DePaul University 2018-2019 Winter

ARCHITECTURE OF REAL-TIME SYSTEM

SPACE INVADERS GAME

Xuefan Wang

Table of Contents

This page intentionally left blank.

Introduction

This document outlines the intricate design patterns employed in the Space Invaders Game. The game engine has been supplied by the professor, with a specific constraint that prohibits the use of modern features in the program.

Notably, this restriction includes the exclusion of containers/arrays, as well as template or generic parameters. Consequently, the entire program has been crafted exclusively using basic data types and fundamental language features.

Implementation

Problem 1: Object for Reuse

Prior to initiating the entire project, we acknowledged our preference for not generating a new object every time it is needed. Our objective is to maximize object reuse after its initial creation. Now, the question arises: how can we effectively store and manage these objects for reuse?

Design Pattern: Object Pool Pattern

Performance can be sometimes the key issue during the object creation (class instantiation). Especially when cost of initializing a class instance is high, and the rate of instantiation of a class is high. The Object Pool pattern offer a mechanism to reuse objects that are expensive to create. For the best performance, the number of instantiations should be finite.

Pattern in the game: Manager and DoubleLinkedList.

In this game, the DLinkedNode is utilized as a "Mixin" class, functioning as the base for every actual object. This ensures that each object is intricately linked to its neighboring elements. To facilitate efficient organization and reuse, a manager class is implemented to oversee a list of objects within the "active list." When an object is not in use, it is moved to the "reserved list."

Details of Implementation

a. The DLinkedNode is an abstract class. For any object that should be in a list must derive from the DLinkedNode. The class has only Prev and Next fields, both of which are instances of DLinkedNode.

b. The DLinkedList utilizes the DLinkedNode and incorporates various functions, including but not limited to AddToFront(), AddToEnd(), and Remove(), to manipulate the linked list.

c. The ManagerBase class makes use of DLinkedList, adding any objects to the manager to the active list. Upon removal, the object is promptly transferred to the reserved list.

d. Recognizing the occasional necessity for a sorted list, I have introduced the Sorted Double Linked List (SDLinkedList) and SDLinkedNode. These classes are derived from their respective base classes, providing a framework for maintaining sorted lists.

UML

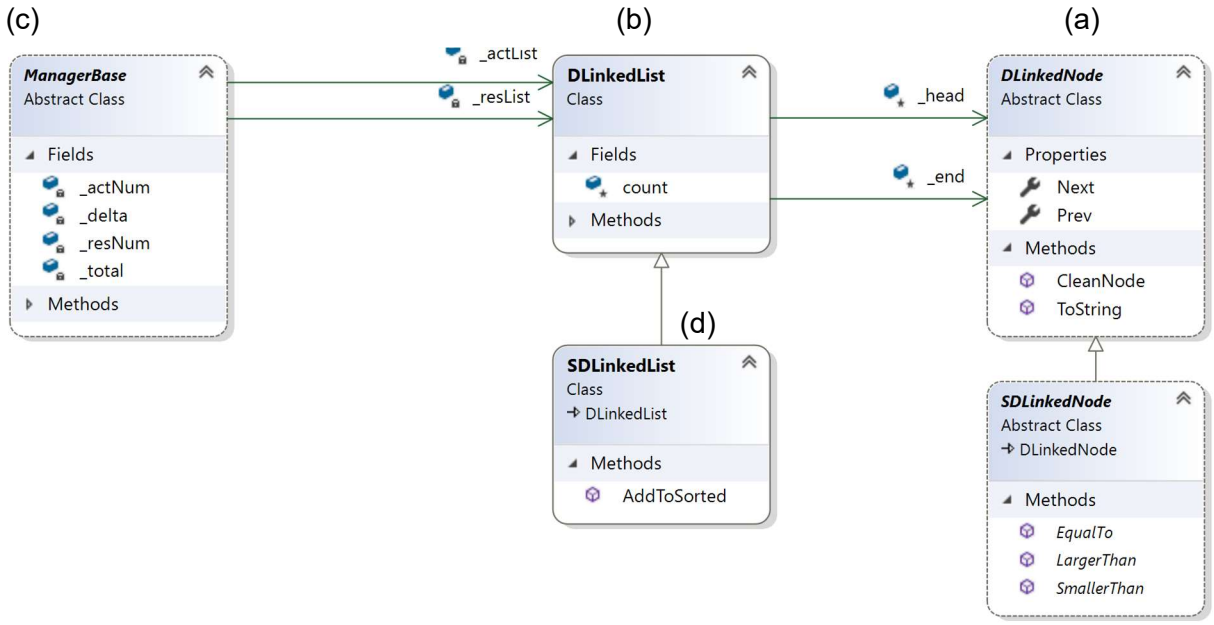

Figure 1: UML of Object Pool Pattern

Problem 2: **Collaborate different Objects**

Given that the Game Engine is supplied by the professor, it is imperative to shield clients from the intricate details of its implementation. Additionally, there is a need for different objects to collaborate seamlessly according to our requirements. How to solve this problem?

Design Pattern: Adaptor Pattern

Adapter lets classes work together, that could not otherwise because of incompatible interfaces. Wrap an existing class with a new interface and convert the interface of a class into another interface clients expect. Impedance matches an old component to a new system

Pattern in the game: Texture, Image, Sprite, and CollisionRect.

In this game, various elements such as textures, images, sprites, and a sound engine are essential components. However, utilizing or modifying the original game engine code violates open-close principle. To overcome this, we have devised our adapter classes, each equipped with pointers to the original game engine code.

As illustrated in Figure 2 (c), classes like Sprite and BoxSprite, originally distinct in the engine, are now harmonized through adapters and can function uniformly as instances of the SpriteBase class. Furthermore, by employing adapters, the previously disparate classes of Texture, Image, and Sprite from the engine can now collaborate seamlessly. The UML diagram provides a clear depiction of the associations between these diverse classes, showcasing the enhanced interoperability achieved with adapter classes.

Details of Implementation

a. In the top-left section, we observe the Texture class and its manager. Our custom Texture class has a pointer (in the blue rectangle) to the game engine's Azul.Texture, and it incorporates functions tailored to its functionality. By adopting this design, clients are shielded from direct interactions with Azul.Texture. Instead, they exclusively utilize the new Texture class, managed by its designated manager, to create or modify texture features.

b. In the bottom-left segment, a similar approach is employed for the Image class.

c. On the right side, we introduce Sprite classes. Given the existence of two distinct types of sprites (GameSprites and BoxSprites), a base class is created to encapsulate them. This base class serves as

a wrapper, ensuring unified usage and facilitating future extensibility.

UML

(a) (c)

(b)

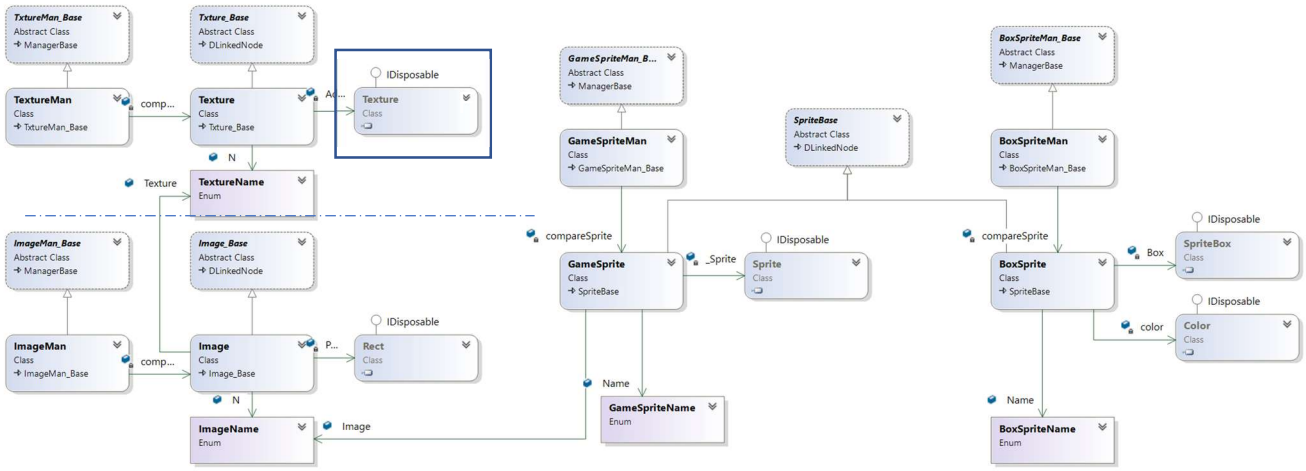

Figure 2: UML of Adaptor Pattern

Problem 3: **Shared State**

As a single texture accommodates numerous font images, the objective is to refrain from generating a new texture simultaneously when creating a font image. Instead, the aim is to utilize a singular texture that multiple font images can collectively share. What steps should be taken next to implement this strategy?

Design Pattern: Flyweight Pattern

Some programs require many objects that have some shared fields among them. Creating many soldier objects is a necessity however it would incur a huge memory cost. So that we can share the common part where the other part can vary. Each "flyweight" object is divided into two pieces: state-dependent (extrinsic) part, and the state-independent (intrinsic) part. Intrinsic state is stored (shared) in the Flyweight object. Extrinsic state is stored or computed by client objects, and passed to the Flyweight when its operations are invoked.

Pattern in the game: Texture, Image, Font

In this game, various images can now collectively share a common Texture. Additionally, all fonts are configured to share a single texture. This design enables the extraction of the shared aspect as an extrinsic component, represented by the Texture class. The intrinsic part of the design now incorporates a pointer to this shared Texture class, enhancing modularity and flexibility.

Details of Implementation

a. Generating a Texture Atlas/Font Sheet: In this game, only three unique textures are created. They collectively encompass 20-30 images and 50-60 fonts, demonstrating efficient resource utilization.

b. The Font Class is structured with a pointer directed to the FontSprite, which, in turn, is linked to the Glyph Class. The Glyph Class contains a pointer pointing to the Texture Class. As a result, all fonts share a common connection to a single texture, establishing a clear distinction between the shared texture and the unique font components.

c. In a similar fashion, every image maintains a pointer to the texture class, even though it is denoted by TextureName. This indirect representation allows for easy retrieval of the actual texture.

UML

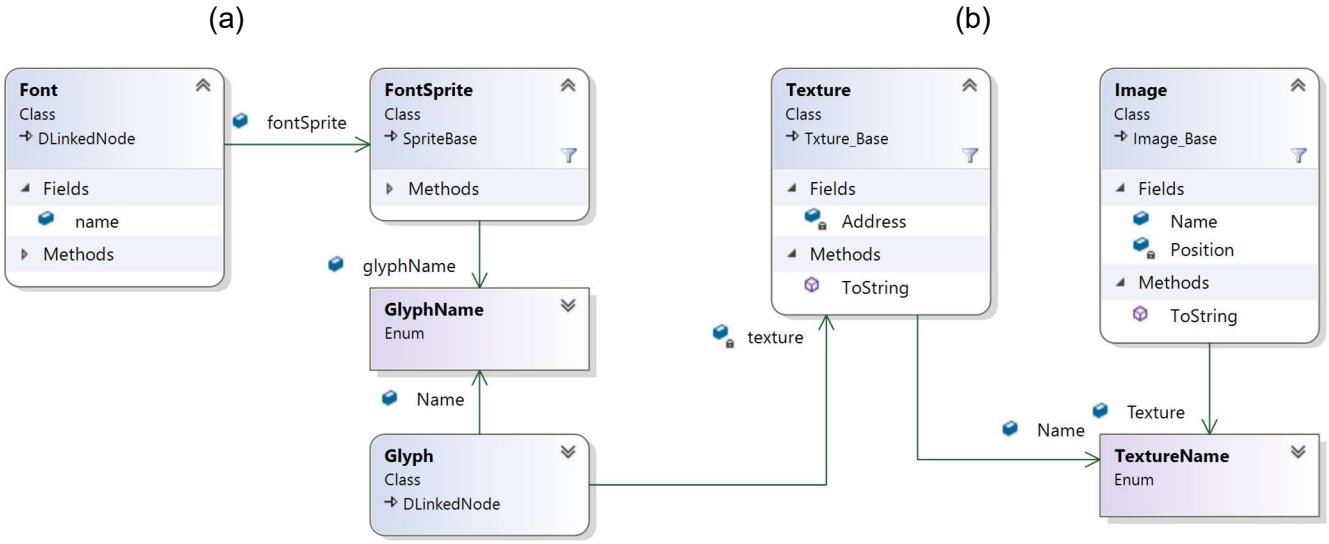

Figure 3: UML of Flyweight Pattern

Problem 4: **Need Lighter Objects**

Creating a Sprite is a resource-intensive task, and within the game, there is a continuous need to add, move, and delete Sprites. Nevertheless, we aim to avoid direct operations on these expensive objects. Instead, the approach involves employing smaller objects to manage and control the behavior of Sprites. What steps should be taken next to implement this strategy?

Design Pattern: Proxy Pattern

Sometimes we need the ability to control the access to an object. For example, if we need to use only a few methods of some costly objects we'll initialize those objects when we need them entirely. Until that point, we can use some light objects exposing the same interface as the heavy objects. These light objects are called proxies and they will instantiate those heavy objects when they are really need and by then we'll use some light objects instead. So that proxy pattern can provide a placeholder for another object to control access to it, and only keep an instance of smaller object using a full feature object

Pattern in the game: Proxy Sprite

This game involves 55 constantly moving aliens and numerous shield objects on the screen. However, for operational efficiency, only the essential (x, y) coordinates of these objects are required. When performing actions like adding, moving, updating, or removing an alien or a shield, only the (x, y) position needs to be updated. The complete object is solely necessary during the drawing phase on the screen. At this point, a proxy sprite can be utilized, and the complete object is instantiated only when necessary.

Details of Implementation

Every game object employs a proxy sprite, and when updating a game object's state (e.g., when Aliens are in motion), only the proxy's (x, y) position needs to be modified. Subsequently, when the proxy is about to be rendered, it conveys the position to the actual object.

The primary distinction between the Flyweight and Proxy patterns lies in the fact that the Flyweight pattern shares a common, immutable part, preventing any modifications. In contrast, the Proxy pattern can also utilize a common part, but this part is mutable, allowing each proxy to have full control over it.

UML

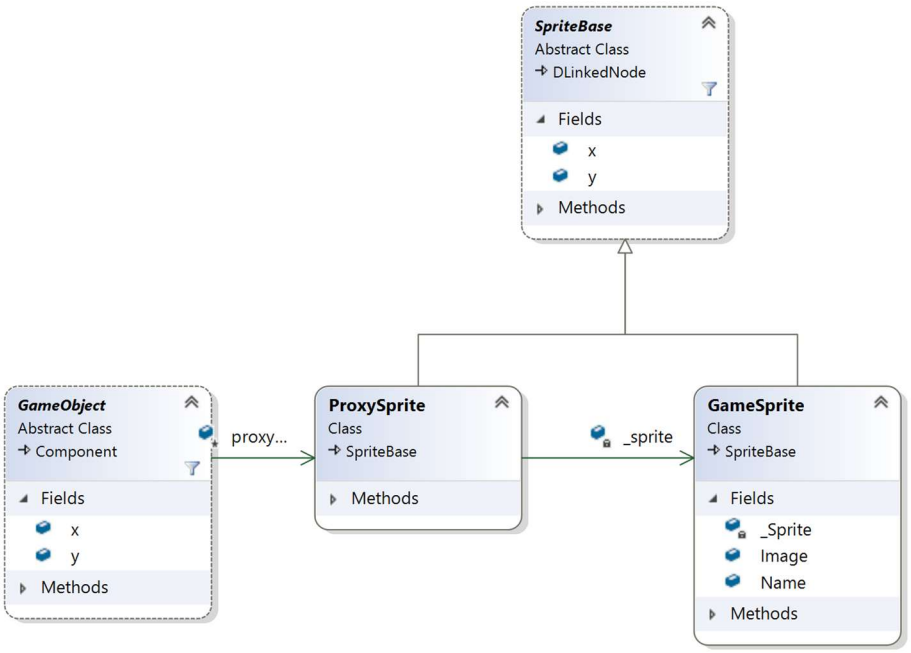

Figure 4: UML of Proxy Pattern

Problem 5: **Only One Instance**

Various objects have been placed into their respective object pool, and some other classes set to utilize them. Our goal is to ensure that whenever we need to reference an object, we consistently use the same and only one pool. What steps should be taken to achieve this?

Design Pattern: Singleton Pattern

The Singleton Pattern ensures a class has only one instance responsible for self-instantiation, providing a global point of access, and enabling usage from anywhere without direct constructor invocation; it also ensures "just-in-time initialization" or "initialization on first use," unlike static classes or fields that constantly occupy memory.

Pattern in the game: Managers

Form previous discussion, we have placed all objects in their respective object pool, and when retrieving an object, the intention is consistently to obtain it from the same pool. To achieve this, we employ the Singleton Pattern to instantiate all managers; upon first usage, we create and maintain a global pointer to the manager, ensuring subsequent accesses refer to the existing instance without re-creation.

Details of Implementation

To implement the Singleton Pattern, we employ a private constructor to prevent external class initialization. Additionally, a private member storing an instance of the singleton is created, along with a public GetInstance() function to retrieve this private instance. Upon the initial call to GetInstance(), a new manager instance is created and stored in the private member. Subsequent calls to GetInstance() return the existing private instance, ensuring no further object creation.

UML

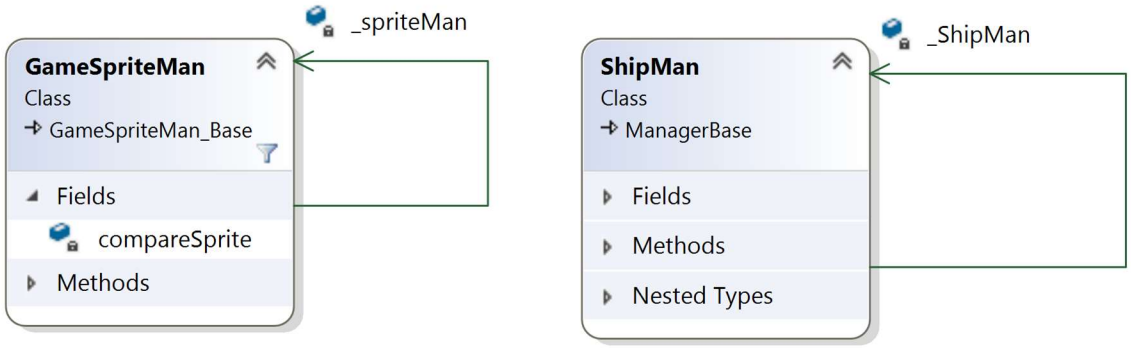

Figure 5: UML of Singleton Pattern

Problem 6: **Execute Command**

As we utilize a proxy to manipulate our sprite's movement, we also aim to alter the sprite's image for animation. Yet, the proxy lacks awareness of the image. How can this challenge be addressed?

Design Pattern: Command Pattern

The Command design pattern encapsulates method calls in objects, enabling request issuance without knowledge of the operation or requesting object; it facilitates queuing commands, undo/redo actions, and other manipulations. This pattern elevates "invocation of a method on an object" to a full object status, suitable for actions requiring more than immediate execution.

Pattern in the game: Animation Command, Remove command

In this game, the Command Pattern is extensively employed. Animation commands alter sprite images, shoot commands initiate bomb firing for aliens, and UFO moving commands facilitate UFO movement. When a game object is shot and slated for removal, I initially mark it for removal and subsequently execute its removal command. To ensure smooth image transitions in animation commands, a timer and an image holder containing all images are incorporated.

Details of Implementation

a. Animation command is derived from command base. When we desire the proxy sprite to render a different image, the proxy sprite lacks the knowledge of the specific method for changing the sprite's image. Its sole responsibility is to draw whatever is present in the sprite. Instead, the images of the sprite are modified using the Animation command.

b. To create an animation, the repeated alteration of the image is necessary. Therefore, a timer is utilized to regulate this process, allowing the command to execute after a specified duration.

c. During the execution of the animation command, it modifies the images within its list of image items, organized in a Double Linked List. Upon reaching the end of the list, the process cycles back to the beginning.

d. Similar commands are introduced to fulfill the same objective. After a specific duration, aliens launch bombs, and the UFO initiates movement. However, the ship does not fire regularly; it only shoots when the space bar is pressed, so it is not included in this set of commands.

UML

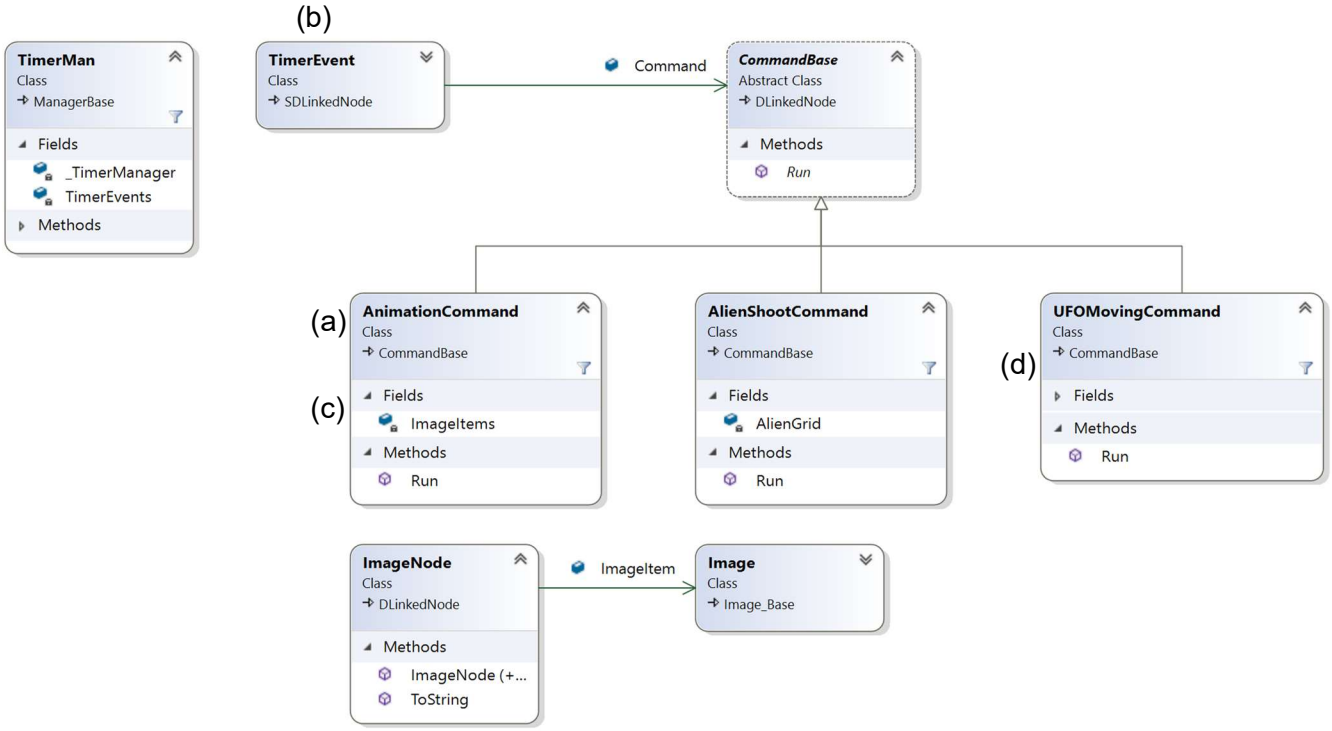

Figure 6: UML of Command Pattern

Problem 7: Hide Details

Whenever there is a need to create a texture or image object, obtaining the precise parameters is essential. However, we aim to conceal these details from the client. How can this objective be achieved?

Design Pattern: Factory Pattern & Factory Method

Both factory patterns are used to create objects without exposing the instantiation logic to the client. And refers to the newly created object through a common interface.

Factory Pattern: When client needs a product, but instead of using “new”, it asks the factory object for a new product, providing the information about the type of object it needs. The factory instantiates a new concrete product and then returns to the client the newly created product.

Factory Method Pattern: Defines an interface for creating objects but let subclasses to decide which class to instantiate. Concrete Creator overrides the generating method for creating objects

Since they look similar, they can be changed into each other very easily. The major difference is that, factory pattern has different create functions to create different object, whereas factory method is an interface which has only create method, and let its subclass decide which it is going to create.

Pattern in the game: Texture Factory

For certain expensive objects such as Shields and Images, the factory pattern is employed in their creation. Instead of specifying the intricate process of creating these costly objects, all creation semantics are encapsulated within a factory. Subsequently, when an object is needed, the factory is invoked for its creation.

Details of Implementation

a. The Texture Manager creates a texture by invoking the relevant methods of the Texture Factory, abstracting away the creation details. Similarly, for shield object creation, the shield factory is called without the need to know the specific locations of individual bricks. The UML of the shield factory is omitted for brevity.

b. The Image Factory, using the Factory Method pattern, directly calls sub-factories to create

corresponding images. In this game project, these two patterns are interchangeable, although the Factory Method pattern can also incorporate additional functions that operate on the created objects, making it versatile for real-life

UML

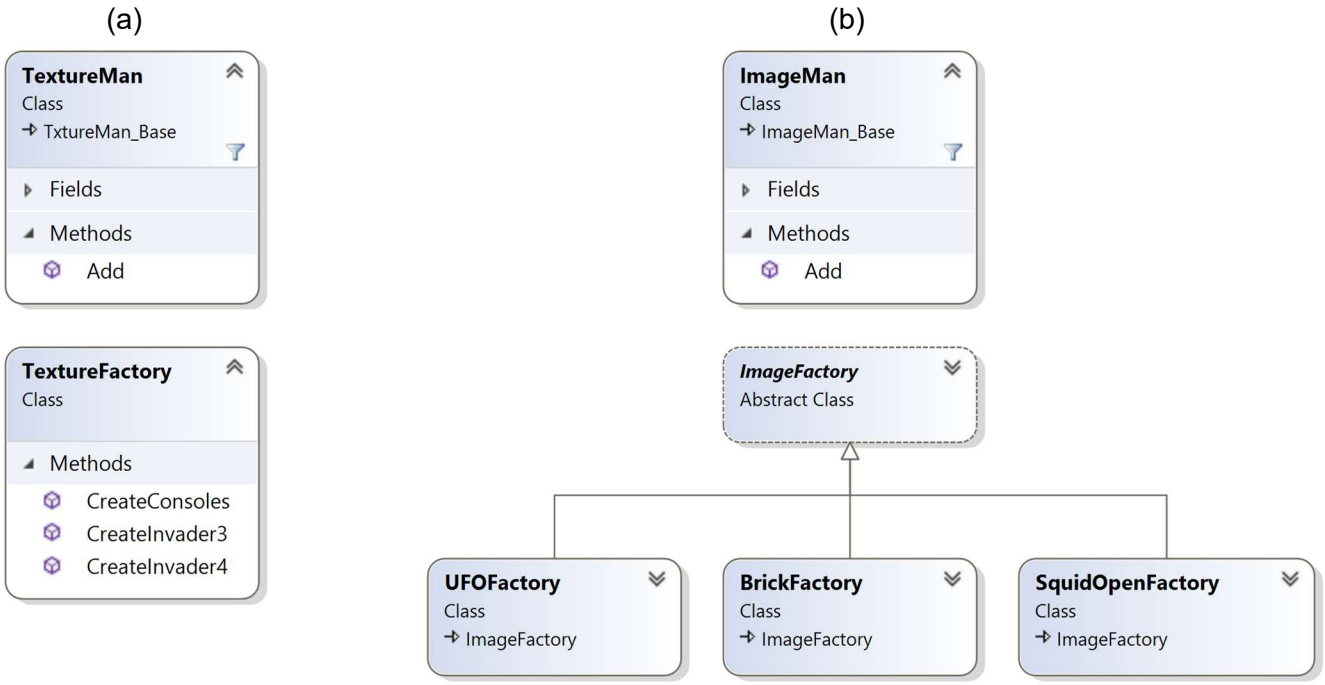

Figure 7: UML of Texture Factory and Image Factory

Problem 8: **Tree Structure**

Now, as the real game development begins, it becomes apparent that organizing all the aliens into columns and placing these columns within a grid is feasible. How can the order of these objects be maintained in the form of a tree?

Design Pattern: Composite Pattern

There are times when a program needs to manipulate a tree data structure and it is necessary to treat both Branches as well as Leaf Nodes uniformly. The intent of this pattern is to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. Every time we call a function, it is recursively called both in the branch and the leaf.

Pattern in the game: All Game Objects (Aliens, Shields, Bombs)

In this game, all actual game objects are organized into a tree structure for efficient collision detection. Instead of constantly checking all movable parts, which would be resource-intensive, placing objects in a tree structure allows us to check if the two roots have a collision. If they don't collide, we can bypass the examination of branches and leaves. When the roots do collide, we can identify the specific branch and leaf involved, leading to substantial time and effort savings during program execution.

Details of Implementation

Initially, we have an abstract class called "component," from which both branches (composites) and leaves inherit. Both types share a common function defined in the component. The only distinction lies in the implementation: when this function is invoked from a leaf, it performs a specific action, while if called by a composite, it recursively calls the same function on its children. This design ensures that with a single call to the root, a recursive invocation is initiated for all its descendants.

UML

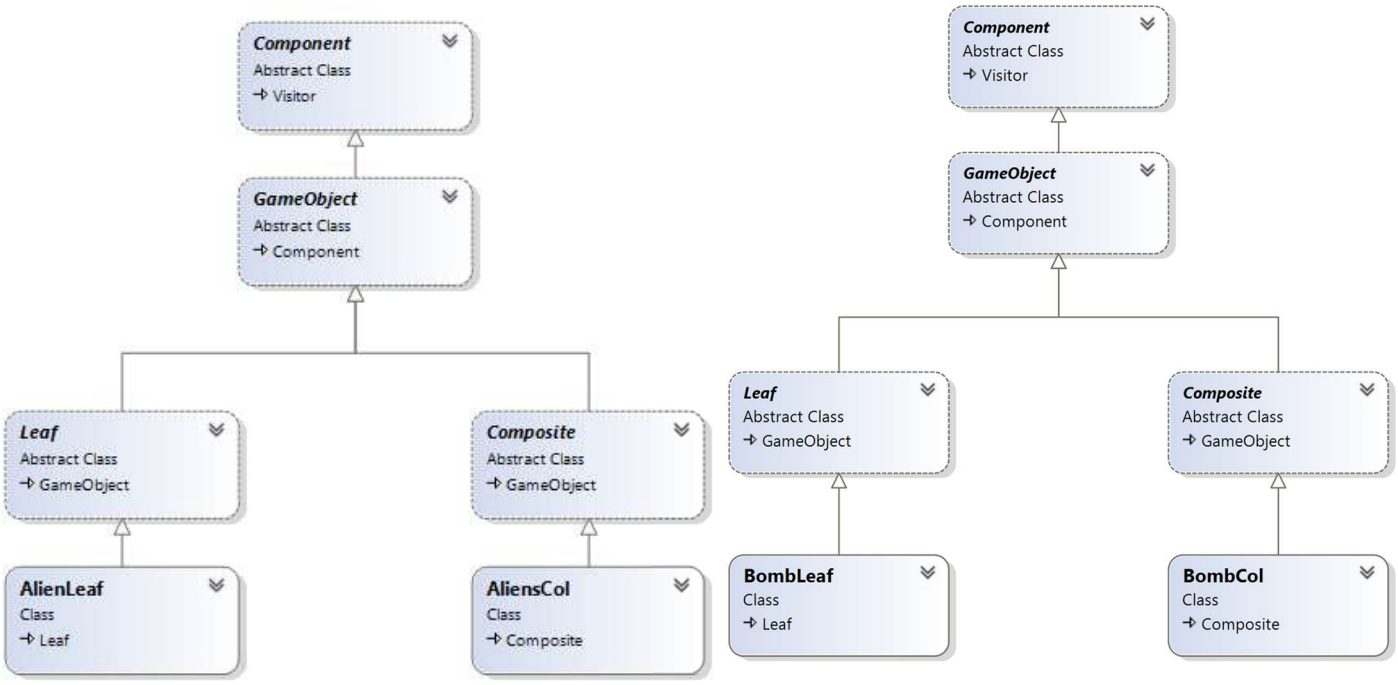

Figure 8: UML of Composite Pattern

Problem 9: **Iterate**

With everything arranged in a composite pattern, a clear structure emerges. Now, when accessing descendants, there is no need to be acquainted with the intricate implementation details of the composite. If we can uniformly treat everything as a tree, obtaining a destination object becomes as simple as calling "next" to navigate to it. What steps should be taken to achieve this?

Design Pattern: Iterator Pattern

A collection should provide a way to access its elements without exposing its internal structure. We should have a mechanism to traverse in the same way a list or an array. It doesn't matter how they are internally represented. The idea of the iterator pattern is to take the responsibility of accessing and passing through the objects of the collection and put it in the iterator object. The iterator object will maintain the state of the iteration, keeping track of the current item and having a way of identifying what elements are next to be iterated. Iterator pattern provides an abstraction that makes it possible to decouple collection classes and algorithms.

Pattern in the game: Game objects, Collision pairs

In this game, identifying collided objects is essential. To navigate the tree structure without delving into its intricacies, we utilize the Next() function to move to the next object. When updating a composite's position and collision box, it's crucial to first update all its children, as the composite's position depends on them. Now, a traverse iterator is necessary to execute function calls starting from the lowest level.

Details of Implementation

Within the Collision Pair Manager, when we identify collisions between two object roots, the goal is to pinpoint which descendants are truly involved in the collision. To achieve this, we traverse through all the children of one tree and compare them with the other tree.

During the update of game objects, the calculation of their (x, y) positions is contingent on their children. Consequently, iteration from the bottom of the tree is necessary. In this case, a Reverse Iterator class is employed, and its implementation essentially involves copying the forward iterator from the end to the head.

UML

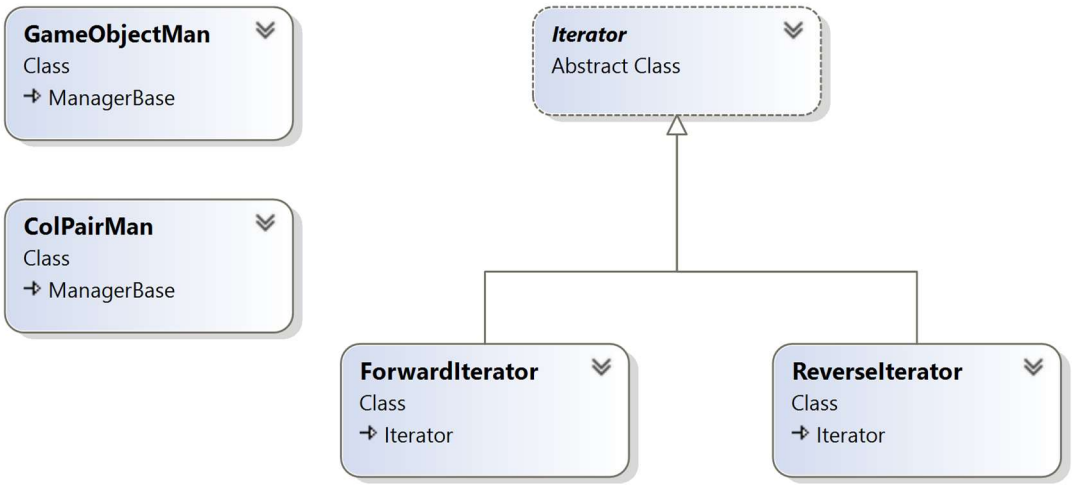

Figure 9: UML of Iterator pattern

Problem 10: **Special action between two Classes**

Having discussed collisions in previous issues, a new challenge arises: when two objects collide, how can one object invoke a function of the other object, especially when they were not connected prior to the collision?

Design Pattern: Visitor Pattern

Visitor Pattern lets you define a new operation without changing the classes of the elements on which it operates. It represents an operation to be performed on the elements of an object structure. The classic technique for recovering lost type information. Do the right thing based on the type of two objects.

Pattern in the game: Game objects

In this game, when two objects collide, as previously discussed, we utilize iterators to traverse both tree structures. However, at each step, we only move one level deeper to identify the specific child that is involved in the collision. Upon detecting the collided child, we allow that child to visit the other tree, and vice versa. In each step, one object accepts the other, enabling the visited object to reciprocally interact, ultimately reaching the leaves of both trees.

Details of Implementation

Initially, an abstract visitor class is created, incorporating Accept() and Visit() functions. All game objects slated for visitation inherit from this Visitor class. To facilitate mutual visits between different game objects, each object must implement the Accept() method. The implementation of the Visit() method is reserved for those functions that will be utilized.

As new collision pairs are introduced, the corresponding Visit() methods should be implemented accordingly. In the final stages of developing the game project, the primary focus revolves around identifying all collision pairs and implementing the Visit() methods along with their corresponding observers, which will be discussed subsequently.

UML

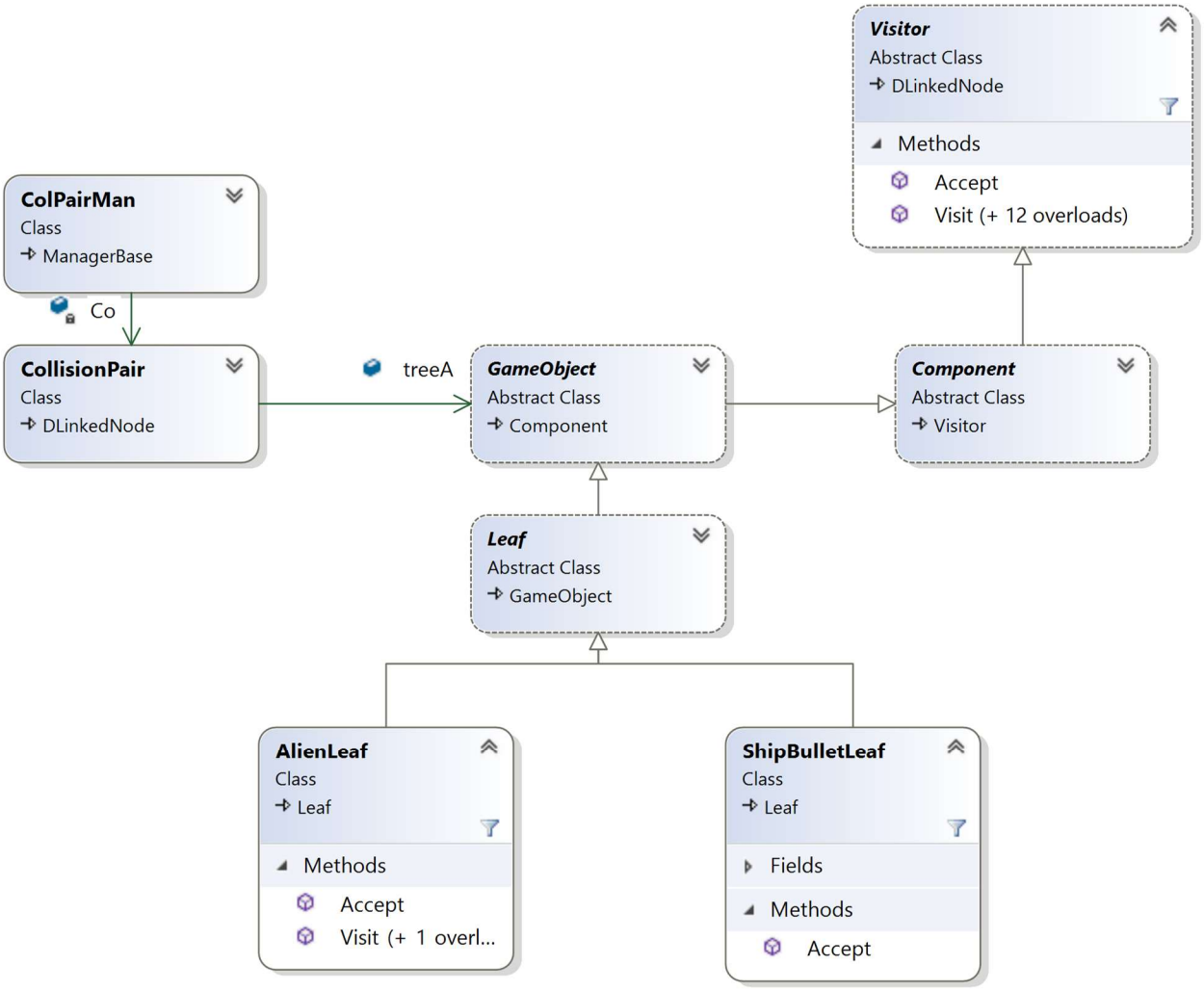

Figure 10: UML of Visitor Pattern

Problem 11: **Notifications**

Upon detecting the collision between two leaves, there's a need to perform specific actions on these nodes, such as deleting the node or resetting it to its original position. Additionally, other unrelated actions, like playing a crash sound or changing an explosion image, may be desired. To facilitate the execution of these tasks, a notification mechanism needs to be implemented to inform all relevant objects about the collision event, enabling them to carry out their respective actions.

Design Pattern: Observer Pattern

All object-oriented programming is about objects and their interaction. The cases when certain objects need to be informed about the changes occurred in other objects are frequent. To have a good design means to decouple as much as possible and to reduce the dependencies. The Observer Pattern can be used whenever a subject has to be observed by one or more observers. Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. Observer is used when the Observable needs to notify other objects about certain events, but doesn't know “which” objects to notify.

Pattern in the game: Collision pairs

In this game, when a ship missile hits an alien, several actions need to be taken, such as deleting the alien, removing the missile, resetting the ship state, playing a "Boom" sound, and more. To handle such collisions, all connected listeners are notified, allowing them to execute their respective actions in response to the event.

Details of Implementation

Initially, an abstract class called Observer is created, featuring a single function, Notify(). All concrete observers are derived from this base class. In the collision pair, a pointer to a subject class is maintained, which, in turn, contains all the observers. When two game objects in the collision pair collide, they notify all listeners stored in the subject.

Upon creating a collision pair, the necessary listeners are added to it, ensuring that nearly every observer can be employed in different situations. For instance, the "Boom" Observer can be utilized in scenarios such as a missile hitting an alien or a bomb hitting a ship.

UML

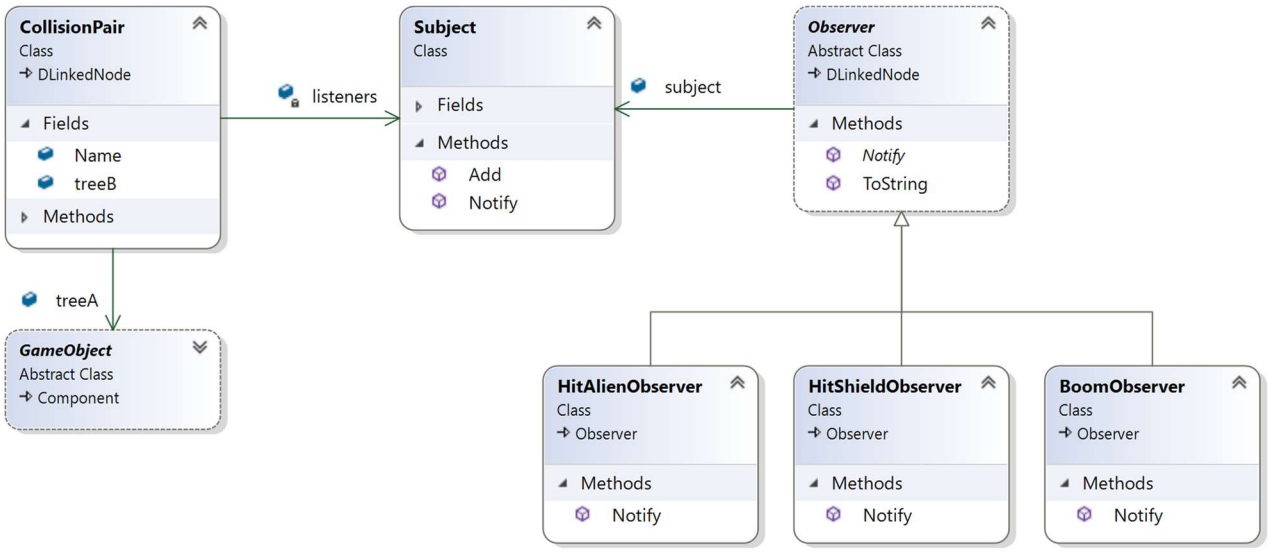

Figure 11: UML of Observer Pattern

Problem 12: **Same Object but Different Actions**

The ship undergoes different states, including a "ready" state that allows movement and shooting, a "missile flying" state that permits movement but not shooting, and a "death" state that restricts both movement and shooting. How can the ship's state be modified without necessitating a change to the entire object?

Design Pattern: State Pattern

State Pattern allows an object to alter its behavior when its internal state changes. But it does not specify where the state transition will happen. State pattern allow the same object changes its pointer to different state classes, which is different from strategy pattern.

Pattern in the game: Ship State, Game State

In this game, a ship can exist in various states, each associated with distinct behaviors. The state pattern does not determine when the ship's state should change; instead, observers are utilized to notify state changes.

Additionally, the game encompasses different states: Select State, Play State, and Game Over State. Despite the visual differences in each state, understanding that the implementation is akin to the Ship state and recognizing that transitions between states rely on observers makes the implementation straightforward.

Details of Implementation

A Ship State abstract class is crafted, necessitating diverse implementations for its inherited functions in different ship states. When a ship initiates a missile launch, the state transitions from Ship Ready State to Missile Flying State. Consequently, the pointer stored in the ship leaf class is updated to the corresponding concrete state.

Similarly, when a game scene undergoes a state change, it should adjust its state pointer to the relevant scene.

UML

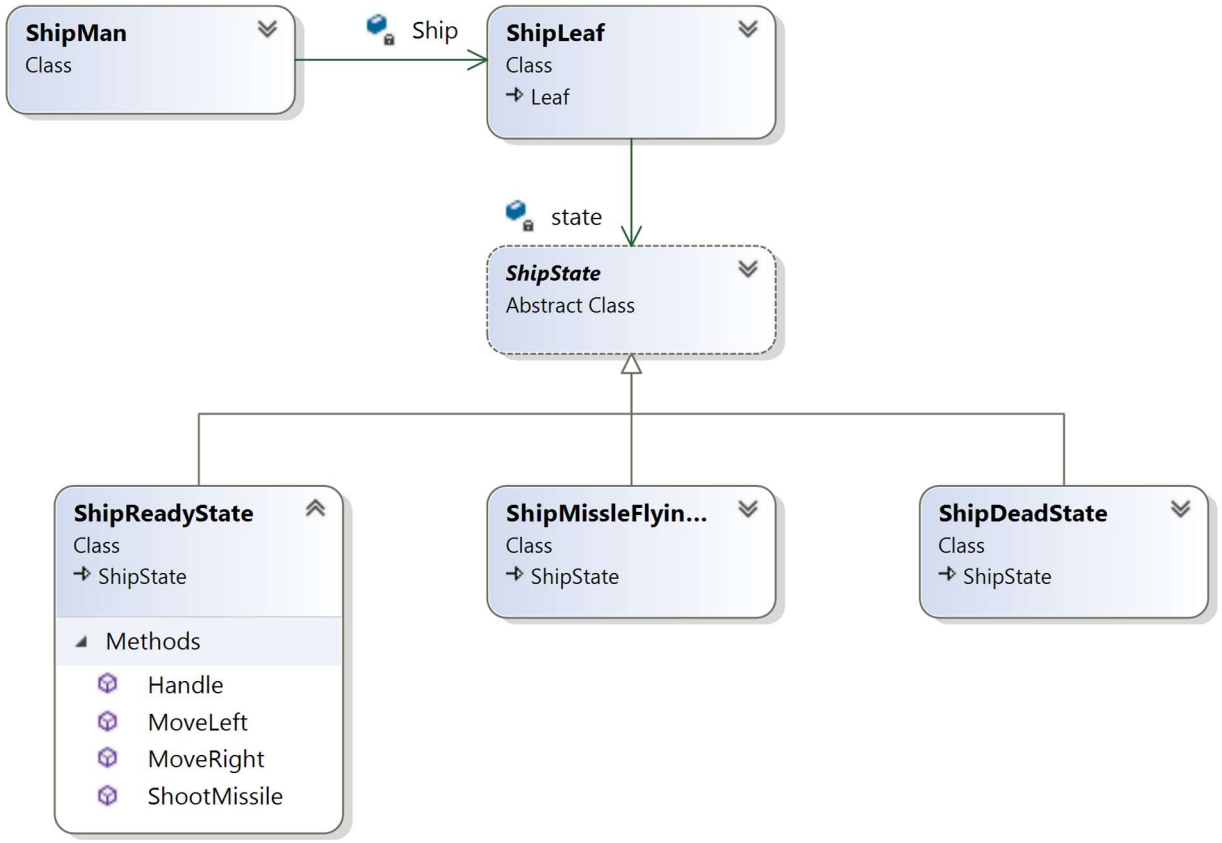

Figure 12: UML of State Pattern

Problem 13: **Same Class but different Behavior**

When aliens release a bomb, each bomb should exhibit a distinct falling style. The objective is to avoid swapping bomb images. What approach should be taken in this scenario?

Design Pattern: Strategy Pattern

There are common situations when classes differ only in their behavior. Isolating the algorithms in separate classes is a great choice in order to have the ability to select different algorithms at runtime. Strategy pattern can define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Pattern in the game: Bomb fall strategy

In this game, bombs need to be dropped in varied styles—some in a zigzag manner, others flipping over, and some falling straight. To achieve this, distinct strategies are developed. Prior to the creation of a bomb, it remains unaware of its specific descent pattern. However, it knows it will adhere to a designated Fall Strategy. Once created, the bomb consistently follows its concrete strategy, and this strategy remains immutable.

Details of Implementation

An abstract Fall Strategy class is established, and various concrete fall strategies implement it differently. When creating a bomb leaf, the desired fall pattern is passed as a parameter to obtain the bomb. Despite the possibility of recycling the bomb later, it consistently retains the same strategy, remaining unaltered. This distinction serves as the primary difference between the strategy pattern and the state pattern.

UML

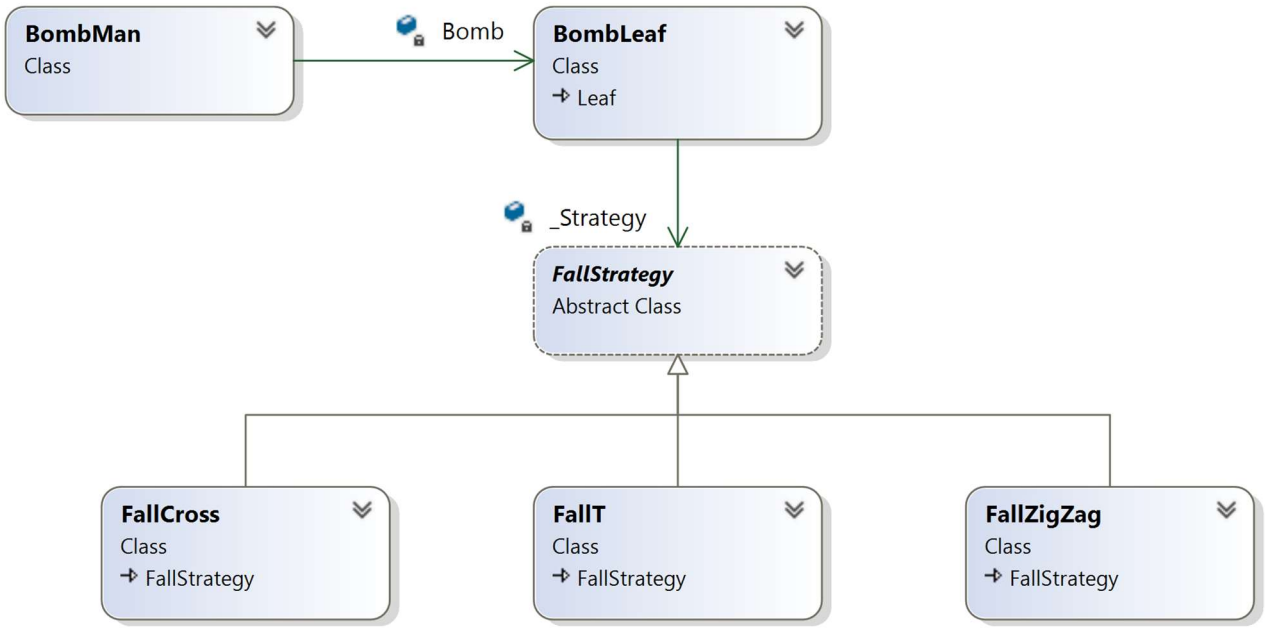

Figure 13: UML of Strategy Pattern

Problem 14: **Abstract Behavior**

In the process of developing the entire program, numerous classes are derived from their base abstract class. The base class exclusively contains abstract operations, with each derived class mandated to define its concrete operation, thereby overriding the base abstract operation. Is there a specific pattern associated with this approach?

Design Pattern: Template Pattern

Template Pattern defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. It lets subclasses redefine certain steps of an algorithm without letting them to change the algorithm's structure. Base class declares algorithm “placeholders”, and derived classes implement the placeholders.

Pattern in the game: All class name end with base, and Game Object

This is a basic approach in writing object-oriented programming. Each class is designed to inherit from an abstract class or an interface, both of which may define functions requiring implementation without delving into excessive details. Consequently, all subclasses derived from the base class must override these functions with concrete implementations.

Details of Implementation

This UML diagram represents the Composite pattern, with a specific focus on their methods. The base abstract class includes three abstract methods: Add, Remove, and Get First Child. Both derived classes, Composite and Leaf, are required to implement all these methods. However, obtaining a child from a leaf does not make sense. Therefore, the Leaf class essentially performs no action, implying that the functions either do nothing or return null, despite the requirement to implement them.

UML

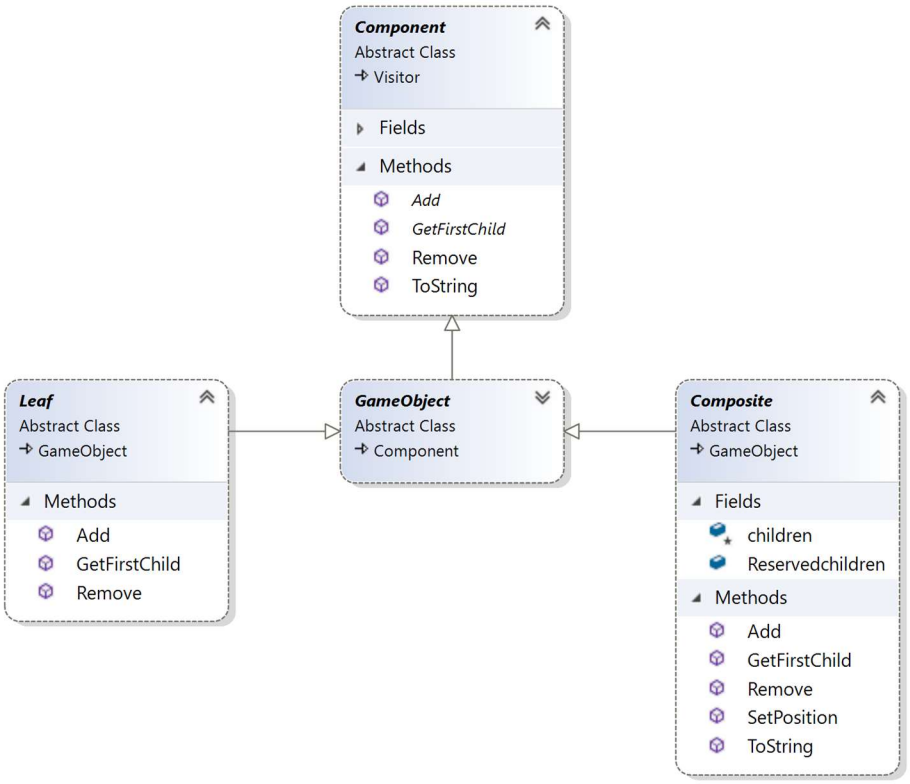

Figure 14: UML of Template Pattern

Problem 15: **Object that does nothing**

As mentioned in the earlier problem, when attempting to retrieve its child, the leaf class takes no action. There are instances when we require an object to perform no operation. What approach should be taken in such cases?

Design Pattern: Null Object Pattern

Null object pattern provides an object as a surrogate for the lack of an object of a given type. The Null Object Pattern provides intelligent do-nothing behavior, hiding the details from its collaborators.

Pattern in the game: Null Game object

In this game, the null game object, is just as it defined, all the methods are returning null, or doing nothing.

Details of Implementation:

Implementing null objects is remarkably straightforward. As illustrated in Figure 15, all methods defined as abstract in the abstract class must receive a concrete implementation in the null object. However, all these concrete implementations essentially involve doing nothing.

UML

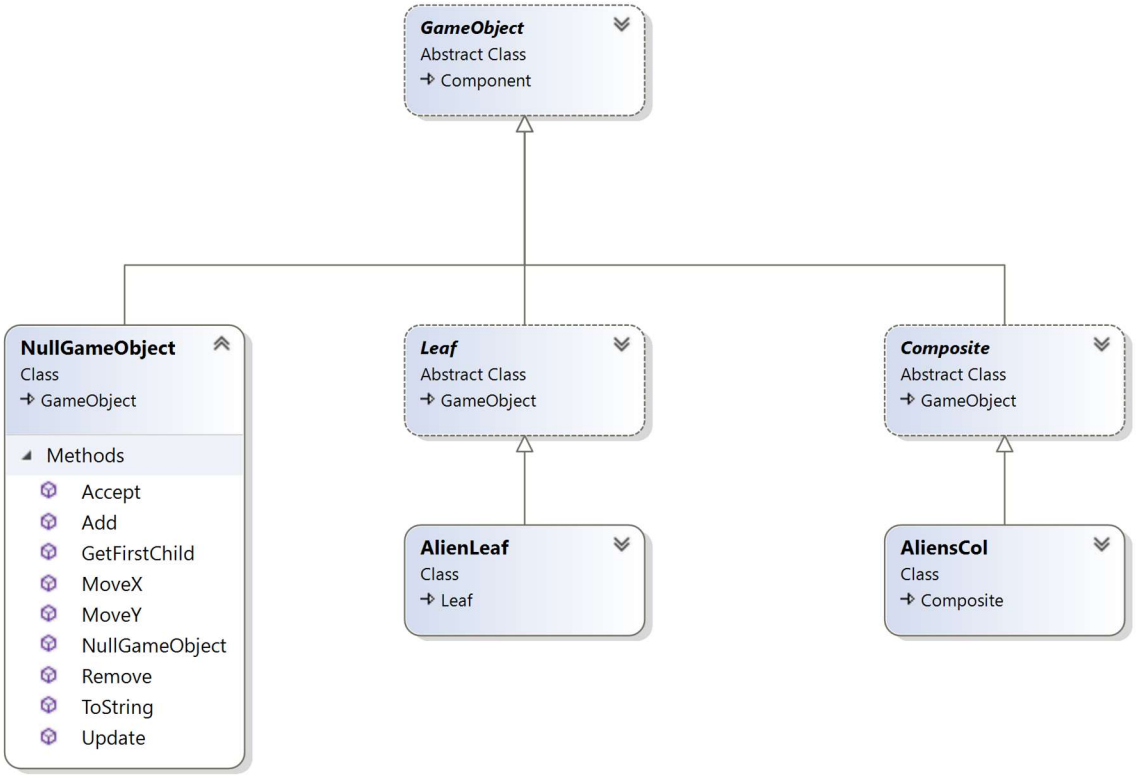

Figure 15: UML of Null Object Pattern

Conclusion

In the development of the entire Space Invaders game, as outlined in the preceding section, a minimum of 15 design patterns are employed. Throughout the development process, efforts were made to adhere to the SOLID principles.