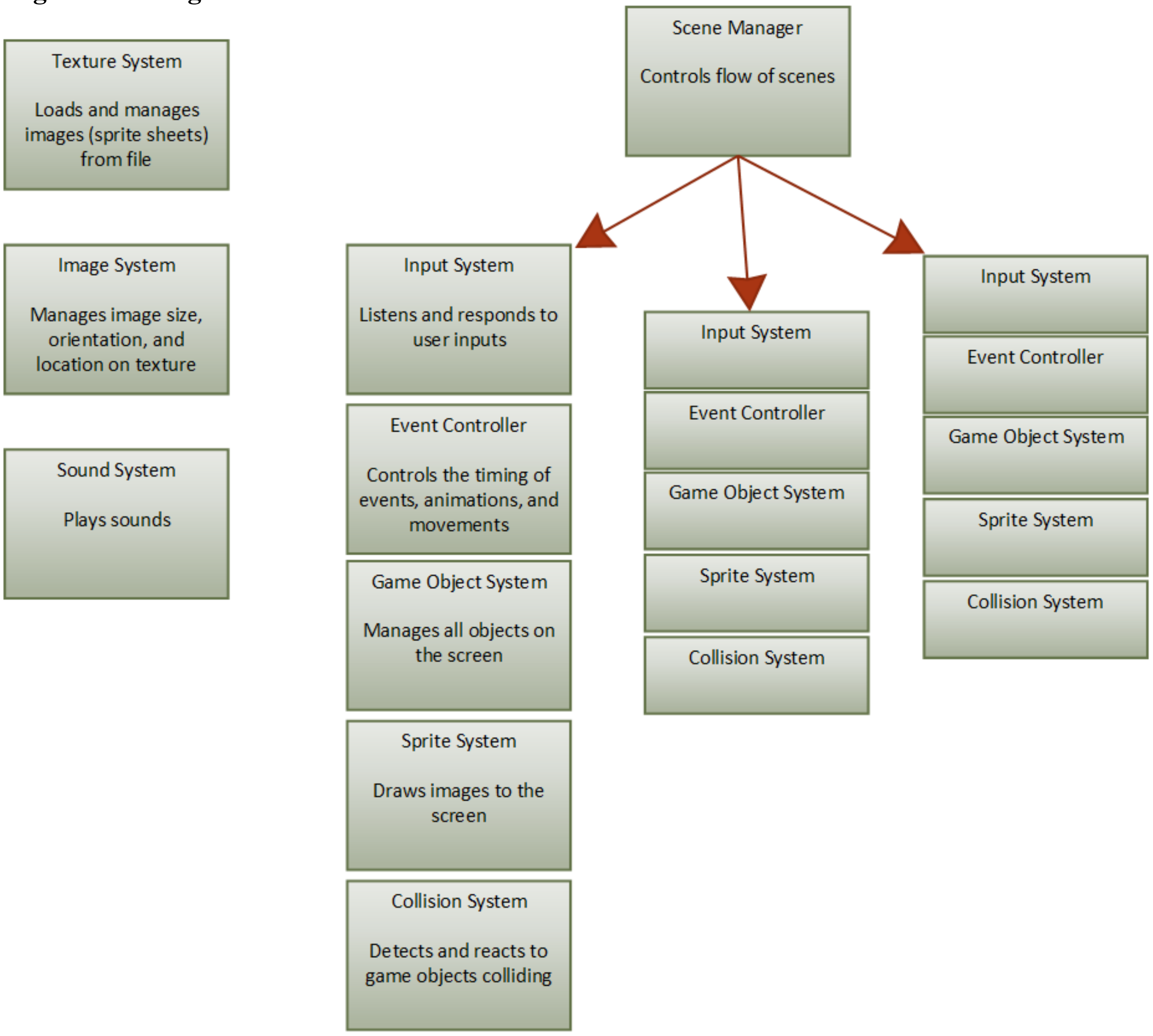
Space Invaders
Design Documentation
February 17th, 2015

Programmer: Chris Jongeward
Demo: http://youtu.be/gApLLkotd_o?hd=1

Game:	Space Invaders (A clone of the original arcade game)
Programmer:	Chris Jongeward
Programming Language:	C#
Game Engine:	Azul
Number of Classes:	

Description: This documentation details the design of the Space Invaders arcade game using modern object-oriented design methodology.

High-level Design



This game is a clone of the original arcade version of Space Invaders. It is built up from the Azul game engine and utilizes the 2D components to create the display window, draw images, and draw text. The game is programed using the basic features of the C# programming language. All data structures are built from scratch and none of the built-in types that come with C# are used.

The design of this game is centered around the use of "Manager" objects which are used to create, store, control, and destroy that various components of the game. The Manager objects make use of the singleton design pattern so that they can be accessed in a static context from any other class in the game. The game is composed of the following components each of which has its own manager object.

Textures:

The texture system controls access to images such as backgrounds and sprite sheets.

Images:

The image system is used to define and store the location of images within the textures. The image system is dependent on the texture system.

Sprites:

The sprite system utilizes images and textures to create sprites. The sprites are organized into groups such as aliens, shields, and bullets as well as optional sprites such as bounding boxes. The sprite system is responsible for managing and drawing sprites to the screen through the Azul game engine.

Game Objects:

The game object system controls the position and organization of the objects on the screen.

Time Event System:

The event system tracks the global time and triggers the various time-dependent events that occur during the game.

Collision System:

The Collision system tests for collisions between the various game objects and triggers the appropriate reactions.

Inputs:

The input system listens for user inputs and sets off the corresponding responses.

Sounds:

The sound system loads and plays the sound files that play during the game.

Scene Manager:

The scene manager tracks the state of the game and switches between scenes.

1 Texture and Image System

1.1 Manager

Many of the systems in the game are built around a "Manager" base class that controls the creation and removal of the nodes in the system. Although it is not a formally recogized design pattern, the Manager architecture is reused frequently in this game and will be referred to as the "Manager design pattern" in this document. The texture and image systems make use of the Manager pattern to manage the various textures and images that appear during the game. The texture and image managers are derived objects that inherit from a manager base class. These derived Manager objects make use of the singleton design pattern.

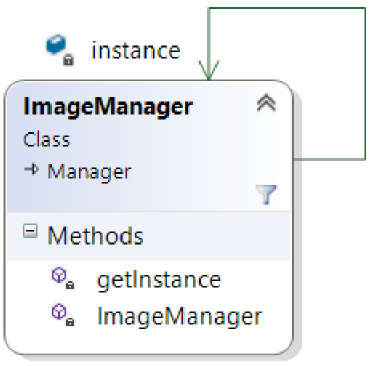
1.1.1 Singleton Design Pattern

The Singleton design pattern is use in cases where a single instance of an object is needed during the execution of the game. Use of the singleton pattern ensures that only one instance of the object is created. The object in question maintains a static reference to the one and only instance and controls access by other classes though a method called "getInstance()." the getInstance method constructs the instance if it hasn't yet been initialized and then returns it.

Several attributes characterize the singleton pattern:

1. Only one instance of the object exists and it is owned by the object itself.
2. The constructor of the object is private so additional instances can not be constructed by other classes.
3. The instance is accessed by an accessor method.

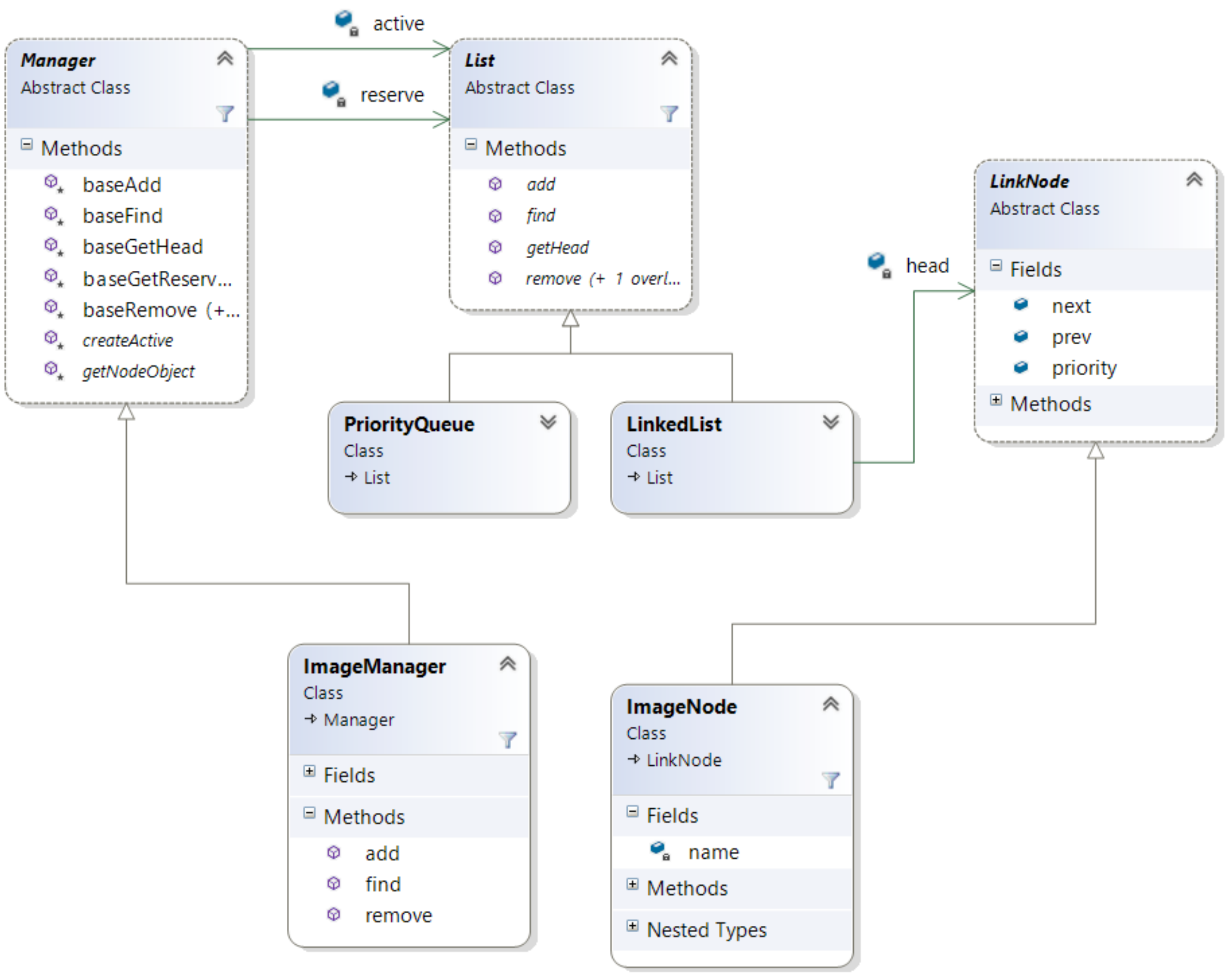
The singleton structure can take many forms as long as the previous three criteria are satisfied. In this game, the instance accessor is private. All public methods are static so that they can be accessed from anywhere in the code. Each static method uses the private getInstance method to obtain a reference to the instance. The advantage to this approach is that singleton implementation is hidden from the user. The user only needs to access the method in a static context and the method itself handles the instancing of the object.



This diagram shows the singleton aspect of the Image manager. The same layout is used in the Texture manager, Sound manager, and Scene manager. In this example, all the members and methods that make up the singleton are private so the user is shielded from the singleton nature of the object. This becomes benificial when a singleton object needs to be expanded to multiple instances. Access to the multiple instances can be controlled through the getInstance method thus minimizing changes to the code. This architecture is used to support 2-player mode by allowing an instance of each manager for each player.

1.1.2 List Base Class

The Image manager is a derived object of the Manager base class. The Manager base class is used by all managers to store the nodes that make up the system. As such, the Manager base class contains a List of nodes. Each node in the list is a base class for a specialized node for the system in question.



This diagam illustrates the use of the Manager and LinkNode base classes as implemented in the Image system. In this case, the Image manager contains a list of Image nodes. Note the following atributes:

1. The Manager base class owns a List object.
2. The List object utilizes the Strategy design pattern. It is a base class with two different implementations, and linked list and a priority queue.
3. The creation of the List is accoplished by the abstract method, createActive(), which is implemented in the derived manager class. This is an example of the Factory design pattern.
4. The list is made up of LinkNode objects and the ImageNode derives from LinkNode.

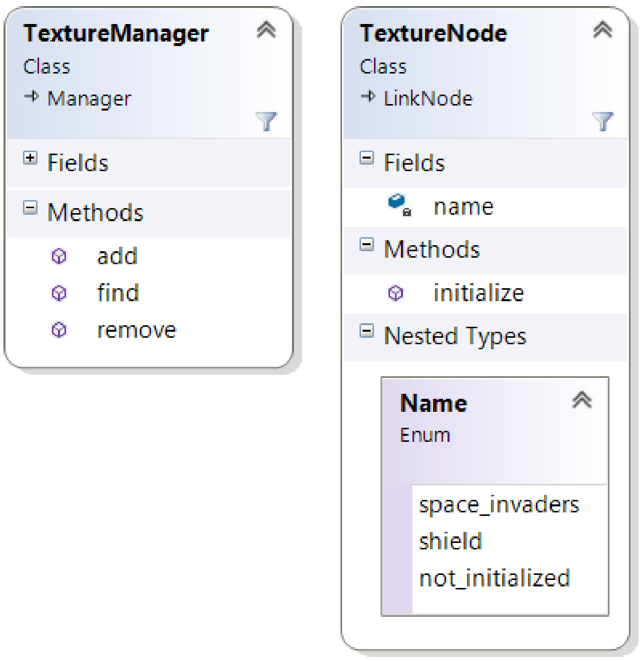
This collection of base classes is reused in all other Manager objects including the Texture system. The Strategy and Factory patterns will be discussed in subsequent sections of this document.

1.1.3 Resource Pooling

Another important note from the previous diagram is the use of two lists in the Manager base class. An important goal of this design is to avoid dynamic memory allocation during runtime. To acheive this result, a predetermined number of reserve nodes are allocated during initialization and added to the reserve list. This creates a resource pool. Whenever a node is needed by the manager, it will pop a node off the reserve list as an alternative to dynamically allocating the memory during runtime. If the reserve list is empty and a new node is needed, the reserve list will repopulate with a predetermined number of nodes. Ultimatly, the goal of the final product is to initialize the reserve list with enough nodes to last for the entire game.

1.2 Texture Manager

The texture managment system utilizes the aformentioned "Manager design pattern" to manage texture nodes. A texture node contains a texture that is read in from the file. In this game, all of the texture nodes contain sprite sheets and there are to textures total. One texure contains the Space Invader images such as the aliens, player, bombs, and explosions. The other texture contains the bricks used in the shields.



The texture manager (and all other managers in the Manager pattern) contains three methods to control list access:

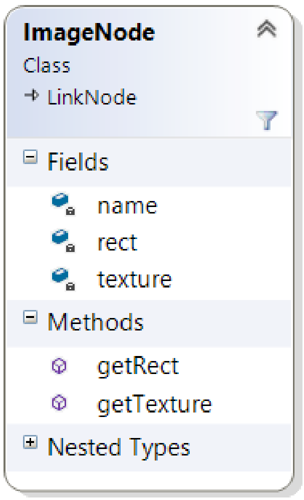
Method: Add(Enum name, String filename)
The "Add" method removes an uninitialized node from the resource pool and initializes it with a texture and a name.
Parameter: name
The "Name" parameter is an enum that is stored within the node itself. The "Add" method will assign the node a name from its own enumeration of possible names. This name is used to uniquely identify the node so that it can be found by the other methods.
Parameter: filename
The "filename" parameter is a string that contains the location of the image in the file system. This image is loaded into an Azul Texture and stored in the texture node.

Method: find(Enum name)
The "find" method searches the active list and locates a node with the specified name. The method returns the node (Or null if the node is not found).

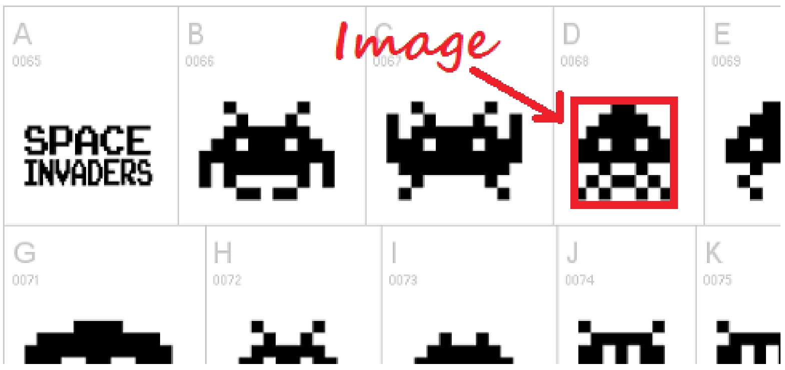
Method: remove(Enum name)
The "remove" method finds the specified node in the active list and removes it. The node is cleared of all data and the name is set to "not_initialized." The node is then placed back into the resource pool. All data must be cleared from the node when it is removed otherwise old data could linger when the node is reused.

1.3 Image Manager

The Image manager is implemented the same as the Texture manager except the image node has a different composition and the image manager "add" method has a different signature.



The Image node contains a reference to a texture node and a rectangle that identifies the location on the texture that contains the image. Since one texture contains many images, the image manager contains many image nodes (two images for each alien, one for the player, an alien explosion, and player explosion, etc...). The following figure shows an ImageNode rectangle superimposed on a Texture. The Sprite system can use this data to draw the selected image on the screen.



2 Sprite System

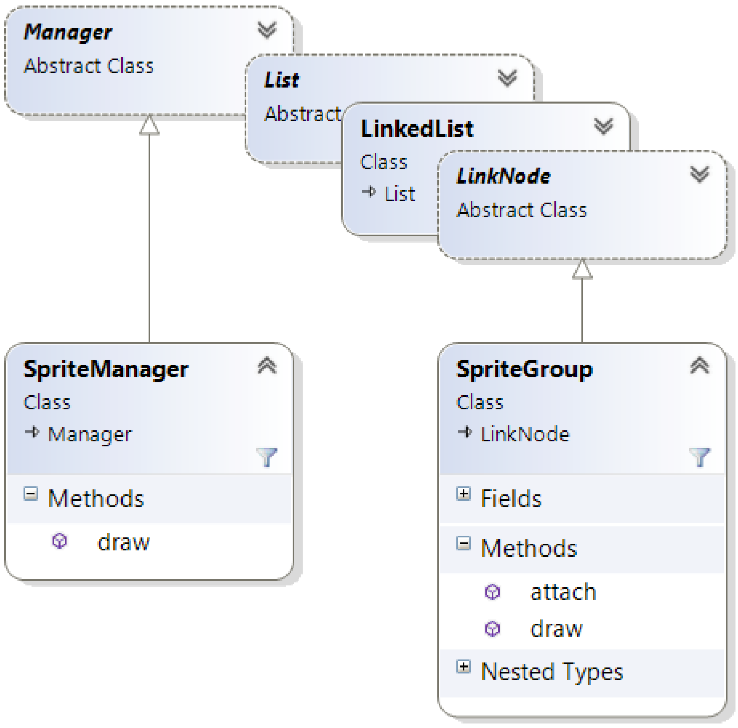
The sprite system is resposible for managing and drawing sprites. A sprite is an object that can be drawn to the screen. The sprites in the sprite system are built from the Azul Sprite object. The Azul Sprite object contains:

– A texture
– A rectangle specifying a location on the texture where the image is located
– A position on the screen to draw the image
– The size of the image on the screen

The Sprites are contained within a sprite management system that utilizes the Manager design pattern to organize sprites into groups

2.1 Sprite Groups

Unlike the image and texture managers, the Sprite manager does not directly manage a list of sprite nodes. Instead, it manages a list of SpriteGroup nodes. Each sprite group contains a sub-list that contains the actual sprite nodes. In this manner, the sprite nodes can be organized into groups.



In addition to the typical Manager methods described in the previous sections, the Sprite manager also contains a "draw()" method. The "draw" method iterates through the list of sprite groups and calls their respective "draw" methods. Each group then iterates through its own list and draws the individual sprites to the screen.

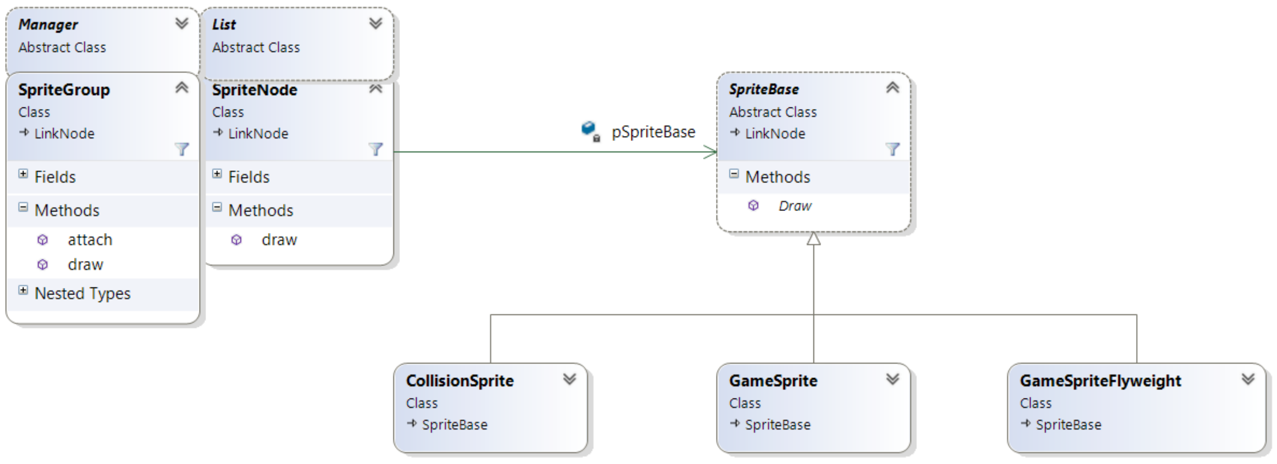
By subcategorizing the sprites into groups, this gives us the ability to disable certain groups of sprites from being drawn. This is specifically handy during development when the developer may want to draw extra data to the screen (such as bounding boxes) that will not appear in the final product.

The sprite manager contains the following groups:

- Aliens: the grid of 55 aliens that start the game
- Shields: the group of bricks that make up the shield
- Bullets: bullets fired by the player
- Bombs: bombs dropped by the aliens
- UFO: the ufo that flies across the top of the screen
- Player: Although their is only one player at a time, the player sprite is stored in a group for consistancy.
- Bounding boxes: These are invisable boxes that surround each object on the screen and are used to detect collisions between objects. Although they are not drawn to the screen in the final product, they can be drawn during development for debugging purposes.

2.2 Sprites

The individual sprite nodes are contained in a list that is managed by a SpriteGroup manager. Each sprite group is a node in the SpriteManager list but is also a manager itself that contains a list of sprite nodes. The Sprite group utilizes the Manager design pattern except that it is not a singleton since there are many sprite groups.



Each node in the sprite group contains a SpriteBase object which indirectly contains the image that will be drawn to the screen. Notice that there are several different types of sprite objects that derive from the SpriteBase class. Each of these derived sprite object can draw something to the screen but the behavior and implementation is different. This neccessitates the use of the Strategy design patter.

2.2.1 Strategy Design Pattern

Strictly speaking, a sprite is anything that can be drawn to the screen. This can take the form of an image, a shape, or text. The Azul Sprite object can be used to draw an image to the screen from a texture but other sprites such as bounding boxes do not require the use of an Azul Sprite object. In the case of a bounding box, the sprite manager should still draw it to the screen but instead of an image, the sprite should only draw four lines to make a box on the screen.

No matter how the sprite is implemented, all sprites must be able to draw something to the screen and therefore must have a "draw()" method. This is captured in the abstract class SpriteBase. The SpriteBase is a base class that contains a "draw()" method but the implementation of the method is handled by the base class. In this manner, a SpriteGroup can contain a list of SpriteBase objects. It can iterate through the list and draw each sprite without concern over what type of sprite it is or how it is implemented.

The SpriteBase class is the base class of three different derived sprite classes, CollisionSprites, GameSprites, and GamspriteFlyWeights. The CollisionSprite draws a box on the screen that is typically (although not always) centered on an actor in the Game. The GameSprite and GameSpriteFlyWeight classes draw an actual Azul Sprite.

This architecture illustrates a use of the Strategy design pattern. In the strategy pattern, different implementations of the same bahavior are encapsulated into seperate classes. Furthermore, the classes are interchangable and the specifics of the implementation are hidden from the user. In the sprite system, the behavior in question is drawing something to the screen. We have two different ways to do this (drawing a box, or drawing an image). Since the implementaion of the "draw()" method is encapsulated in a subclass of the SpriteBase class, each implementation is interchangable and invisible to the SpriteManager.

The main benifits to the strategy pattern are to facilitate maintenance and expansion of a system.

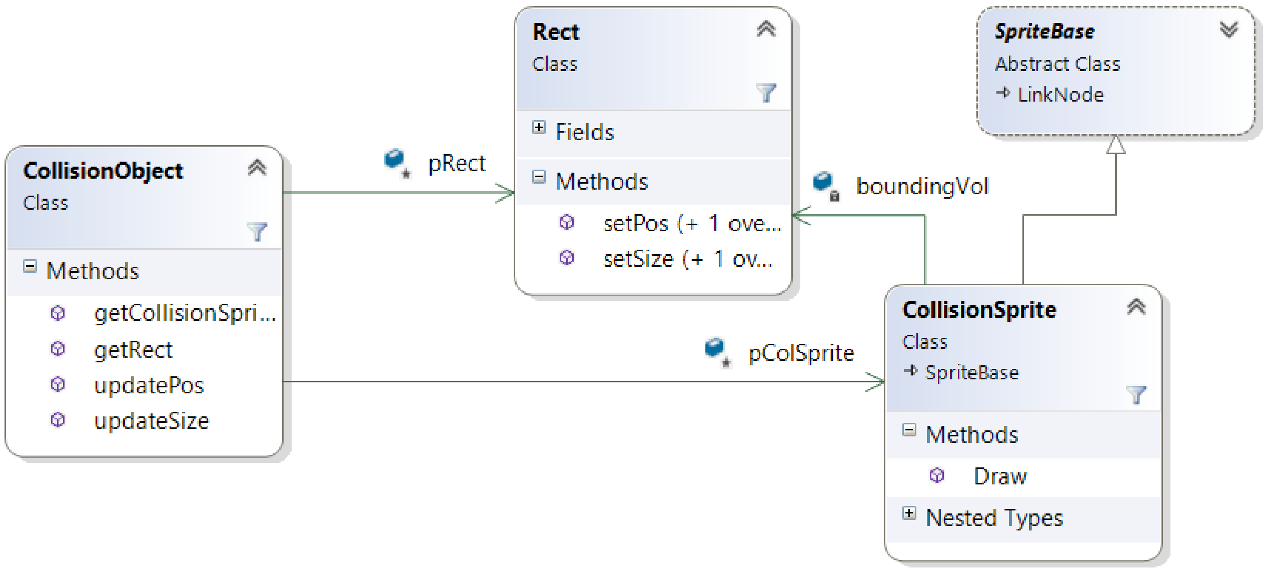
Maintenance: Each implementation of the behavior is encapsulate in its own class. So if a defect is discovered in a specific implementation, only that one class needs to be modified. This preserves and protects the functionality in other classes. Fixing the defect involves "breaking the seal" of only one class. Since other classes are likely already complete and fully tested, it is undesirable to modify them unless necessary.

Expansion: When the need arises to add new implementations of a behavior, it can be encapsulated in a new class that implements the proper interface. Suppose we were to use the sprite system to draw text to the screen, we could create a new derived class of SpriteBase and use it to draw text just as we would draw an image or counding box.

Seperating sprecific implementations into classes promotes proper object-oriented design by supporting the single resposibility principle. The single responsibility princle states that each class shoud have one and only one responsibility. It also supports the open-closed principle which states that a system should be open to expansion but classes should be closed to modification.

2.3 Collision Sprites

The Sprite system is composed of two different types of sprite objects, Collision sprites and Game sprites. The collision sprite maintains a reference to a rectangle that can be drawn to the screen on command. The position, size, and orientation of the collision rectangle is controlled by a CollisionObject that is composed of the rectangle and references the CollisionSprite.

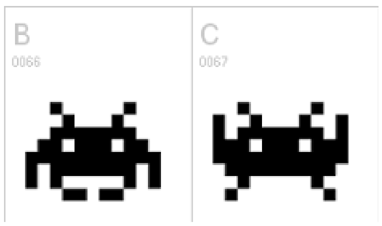


2.4 Game Sprites

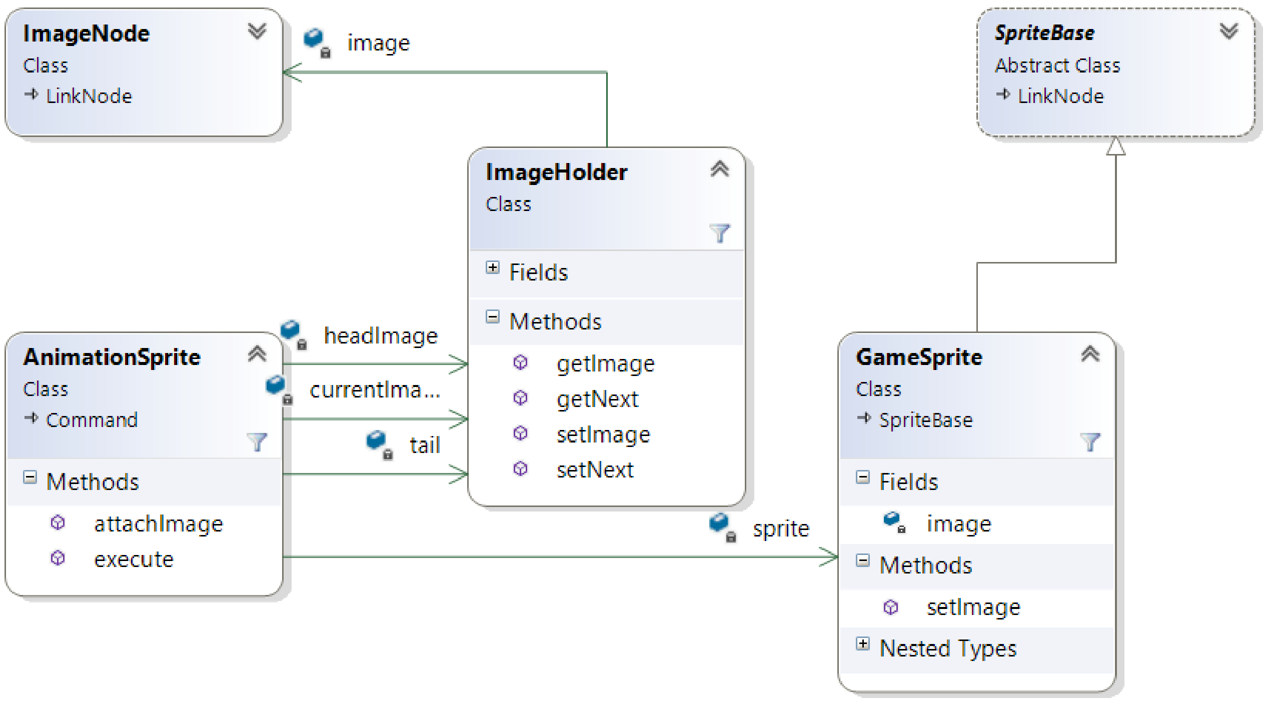
The GameSprite object is an implementation of SpriteBase that contains an actual Azul Sprite Obejct. The GameSprite is an aggregation of a texture and image obtained from the TextureManager and ImageManager. It uses the texture and the image data to compose an Azul sprite that can be drawn to the screen on commad.

2.4.1 GameSprite Animation

In the case of the aliens, it is necessary to animate the sprite. As the aliens move across the screen, they animate between two images.



To accomplish this, an animation system is used.



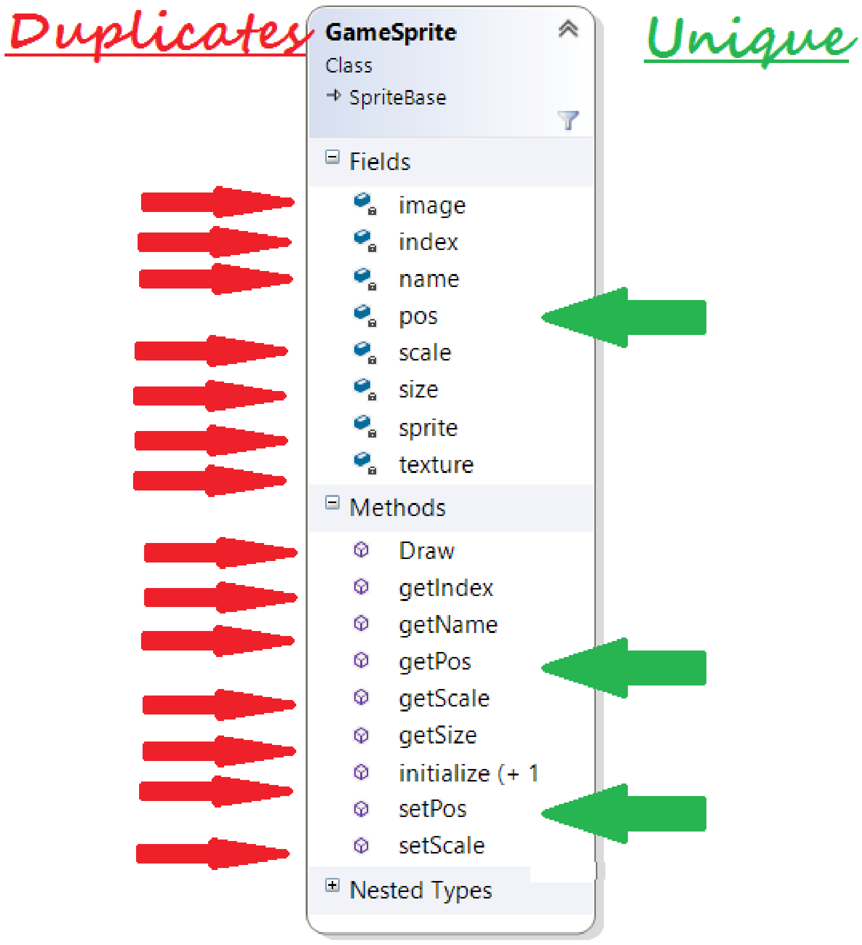
The animation system is composed of an AnimationSprite which maintains a reference to the GameSprite that it is animating. The Animation sprite contains a circularly linked list of ImageHolder objects and cycles through them at a specific time interaval to create the appearance of animation. The alien animation sequence contains two images that are provided by the ImageManager. For each image in the animation sequenct, the ImageHolder object maintains a reference to the image node.

When the animation sprite is triggered, it will cycle to the next image in its list and call the "setImage()" method in the gamesprite to change its image. Subsequently, the next time the GameSprite is drawn by the Sprite system, a different image will appear on the screen. This approach is adventagous because the animation of the GameSprite is decoupled from the rest of the Sprite system. The Sprite system draws the game sprite each frame and has no knowledge that the image is periodically changed.

2.4.2 Flyweight Design Pattern

A recurring problem throughout this game is the GameSprite redundancy. The grid of Aliens at the start of the game contains 55 actors but only three unique aliens (squids, crabs, and octipuses). The top row of aliens contains 11 identical squids, the next two rows contain 22 identical crabs, and the final two rows contain 22 identical octopi. Each of the three different alien types are the same size, and image, and are animated in unison.

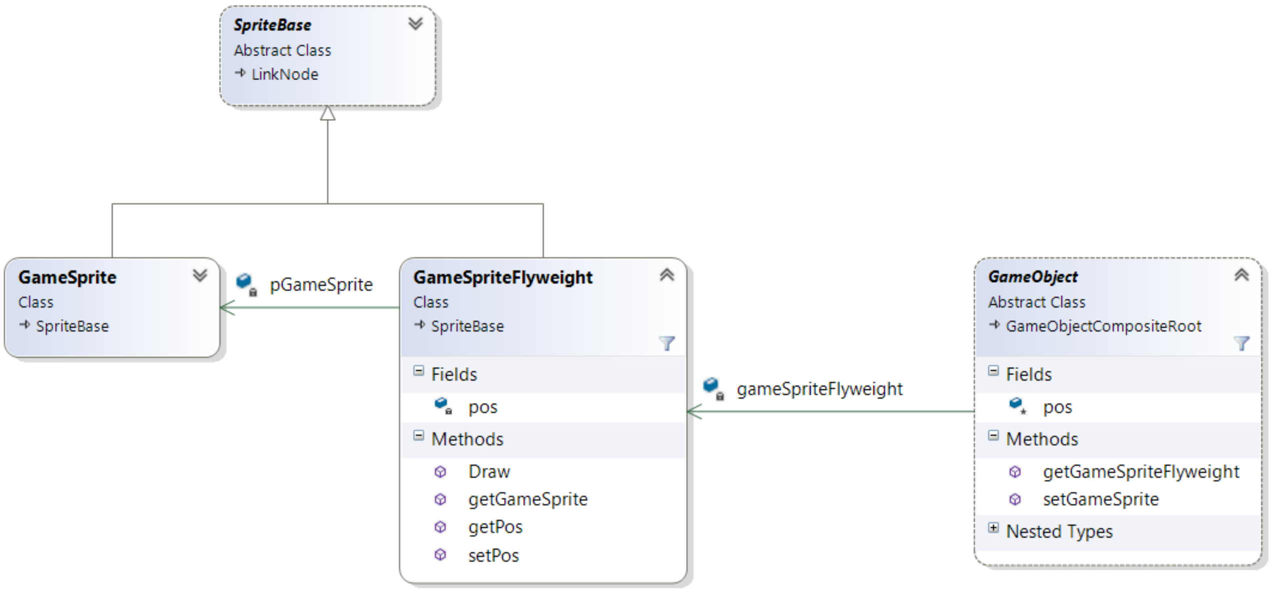
In one approach to the GameSprite system desing, each individual alien could have its own GameSprite. This would result in the duplication of many of the GameSprite parameters. If we have 22 crabs and each of them has duplicate parameters, then we have uneccessary redundancy or variables in memory which could be a detriment to the performance of the game.



Obviously, this redundancy is unacceptable. The solution to this problem is the FlyWeight Design Pattern. In the flyweight pattern, a second smaller version of the subject class is created that contains only the parameters that vary between actors as well as a reference to the subject class. This smaller class is called the flyweight. Each actor contains its own flywieght to control its variable parameters and the flyweight only calls the subject class when neccessary.

In Space Invaders, there are three GameSprites, one for each type of alien. Each of the aliens on the screen contains its own GameSpriteFlyweight object that allows it to change its position. The shape, size, image, teture, and animation of aliens is common to all aliens so those parameters are not included in the flyweight. Whenever an alien changes position, it calls its flyweight "setPos()" method. Then later, when the sprite is drawn, the flyweight "draw()" method is called. The "draw" method calls the "draw" method of the GameSprite but first it sets the position of the GameSprite to its own position. In effect, if there are 22 identical aliens on the screen, the same GameSprite is being drawn 22 times in 22 different locations by 22 different GameSpriteFlyweight instances.

The Flyweight pattern can potentially drastically cut down on the amount of memory that is used to store sprites. The more identical objects that exist on the screen at one time, the more this savings is realized. The flyweight pattern is also used on the sheild system where there are nearly 200 identical sheild bricks arranged to protect the player from bombs.


SpriteBase
Abstract Class
LinkNode
GameSprite
Class
SpriteBase
pGameSprite
GameSpriteFlyweight
Class
SpriteBase
Fields
pos
Methods
Draw
getGameSprite
getPos
setPos
gameSpriteFlyweight
GameObject
Abstract Class
GameObjectCompositeRoot
Fields
pos
Methods
getGameSpriteFlyweight
setGameSprite
Nested Types

3 Game Object System

The Game Object System controls the actors within the game. As with the other systems in this game, it utilizes the Manager pattern to manage a collection of Game objects.

3.1 Game Object

The Game Object is a class that can represent any entity in the game. Typically, this is a physical actor that is drawn on the screen and collides with other actors. But it can also represent images that do not interact with actors or areas that do interact with actors but have no image.

<u>Visible Game Objects that interact:</u> Aliens, Player, bullets, bombs, sheilds

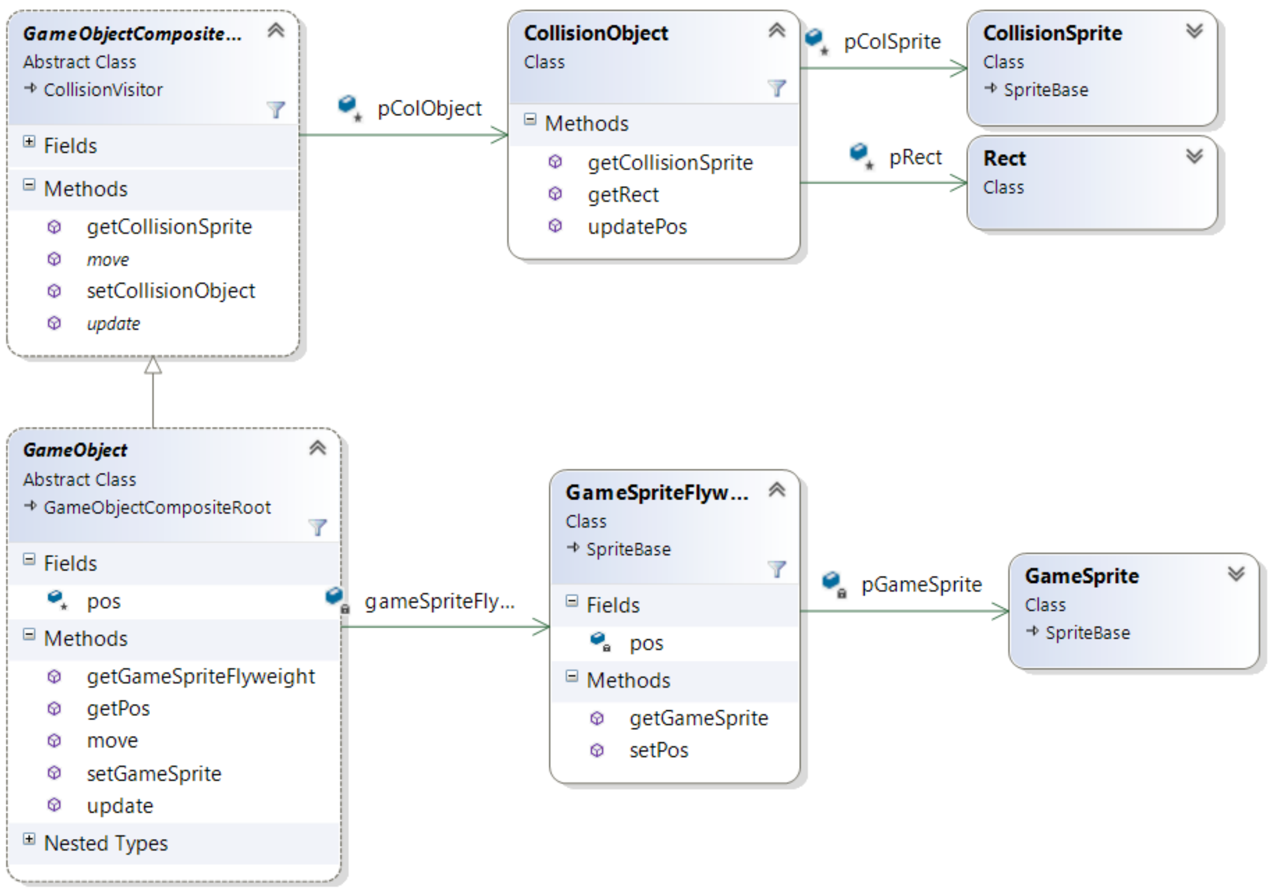
These Game Objects are all visable and make up the bulk of the game. They move around the screen and interact with each other.

<u>Invisible Game objects that interact:</u> Walls, Ceiling, Floor

These objects are invisable but still interact with the actors on the screen. Aliens and the Player cannot move past the walls. Bullets and Bombs do not go through the ceiling or floor.

<u>Visible Game objects that do not interact:</u> Explosions, Extra Lives

These objects are drawn on the screen but do nothing to impact the game. These typically exist as information or accents. The explosions are display when an actor is killed. They remain on the screen for a short period of time and allow other actors to pass through them. The extra lives are player ships that are drawn in the lower left corner of the screen to indicate how many lives a player has remaining. Additionally, all of the game objects that are displayed in attract mode do not interact.



The Game Object is a derived class of the GameObjectCompositeRoot class. This class will be discussed in the next section. For now, these two classes can be view as one Game Object. The Game object is an aggregation of a GameSpriteFlyweight and a CollisionObject. The Collision object contains the Collision Sprite and the bounding box and will be discussed later in this document. The GameSpriteFlyweight contains the image that will be drawn on the screen to represent the Game Object.

Game objects move around the screen using the "move()" method.

<u>Method:</u> move(float x, float y)
The move method causes the game object to move from its current position on the screen by "x" pixels horizantally, and "y" pixels vertically.

The new location is propigated to the GameSpriteFlyweight and the CollisionObject so that the image and the bounding box move to the new location of the game object. The new position is passed from the game object when the "update()" method is called. The "update()" method is called for this and all other game objects once each frame by the game object manager.

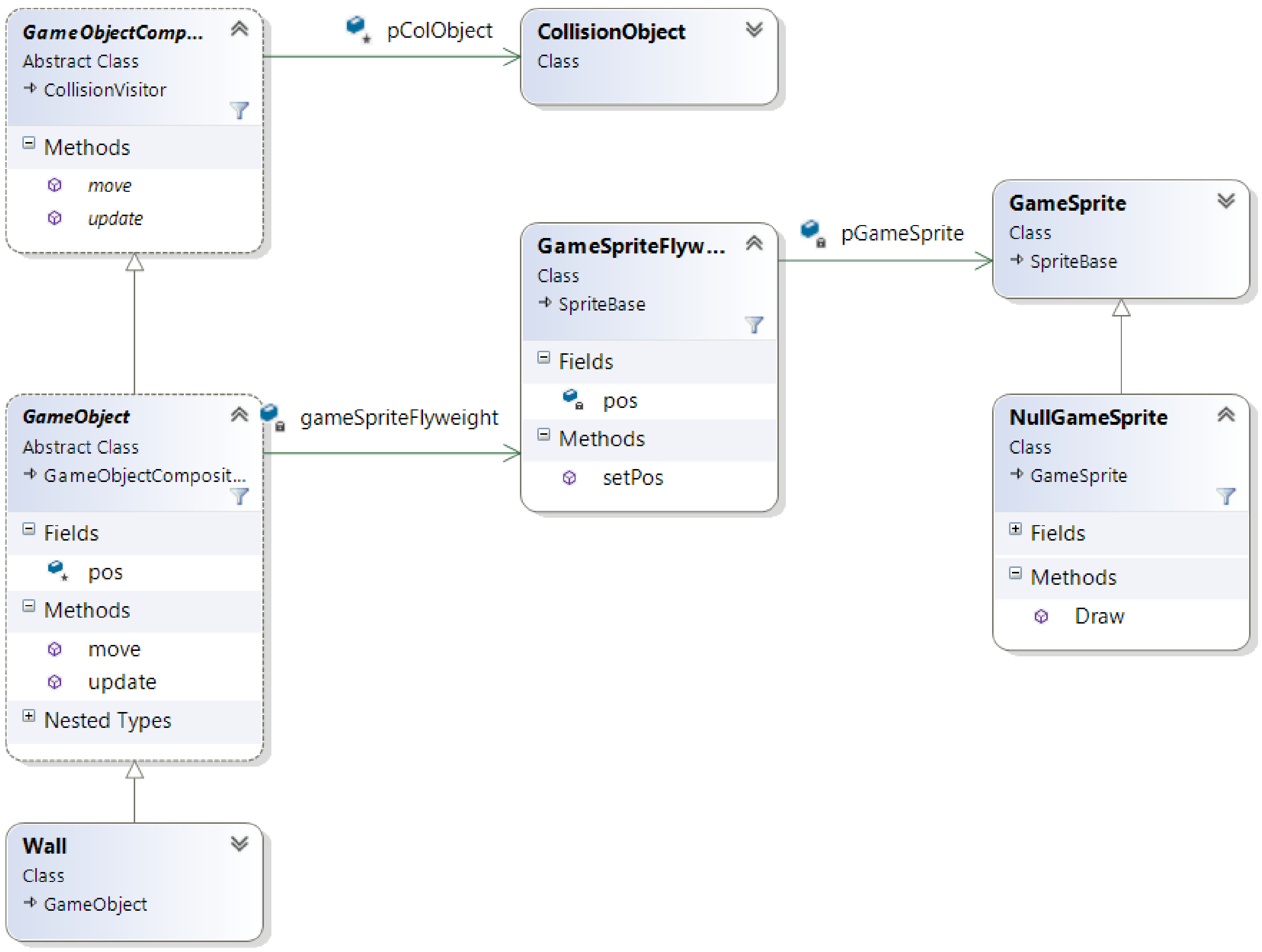
3.1.1 Null Object Design Pattern

Visible Game Objects that interact with each other do so through the GameSpriteFlyweight and the CollisionObject respectively. The image on the screen is managed by the GameSpriteFlyweight and the physical bounaries of the object are managed by the CollisionObject. But some Game Objects do not contain images and thus contain no sprite, and others do not interact and thus contain no Collision object. But it is still desirable for these Objects to behave in a uniform manner.

The "update()" and "move()" methods have calls to both the CollisionObject and the GameSpriteFlyweight. So when an object does not contain one of these objects, a mechanism needs to be in place to ensure methods are not called for an object that does not exist.

For instance, the Wall is a GameObject that contains a CollisionObject but no GameSpriteFlyweight because there is nothing to draw on the screen. Although the wall does not move once it is initially positioned, it still has a position attribute and a "move" and "update" method that act on the GameSpriteFlyweight and the Collision object. So when the GameObjectManager goes through and calls the "update()" method for all GameObjects, the Wall game object must ensure that it does not try setting the position of the non-existant GameSpriteFlyweight.

The solution to this problem is the Null Object Design Pattern.



One possible, albiet undesirable, solution would be to have the GameObject check that the GameSpriteFlyweight is not null before performing any operations on it. But this adds complexity to the method and creates non-uniform behavior. In the Null Object Pattern, a specialized GameSprite is created called the NullGameSprite. This Class contains all the same methods of a normal GameSprite object but performs no action when the methods are called.

In this example, the GameSpriteFlyweight is pointed to the NullGameSprite instead of a real GameSprite object. The NullGameSprite overrides the "Draw()" method of the GameSprite and draws nothing to the screen. In this approach, the GameObject does not need to be modified to handle the special cases where no GameSpriteFlyweight or CollisionObject is needed.

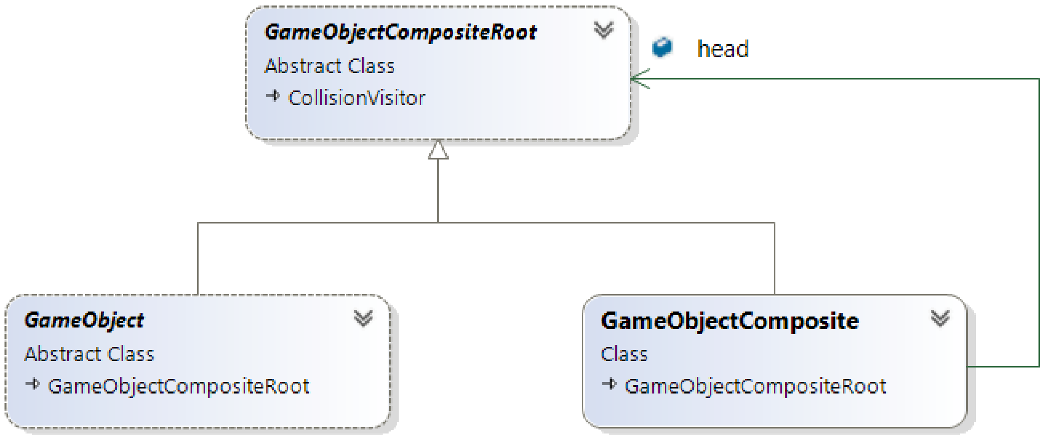
3.2 Game Object Heirarchy

Similar to the SpriteManager and SpriteGroup architecture, the GameObjects should be organized into groups. Specifically, the sheilds, aliens, and bombs each make up their own GameObject group. Within each group, the objects can be further subcategorized thus leading to a neccesity for a GameObject heirarchy.

3.2.1 Composite Design Pattern

All of the GameObjects have a position on the screen and a bounding box for collision detection, but some of the exist in groups and it is useful to be able to treat them as a group. As discussed earlier, the GameObject class is really a combination of a GameObject derived class and a GameObjectCompoiteRoot class. The GameObjectCompositeRoot contains the position attribute and the abstract declaration of the "move()" method to control position. This setup represents a single GameObject within the game.

Another implementation of the GameObjectCompositeRoot is the GameObjectComposite class. The GameObjectComposite is treated similarly to a single GameObject except that it contains a list of GameObjectCompositeRoots each of which can also be a composite. The GameObjectComposite contains all the methods of a single GameObject. The GameObjectComposite implementats these methods by calling the same method in all of the objects in its list. If effect, an entire heirarchy of game ojects can be moved uniformly simply by calling the top level "move()" method. The method will filter down to all of the lower level GameObjects and GameObjectComposites.



This approach is an implementation of the Composite Design pattern. The Composite pattern solves the problem of how to issue commands to an entire group of GameObjects but has a secondary benifit as well. The bounding boxes used for collision detection can be extended to cover entire composite heirarchies instead of just single GameObjects. This benifit will be discussed in the Collision section of the document.

3.3 Alien and Shield Grid

In this game, all GameObjects on the screen are part of a GameObject composite heirachy, even the GameObjects that are alone in their group. The player's ship for instance, is the only ship on the screen and therefore does not need to be part of a heirarchy but it is for consitancy. Even if there is never a need for multiple ships on the screen, the possibility exists and this approach supports the open-closed principle of object oriented design stating that software should be open to expansion and closed to modification.

The Sheild grid however, fully utilizes the composite structure. The top level node is the sheild group and is a composite of individual sheilds. Each sheild is in turn a composite of individual sheild bricks that can be destroyed one at a time to deteriorate the sheild.

The alien grid also fully utililizes the composite structure. The top level node of the alien grid is a composite of columns. Each column is a composite of five individual aliens. Again, this facilitates collision detection between the aliens and the bullets. It also faciltates uniform motion of the alien grid

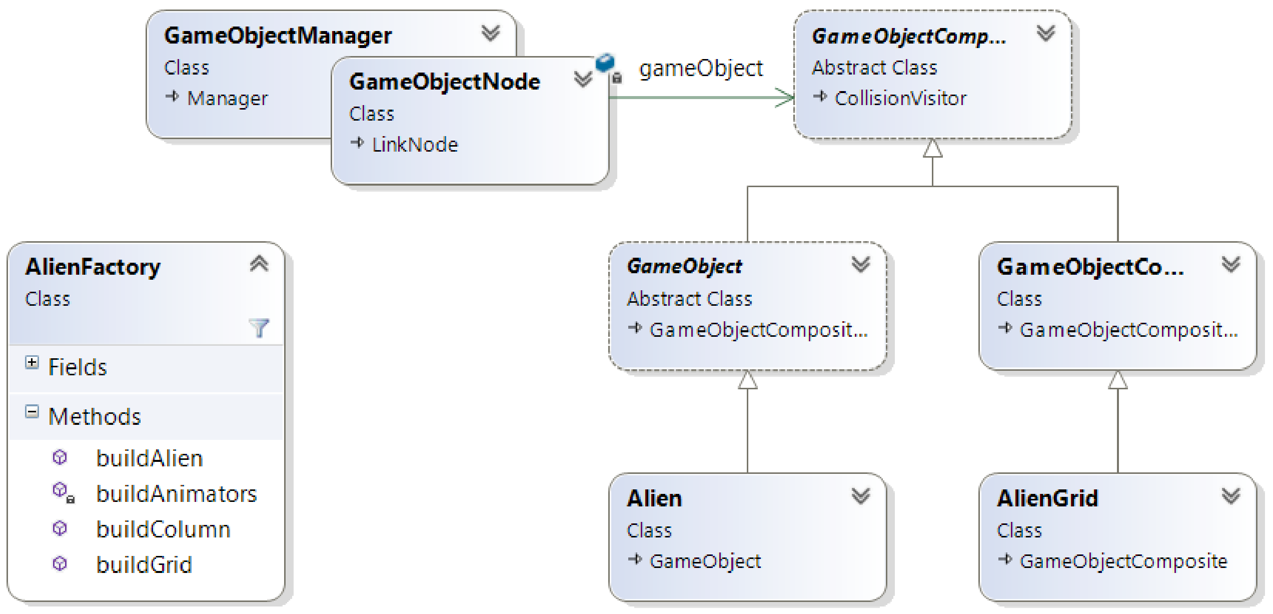
In the game, all of the aliens move across the screen in uniform steps. When the edge of the grid reaches the edge of the screen, the grid uniformly drops down a level and reverses direction. When any one of the aliens collides with a wall, it sends a notification to the top-level node which causes the entire grid to drop down and reverse direction.

3.3.1 Factory Design Pattern

The top row of the alien grid consists of 11 Squids, the next two rows are 22 crabs, and the bottom two rows are 22 Octopuses. The grid is organized in columns to support optimization of the collision detection system. All of this information is useless to the GameObjectManager and to any of the other classes in the game. Yet, one of these classes needs to build the grid to these exact specifications.

If another level of the game has a different layout, or is composed of new types of aliens, or we decide to organize the aliens into rows instead of columns, the class that creates the alien grid needs to know this.

The GameObjectManager manages a list of GameObjects all of which contain the same methods so it would not make sense for it build the alien grid. In object oriented design, the single-responsibility principle states that any class should have one and only one responsibility. As such, the game requires a class whose sole purpose is to build the grid of aliens and return the GameObjectCompositeRoot for the top level of the grid. This function is accomplished by the AlienFactory class and is an implementation of the Factory Design Pattern.



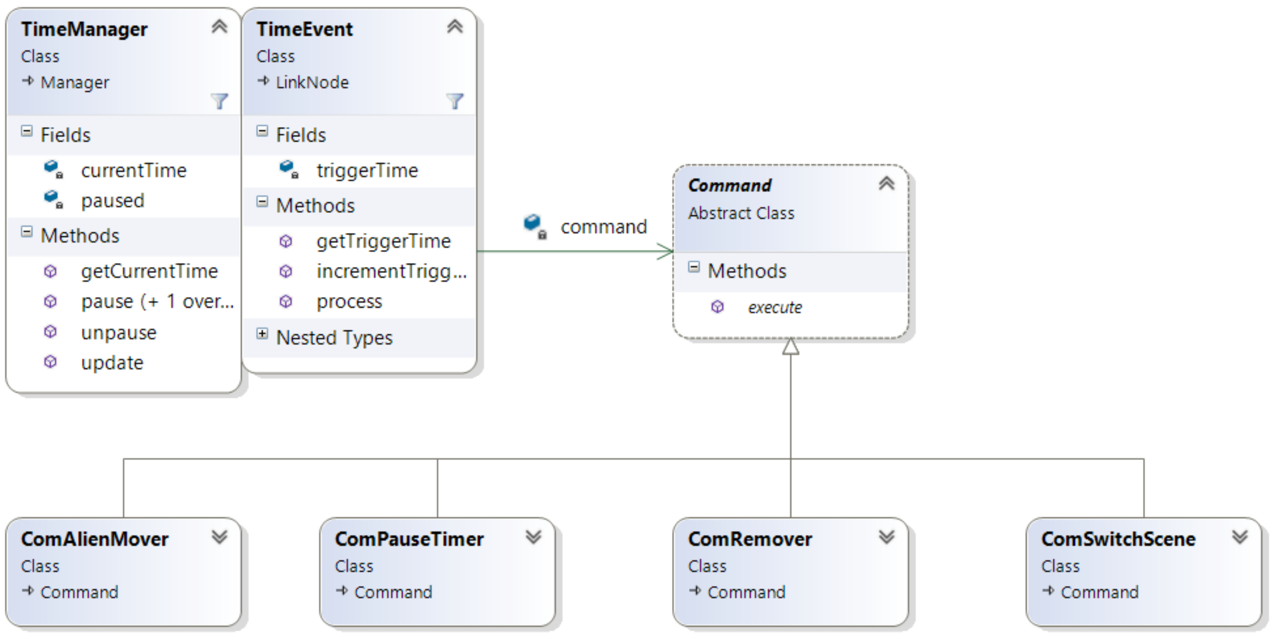
In this example, the AlienFactory has the knowledge of the different alien types and the layout so that it can create the grid and build each alien complete with a GameSpriteFlyweight, CollisionObject, and AnimationSprite.

4 Time Event System

Events that occur during the game can be triggered by one of serveral possible stimuli:

- The event is triggered by a user input
  player moving the ship from side to side
- The event is triggered by an interaction between two game objects
  bullet colliding with an alien triggers an explosion
- The event automatically occurs every frame
  The movement of a bomb toward the bottom of the screen
- The event is triggered at a specific time
  The alien grid steps to the side periodically (roughly once per second at the beginning of the game)

Events that occur at a specific time are managed by the Time Event System. The time event system starts with a manager class called TimeManager that keeps track of time and contains a list of TimeEvents that are scheduled to execute at a specific time. Each TimeEvent executes a command on a GameObject or system.



4.1 Time Server

The root of the Time Event system is the TimeManager class which acts as a time server for the game. The Azul game engine provides absolute time at the beginning of each frame. This data is passed to the TimeManager which provides the current time to other systems through an accessor method.

The time manager contains a list of TimeEvent nodes that are each scheduled to execute at a specific time. Unlike other manager classes, the TimeManager stores its list as a priority queue. Whenever current time is updated in the TimeManager, it checks the first TimeEvent nodes in the priority queue. If the new current time is past the trigger time of the TimeEvent, the the event is executed and removed from the queue.

The TimeManager contains the following unique methods:

Method: getCurrentTime()
Description: This method returns the current time stored in the TimeManager.

Method: update(float currenttime)
Description: This method is called to update the current time stored in the TimeManager. After the time is updated, the method checks if any of the TimeEvents need to be executed.

Method: pause()
Description: This method pauses the execution of all TimeEvents. As the current time updates, no TimeEvents are executed.

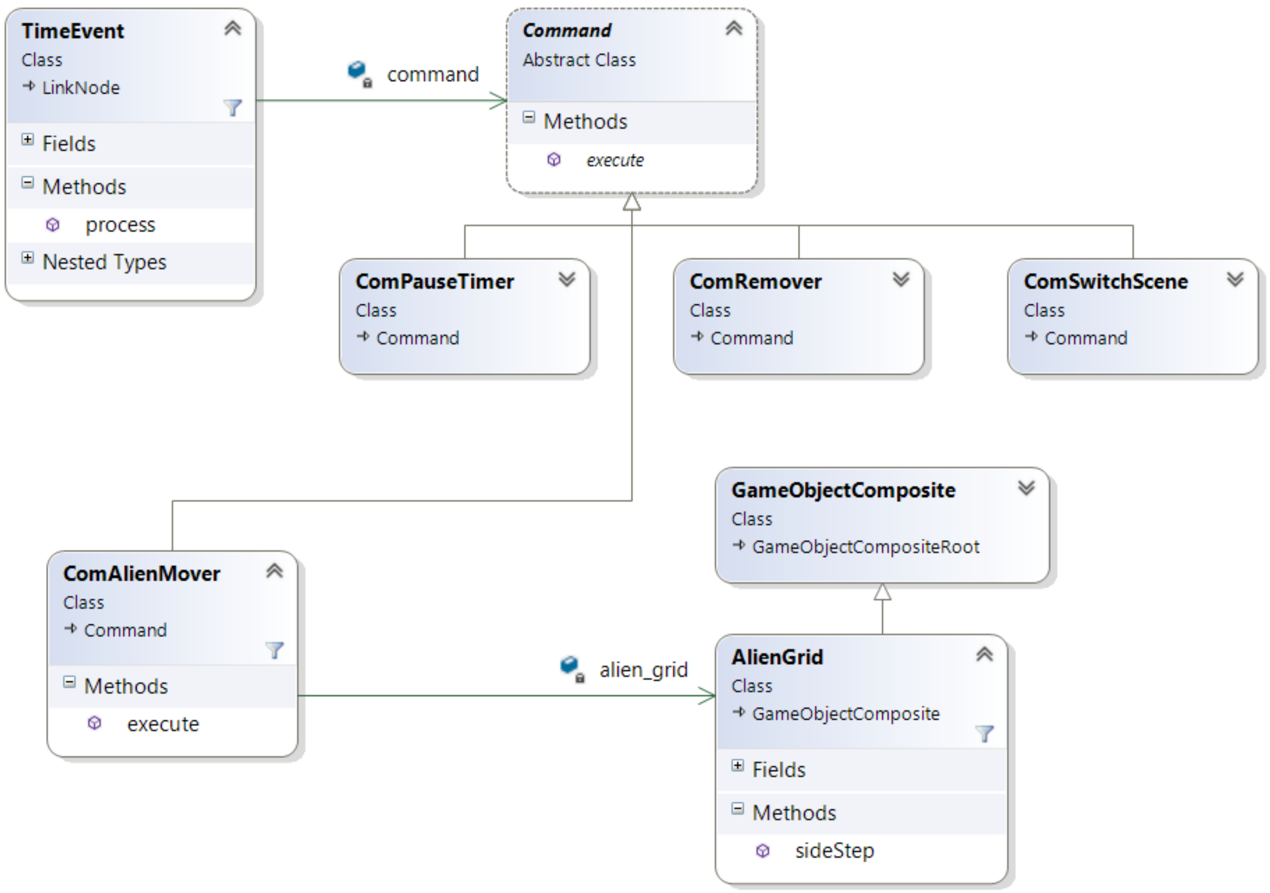
Method: unpause()
Description: This method unpauses the execution of all TimeEvents. When this method is called, the delta between the current time and the time that pause was activated is calculated. Then the method iterates through the entire queue and adds the delta to all of the TimeEvents. This effectively resumes execution of TimeEvents where it left off when the pause feature was activated.

4.1.1 Command Design Pattern

Once a TimeEvent is executed, there is a question of how to execute the event. Numerous different types of events could occur but the TimeEvent object needs to be generic enough that it could be called by the TimeManager without any specialization.

To enable the ability for a TimeEvent to execute any type of event, the Command Design Pattern is used. In the command pattern, the TimeEvent node contains a refernce to a Command object. The command object contains only one method called "execute" that is used to execute the command.

The behavior of the command itself is encapsulated in a class that derives from the Command class using interface inheritence. Any subclass of the Command class must implement an "execute" method that performs the event.



The example above shows how the command pattern is used in conjunction with the TimeManager to trigger various events during the game, specifically, the alien grid side-stepping across the screen.

When a TimeEvent is executed that contains an Alien Mover command, the command is executed. The Alien Mover command contains a reference to the alien grid and calls the AlienGrid "sideStep()" method to cause the aliens to move to the side. Once the execute method of the command is complete, the command may push itself back into the TimeManager list to be executed again at a later time.

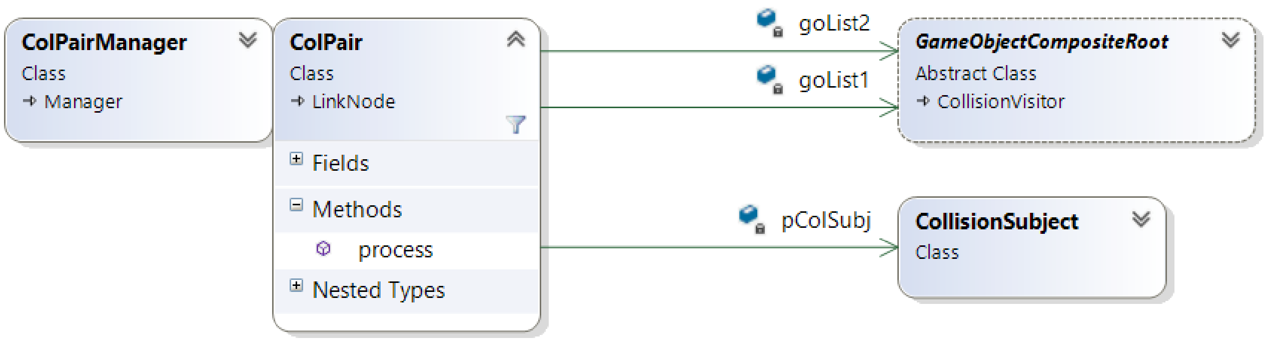
5 Collision System

The collision system continually tracks all of the combinations of GameObjects on the screen that can collide with each other. The following combinations of GameObjects can collide:

- Player VS Bomb
- Floor VS Bomb
- Shield VS Bomb
- Alien VS Bullet
- Ceiling VS Bullet
- Shield VS Bullet
- Alien VS Wall
- Player VS Wall
- Alien VS Shield

5.1 Collision Pair Manager

The composite nature of the GameObjects enables the Collision system to monitor pairs of GameObject groups. For each of the above combinations, the Collision System manages a pair GameObjectCompositeRoot objects. This neccessitates another manager called the CollisionPairManager. The CollisionPairManager class manages a list of CollisionPair nodes. Each node is an aggregation of two GameObject composites that can collide with each other.



During each frame, every ColPair is evaluated. This means that the position of every object in one list must be compared with the position of every object in the other list. If the positions overlap, then a collision has occured and the proper processing is set off. This presents an performance problem. It is computationally demmanding to compare every object that can collide with every other object that it can collide with. This is where the heirarchical nature of the GameObject composite demonstrates its value.

Each GameObject is contained by a bounding box. When the bounding boxes of two objects intersect, then they have collided. But each GameObjectComposite also has a bounding box that is a union of all the objects it contains. When an object (such as a bullet) is tested against the 55 aliens in the grid, the bounding box of the bullet is first tested against the bounding box of the entire grid. If the two boxes do not intersect, the there is no need to test the bullet against each individual alien. Only when the bullet intersects with a composite bounding box is it tested against the individual bounding boxes. Since the alien grid is divided into 11 columns with 5 aliens each, a bullet must only be tested against a maximum of 17 boxes (1 grid, 11 columns in the grid, and 5 aliens in the column) instead of all 55 aliens.

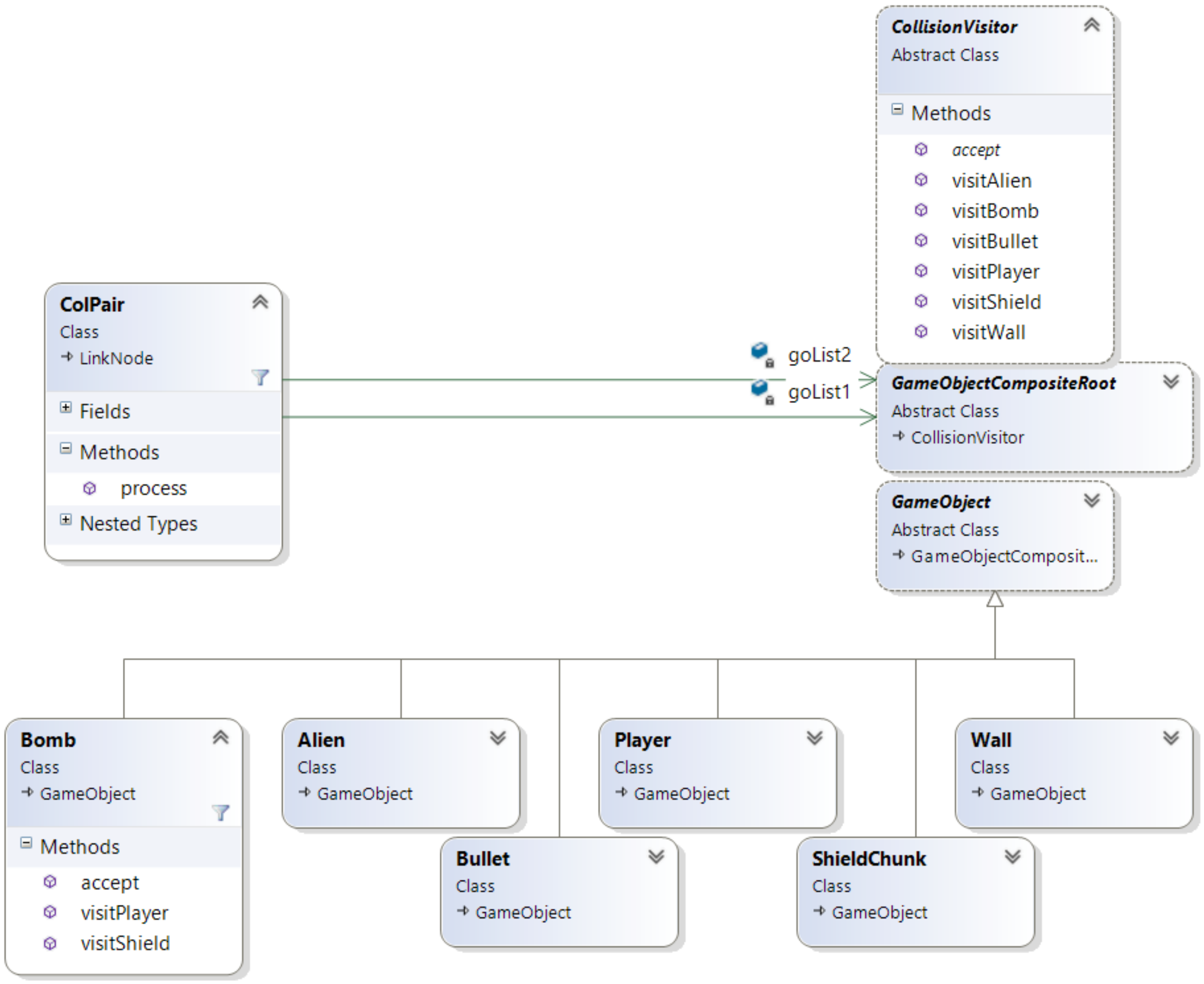
The same is approach is taken with the shields. The entire group of shields in divided into 4 individual shields, then further subdivided into 49 bricks per sheild. Reducing the maximum number of tests for a shield collision from 196 to 54.

5.1.1 Visitor Design Pattern

When two objects collide, it is important to determine which two types of objects have collided and trigger the appropriate action. If there was only one type of object in the game and it could only collide with other objects of the same type, then there is only one possible combination of objects that can collide. As the total number of object types increases, the number of different collision combinations increase proportionally to the square of the total number of objects. If we just consider bullets, bombs, aliens, and the player, that is 16 different possible combinations that may need to be handled. The game also contains sheilds, walls, ceiling, and floor bringing the total number of possible combinations up to 64.

One possible approach to handle all these combinations is a complicated conditional structure or perhaps a switch statement with embedded switch statements in each case. This would create code that is difficult to read and debug and would present a unique maintenance challange if changes or improvements need to be made. Additionally, most combinations will never occur and dont need to be compared.

A better approach to this problem is the Visitor design pattern. In the Visitor pattern, all possible objects that can collide inherit from a CollisionVisitor base class. The Visitor base class contains one method for each possible object type. In this way, each object type inherits the method that compares it with all other object types. Furthermore, the derived object can implement only the methods that compare it to objects that it could possibly collide with. As a result, the derived object does not need to consider impossible collision cases (such as the Sheild VS the ceiling).



Consider the Bomb in the above class diagram. The bomb can collide with three possible other object types, the Player, the Shield, and the floor (which is handled by the Wall class). When a collision is detected between a bomb and a shield, the ColPair will call the shield's "accept()" method and pass it the bomb object as a parameter. The Shield has no knowledge of what the other object is, but it is gaurenteed that the other object has a visitSheild method, so the shield calls the bomb's "visitShield()" method. Within this method, both object types are known and the specific actions that need to occur can be triggered.

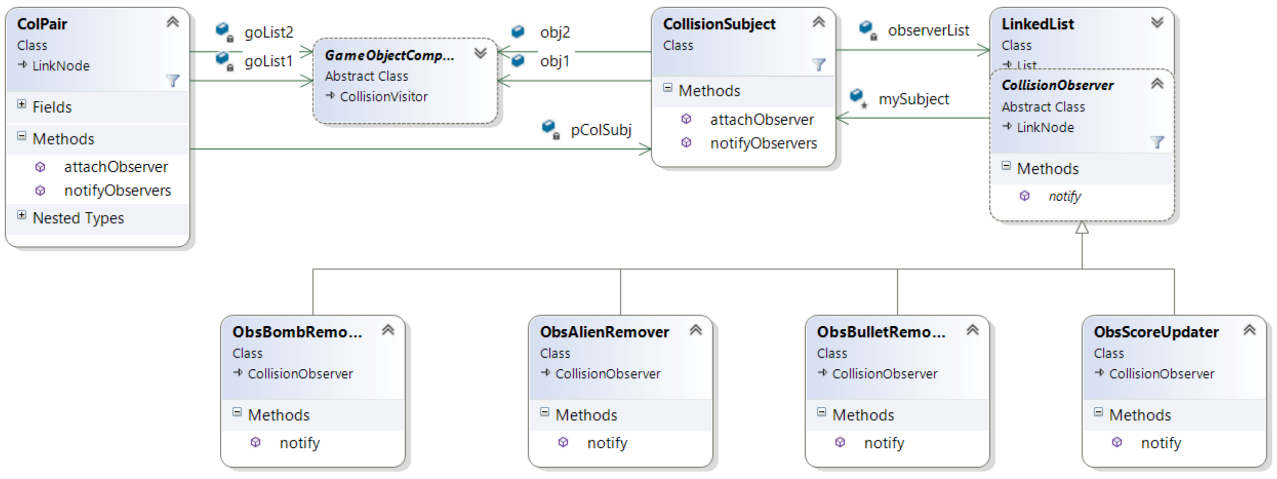
5.1.2 Observer Design Pattern

Once the two colliding object types are known, the collision must be processed. An action of some sort is typically required as a reaction to the collision. This could be one action or many actions depending on the two colliding types and the state of the game. It would be unneccesarily complicated for the colliding objects or the ColPair to keep track of which other systems need to know about the collision. This would violate the single responsibility principle of object oriented design. Therefore, a method is needed to make sure that all systems that need to respond to a collision know about it.

Consider the case where a bullet collides with an alien. Several events need to occur:

– the bullet must be removed from the game
– the Alien must be removed from the game and replaced with the "splat" graphic
– the "Splat" graphic must be removed after a short period of time.
– While he splat graphic is visible, nothing should be able to collide with it
– The grid of aliens should speed up
– The player's score should be incremented by the value of that alien
– The player's ship must be notified that the bullet is destoyed because only one bullet can exist at any time.

It would be impractical for the alien, or bullet, or the ColPair to know that all these actions need to take place and inform the appropriate system. The solution to this problem is the Observer design pattern.



The Observer design pattern encapsulates the notifications that need to be sent to other systems. For any event, the systems that need to be informed are known as the observers. Each observer derives from an Observer base class and must implement a "notify()" method. All observers register with the ColPair so that their "notify()" method will be called when the collision event occurs. In this manner, the ColPair or the colliding objects do not need to know what will happen when they collide. They simply inform their observers and the observers perform the neccessary action.

When the Bullet/alien ColPair is initially created, four observers are registered:

- the AlienRemover
  - This observer handles the removal of the alien and the resulting "splat" graphic and explosion sound.
- the BulletRemover
  - This observer handles the removal of the bullet from the game
- The ScoreUpdater
  - Sends a command to update the score with the value of the alien that was just destroyed
- The BulletReady notifier
  - Notifies the player's ship that it can now fire another bullet.

6 Input System

The input system contains an InputManager class that listens for keyboard inputs and triggers the corresponding actions. Only 5 keys are used in this game.

To control the player:

- The left arrow and right arrow
  - Move the player left and right across the screen
- The spacebar
  - Fires a bullet

To Start the game during the attract-mode screen:

- the number 1 or 2
  - Selects one or two players

For debugging purposes, the number 3 toggles the drawing of the bounding boxes on the screen. This aides in the development of the collision detection system but should not be included in the final product.

The Input manager is a simple singleton class that is called once each frame and handles two types of keyboard events: when a key is pressed, and when a key is held down.

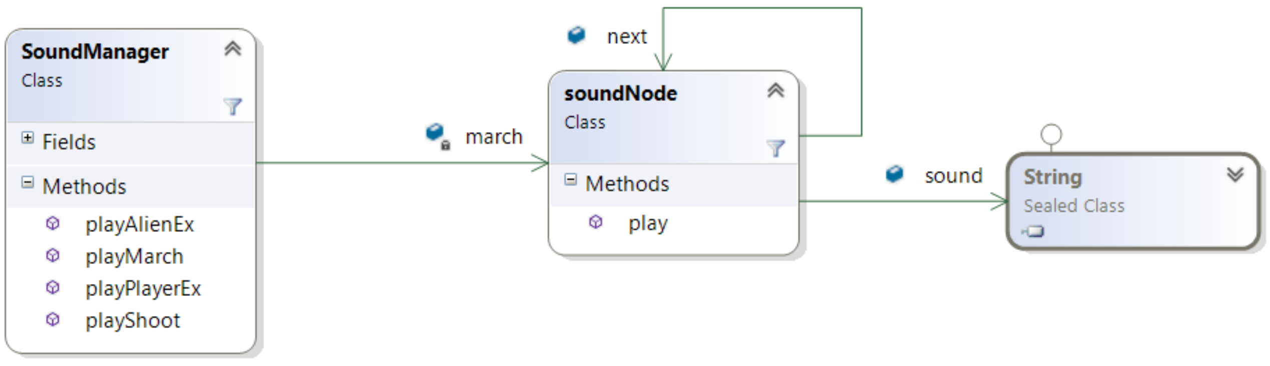
The movement of the player left and right is triggered by a key being held down. A simple check sets a boolean to TRUE is the key is pressed or FALSE when the key is not. When the key is down, the player's "move()" method is called each frame to create a smooth movement.

For events such as player selection and firing a bullet, a single key press should trigger one event. But, since the keyboard is processed once each frame, a single key press can linger for many frames potentially triggering multiple events. Some simple logic handles this case. The key position is monitored and its current value is stored for the next frame. If the current value of the key is TRUE and the previous value is FALSE, then the event is triggered.

7 Sound Manager

The Sound Manager is a very simple singleton class that has a method for each sound that plays durring the game. When an event happens that requires a sound, the corresponding method in the sound manager is called.

Each time the aliens step across the screen, a sound is played. This sound cycles between four individual sounds that are played in repeating order. This is handled by a circular linked list of sound nodes. Each of the four sounds is placed in a node that contains a "play()" method. When the alien side-step sound is requested from the sound manager, it will play the current sound in the list, then increment to the next sound for the next time the sound is requested. The sound is stored in the SoundNode as a string which contains the filename of the sound.



8 Scene Manager

The Scene Manager controls the mode of the game. The game has the following modes:

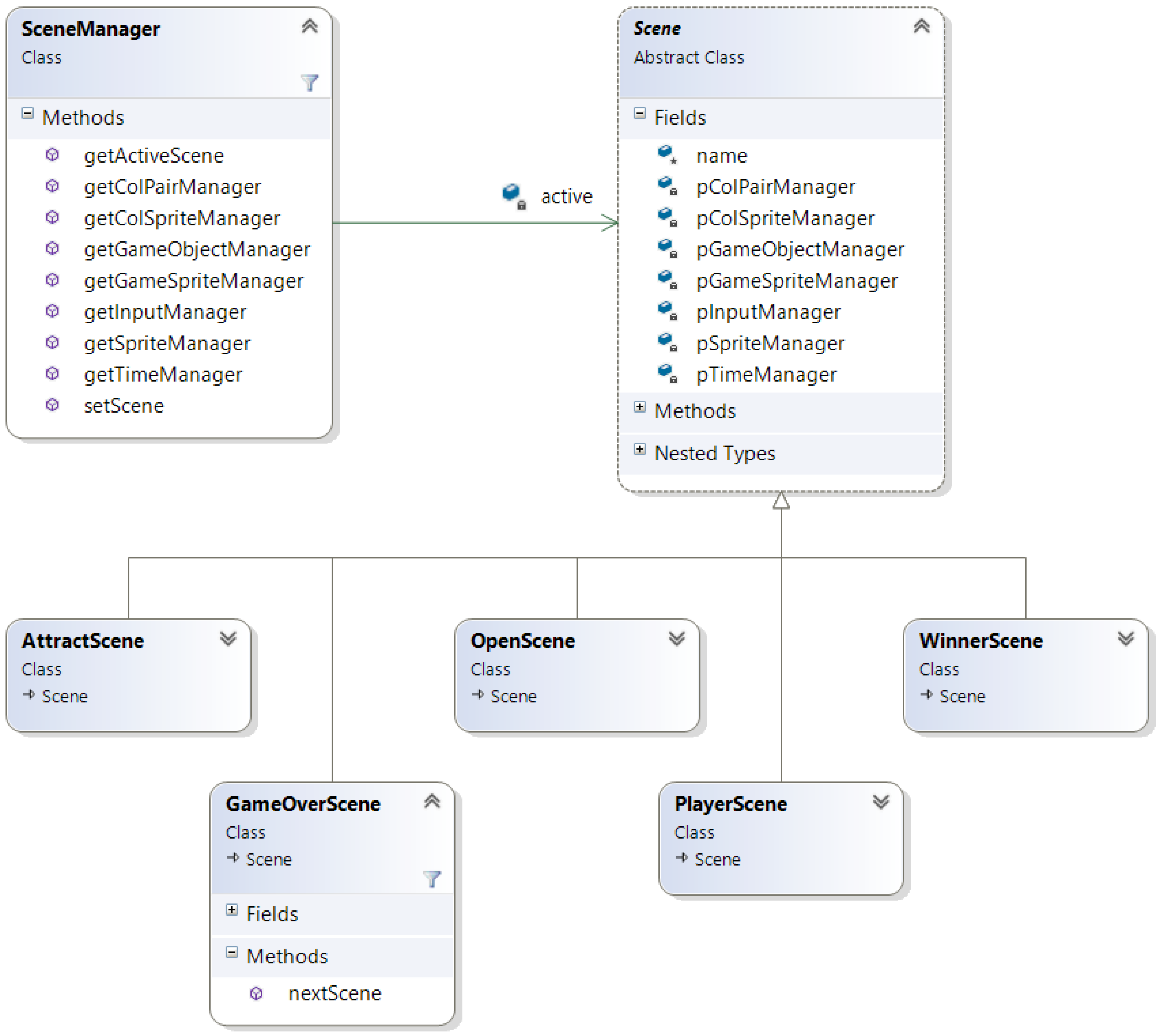
- Opening screen
  - Displays the title of the game for several seconds
- Attract mode
  - Displays the value of each alien and instructions to start the game
- Gameplay mode
  - The actual game can be played in 1 or 2 player mode. In 2-player mode, the scene switches back and forth between the two players until they both run out of lives.
- Game over screen
  - Displays a "game over" message when both players have run out of lives
- Winner screen
  - Displays a winner message when one of the players wins the game.

8.1 State Design Pattern

Switching between each scene can be a complicated challege. The Scene system is essentiall a state machine that switches between different modes based on a set of rules. An example of a possible path through the states is:

1) The Game starts in opening mode
2) Transitions to attract mode after 3 seconds
   1. player selects 2-player mode
3) Transition to player 1 gameplay
   1. player 1 dies
4) Transition to player 2 gameplay
   1. Player 2 dies
5) transition to player 1 gameplay
   1. Player 1 wins the game
6) transition the winner screen
   1. wait 3 seconds
7) transition to attract mode

Due to the complex set of rules that dictate which state comes next at any given time, writing the code to transition between states is a challenge. The solution in this game is the State design pattern.



In the state design pattern, the behvior of each state is encapsulated in its own class that derives from a State base class. Each derived state class contains all the code necessary to execute the behavior of that state. Additionally, it contains the rules to switch to the next state. In this game, each state (or scene) has a "nextScene()" method that determines which scene should come next based on the set of rules that governs the state machine.

The player scene contains an instance of all the managers that contain unique data. The following managers contain data that is unique to each player and is therefore included in the playerScene class:

- ColPairManager
- CollisionSpriteManager
- GameObjectManager
- GameSpriteManager
- SpriteManager
- InputManager
- TimeManager

Each of these classes were originally implemented as a singleton but the "getInstance()" method was replaced with a method that retrieves the player's instance of each of these classes from the PlayerScene state class.

9. Commentary

Designing and building a game of this magnitude presents several challenges and certain guidelines should be followed.

1) Work small problems at a time.
   The human brain can only keep track of so many things at once, but complex software such as this game may be tracking thousands of items at a time. It is imperitive as programmers that we tackle small problems first and then integrate these units together to solve the larger problems. I found it interesting that progress went slowly at first. But once most of the small problems were solved, the game came together very quikly and surprisingly easily.

2) Prototype implementations
   A large-scale solution to a problem will never work if you cant get it to work at a small level first. There are many creative ways to solve problems but they should be tested out in a simple and forgiving environment before they are implemented in the overall software.

3) Try out solutions
   Sometimes it is difficult to figure out how to start. In these cases it may be best to just start even without all the information or a good design. By working on the problem, you may figure out a better way of doing it that you would not have though of otherwise.

4) Manage time wisely
   Complex software takes time so you must give yourself enough time to do it right. Writing complex code means that you will spend a lot of time trying out ideas that never make it into the finished product.

5) Take care of yourself
   Sitting for a long time is bad for the body. Typing too much without breaks is hard on hands and wrists. As profesional programmers, we must take care of ourselves so that we can continue to program. That means proper posture, good ergonomic practices, and taking breaks every now and then