Space Invaders Design Documentation

Andrew Richards

3/17/2020

DESIGN OVERVIEW

Space Invaders is built on top of a custom game engine provide by DePaul University called Azul. The game itself is written in C# while the engine is written in C++. Azul provides a base class called Azul.Game which loops in real-time. A class called "Space Invaders" inherits from this base class and overrides the Initialize, Load Content, Update, and Draw methods. The game consists of multiple components built using fifteen design patterns.

The Initialize method creates an OpenGL window through the Azul engine. The derived definition of this method in the Space Invaders class sets the properties of the window. LoadContent uses Azul to load content files from the directory and initializes the components built in the game. Update and Draw loop in real-time and are called each frame.

Load Content is called once when the game is first launched. It begins by constructing the managers for the different components. The managers are static and derived from an abstract manager class and used to manage collections of objects. All the collections are in the form of a linked list written from scratch using references to link nodes. There are no generic collections used in the game.

After the managers are constructed, a Texture Manager loads in a texture file in TGA format. It manages a linked list of Texture nodes which serve as wrappers for an engine class called Azul.Texture. Azul.Texture is responsible for accessing and storing the file from the directory.

A Sound Manager then loads in WAV files from the directory to be used by other components. This manager is a wrapper for the IrrKlang sound engine which is an external dependency. The sound manager manages Sound nodes which hold references to the WAV files. When a sound needs to be played, the manager will search for the node by name and call the Play2D method of the IrrKlang engine.

An Image Manager then creates images out of the textures in the texture file. Each Image node consists of a name and pixel coordinates for a specific image in the texture file. This breaks up one large texture file into multiple smaller textures called Images. The Images are added to a linked list of images.

Similar to an Image Manager, the Glyph Manager creates smaller textures out of a large texture file using pixel coordinates. The Glyph Manager is used for images which represent alphabetical, numerical, and special characters. In addition to the attributes of an Image Node, a Glyph node holds an int which represents the ASCII decimal representation of a Char. This allows for a Font node to take in a String, convert each char to a byte, find the corresponding Glyph, and display it in the form of a sprite. This sprite is called a Font Sprite and is made up of glyphs which represent each character of the String. Font nodes are managed by a Font Manager.

Once the Image nodes are added to their collection, they are used by a Game Sprite Manager to create Game Sprite Nodes. Game Sprite are the objects which will be rendered in the draw loop of the game. They set the scale and color of an image.

These Managers are loaded first because they are constant for every scene of the game cycle. Managers that are unique to each scene are cycled through in a static context depending on the current scene. The game cycles through five scene state classes. There is a title scene, a select player scene, two player game scenes, and a game over scene. These are initialized next through the construction of a Scene Context class at the end of the LoadContent method. This class constructs and owns a reference to each scene. When each scene class is constructed, it calls its own Initialize method similar to the load content method of the Space Invaders class. This constructs managers which manage collections unique to that scene. The Update and Draw loop of the Space Invaders class call the Update and Draw loop of the scene that is currently active.

The Initialize method for the player game scenes constructs a Sprite Batch Manager for managing collections of Sprite Batches. A Sprite Batch is a collection of game sprites. They are used to draw a collection of Game Sprites on the screen with one call rather than through individual calls.

A Game Object Manager is constructed next which manages collections of Game Object composites. Game Objects are objects that hold location data and own a reference to a sprite and to a collision object. A Collision Object owns a reference to a Collision Rect and a Box Sprite. Collision Rects are used to calculate an intersection with the Collision Rect of another game object. Each frame there is a check to see if an intersection between any two collision pairs has occurred. A Box Sprite is a sprite that is used to visualize the collision boundary of a game object. It can be toggled on and off for debugging purposes. Each frame in the update method, the Game object manager updates its location data and references.

Collision pairs are explicitly defined through a Collision Pair class. Collision Pairs hold references to two Game Objects and are stored in a collection managed by a Collision Pair Manager. This manager iterates through the collection and processes the collision checks each frame.

A Timer Manager is used for handling timed events in the game. It owns a collection of Timer Events which store a float representing a trigger time. Trigger times are the total amount of time that has passed at the time the node is added plus the delta for the frequency that an event occurs. When the Update method is called each frame, the Timer Manager iterates the collection and triggers each event node whose trigger time is less than the amount of real-time that has passed. The animation of the alien movement is accomplished through a Timer Event. Every time the timer event is fired off, the alien sprite flips the image associated with it to another image texture. Then it adds itself back as a new timer event and flips the image texture again.

Input from the user is handled through an Input Manager. The Azul engine can detect when input is received. The Input Manager wraps the detection of keys that are relevant to the game. The manager is unique to each scene so that the same keys can be reused to perform different actions. The space key is used in the main menu to cycle to the next scene while it is also used in the game scenes to fire the ship's missile.

After the Timer and Input managers are constructed in the player game scenes, the Fonts specific to each scene are created and added to the Font Manager. A String representation of the message to be displayed and a location are given as arguments. This is used to display the scores and the number of player lives remaining. The Fonts are updated each frame in the update method to dynamically display the change in the score and number of lives.

The player game scenes contain a Game Object composite which represents wall boundaries. They are Game Objects that own location data and references to a Collision Object. They do not contain a sprite that is rendered in game. These walls detect collision on all sides so that moving Game Objects do not exceed the boundary of the game window. This ensures the player ship, the player missile, the aliens, and the alien bombs all remain within the playable area of the game.

The player ship and missile are constructed during the construction of a Ship Manager. This manager owns these two references and does not manage a collection. The aliens, UFO, and shield Game Object composites are then constructed at the end of the Initialize method of the player game scene. They are then added to the Game Object Manager and Collision Pair Manager with the appropriate collision relationships.

Both player game scenes construct all their own instances of the scene-specific managers and Game Objects. These are unique to each player scene so that each scene can preserve their own state when the players switch turns. When cycling the to a new scene, the transition method of the new active scene is called which swaps out the scene-specific managers. Once the Sprite Batch Managers are swapped out, only the sprites of the current scene are rendered. Only the managers of the active scene are updated in the update method. This pauses the gameplay of a scene when it is not active.

The main menu scene, player select scene, and game over scene only display fonts and static sprites. They only have an Input Manger and Sprite Batch Manger Manager which are unique to them.

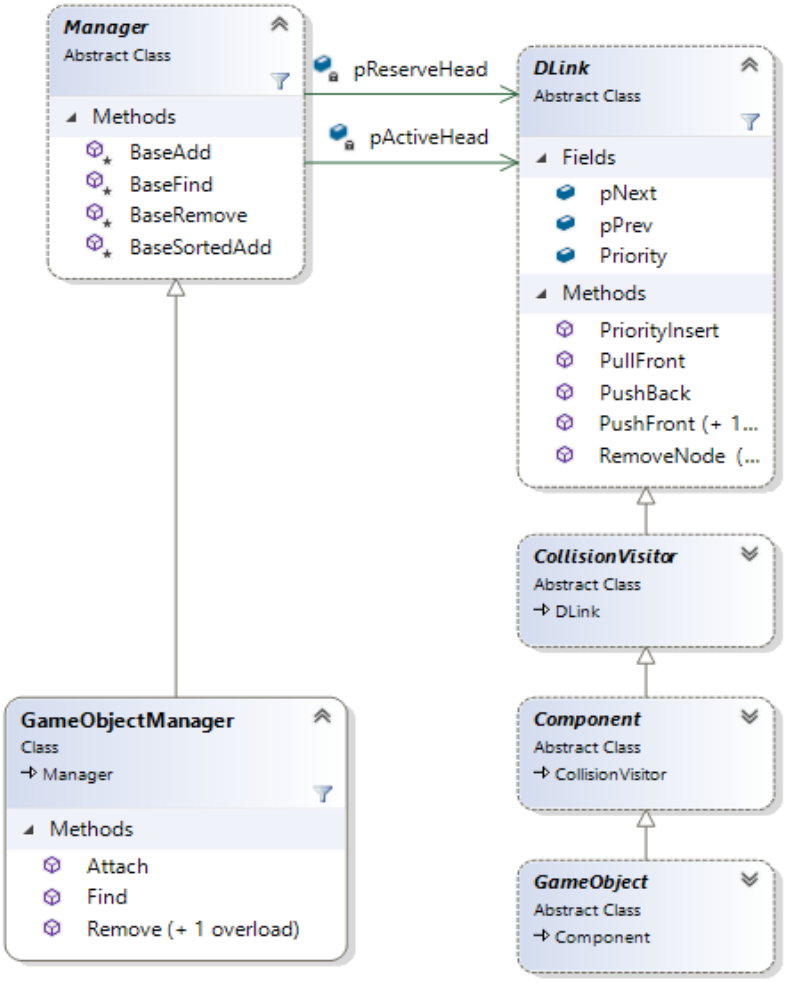
When a new game is started, the player game scenes are cycled to for the first time and their Reset All methods are called. This resets all the Game Objects back to their starting locations and reactivates any Game Objects that were removed during previous game play. Once Game Objects are initially constructed, they are never deleted and reconstructed. They are reused each game cycle in order to optimize performance.

COMPONENT DESIGN

OBJECT POOLING

Real-time systems built with a managed language like C# face the issue of automatic garbage collection. It is important to ensure inactive objects that will be needed later are not deleted. It is also very expensive to call large quantities of constructors to create new objects while the game is running. Constructors should be called during initialization and never called during gameplay. Re-using objects can be achieved through an Object Pooling design pattern.

This pattern allows for a client to pull an object from a pool of reserve objects instead of constructing a new object. Instead of leaving the object to be destroyed by the garbage collector, it is washed and returned to the reserve pool once it is no longer needed. This removes the need to construct and destroy objects while the game is running. The reserve pool is constructed when the program is first initialized.



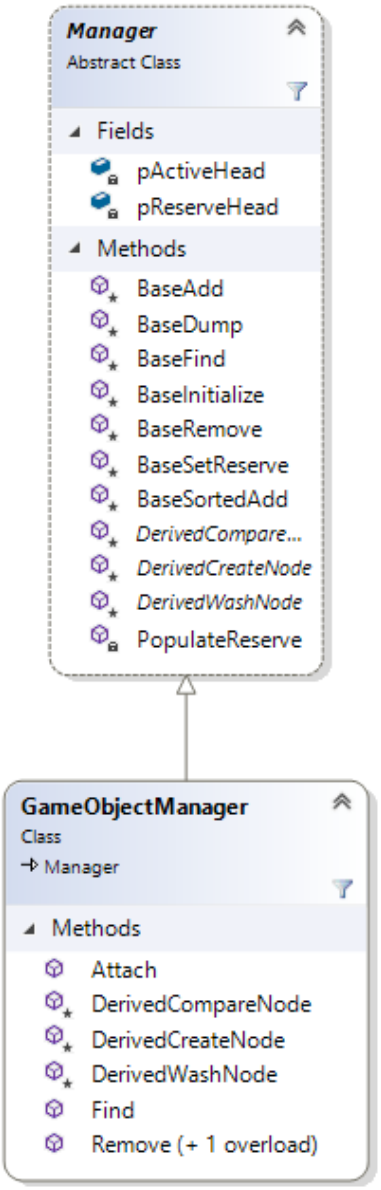
The Object Pool pattern is used by the managers in Space Invaders. There is an Abstract base class called Manager which owns two references to another abstract base class called DLink. DLink represents a node of a linked list while Manager handles adding, removing, and searching for a DLink on a linked list. Every object type that belongs to a collection, such as a Game Object or a Game Sprite, inherits from the DLink class. Every manager that manages a collection inherits from the Manager base class. The Manager base class has methods for adding, removing, and finding DLink nodes which are used by the derived managers.

The Managers hold references to the head node of two linked lists. One of them is an active object linked list and the other is a reserve object linked list. When a new object is needed, a Dlink is pulled off the reserve list and added to the active list. It is then down casted to the type of the derived DLink class in its corresponding manager. This downcast is acceptable since it only occurs in a manager specific to one type. By deriving from the Manager and DLink abstract classes, this linked list pooling structure can now be reused for as many object types as the game requires.

TEMPLATES

The functionality for adding to the active list, removing from the active list, and searching an active list needs to be unique for each manager so that DLinks can be downcast to different types. The functionality for initially creating the reserve DLinks, comparing DLinks during searches, and washing DLinks that are returned to the reserve, does not need to be redefined for each manager. In order to simplify the abstraction of the managers, a Template Pattern is used.

This pattern occurs when an abstract class has an abstract method that it forces a concrete derived class to override. That abstract method is then called in a concrete method of the abstract base class. The concrete derived class calls the concrete method of the abstract base class which uses the derived class's overridden definition of the abstract method.

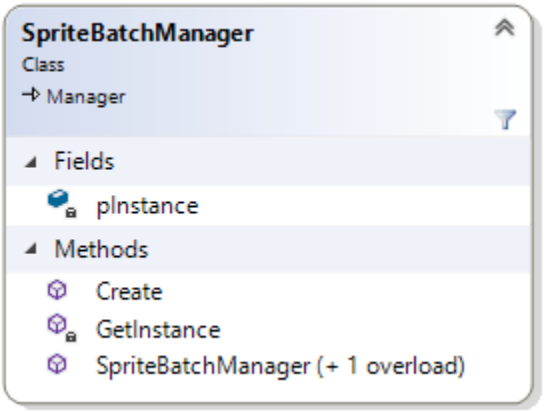


The derived managers, such as the Game Sprite Manager, use this pattern by calling the Base Initialize, Base Add, Base Remove, and Base Find methods of the abstract manager base class. These base methods call the abstract methods Derived Create Node, Derived Compare Node, and Derived Wash Node which have overridden definitions in the derived managers. Each overridden definition downcasts a DLink node to the corresponding type for that manager.

SINGLETONS

Since managers handle collections of a one specific type, there should only be a single instance of each type of manager. The Singleton pattern is used to ensure only one instance of each manager exists.

A Singleton pattern occurs when a class owns a reference to an instance of itself. The constructor for the class is private and called once through a public Create method. This method checks to see if the instance of the class is null. If it is, then the constructor is called. If it is not null, then it does nothing. A Get Instance method is used to access this single instance of the class.

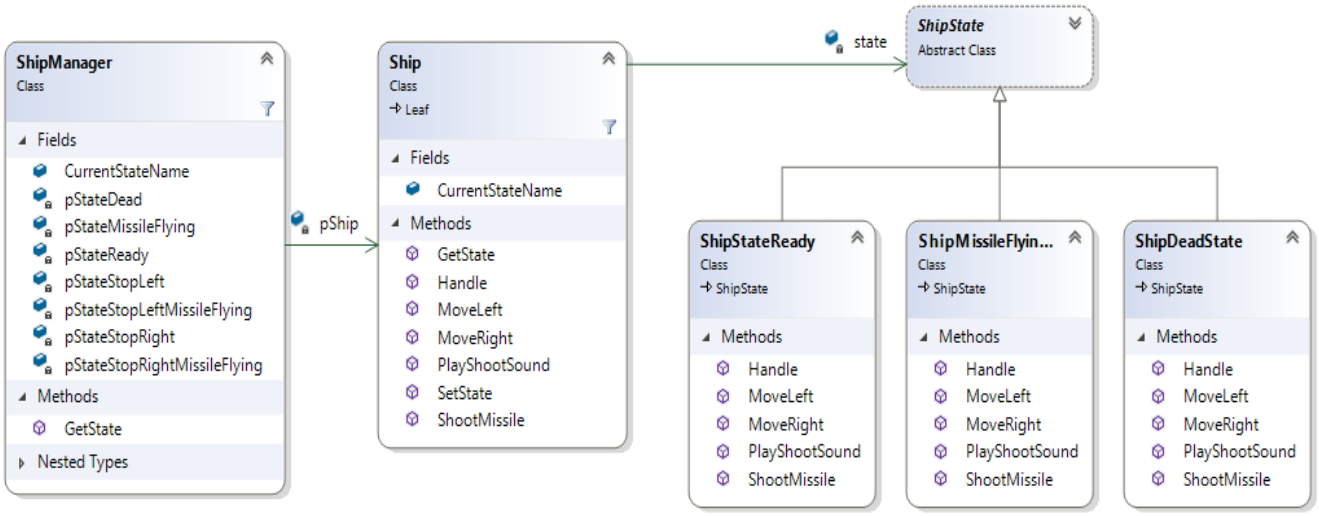


When the Load Content of the Space Invaders class is called. All the managers call their Create methods in a static context. Since the references to their instances of themselves are initialized to null, their private constructors are called. If create is accidentally called again, their constructors will not be called again. They are created in a static context since there is only a single instance. The player ship and ship missile are also singletons. There is only one instance of each which is used by both players.

STATES

The player ship has the ability to fire a missile and move side to side when it receives input from the user. There are times when it should not be allowed to perform all of these actions. When the ship is at the left or right boundary of the screen, it should not be able to continue to move left or right. When the ship fires a missile, it should not be allowed to fire another missile until the first missile is destroyed. In order to handle these conditions. A state pattern is used to limit functionality.

The State pattern uses an abstract base class called State with an abstract method called Handle. Multiple concrete states are derived from this abstract class with their own implementations of the Handle method. Another class called State Context owns a reference to the concrete states and has the responsibility of switching between them. It has a Set State method which sets a specified state as the current state. This set method is called in each overridden implementation of the Handle method. It allows one state to transition to another specified state.



When the Shoot Missile method of Ship is called, it calls the Shoot Missile method of the state that is currently active. Each state has a different implementation of the method. If the active state is the Ship Ready State, then the Shoot Missile method will fire a missile. When the missile is fired, the Handle method of the Ship Ready State is called which transitions the active state to the Missile Flying State. The implementation of the Shoot Missile method in the Missile Flying State does nothing which prevents a second missile from being fired. Once the missile is destroyed, the Handle method of the Missile Flying State is called which transitions the current state back to the Ship Ready State.

There are additional states for when the ship collides with the left and right boundary walls, and for when the ship fires a missile while colliding with the left and right boundary walls. These additional states are required to prevent the ship from moving too far off the screen while simultaneously preventing the ship from firing two missiles at once.

The State pattern is also used for the scenes of the game cycle and the bombs dropped by aliens. The scenes of the game cycle manage which scene specific managers are currently active including the Sprite Batch Manager which controls what is being rendered. Similar to the

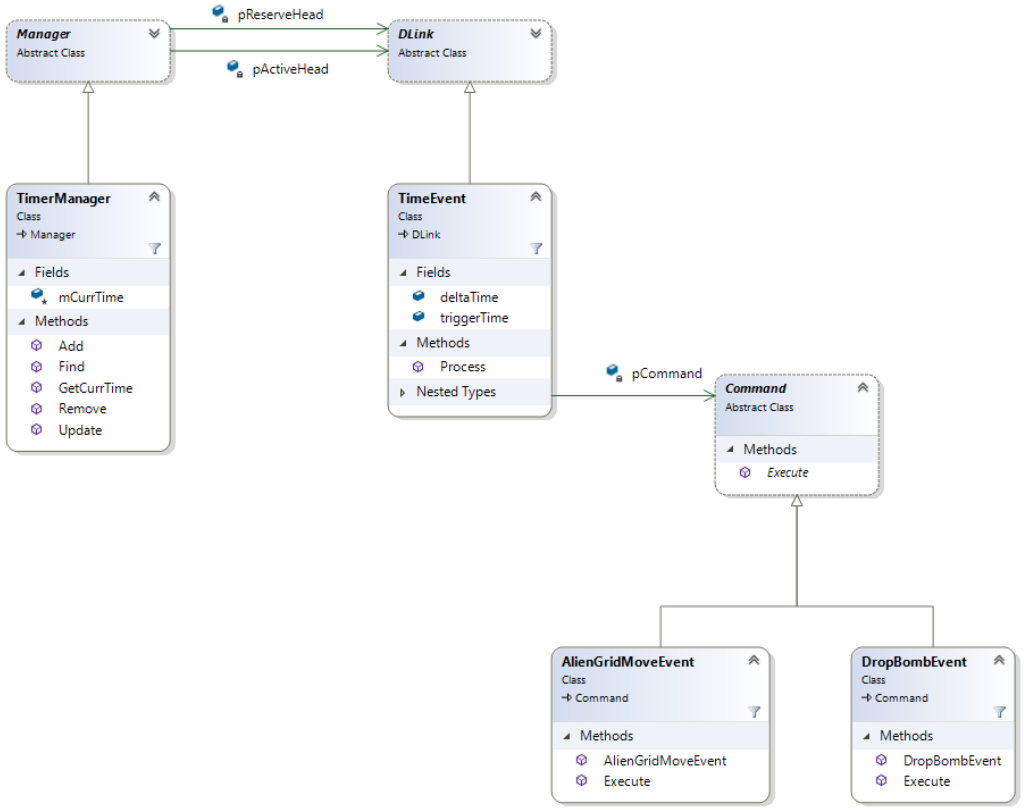
Ship states, the Bomb states prevent more than one Bomb from dropping at once. A new Bomb is not activated until the previous Bomb is destroyed.

COMMANDS

Events take place in Space Invaders that need to be based on time. There needs to be a system for managing the frequency that events occur such as moving aliens, playing animations, dropping bombs, triggering sounds, and spawning UFOs. These all may have a different frequency which they occur.

This is accomplished with a sorted collection of events that are fired off based on the amount of real-time that passes. A priority queue in the form of a linked list is used to manage the events. The collection will remain sorted by inserting new events based on their delta time. Each frame the list will be iterated while triggering events if their delta time is less than the current time passed. The iteration will stop once the current time exceeds the trigger time of the current event node in order to prevent unnecessary iteration. Once an event is triggered, it will be removed from the list and inserted again based on its frequency.

The triggering of events is done through the use of a Command pattern. This allows for the ability trigger multiple types of events with only one implementation. There is one Time Event type in the timer collection that holds a reference to multiple types of events. Calling the abstract method in Time Event can fire off any overridden derived Execute method from any type of event.



This pattern works by having one Abstract base class called Command with an abstract Execute method. Any class derived from this base class will override the Execute method with its own implementation.

Another class holds a variable of the base class Command type which stores a reference to a class derived from Command. When the abstract Execute method of the Command base class variable is called, it will call the overridden Execute method of the derived class being referenced regardless of its type. Its type does not matter since it is derived from the Command class and the variable storing the reference is of the base class type.

In Space Invaders, the Time Event class is a node in the priority queue for triggering timer events. This class has an aggregate relationship with the abstract Command base class and holds a reference to it called pCommand. When a Time Event object is constructed, a reference to an event such as Alien Grid Move Event or Drop Bomb Event is stored in the pCommand variable.

The Timer Manager handles the management of the collection. It is used to add, remove, and iterate through Timer Events of the collection. The Update method of the Timer Manager is called each frame in the main update loop of the game. Timer Manager's update method iterates through the priority queue. If a Time Event node has a trigger time that is less than the delta time that has passed, the Time Event node calls its Process" method. The Time Event node is then removed from the collection and reinserted back into the priority queue based on its trigger time. Once there are no more nodes with a trigger time less than the delta that has time passed, the iteration loop is broken out of in order to prevent unnecessary looping.

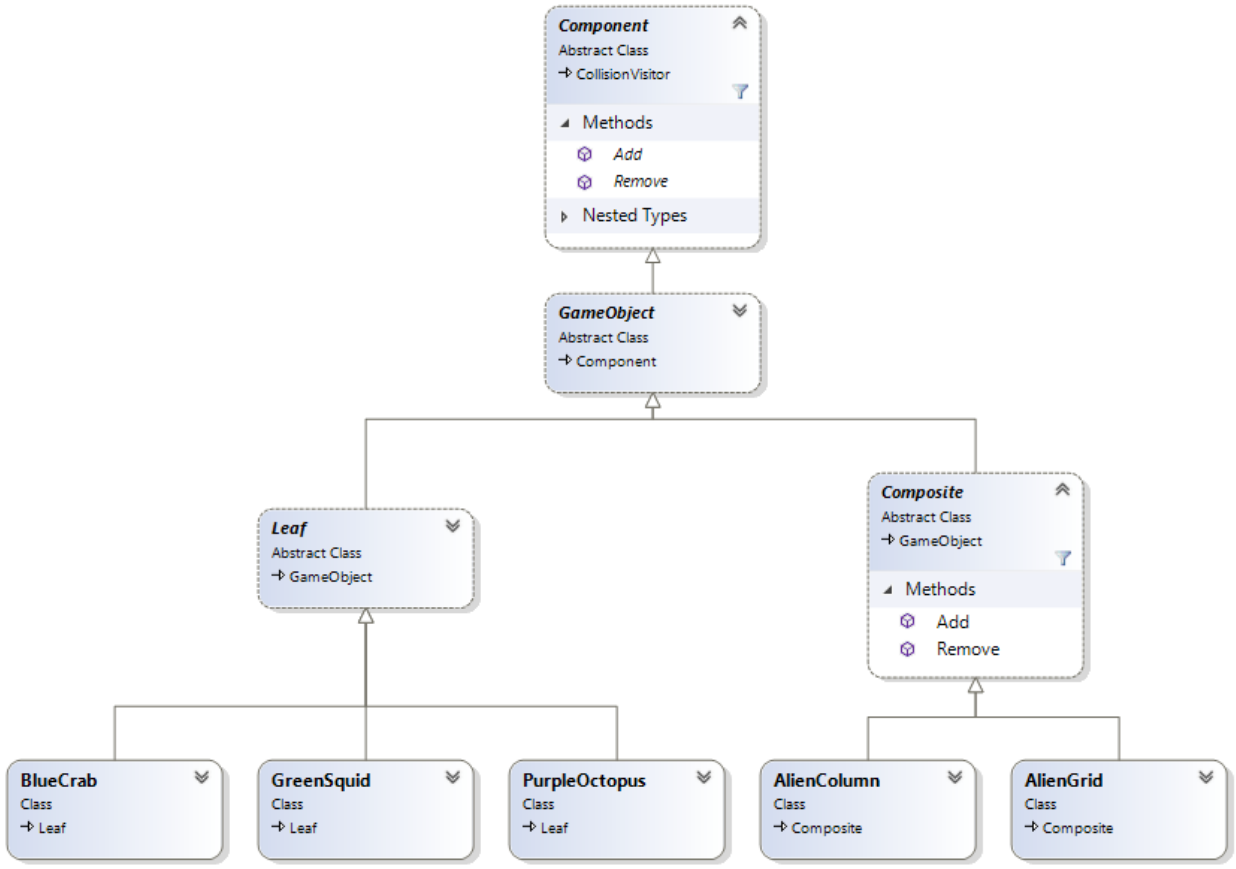
In the Process method, the Time Event calls Execute on its pCommand variable which calls execute on the overridden derived method of whichever derived type pCommand is holding a reference to. Since the pCommand variable is of type Command. It can be used to call the execute method on any derived type.

Time Events are initially added to the Timer Manager in the Load Content method of the game. After they are processed and removed, they are reinserted with the Timer Manager inside of the overridden derived execute methods.

Composite

There is a total of 55 aliens in the scene during game play. They all move together across the screen in a uniform way. Moving these objects individually while maintaining their synchronization is a challenge. There also needs to be a clean way to perform collision check against each alien. This problem is simplified by grouping the objects together into composites.

A Composite pattern is a collection of objects in a tree hierarchy. There is an abstract Component base class with abstract methods for adding and removing from the tree. Derived from this base class is Leaf class and a Composite class. Leaf represent individual objects while a Composite represents a group of objects. Composite classes override the abstract methods while leaf classes leave them undefined. Leaves can be added and removed from composite groups.



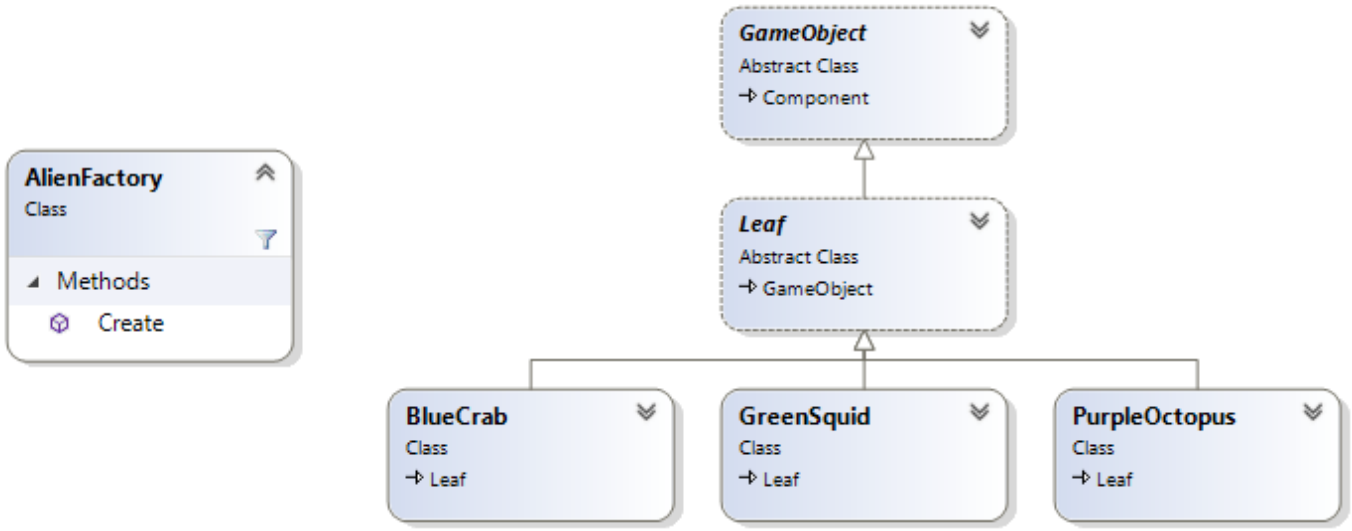
Aliens and shields each belong to their own Game Object tree composites. The aliens and shield bricks themselves are Leaf nodes which are children of alien column composites. These column composites are children of grid composites. The grid composite is used to move all the aliens at once rather than through individual calls. There is a composite of null objects used for the player lives as well. This is used to display the sprite of the player lives in the bottom left corner of the game scene. Although there is only one instance of the player ship and missile, they are also store within composites containing only one item. This allows for all game objects to be handled in a consistent fashion.

The Alien Grid composite simplifies the process of performing collision checks against aliens. When two composites collide, there is then a collision check against the 11 columns of aliens. If there is no collision with a column, then no collision checks are performed against the individual aliens. If there is a collision against a column, then a collision check is run against the 5 aliens of that column. This prevents from having to perform 55 collision checks against the aliens each frame when the collision pair manager preforms its collision checks.

FACTORY

Some of the Game Objects have similar behavior but have slightly different attributes. Aliens and shields have different name enumerations and own different references to sprites. A clean way to construct a new alien object is through the use of a factory. A factory method has the ability to switch between which object is being constructed when all of the objects inherit from the same base class.

A Factory pattern consists of a factory class with a Create method. The Create method takes in an argument which is used as the argument for a switch statement within the method. It switches between constructor calls and constructors an object in a reference to the base class.



Factories are used for the creation of the Alien composites and the shield composites. A name enumeration is passed into the factories create method as an argument. The Alien Factory has the ability to construct an AlienGrid, AienColumn, UFO, Squid, Crab, or Octopus. All

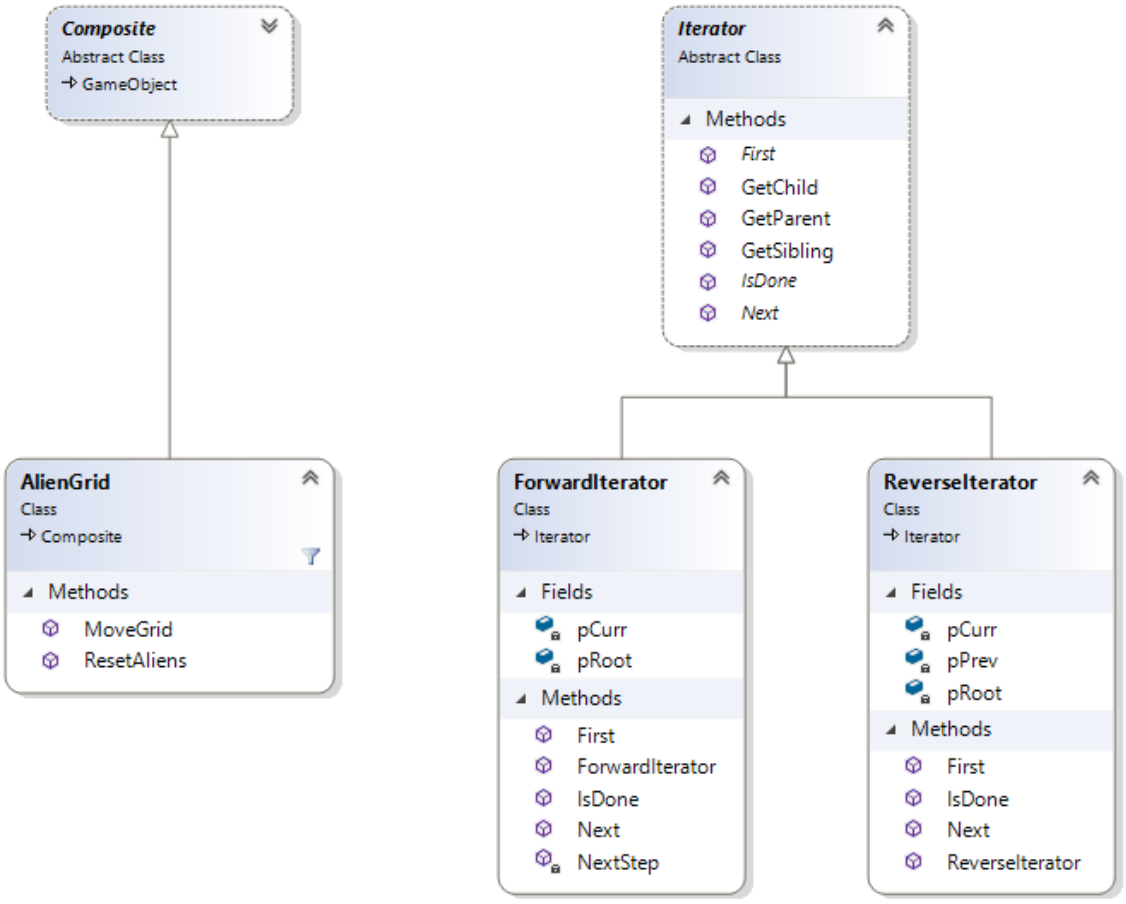
these classes are derived from the Game Object class. The factory stores and returns a reference to one of these newly constructed object as a Game Object type. The Game Object reference is then downcast to the type of the corresponding object.

The Shield factory behaves in the same way but for shield bricks and shield composites. These are also derived from the Game Object class and downcast to their appropriate type.

Iterator

There must be a way to access the Game objects stored inside of tree composites. There are occasions when every leaf of a tree needs to be access such as when the alien grid moves its children across the screen. This is achieved using an iterator that walks the nodes of the tree.

An Iterator pattern consists of an abstract iterator class and an abstract aggregate class. The abstract iterator contains four abstract methods called First, Next, Done, and Current Item. A concrete iterator inherits from the abstract iterator and overrides these methods. The concrete aggregate owns a reference to the concrete iterator and uses it to iterate its aggregation.



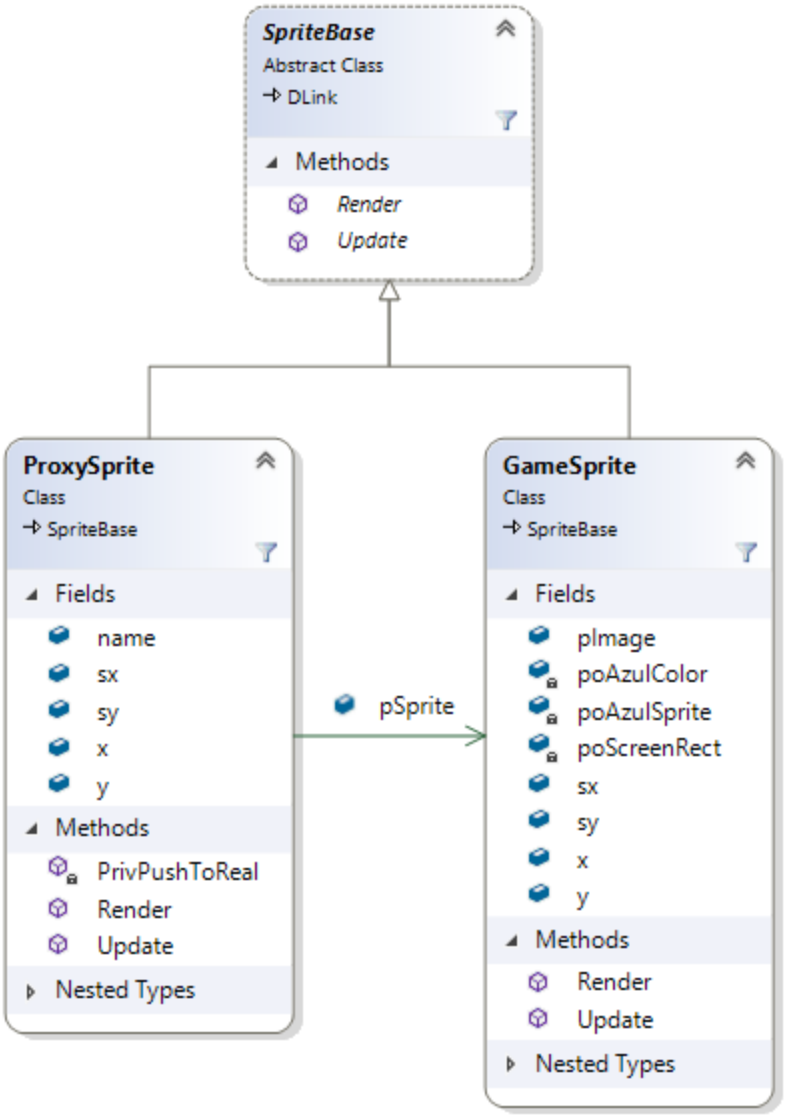
The iterator pattern is used on the alien and shield composite aggregates. A concrete iterator class called Forward Iterator is used to walk the collections starting from the root and working its way down. This uses a depth first search by iterating all the way down the length of the tree before coming back to the root and iterating down the tree again. This is used to move the alien grid composite, reset the alien grid composite, reset the shield grid composite, and remove from the player lives composite.

A Reverse Iterator concrete class is used to walk the collections starting from a Leaf and working its way back up to the root. The Game Object Manager uses this iterator to update all the Game Objects each frame. The children of the composite are updated first while the composite itself is updated last. The is to prevent a bug that would cause the children to be updated out of synch with one another.

PROXY

There is a maximum of 55 aliens in the game scene at one time. This adds up to a large amount of Game Sprite data. Many of the Game Sprites are identical or have slight variations from one another. It is more efficient to reuse this Game Sprite data where possible. Proxy Sprites are used to hold the place of duplicate Game Sprites.

A Proxy pattern has a subject base class with at least two concrete derived classes. One of the derived classes is the real subject and the other is the proxy subject. The proxy subject shares the fields of the real subject which are variable. It does not share fields that are constant in the real subject. The proxy has a method that pushes the changes of its variable fields to the real subject. This way multiple proxies can represent one real subject.



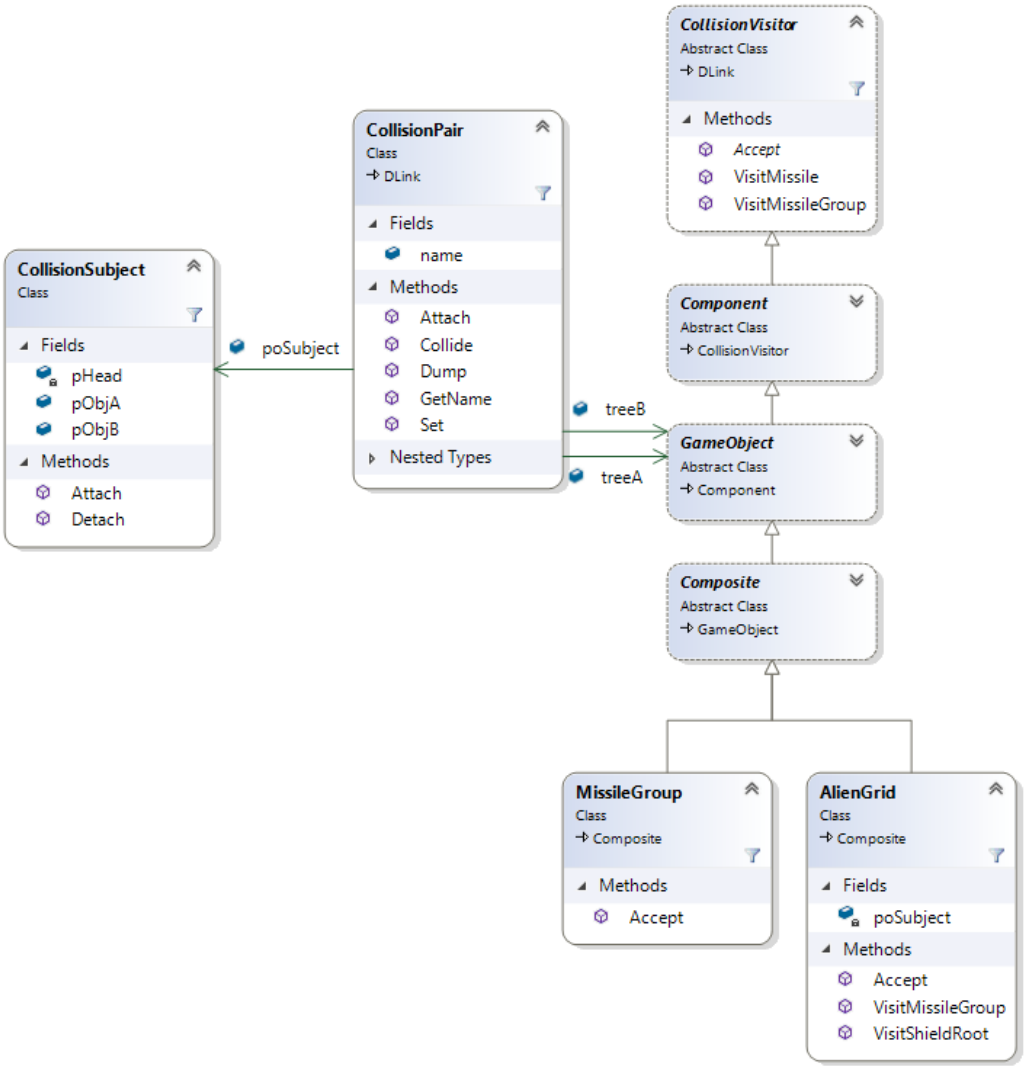
All of the aliens own references to Proxy Sprites. Each Proxy Sprite has a reference to one Game Sprite. As the location and scale data of the Proxy Sprite changes each frame, it is pushed to the real Game Sprite. This allows for multiple aliens to have a reference to one instance of the Game Sprite. The single instance of the Game Sprite is rendered multiple times in the same frame with different positions.

VISITORS

The Collision Pair Manager checks for collision between two Game Object composites each frame. When a collision is detected, there must be a way of checking collisions with the

children of the composites. This is needed to find out what exactly which object in the composite has been collided with. This can be achieved with a Visitor pattern.

The Visitor pattern uses an abstract class called Visitor made up of multiple concrete visit methods and an abstract method called Accept. The Accept method and concrete Visit methods are overridden by one of the two elements that want to interact with one another. Each element inherits from the abstract Visitor method. When the Accept method of one visitor is called, it takes in another visitor as an argument. This argument then calls the Visit method associated with the first visitor and takes the first visitor in as an argument. Inside of the overridden visit method, any interaction between the two element is defined.



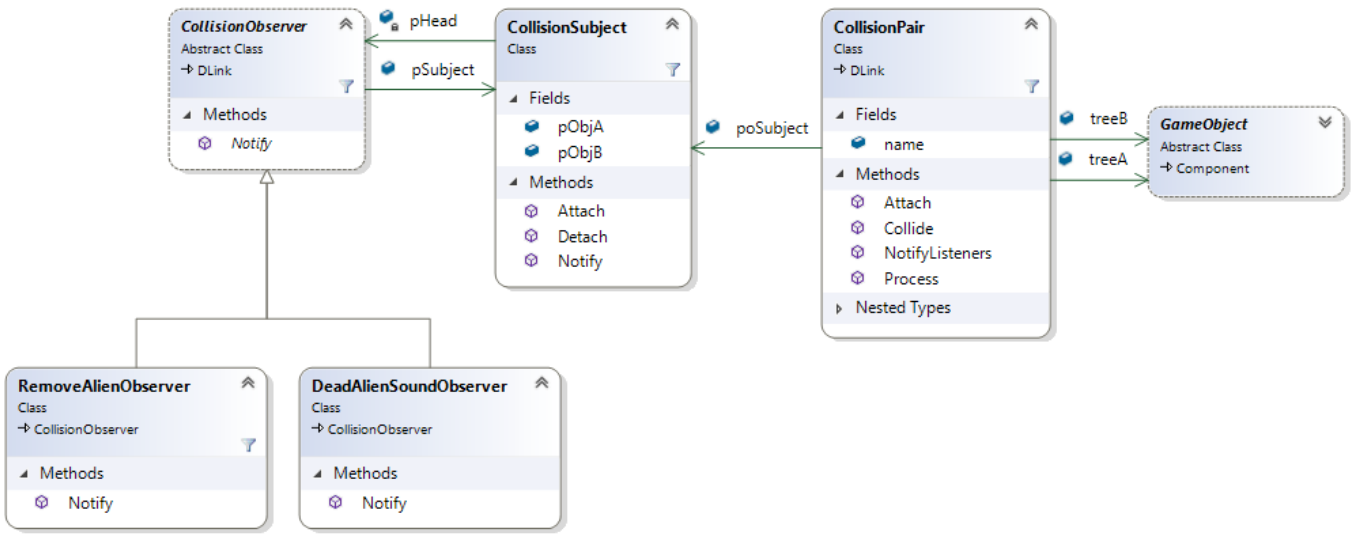
When a missile composite collides with an alien grid composite. The Missile Group composite calls its Accept method and passes in the Alien Grid composite as an argument. The Alien Grid composite then calls its overridden Visit Missile Group method inside of the Missile Group accept method. This Visit Missile Group belonging to the Alien Grid checks for a collision

between the missile group and the children of the Alien Grid composite. The process is repeated until the missile composite visits all the children of the Alien Grid composite that it collides with. This allows for the ability to check collisions against an entire composite. If a Game Object does not collide with a parent in the tree, then it does not waste resources checking collisions with the children.

OBSERVER

If a collision is detected with a child, there needs to be a way to trigger off an appropriate response. When a missile collides with an alien, the alien needs to be removed from the game, the missile needs to be removed from the game, a sound needs to play, and an explosion sprite needs to appear. By using an Observer pattern, all these events can be triggered after a collision.

An Observer pattern is structured with an abstract Observer class and an abstract Subject class. The Observer class has an abstract update method which is overridden by concrete observers. The concrete observers inherit from the abstract observer. Concrete subjects inherit from the abstract subject class and own references to concrete observers. The concrete subject overrides an abstract method called Notify. Every time Notify is called; it calls the update method of all the observers referenced by the subject.



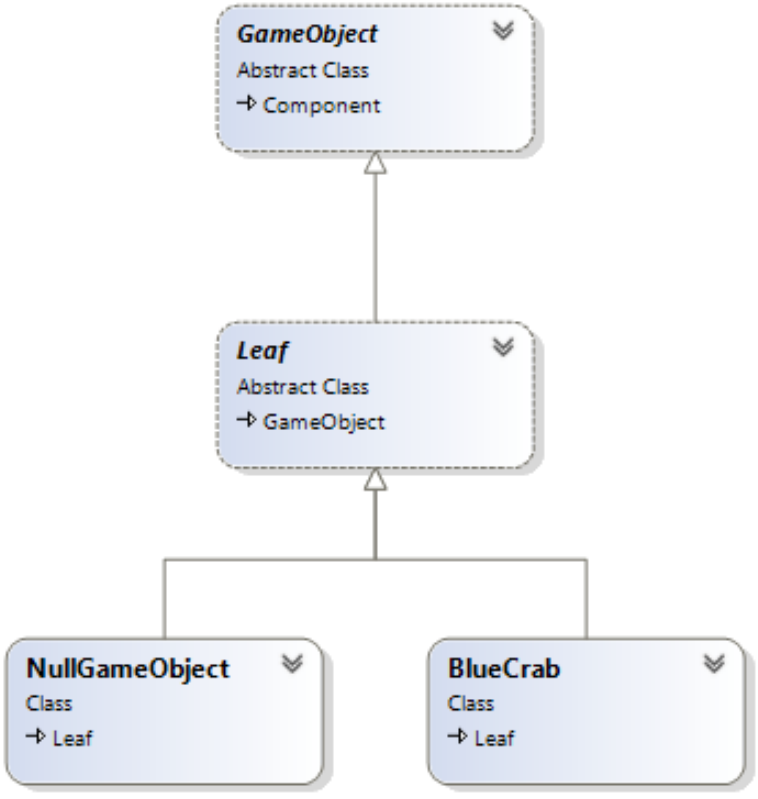
Collision Pairs act as observer subjects. They have an Attach method which attach observer classes to a linked list. Then when a collision occurs between a Collision Pair, it iterates the list of observers and calls the Notify method for each. This allows for a different collision reaction for each unique Collision Pair of objects.

The Input Manager uses observers when input is detected from the user. There are input observers for firing the player missile and moving the player ship left and right. There is an input subject for each key. Each input subject has a linked list of observers for when that key is pressed.

Null Object

There are occasions when Game Sprites need to be displayed but do not need all the attributes of a game object. For these cases a Null Object Pattern is used.

A Null Object Pattern is used when an object needs to be replaced with another object that does nothing. Both objects inherit from the same base class. The null derived class can be swapped out and take the place of a derived class that has functionality.

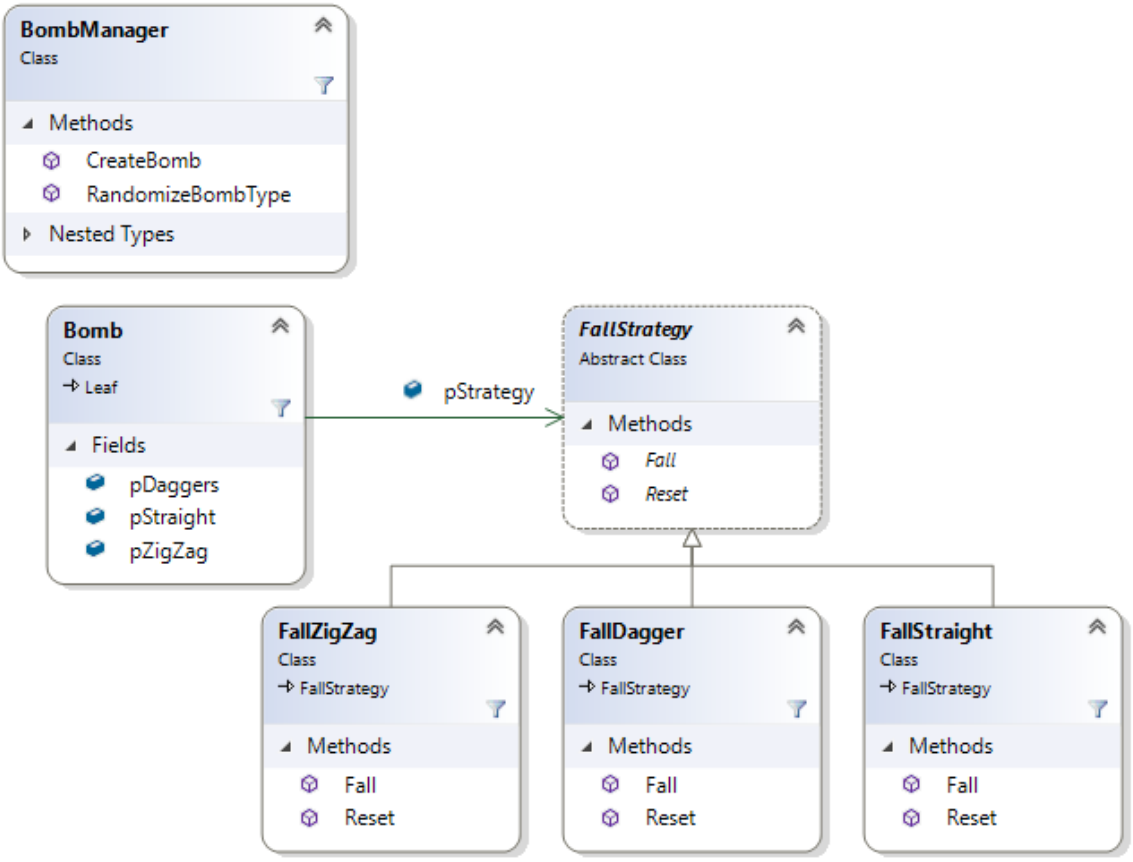


The static sprites in the main menu of the game do not need to own a reference to a Collision Object. The sprite representing the remaining ship lives of the player do not need collision functionality. For these objects, a Null Game Object takes the place of the Collision Objects. This done by passing in a null object argument into the constructor.

STRATEGY

There are three bombs that fall from the aliens. A straight bomb, a dagger bomb, and a zigzag bomb all have different behavior as they fall. A straight bomb has no behavior as it falls. A dagger bomb flips on its x-axis and a zig zag bomb flips on its y-axis as it falls. A strategy pattern is used so that one bomb has the option to fall in three different ways.

A Strategy pattern consists of an abstract Strategy base class with an abstract method. Multiple classes can inherit from this base class and each represent a different strategy. Every derived concrete class overrides the abstract method from the base class. A context class owns a reference to the abstract Strategy base class. This context class has the ability to swap out and cycle between the derived concrete strategy classes. The implementation of the overridden method that belongs to the strategy that is currently active will be used.



The Bomb class owns a reference to a Fall Zig Zag strategy, Fall Dagger strategy, and Fall Straight strategy. A fourth reference called Strategy represents the currently active strategy. The Bomb Manager calls its Randomize Bomb Type method each time a bomb is launched. This uses a switch statement to randomly set the current strategy as one of the three available

strategies. Once the current strategy is set, the bomb falls with the behavior of the current strategy.

IMPROVEMENTS/COMMENTARY

Each scene-specific Timer Manager needs to be offset by the amount of delta time that passes before their scene is made active. Currently these calculations are hard coded for each scene. It would be better design to have a static manager handling the offset of these scenes.

Currently the when Game Objects are deleted from a scene, their proxy sprite is taken out of the render loop and their collision is turned off. They are still present inside of their composites. When a game scene is reset, they are set back to their original proxy sprite and collision is turned back on.

A better design would to remove them from their composites and store them inside of a separate collection of dead Game Objects. Then when the scene is reset, the game Objects would be added back into their respective composite. This is a more robust solution than simply taking an object out of the render loop.