result as that of having a null check. This helps the code be cleaner and less error-prone.

UML Diagram:



Key Object-Oriented Mechanics:

In the project, the GameObject class points to SpriteGameProxy. SpriteGameProxy has various behaviors to update the SpriteGame that it holds. Here, a null object, SpriteGameProxyNull was created by inheriting from the SpriteGameProxy. So in effect, the SpriteGameProxyNull implements all behaviors on SpriteGameProxy.These behaviors are then overwritten in SpriteGameProxyNull with no implementation to get the null object behavior, that is, to do nothing.

FACTORY PATTERN

Challenge:

To create the alien grid in the game, multiple lines of the same code have to write for each alien creation. Thus to generate 5 X 11 aliens, the number of lines repeated was high and the logistics were exposed.

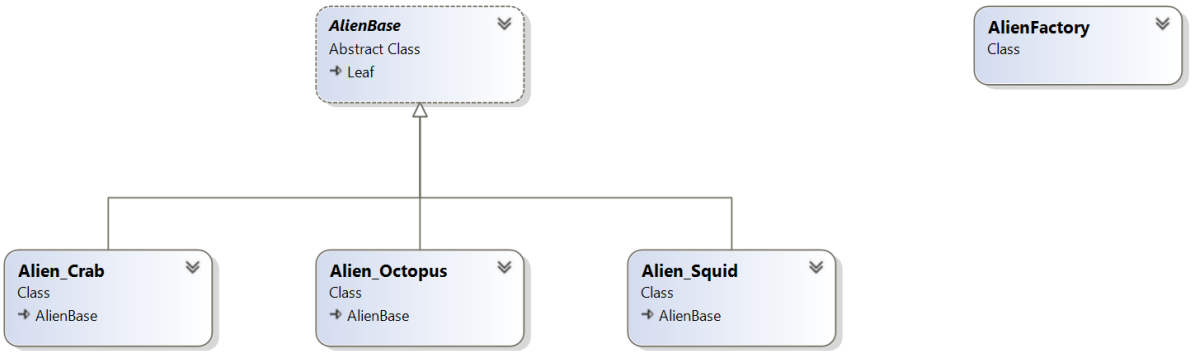
Solution:

A class that generates the required alien at the required location through a common interface to give cleaner code.

Pattern Description:

The factory pattern is a creational pattern that returns the required object for the provided information.

UML Diagram:



Key Object-Oriented Mechanics:

In the project, the client asks the AlienFactory to create the required alien by providing it with the information, alien

type, and position. The alien factory instantiates a new product and returns it to the client after casting it to AlienBase. This was the client can create different types of aliens all through AlienFactory and recast the result into the desired type.

PROXY PATTERN

Challenge:

The game contained multiple copies of the same resource, SpriteGame that help a required sprite. The same sprite was required by various game objects, this increased the number of copies of the same data and hence used too much memory.

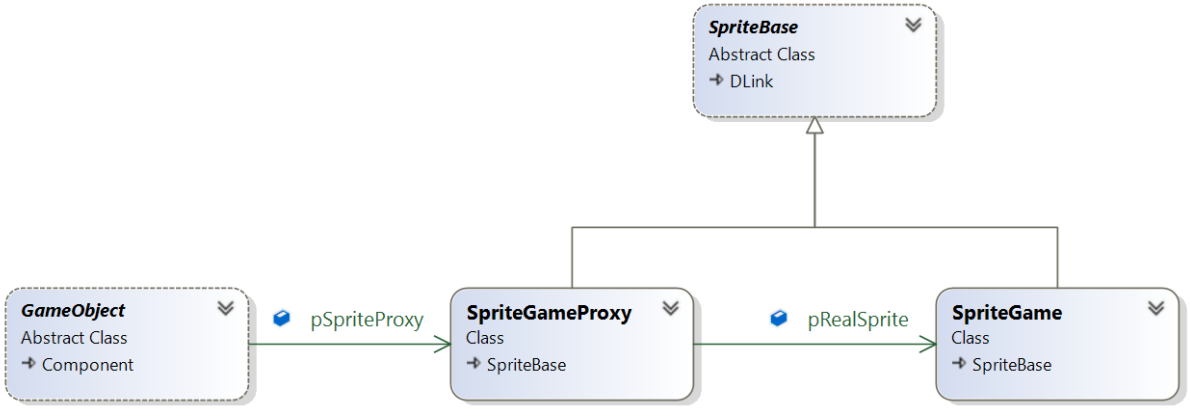
Solution:

A lightweight object that holds the required data in one memory location to which multiple objects can point to with the same behavior as that of the real object, in this case, the SpriteGame.

Pattern Description:

The proxy pattern is a structural design pattern. This pattern is mainly used to create an intermediary object between two objects, the client object and the real object. With the proxy object in between, the system can have control over the requests. The obtained control can be used to filter, cache, and log requests. The proxy pattern has 3 major components: the real service object, the service interface, and the proxy object. For the client, to use the real service they must go through the proxy object. The requests sent into proxy are processed and then send to the real service provider.

UML Diagram:



Key Object-Oriented Mechanics:

In the project, the real object is the SpriteGame. SpriteGame stores the sprites, and variables controlling the sprite, and Azul Sprite. The intermediary object is the SpriteGameProxy. SpriteGameProxy is very lightweight, it only stores the minimum variables required to manipulate the sprite and pointer to the SpriteGame. Both SpriteGame and SpriteGameProx inherit from the same SpriteBase interface. This allows any client to treat SpriteGameProxy just like SpriteGame. The client in this scenario is the GameObject. The GameObject points to the SpriteGameProxy and not to the SpriteGame.

With the proxy pattern implementation, GameObjects in need of sprites point to a SpriteGameProxy object and multiple SpriteGameProxy objects point to one SpriteGame. This solves the issue of too much memory usage as the sprites are only loaded once. In the case of the sprite for Alien Crab, one SpriteGame exists that stores the loaded sprite, and every game object that uses the sprite Alien Crab, points to the same SpriteGame.

With this update, the GameObjects update the variables to manipulate the sprite in the SpriteGameProxy.This includes position, scale, and color. These variables still need to reach the SpriteGame, and this was implemented in SpriteGameProxy. As both SpriteGame and SpriteGameProxy implement the same interface, the GameObject treats SpriteGameProxy just like SpriteGame.So all requests go through the SpriteGameProxy. So when the Update() is invoked, the SpriteGameProxy pushes the variables stored to the SpriteGame. This is done before rendering the SpriteGame and hence the GameObject required changes are updated in the sprite.

STATE PATTERN

Challenge:

The player ship to have different behaviors according to the current state of the machine without many conditional checks.

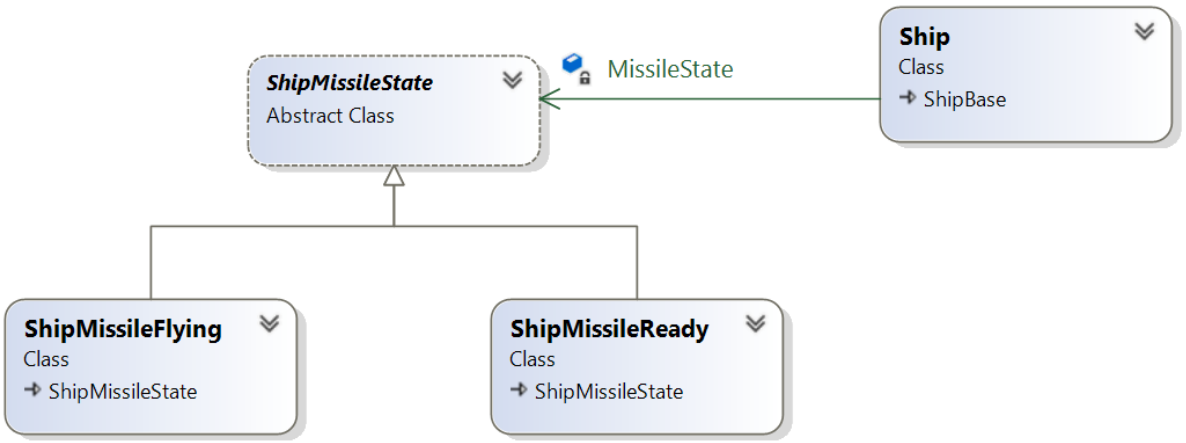
Solution:

Dedicated classes with logic for each state of the ship.

Pattern Description:

A state pattern is a behavioral design pattern used to change the behavior of a class based on its internal state. To implement the pattern, a context class, a state interface, and concrete state classes are required. The context class maintains references to concrete state objects which may be used to define the current state of objects. The state defines the interface for concrete states.

UML Diagram:



Key Object-Oriented Mechanics:

In the project, the ship points to a ShipMissileState abstract class that holds definitions for ShootMissile(). When the ShootMissile() is invoked in the Ship, the Ship invokes the ShootMissile() of the

MissileState it points to and the behavior of the state is then executed. The states are controlled in the Ship and can be changed to get different behaviors. Thus, getting the required state changes without multiple conditionals.

ADAPTOR PATTERN

Challenge:

To integrate an external interface into the project to bridge the gap.

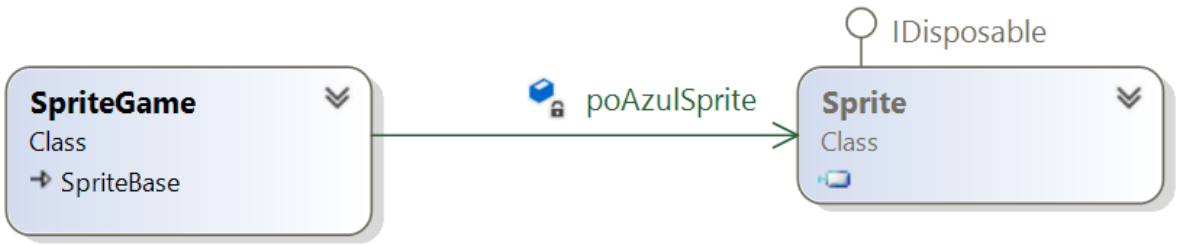
Solution:

An object that converts the interface of the external object so that the client can use it.

Pattern Description:

The adapter pattern is a structural design pattern. To implement the adapter pattern, a class adapter is created which lets classes work together. The pattern consists of an adapter and an adaptee. The adaptee delegates all the requests that it gets to the adaptee in the required format.

UML Diagram:



Key Object-Oriented Mechanics:

In the project, the adapter pattern was used to implement the Azul.Sprites. Here the class SpriteGame is the adapter. All requests that come to the SpriteGame are delegated to Azul.Sprite.
The client only needs to use SpriteGame and does not need to be aware of the requirements in Azul.Sprite.

OBSERVER PATTERN

Challenge:

When collisions occur, certain sets of behaviors need to be added to various collisions. For example, For collision between the Player ship and a bomb, two behaviors need to be added, one to destroy the bomb and the other to destroy the ship and update the Life

Solution:

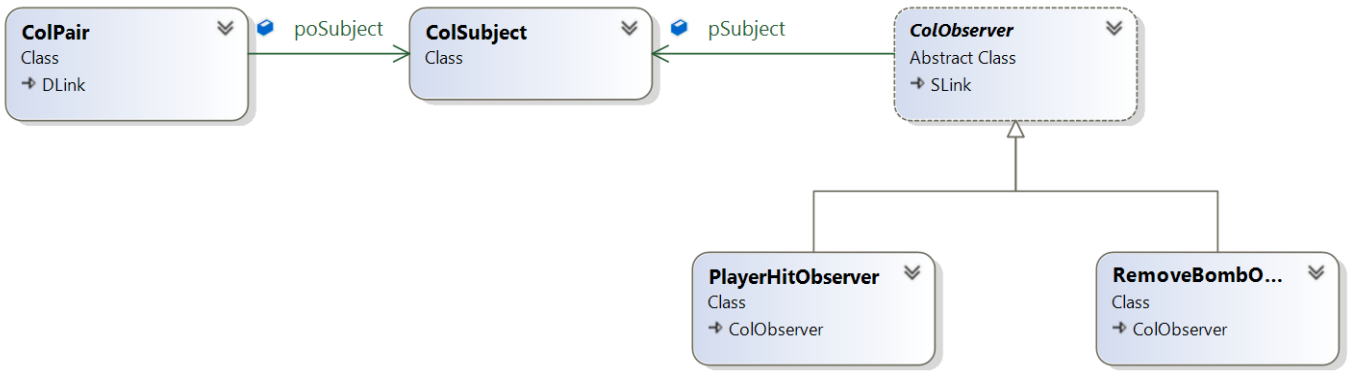
A one-to-many dependency between objects such that when a state change is triggered, all the depending objects are notified automatically.

Pattern Description:

The observer pattern is a behavioral design pattern. It helps define a subscription mechanism to notify more than one observing object when a state change occurs.
An observer pattern contains an observable class, a concrete observable class, an observer class, and multiple concrete observers. The observable class implements an interface that defines how to add and remove observers. The concrete observable maintains the state, when the state is changed it traverses through all observers and notifies them. The observer class defines an interface that declares behaviors the observable class uses to notify the

observers, and finally, the concrete observers implement various behaviors as required

UML Diagram:



Key Object-Oriented Mechanics:

In the project, the PlayerHitObserver and the RemoveBombObservers are two Concrete Observables that inherited from the ColObserver, which is the Observer. In the ColObserver, the abstract behaviors are declared that implement the operations required to notify the Observers, such as Notify() and execute(). The ColPair holds various ColObserver which are notified when the ColSubject has a change of state. The ColPair is the Concrete Observable, and the change of state is the detection of a collision between 2 poSubject's.

When the collision is detected in the ColPair, the ColPair notifies the poSubject, the ColSubject. The ColSubject holds a list of all the Observers attached, iterates through each of them, and invokes their Notify(). Thus, all the observers attached to the ColSubject, in this case, PlayerHitObserver and RemoveBombObserver are notified of the collision, and required behaviors are executed.

COMMAND PATTERN

Challenge:

To execute a set of actions after a timed delay in a user-defined order.

Solution:

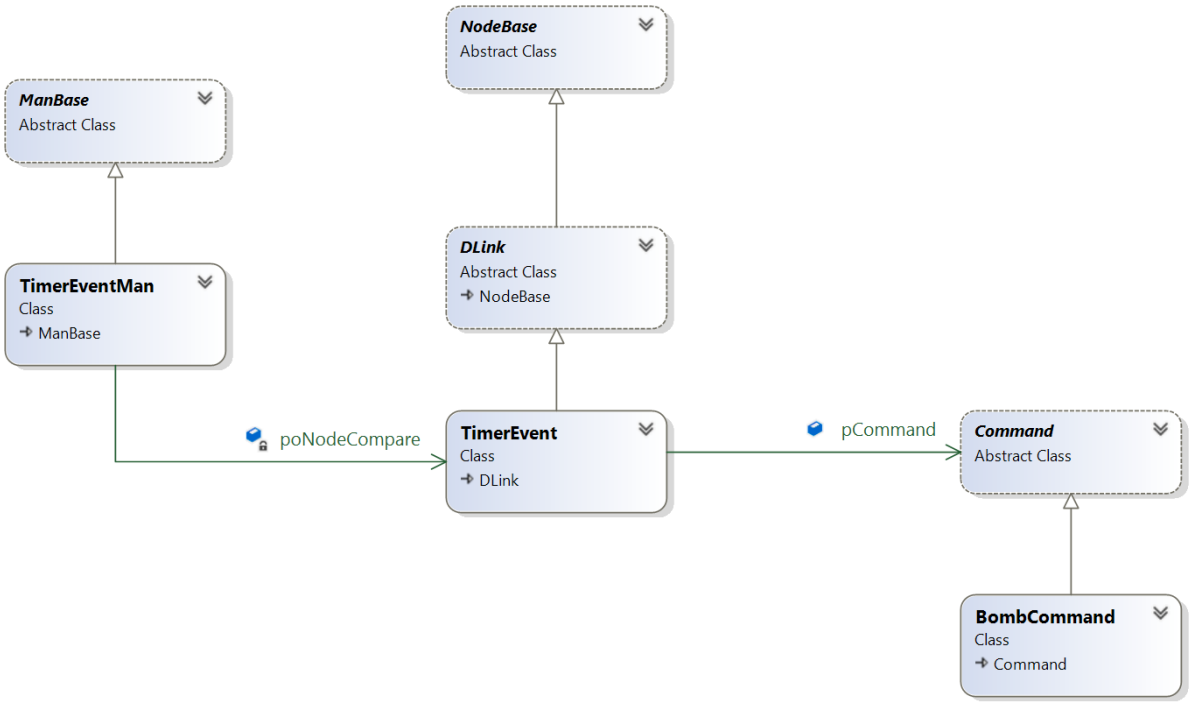
An object that can hold multiple objects of required behaviors in a collection, which then triggers the required behavior from the collection of objects in a user-defined sequence.

Pattern Description:

The command pattern is a Behavioural pattern. It consists of an Invoker class, a Command interface, and concrete commands. The invoker class maintains the collection of commands and iterates through it in a specific sequence and invokes the behavior in attached commands when prompted to. The common interface declares the required behaviors in a command, all concrete command patterns inherit from the command interface and so, implement the same behaviors.

When the client prompts from command executions, the invoker takes the command and places it in a queue, then when the invoker executes each command from the queue, it invokes the defined behavior in the Command interface, which invokes the required behavior to be executed.

UML Diagram:



Key Object-Oriented Mechanics:

In the project, the TimerEventMan and the TimerEvent together are the invokers. The TimerEventMan stores a collection of TimerEvent. Each TimerEvent points to a command that is to be executed. The TimerEvent class also stores the time at which the commands are to be executed.

The Command interface defines Execute(), this function will hold different implementations for different concrete classes.

When a command is to be executed at a certain point in time, from the client's point of view, a concrete command is created which has the required behavior. The new concrete class is then added to the TimerEvenMan along with the time at which the command is to invoked. The TimerEventMan updates the time requirements to a new TimerEvent class and attached the concrete command to the TimerEvent class. Then the TimerEvent is added to a queue stored in the TimeEventMan. The TimerEventMan is updated with the current time in every game cycle. In this update, the TimerEventMan calls Execute() for TimerEvents commands if the time has matched. This then executes the

required behavior and later removes the used TimerEvent from the queue.

STRATEGY PATTERN

Challenge:

To implement different behaviors in how the bomb sprite moves down for the Bomb class without the use of many conditionals.

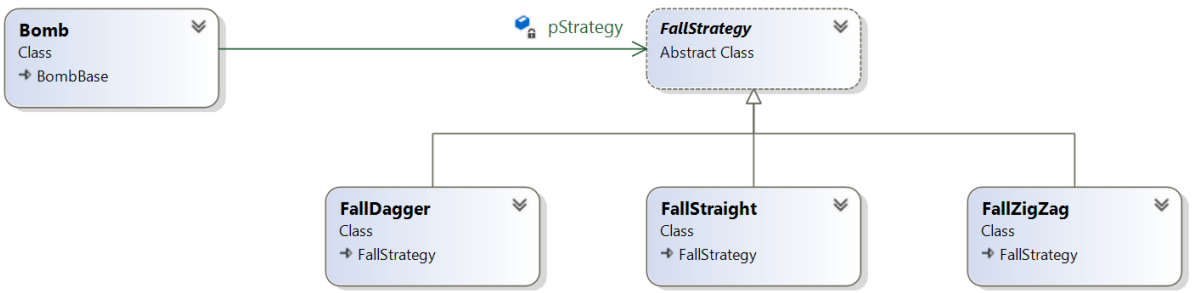
Solution:

To have separate classes for each behavior that are easily interchangeable for different objects.

Pattern Description:

The Strategy pattern, is a behavioral design pattern. To implement this pattern is to extract the algorithm of a class into separate classes called strategies. The original class context will delegate work to the extracted classes called strategy objects, which will execute the desired behavior. The abstract strategy class will declare the required behaviors which are defined in the concrete strategy classes.

UML Diagram:



Key Object-Oriented Mechanics:

In the project, the Bomb class contains a reference to the strategy object, which is set to require concrete strategy classes when the objects are created. In this case, FallDagger, FallStraight, or FallZigZag. This was implemented to get different behaviors for different missiles while it is moved down the screen. When creating a Bomb object, the required strategy is set while instantiating the object. With this implementation, when a function is invoked that required the behaviour of one of the strategies, the class accesses the already set pointer , and invokes its function. Thus, a conditional was not required to check the type of bomb it was, rather the strategy was already set to its pointer, which updated the behavior.