Proxy:

Use: Minimize the amount of data consumption by having multiple objects share/ reference the same resource(s).

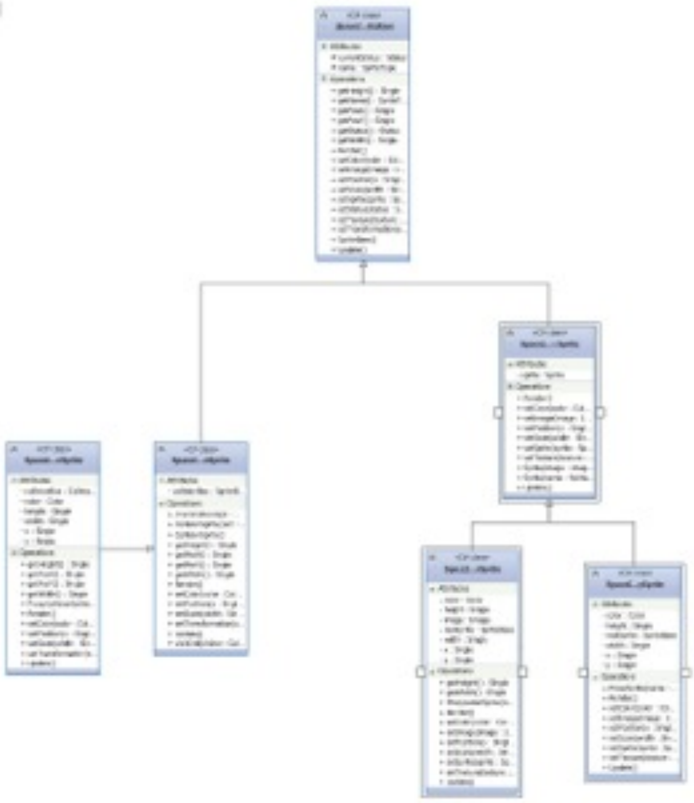
Where is it used?

Sprite Rendering:

The sprites are divided into two kinds, a real sprite and a proxy sprite. The real sprite holds a reference to fan Azul.Sprite object that is needed to draw images on the screen. The Azul.Sprite maintains a reference to a texture and a coordinate on the texture in order to draw the requested image or the texture in its entirety. The coordinates of each image are not maintained by the real sprite but are supplied by objects referencing the real sprite.

The proxy sprite maintains the same functionality as the real sprite but maintains a reference to the real sprite instead of the Azul.Sprite. The proxy sprite maintains a position and possibly a reference to an image coordinate but not the texture. During render loops, the proxy sprite sends its coordinates and image to the real sprite in order to have the sprite draw where the proxy sprite wants it to draw during update and render loops. By having each proxy instruct their respective real sprites on where to draw, the illusion of having many objects being draw is maintained by having only a handful of sprites perform this function.

Proxy sprites overwriting the real sprite only works when they update and render the real sprite during the render loop. As a result, the iterator responsible for instruction the sprites to update and render is placed inside the render loop.



Flyweight:

Use: Minimize the amount of data allocation by having multiple resources designed for reuse.

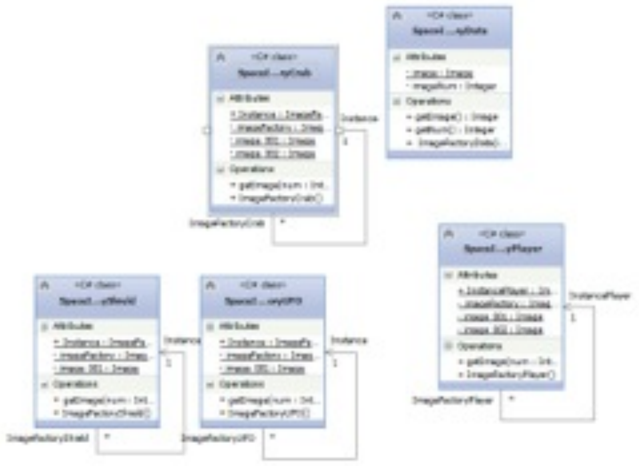
Where is it used?

Textures, Images, and Real Sprites:

The textures, coordinates on the textures, and the sprites used by the proxy sprites are all made to be reused by multiple objects. Factories maintain a list of each image, texture, and sprite. The resources are supplied for every new proxy sprite but the proxy spites don’t create a new instance of the images, textures, or real sprites.

The reuse of various resources helps to condense the amount of allocation for each resource. By having the resource being created on a list, factories can make a request to get the resources needed rather than having to continuously create new resources for every instance of the sprite.

The coordinates of texture images kept inside the Image class. The image class references the Azul.Rect, a component necessary for Azul.Sprite to set the image coordinates properly. Because Azul.Rect is also needed for the position and scale for the sprite, the Image class is used to avoid confusion in both the wringing of code and the acceptance from functions. The scale of functions is controlled by the struct, Size, and is used after the sprite has been created in order to avoid errors with size. Textures, needed by the Azul.Texture, are kept inside the Texture class. The various components are found using the enum, SpriteType, in order to acquire the necessary components.



«C# class»
SpaceI...yCrab
Attributes
+ Instance : ImageFa...
- imageFactory : Imag...
- image_001 : Image
- image_002 : Image
Operations
+ getImage(num : Int...
+ ImageFactoryCrab()
Instance
1
ImageFactoryCrab
*
«C# class»
SpaceI...yData
Attributes
- image : Image
- imageNum : Integer
Operations
+ getImage() : Image
+ getNum() : Integer
+ ImageFactoryData(...
«C# class»
SpaceI...yShield
Attributes
+ Instance : ImageFa...
- imageFactory : Imag...
- image_001 : Image
Operations
+ getImage(num : Int...
+ ImageFactoryShield()
Instance
1
ImageFactoryShield
*
«C# class»
SpaceI...oryUFO
Attributes
+ Instance : ImageFa...
- imageFactory : Imag...
- image_001 : Image
Operations
+ getImage(num : Int...
+ ImageFactoryUFO()
Instance
1
ImageFactoryUFO
*
«C# class»
SpaceI...yPlayer
Attributes
+ InstancePlayer : Im...
- imageFactory : Imag...
- image_001 : Image
- image_002 : Image
Operations
+ getImage(num : Int...
+ ImageFactoryPlayer()
InstancePlayer
1
ImageFactoryPlayer
*

State:

Use: Alter the behavior of the object to behave in a different matter

Where is it used?

Movement:

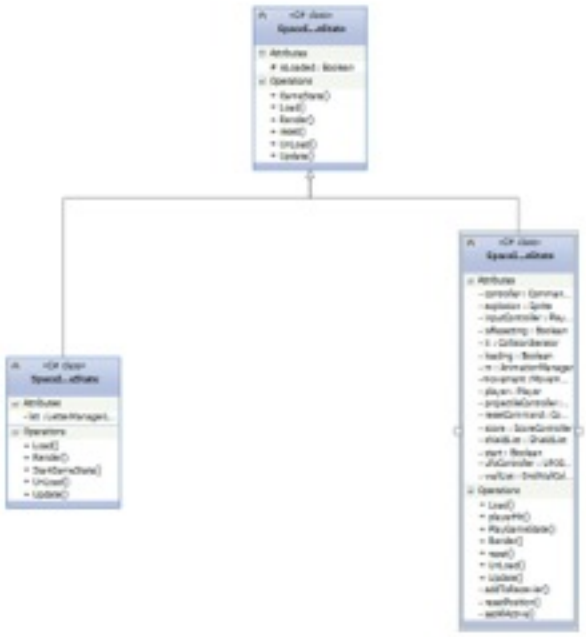
The controllers for the alien and player handle their translation by using a movement state. The movement state supplies the direction for the player/object to be moving. The direction can be changed by meeting the criteria for each object.

The grid of aliens movement is controlled by a movement controller that translates the aliens. The alien grid moves in a single direction continuously until its notified that it has collided with the wall. As a result, the movement controller reverses it controls to the opposite. After changing its state, it ignores any notification of collision with the wall until the wall collision is not occurring.

The player movement controller, similar to the grid, changes its direction but sets the state based on input from the player and collision detection. The player changes its state to move left or right based on the respective arrows pressed. However, the state is set to stationary when a collision is detected in the direction of movement.

The direction of movement is not the same as the speed/distance of the movement. The speed and distance of each moving object is influenced by how the movement is being controlled by the objects applying translations.

The game modes are segmented into a series of state patterns but are influenced by the current state of the game. Players are able to shoot and avoid enemy fire fire while inside the play gamestate. However, the mode is reset when the player runs out of lives. Two player mode only has the game state check to see if both players are active before it determines that the game is over. After exiting or when starting the program, the player is introduced to the start screen. Pressing one or two will determine the multiplayer status and begin the game.


«C# class»
SpaceI...eState
Attributes
# isLoaded : Boolean
Operations
+ GameState()
+ Load()
+ Render()
+ reset()
+ UnLoad()
+ Update()
«C# class»
SpaceI...eState
Attributes
- lm : LetterManager...
Operations
+ Load()
+ Render()
+ StartGameState()
+ UnLoad()
+ Update()
«C# class»
SpaceI...eState
Attributes
- controller : Comman...
- explosion : Sprite
- inputController : Play...
- isShooting : Boolean
- li : CollisionIterator
- loading : Boolean
- m : AnimationManager
- movement : Movem...
- player : Player
- projectileController : ...
- resetCommand : Co...
- score : ScoreController
- shieldList : ShieldList
- start : Boolean
- ufoController : UFOS...
- wallList : EnemyWallCol...
Operations
+ Load()
+ playerHit()
+ PlayGameState()
+ Render()
+ reset()
+ UnLoad()
+ Update()
- addToRecevier()
- resetPosition()
- spawnArmy()

Observer:

Use: Subject notifies its dependency on any state change.

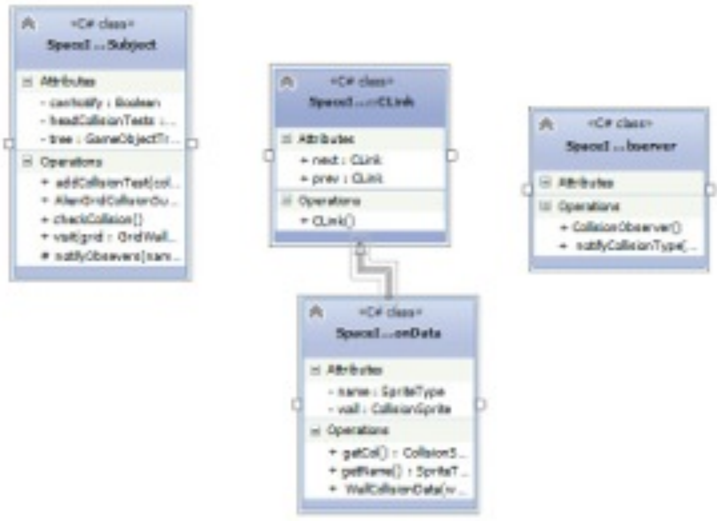
Where is it used?

Alien Movement/Player Movement:

The alien and player movement observers are responsible for notifying their respective movement controllers for any collision with wall colliders. The subjects test collisions between the alien grid and each wall and the player's collision boxes against the wall. When each detects a collision, both notify their observers about the type of collision that has occurred. The observers take this information and use it to notify the movement controllers about the controllers. The observers and the subject only notify about collision and may choose to react or not react to the current collision.

The subject of the grid collision is concerned with collision from the wall components. When detected, the subject notifies an observer looking for a collision from the wall with the alien grid. After receiving the collision type, the observer instructs the movement controller to change the direction of the aliens. The movement controller begins to ignore any other detection of collision until the collision with the wall is no longer occurring.

The victory subject also checks itself against the alien grid. The victory subject is checking against a specific collision box placed near the bottom of the screen, When the alien grid collides with this object, the observer is notified about the hit. In the play game state, the game is notified about this change and instructs itself to end the game due to the aliens winning the game.


«C# class»
SpaceI ... Subject
Attributes
- canNotify : Boolean
- headCollisionTests : ...
- tree : GameObjectTr...
Operations
+ addCollisionTest(col...
+ AlienGridCollisionOu...
+ checkCollision()
+ visit(grid : GridWall...
# notifyObservers(nam...
«C# class»
SpaceI ... CLink
Attributes
+ next : CLink
+ prev : CLink
Operations
+ CLink()
«C# class»
SpaceI ... bserver
Attributes
Operations
+ CollisionObserver()
+ notifyCollisionType(...
«C# class»
SpaceI ... onData
Attributes
- name : SpriteType
- wall : CollisionSprite
Operations
+ getCol() : CollisionS...
+ getName() : SpriteT...
+ WallCollisionData(w...

Visitor:

Use: introduce components without expanding original striation

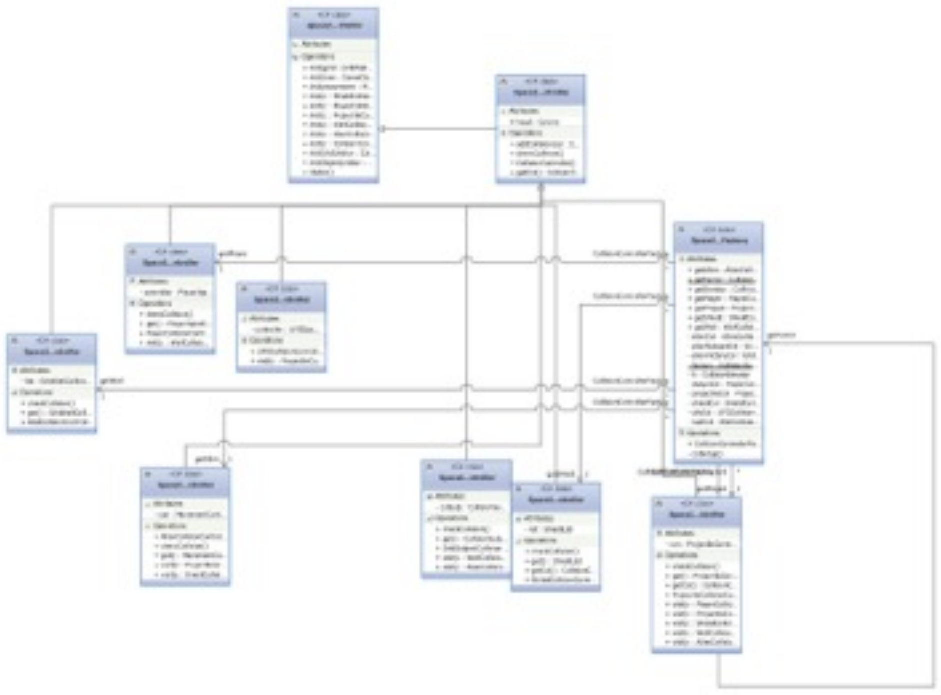
Where is it use?

CollisionController:

Each object being tested for collision is placed inside collision controllers. Each Collision controller inherits from the visitor pattern to allow each collision component to interact with each other. Each collision controller adds the collisions that its testing against to a list and visits each object on the list. Each collision controller doesn't know what is going to visit but each collision does not what to expect from each visitor and how to begin testing collision for each respective visitor.

Each object referenced in each respective collision component is expecting a specific type of collision. The collision components inheriting from the visitor is designed to help introduce each component without requiring a complex or explicit call for each collision component. Each collision component is allow to visit each other and the collision component can be setup to respond to each visit properly and help get each component tested properly.

The collision components accepting/receiving visitors do not perform any collision tests. Instead, they function as a pathway to get one part of a collision test to another. The collision tests are handled by the object being reference and the object received.



Composite:

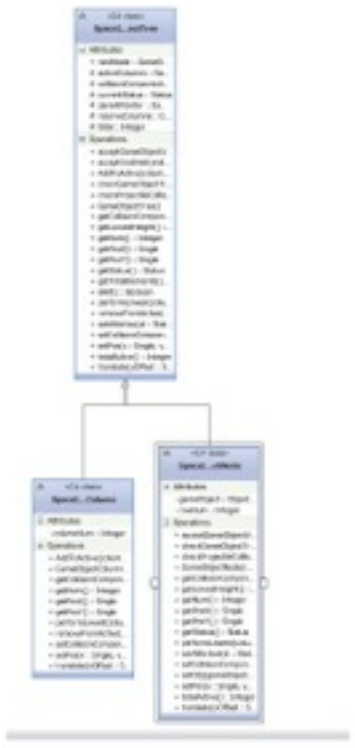
use: treat a group of objects as one object

Where is it use?

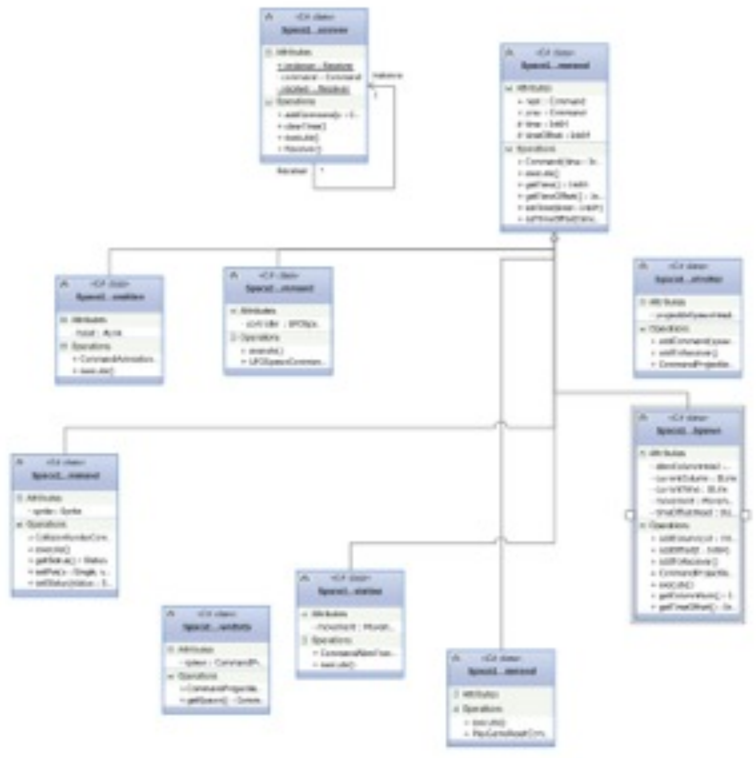
Shield/Alien Grid:

Both the shield and the alien grid are treated as a similar object and share a similar hierarchy. The Composite is segmented into three different components, the tree, the column, and the node. The tree functions as the root of the hierarchy, the columns are branches and the nodes are leaves. Each node is maintained in one of the columns and the tree maintains a list of all the columns. As the names suggest, the hierarchy is organized into a series of rows and columns.

The each element in the hierarchy maintains a collision component responsible for the individual node, the collection in columns and the grid itself. The segmentation of the collision components is to reduce the amount of collision checks being performed and only allow checks on the sub-objects during successful collision hits to the root.



Command:



Use: placing objects onto a queue in order to be executed at specific time.

Where is it used?

Movement and player swap

The Commands are separated into two components, the command and the receiver. The receiver is responsible for receiving the commands and placing the command onto the list. The command maintains a function to be performed at the designated time by the user. The command contains two components, a time offset and a time. The time offset is used by the receiver to set the time for execution. The time is to notify when the command needs to be executed.

The movement of the alien uses a command in order to know when to translate itself. The movement of the alien grid changes its command by changing its offset based on the total amount of objects active in the hierarchy. The offset gets smaller the more aliens are being deactivated during gameplay. As a result, the movement gets faster by appearing closer to the current time.

The spawning of the projectiles is controlled by a command pattern but the offset is a list maintained by the command. After each execution, the projectile command gets the next time offset and uses it as the next time to spawn a projectile. The alien to spawn the projectile is also influenced by the elements on the list. The list of time and aliens to spawn projectiles helps to maintain the illusion of random projectile spawns.

Factory:

Use: Responsible for creating objects.

Where is it used?

Proxy Sprite/Tree/Player

Proxy sprites, game objects, and the objects that maintain references to each object are created inside a factory. The factory create proxy sprites by collecting an image and the real sprite associated with the object being created. An alien proxy sprite is created by finding the real alien sprite associated with that alien and associates it into its creation and the letter proxy sprite collects an image in order to overwrite the sprite's original image. The alien object uses the newly created proxy sprite and uses it as its own image.

The game object tree for aliens and the shield is also created within a factory. For the alien tree, the factory requests the creation of various sprites in order to create each Alien class to be placed onto the list. The factory also assigns a row and column for the object to be added to the object to assent with the organization of each aloe.

Singleton

Use: only one instance of the object

Where is it used?

Factories/sprite/

Singleton, as the name suggest, intends to have a single instance of the object for access. For objects responsible for handling collisions, iterating through update loops, and creating new objects, responsibility should be given to a single object for these responsibilities. As mentioned in the flyweight section, the images of each alien are only created once and both the images for each alien and letter are accessed through a single source.

The images, textures, and composite objects have their creations restring within a single factory. Factories needing to find the specific image are able to go to one class and find the image or texture needed for creating either the animation controllers or a new sprite.

The score controller is another example of another single instance object. Because aliens are going to be constantly shot down, the access to the score controller needs to be limited to a single place. As a result, every time an alien is hit, the alien is able to send its point value to the score controller and deactivate in order to prevent sending multiple joints to the score controller.

The state for each game mode is another example of an object requiring a single instance. Because the game only needs one instance of a play mode, the list of other game states are created in order to limit the creation of each game mode to one instance.

Player input controller during the play state is another example of limiting the creation of an object to one instance.

Iterator

Use: perform function without accessing the function itself

Where is it used?

Sprite/collision test

The sprite rendering and collision tests are placed onto a list that instructs them when to run their tests. The list doesn't have any knowledge about which sprite is being rendered first or which object is being checked first. The list will traverse through all the objects, instruct them to perform their function. The list tells the objects when to do their job but not how.

Each proxy sprite is placed onto a sprite manager list and each manager is placed onto another list. The list holding the sprite managers instructs each sprite manager to update. The sprite managers then instruct each sprite to update and the sprites send their data to the real sprite.

The collision tests are also placed onto a list. The collision components are placed onto a list and are told when to check for collisions. Each collision component then sends their component to visit other components on the list.