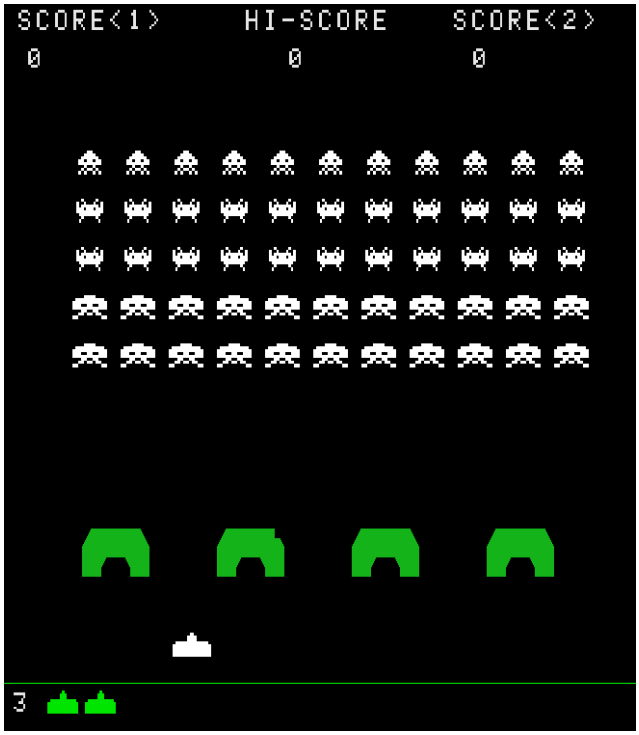
Space Invaders Design Document

Video: https://youtu.be/P91_LJXZGUc



By Robert Grier

Table of Contents

1 Overall Design

1.1 Game Loop

At the highest level, Space Invaders updates the state of the game and then renders the visual representation of that state. Input is taken from the player and combined with the game mechanics to create a game. The objects are updated through the object hierarchy. Then, collision checks between registered object types are checked. These collisions trigger events when their collision geometry overlaps using event observers. Timed events are triggered based on real time that has passed. Timers can be set throughout the game to schedule behaviors. For rendering, the objects are broken out into batches which are rendered with a priority order so that particular groups can be rendered on top of other groups.

1.2 Object Hierarchy

Game Objects in Space Invaders are organized into a Composite hierarchy. This allows for more efficient collision checks where most combinations of objects can be ruled out. The object hierarchy also allows groups of objects to be updated and operated on by iterating over them with custom Composite Iterators.

1.3 Memory Management & Collections

The memory of game assets is managed in a Manager Base class that uses Object Pools. The object pools hold onto memory from discarded objects and recycles them when new objects are requested. This enables a performant real time system. Additionally, list structures in Space Invaders are all custom linked list, both single-linked and double-linked. These link types are “mix-ins” with the game asset types to provide collections with minimal overhead.

1.4 Game Mechanics

The game has unique mechanics based on input, timers, and collisions. The ship and missile are controlled by the player input. Alien enemies move around based on timer events. The actors in the scene can collide to destroy each other, play sounds, and spawn death splatters. The Observer and Visitor Patterns combine well to enable these mechanics. The game supports single and multiplayer modes, as well as various scenes states for game select, play, and game over.

2 Design Patterns

2.1 Proxy Pattern

Problem

A Sprite object holds a reference to an Image, an underlying Azul Sprite, and positional data, allowing the game to display the Image via the Azul Engine at the location described by the data. The problem is that there may be many objects that wish to use the same Sprite, but only want to change a few properties. In particular, an ideal solution would allow sharing a Sprite with a single internal animation but with different locations on the screen. Using just the basic Sprite class, there would need to be a separate copy of each component of Sprite, even if only some properties need to be changed by the client.

Solution

We create a Sprite Proxy, which holds a reference to a Sprite object and the individual location properties of the Proxy, such that we can display the Sprite at that location while maintaining only one copy of the underlying Sprite, including its animation. This allows us to have what appears to be many instances of the sprite on screen at different locations.

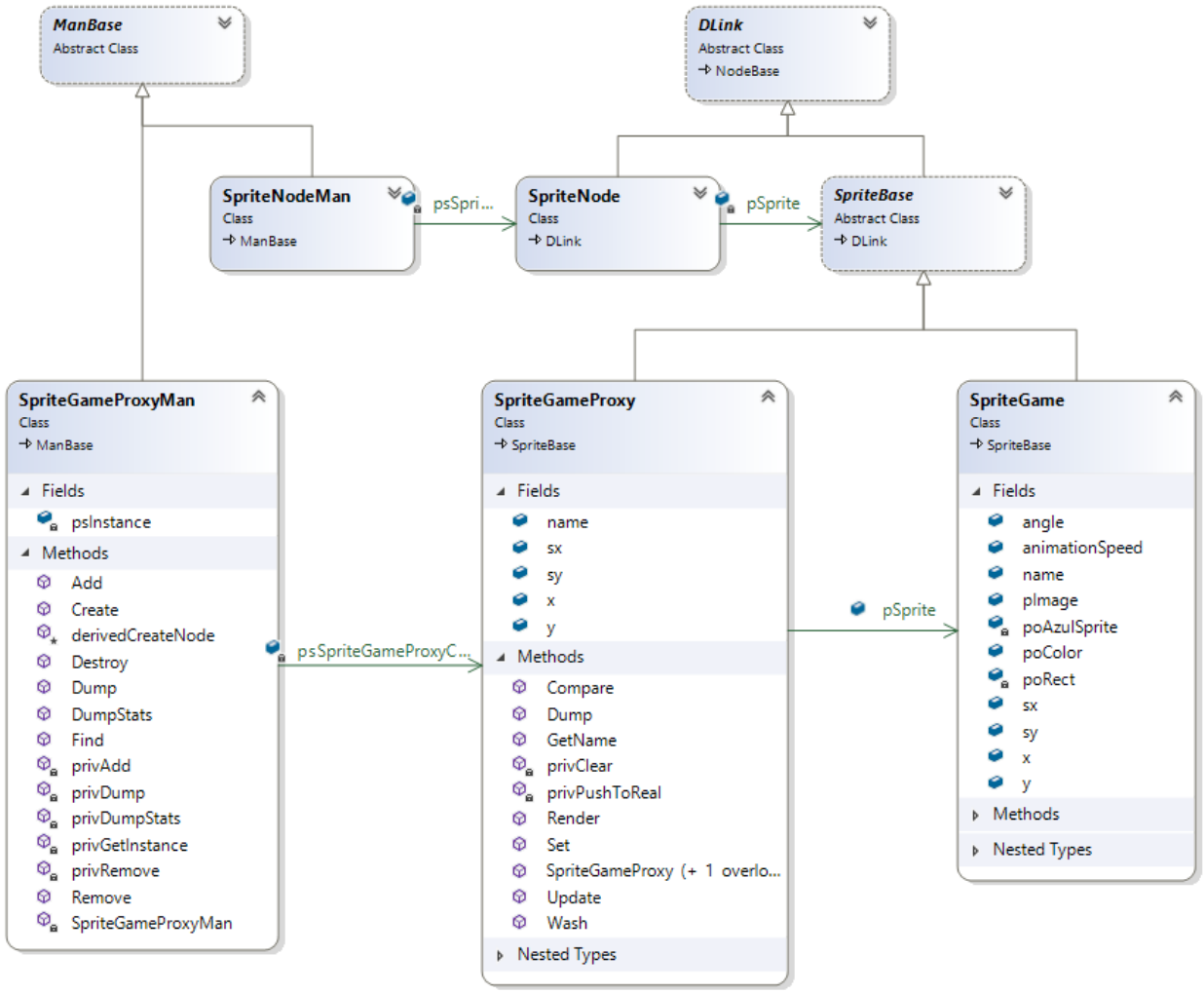
Intent

In its simplest form, the proxy pattern is an added level of indirection that allows us to manipulate how client code interacts with an object. This layer of indirection could do anything. The key elements of the pattern are the proxy object and the real object, which derive from some base class. The proxy holds a reference to the real object, and delegates or changes how a client interacts with the real object.

In the Sprite Proxy case, we can narrow the interface of the Sprite to be just the elements relevant to the client code, such as the x and y coordinates. This forces all client code to share the intrinsic properties of the Sprite, while also having their own extrinsic properties, like the location data, to play with in the proxy. From the client code's perspective, however, it should not be relevant whether they are using the real object or a proxy to that object.

Theory

The concept of intrinsic versus extrinsic data is something that augments our Proxy pattern in Space Invaders. It forces you to be more honest about which properties of an object are truly part of that object, and which properties should be injected by the client in the context of the object's usage. We use the added Proxy layer to only expose the extrinsic data to the client, and force shared usage of the intrinsic data. This concept is also relevant to the Flyweight pattern. However, rather than provide a selection of glyphs, our intent here is to obfuscate usage of Sprites so that the client code can benefit from this behavior and still use it as a Sprite, so it is a just a special Proxy.



Implementation

In the Space Invaders project, the Sprite Proxy is derived from the Sprite Base class, just like the real Sprite Game class. The Proxy holds a reference to a Sprite Game, as well as a set of x, y coordinates. Many of the Proxy's methods just call through to the Sprite Game, but the interface is limited to properties that are relevant to the client. The Proxy Render() method is interesting because each proxy needs to be rendered in order to show up at each location on screen. It does this by quickly updating the position data of the Sprite Game, then "stamping" it onto the screen by calling Render(). The next Proxy will just overwrite the data on the real Sprite Game and do the same thing.

Client-code such as Game Objects hold a reference to the Proxy and change the x and y values. The Game Object cannot manipulate parts of the underlying Sprite Game because the Proxy hides the intrinsic elements such as animation, Images, and Azul references. On the other hand, the Game Object can move their Proxy around as if they had their own copy of the Sprite by manipulating the x, y coordinates.

2.2 Strategy Pattern

Problem

The bombs in Space Invaders need to show one of three separate visual effects on screen while falling. This effect needs to modify the appearance of the sprite independently of the other bomb sprites. It also needs to be interchangeable so that we can pick a visual effect at random when the bomb enters the scene. We cannot implement this as an animation because we are not artists, so we need to create an analog visual effect in-game that is still easy to modify.

Solution

We create a Fall Strategy base class that defines the contract for a given "fall" visual effect, mainly a simple Fall() method. We then implement all of the falling effects as concrete derived classes of the Fall Strategy, which will manipulate the sprite in some way to create an effect. The Bomb class will now hold a reference to a Fall Strategy, and execute the Fall() method on update. When we create a bomb, we can assign it one of the concrete Fall Strategies at random.

Intent

The intent of the Strategy Pattern is to encapsulate a behavior in a separate object, and reference it through a generic interface, such that the underlying behavior can be modified and interchanged with other implementations without affecting the calling code. At first glance, the Strategy Pattern seems to be a restatement of the mechanics of inheritance. However, the real intent of the pattern is to extract implementation details into their own object to gain more dynamic behavior through the composition of strategies.

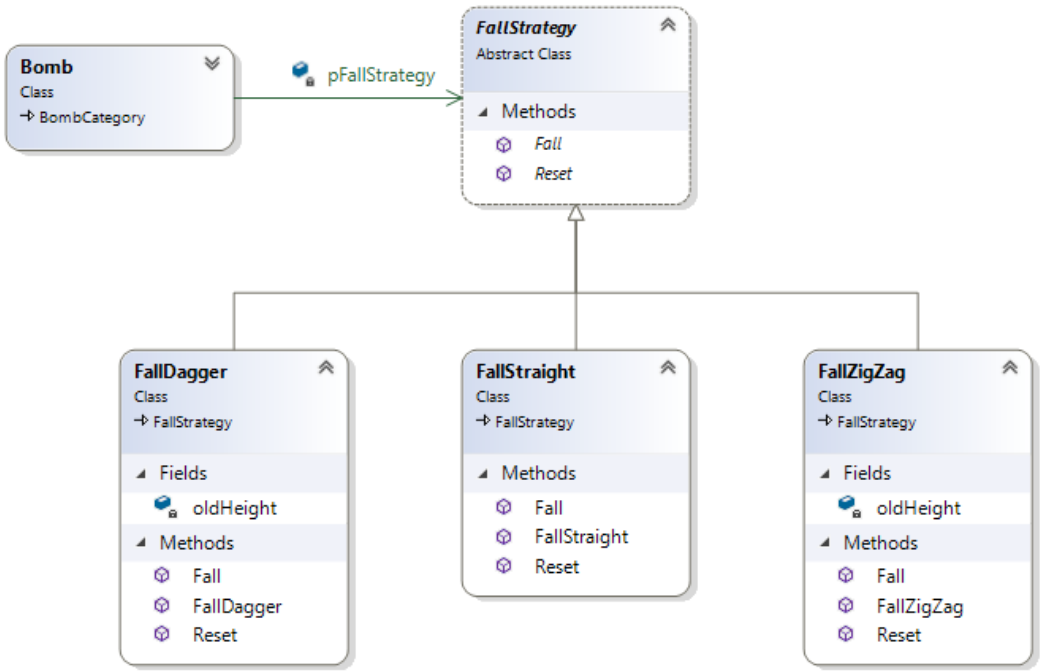
The benefits of Strategies become apparent when compared to the alternatives. You could select between a given implementation of the behavior with conditional statements or a table, then execute the right one. This is probably slower and requires modifying the conditions every time a new implementation of the behavior is added. A more reasonable alternative would be creating several subtypes of Bomb, each overriding a Fall() method to customize the behavior. The drawbacks here are more subtle. Let's say we wanted to add an Explode() behavior to the Bombs, that has several different ways to make the bomb explode. Now, we would need to create a subclass for every possible combination of each Fall() behavior and each Explode() behavior. This would grow exponentially as more behaviors are added. Instead, with the Strategy Pattern we can create Fall and Explode strategy objects. We can then compose any combination of behaviors when the object is constructed.

The flexibility of the Strategy Pattern enables the idea of a "designer" object where the creator of the object can pick and choose from various behaviors to construct a highly customized version. The core code of the object never needs to be changed, but infinite new

combinations can be added. This could be useful in game design, where the code must be able to handle a demand for diversity of mechanics and the whims of designers.

Theory

A common idiom in software engineering is composition over inheritance, which says that objects should gain functionality through composition of other objects, rather than through inheritance. The Strategy Pattern is a great example of how composition and inheritance are not at all competing with each other, but are instead both essential to good object oriented design when used properly. The primary mechanism of the pattern is inheritance because the implementations of the strategy inherit from the strategy base. However, the usage is purely composition, so both concepts are working together to create a more cohesive design.



Implementation

In Space Invaders, the Fall Strategy has a very simple implementation. The base class has an abstract method for Fall() and also for Reset(). The Bomb class holds a reference to Fall Strategy and only interacts with it through the common interface. The Bomb client is completely agnostic to any given Fall Strategy implementation.

The concrete strategies override the Fall() method by adding special effects to the missile sprite every update. The dagger and zig-zag strategies flip the sprite vertically and horizontally, where as the fall-straight strategy is effectively a null object that has no effect. These effects are interesting because they create special effects using the tools available in the game framework. When the bombs are constructed or resurrected, the strategy is injected into the object.

2.3 Factory Pattern

Problem

The Aliens in Space Invaders need to be initialized in a particular way. Firstly, the object needs to be resurrected from the Ghost Manager if there is one to be resurrected. Alternatively, a new object needs to be constructed with the correct parameters. Then, the object needs to be attached to the correct systems in the game so that it can be rendered and updated. Lastly, the Aliens need to be properly organized in the grid hierarchy. This initialization process is potentially bug-prone if repeated wherever aliens need to be constructed.

Solution

By creating an Alien Factory, we can encapsulate the complex initialization process of Aliens inside a set of reusable methods. This adds a layer of indirection between the game code that creates the Aliens and the actual creation logic. Now, the Alien grid can be completely initialized and reconstructed at any point with a single method call.

Additionally, we can ensure that no Alien is ever constructed in the wrong way, because it only ever occurs inside the Factory. When something new needs to happen as a part of this initialization, it is just added to the Factory, and the change immediately applies to all of the Alien constructions.

Intent

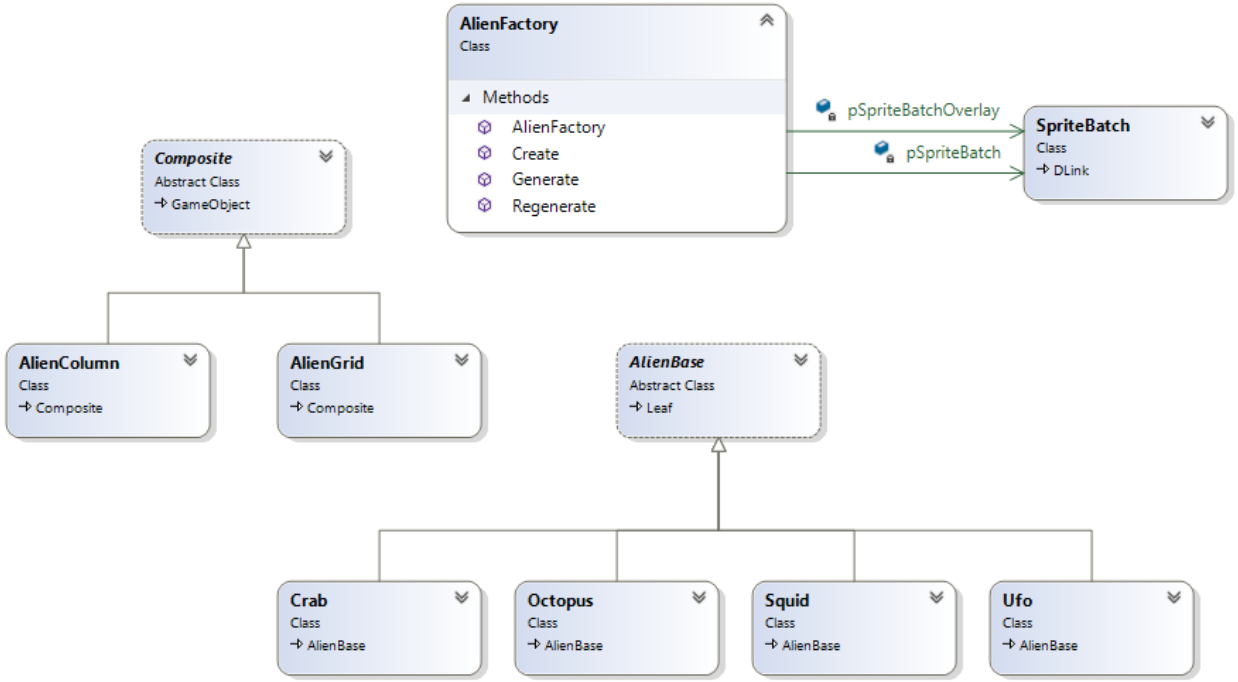
Using the Factory Pattern means encapsulating the creation process of an object such that client code can more easily, safely, and consistently create the object. In the simplest case, a Factory can be a static method that wraps a constructor and some specialized code, such as setting additional properties. In a much more complicated case, a Factory might be an entire system in and of itself, which goes through a complex set of process just to return a reference to an object.

Part of the Factory Pattern is ensuring that anything returned by the Factory is already in a state where the caller can make certain assumptions about what the object can do. For example, a Game Object created through a Factory should always have non-null properties and should be ready to update and draw. By providing this guarantee, the Factory allows its users to write simpler code.

Theory

The Factory is easily identifiable as a creational pattern. As a creational pattern, it can play a role in the RAII idiom, or “resource acquisition is initialization”. The Factory decorates acquisition with initialization and returns the resource in one atomic step. I think this idea is sometimes misread as only applying to memory allocation. In reality, resources can be acquired in many different contexts and across many types of boundaries. For example, objects may be acquired from an Object Pool or even from a synchronized data structure in

a concurrent system. The Factory pattern illustrates how RAII can be applied to any form of resource creation, acquisition, and initialization. The point is that the user of the Factory does not have to be aware of all of the work that really needs to be done to safely acquire an object.



Implementation

The Alien Factory in Space Invaders is complicated and provides several methods. The creation of a single Alien happens in the Create() method. This constructs the object or alternatively retrieves it from the Ghost Manager Object Pool. Then, it sets up the Sprites by attaching them to the Sprite Batches. Each Alien has a Game Sprite and a Collision Sprite, so both of these need to be setup. The batches are passed into the Factory at the start and stored as members to make this process easier. This way, we can be confident that any objects created with a given factory will have the same Sprite Batches.

The Factory also has the Generate() and Regenerate() method. These are more involved and encapsulate the creation of the entire Alien Grid. This allows other code in the game to generate or regenerate the Grid without having to worry about all of the objects being attached correctly.

In summary, no other code in the system has knowledge of how Aliens are created. If we need to change that process, it only needs to be changed in the Factory. From the caller's perspective, it does not matter whether or not there is a memory allocation, an Object Pool, or any other idiosyncrasy of how the Aliens work.

2.4 Iterator Pattern

Problem

There are many different types of collections in the system, and iterating over them is not always uniform or easy. Also, code that iterates over a collection to perform some operation on its elements does not necessarily need to know how that collection is stored or ordered. The memory locations of the elements, how they are linked together, and the logic of how to pick the "next" element, are all irrelevant to most code that will use the collection. This code needs to be able to write a for-loop or while-loop for the collection as if it were a simple array.

Solution

An Iterator object can internally handle the logic for how to iterate a given collection. Its interface needs to provide a way to get the first item, the current item, the next item, and a way to check if there are no more items. With this information, all of the elements of a basic loop can be written to iterate over the collection and perform some operation on the current item.

We can create an abstract Iterator interface that provides these methods. Then we can create implementations of the interface for different types, such as a DLink Iterator and SLink Iterator. We can even create iterators for Composite trees that visit each object in whatever order is correct. We can also create reverse iterators to process collections in reverse order.

Intent

Any complex application will require some form of collection, or at least the concept of a sequence of objects. Additionally, one of the most fundamental mechanisms in structural programming is a loop, which is frequently used to iterate over each object in a collection. So, in order to perform these iterations over sequence of objects, there is some minimum amount of information that needs to be retrieved from the collection to get all the steps right. These are the first element, the next element, the current element, and the condition indicating there are no more elements.

The Iterator Pattern is the encapsulation of these methods in an object. This way, the logic implemented does not have to be repeated for each walk of the collection. Instead, the loop can be written on one line, and never needs to be more complicated than the most basic loop using an array and integer index.

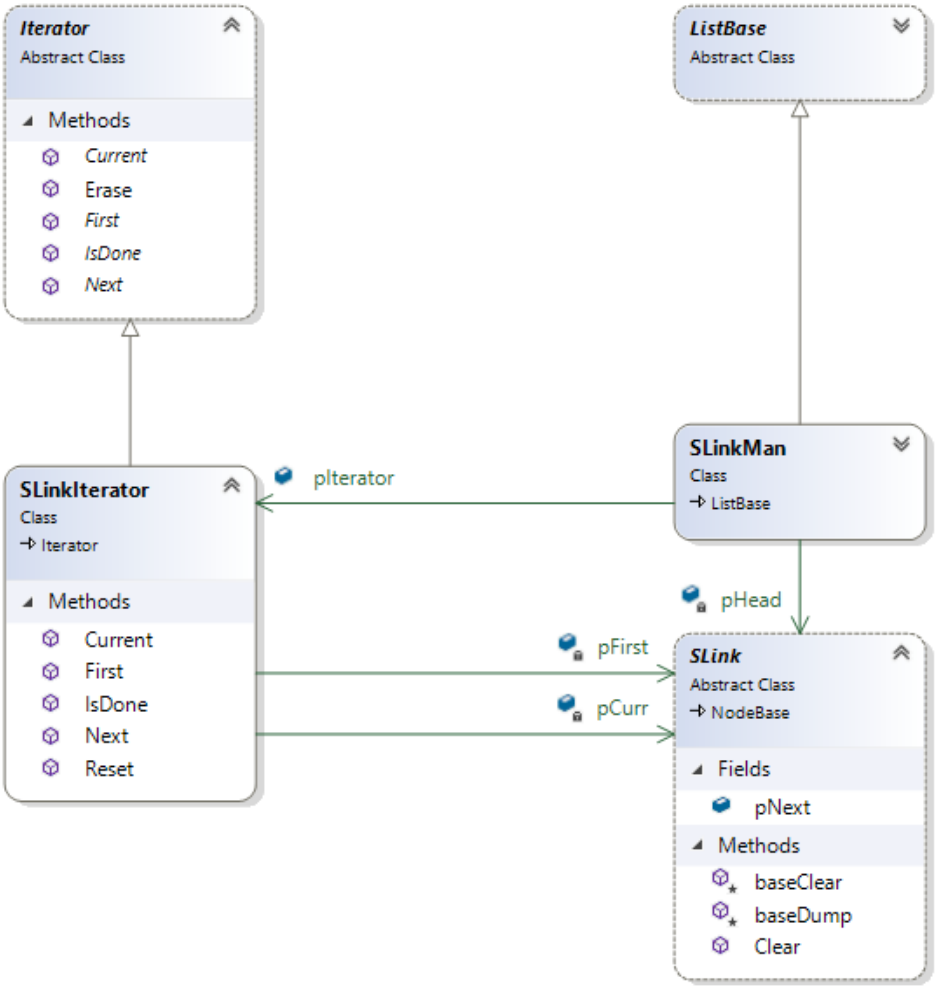
Because there is an abstract base class for the Iterator, new concrete Iterator types can be added for new collections, or for new ways to iterate over existing collections. Not only does the calling code not need to know how to iterate, but it does not even need to know the type of collection. The underlying iterator can be completely swapped out with something new, but the loop using the base Iterator type remains the same.

Implementation

There are several Iterator implementations in Space Invaders. The DLink iterator and Slink iterator are similar because both node types have a "Next" pointer which is all that is really needed to iterate. One difference is in the Erase() method because DLink nodes can remove elements in a different way. If there were a reverse version of these iterators, they would need significantly different implementations. Either way, code where we use the iterators does not need to know the details about the collection. For example, the Sprite Batches iterate over a collection of Sprites and render them. The Timer Event Manager iterates over the Timer Events and triggers their commands.

There are also the Composite Iterator and Reverse Composite Iterator which iterate over the Game Object Node structure to update each Game Object in scene. It plays a role in processing collisions and basically everything performed on a collection of Game Objects. The basic usage of the iterators looks like this:

$$Iterator\ pIt = new\ DerivedIterator(collection);$$
$$for\ (pIt.First(void);!pIt.IsDone(void);pIt.Next(void))$$
$$\{\ var\ pCurr\ =\ pIt.Current(void);\ \ldots\ \}$$



2.5 State Pattern

Problem

The player ship in Space Invaders has distinct behaviors depending on whether or not it has fired a missile. If a missile is currently in mid-air, the ship cannot fire another missile because there should only ever be one on the screen at a time. When there is no missile on the screen because it hit an Alien or the top of the screen, the player ship should be able to fire another missile immediately. Once the missile has been fired, the behavior of not being able to fire should immediately take over.

Solution

There are two distinct States that the player ship can be in. One state is ready to fire. The other state is not ready to fire, or waiting for the missile. There are also clear transitions between these states. The ready state transitions to the unready state when the missile is created. The unready state transitions back to the ready state when the missile exits the scene for whatever reason.

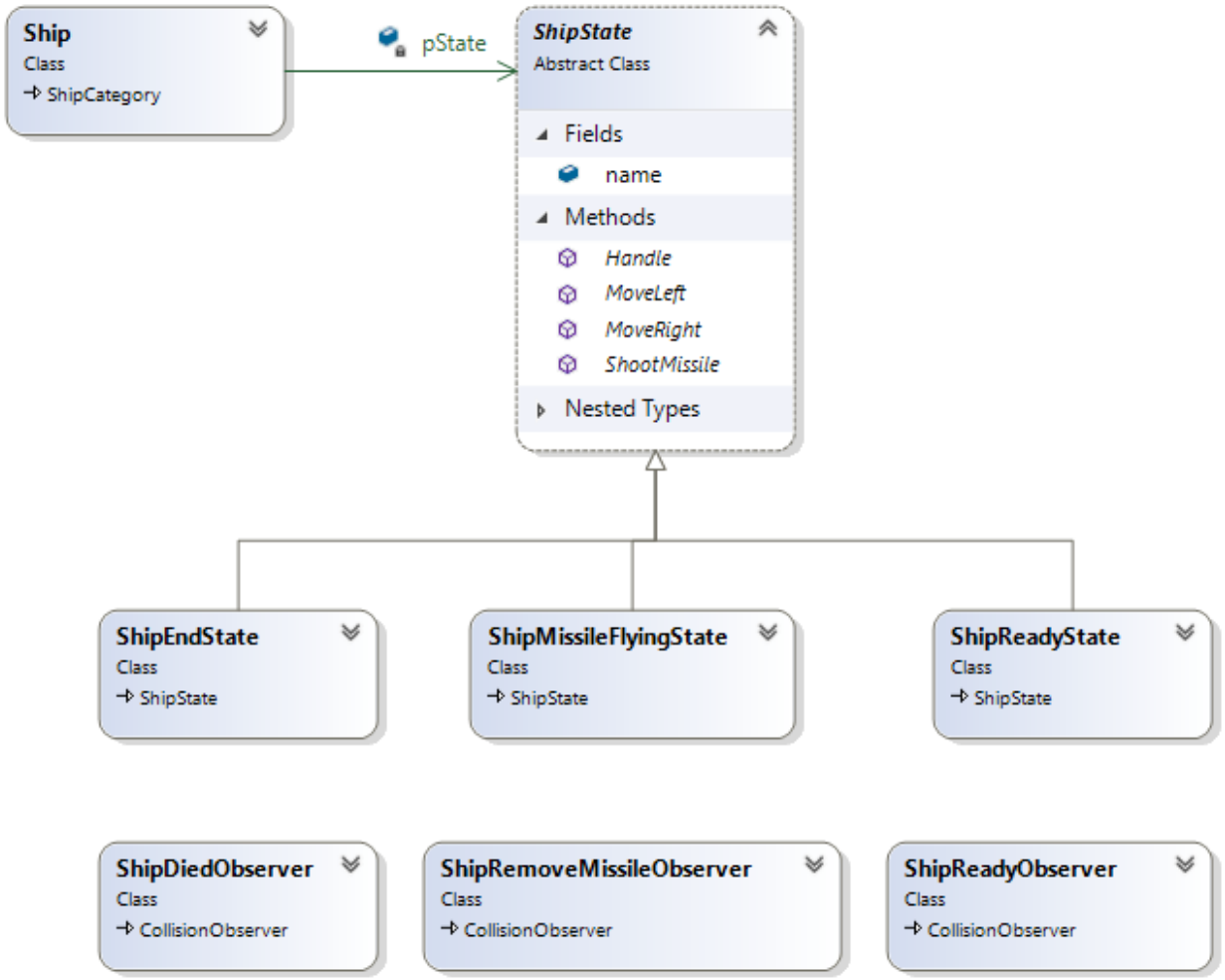
We can implement a State Pattern to encode these States and their transitions. There can be a Player State base class that defines the methods for a given Player State. Then the ready and unready states can be implemented as derived classes. One of the methods should be FireMissile(). In the ready state, this fires the missile. In the unready state, this does nothing. We can then create events in the game that trigger the transition of states at the correct time.

Intent

The intent of the State Pattern is to model a finite state machine using object oriented techniques. There is a base State class that is the common base of all states in the machine. The state machine has a current state and a reference to the other potential states. The states have transitions which define what the next state should be. A key aspect that distinguishes the State Pattern is the fact that the current state is expected to change many times during the state machine's lifetime. Not only are the states interchangeable, but the changing of the current state at runtime is what gives the state machine value.

The State Pattern takes a complex sequence of behaviors and transitions and breaks it up into individual state objects. Each state only has to worry about its own behavior and transitions. The states can be built up into an intricate machine that seems complicated from the outside, but each state is simple and easy to maintain. New states can be implemented and easily plugged into the graph of transitions.

Other code interacts with the State Pattern by executing methods from the current state. They do not need to know what that state is or write any conditionals, they simply call methods, and the current state will take over. The calling code can also indicate that particular events occurred. The state implementation may react to this event and transition to a new state.



Implementation

There is no real "State Machine" in the Space Invaders implementation, but existing classes like the ship become state machines in their own way by owning state objects. The Ship class has ShipReady, ShipMissileFlying, and ShipEnd. These states override the ShootMissile() function. Naturally, only the ShipReady state can fire the missile. The ShipEnd state is unique because you cannot fire the missile or move the ship. To make this work, the behavior for moving left and moving right has also been encoded in the Ship State base class, and the Ship delegates that behavior to the current state. The observers are shown here because they are what trigger the state transitions.

Another place where the State Pattern is used in the game is the scene system. The current scene of the game is effectively the current state of the game. It defines what will be updated and what will be rendered in that state. To do this, there is a base Scene State class with methods for updating and rendering, and the game just calls through to the current scene state. This also enables multiplayer by having a copy of the play scene, or state, for each player.

2.6 Visitor Pattern

Problem

When two Game Objects collide, the collision system does not know what type of objects they are. All we know is that the objects derive from the Game Object base class. We want to trigger a specific reaction for particular object collisions. For example, a missile hitting an alien should destroy the missile and alien. On the other hand, the alien grid hitting the side wall should reverse the grid direction and move it down. We want to have a specific function that we know will be called when the relevant collision occurs.

Solution

The Visitor Pattern can be implemented here to call the right collision handling function. We provide a base class Collision Visitor with overridable Visit() methods for each type of object that can be visited. We combine this with the Accept() method which must be overridden by all objects. Each Game Object will be able to accept another generic Game Object reference in Accept() and will call the appropriate Visit() method with itself as a parameter. The collision system will leverage these methods by taking two Game Objects of unknown types, and simply calling Accept() on one of the objects. The Visitor Pattern will handle the rest.

Intent

The intent of the Visitor Pattern is to leverage the nature of method overriding and the "this" pointer to gather type information about a set of objects. When a method is overridden, such as the virtual Accept() method, the body of code defining Accept() in the derived class has more information about the object's type than the client who is invoking Accept() from a reference to the base class. This more specific information about the type of "this" can then be used to call a more specific method, or overload of a method, on another object. This method is called Visit() and takes the more specific type as a parameter. Visit() is also overridden, so the body of the Visit() method in the derived class now has the more specific type from the parameter, and also the more specific type of itself from its own "this" pointer.

In the end, we have a body of code where we know the specific, derived types of the two objects that were previously ambiguous. Here, we can add specialized behavior. We can even change where the methods are overridden in the inheritance hierarchy to have different granularities of reactions.

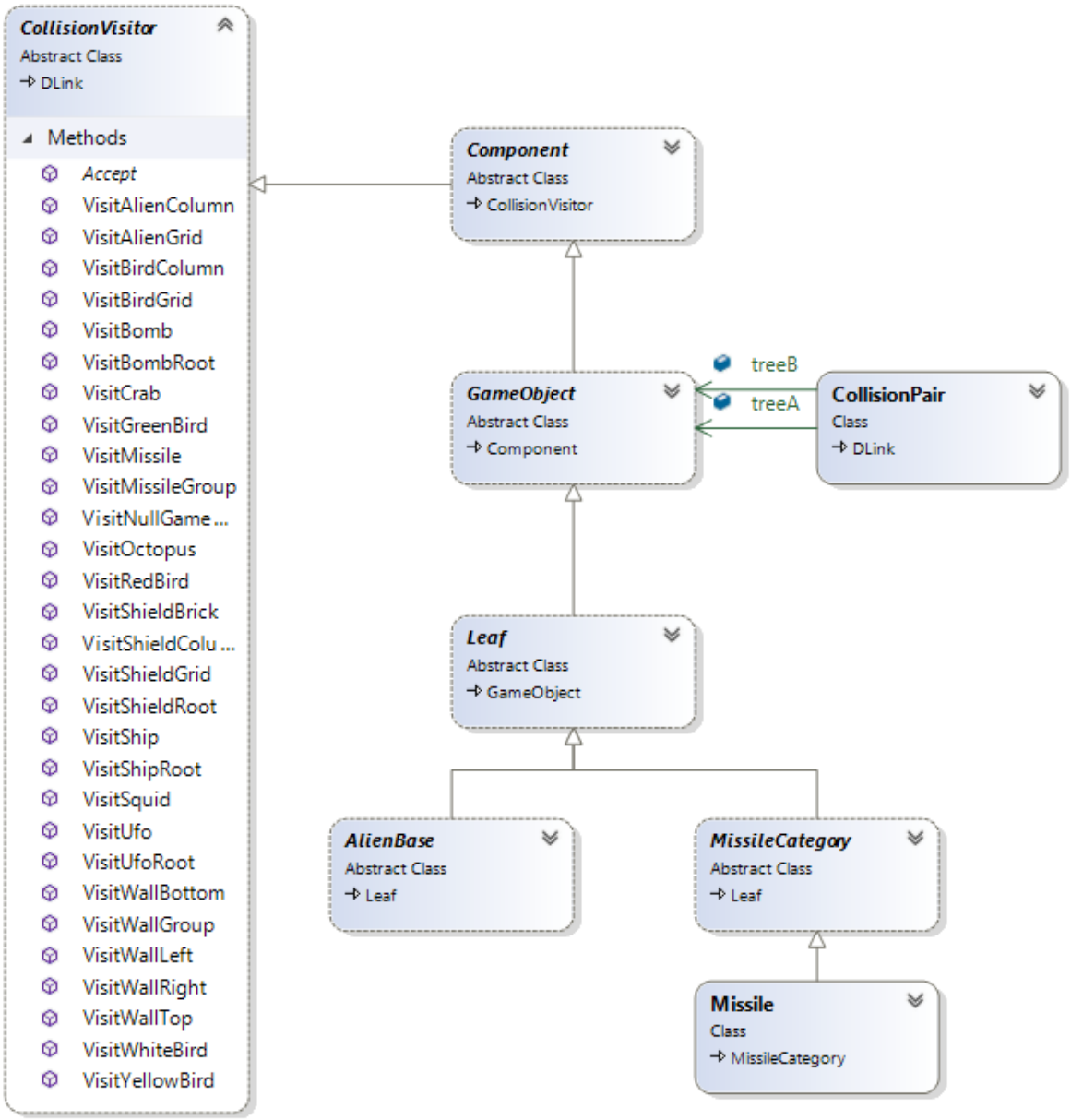
Theory

The Visitor Pattern requires a lot of code to work properly. The namespace of the class hierarchy becomes filled with a lot of methods that are potentially irrelevant to most objects. Additionally, the overridden methods are mostly boiler plate code with small tweaks that need to be repeated many times across many different objects. The end result, however, is very powerful and avoids a lot of bad alternatives. The Visitor shows how

object-oriented design can solve problems of type information without having dynamic casts or extreme conditional logic.

Implementation

The Visitor Pattern is implemented in the Collision Visitor base class of all Game Objects. Each Game Object must override Accept(). The visit methods in this instance have different names for each object rather than being overloaded. The methods are not pure abstract, but instead assert false by default. This prevents any combination being used unintentionally and requires any registered reaction to be explicitly implemented. The collision system can simply call Accept() with the two objects and trigger the correct reaction. Typically, the Visit() method with all the type information solved will trigger a collision Observer to implement some game mechanic.



2.7 Observer

Problem

When a collision occurs, we want a particular set of mechanics to kick in. For example, when an alien dies, it should play a sound, add score to the player, and spawn a death splatter icon. When the missile hits something, it should be removed from the scene and signal that the player is ready to fire again. Ideally, these reactions should be reusable because several collision events might have similar mechanics. It should also be easy to change and add to the set of effects when a particular collision occurs.

Solution

The Observer Pattern can be used to decouple a collision event from the set of effects that happen as a response to it. The Visitor Pattern will figure out the types of the objects and trigger a Collision Subject class by calling Broadcast(). Observers can be added to this subject so that some reaction can be implemented when the subject is broadcasted. The observers can be attached to the collision event when the scene starts.

Intent

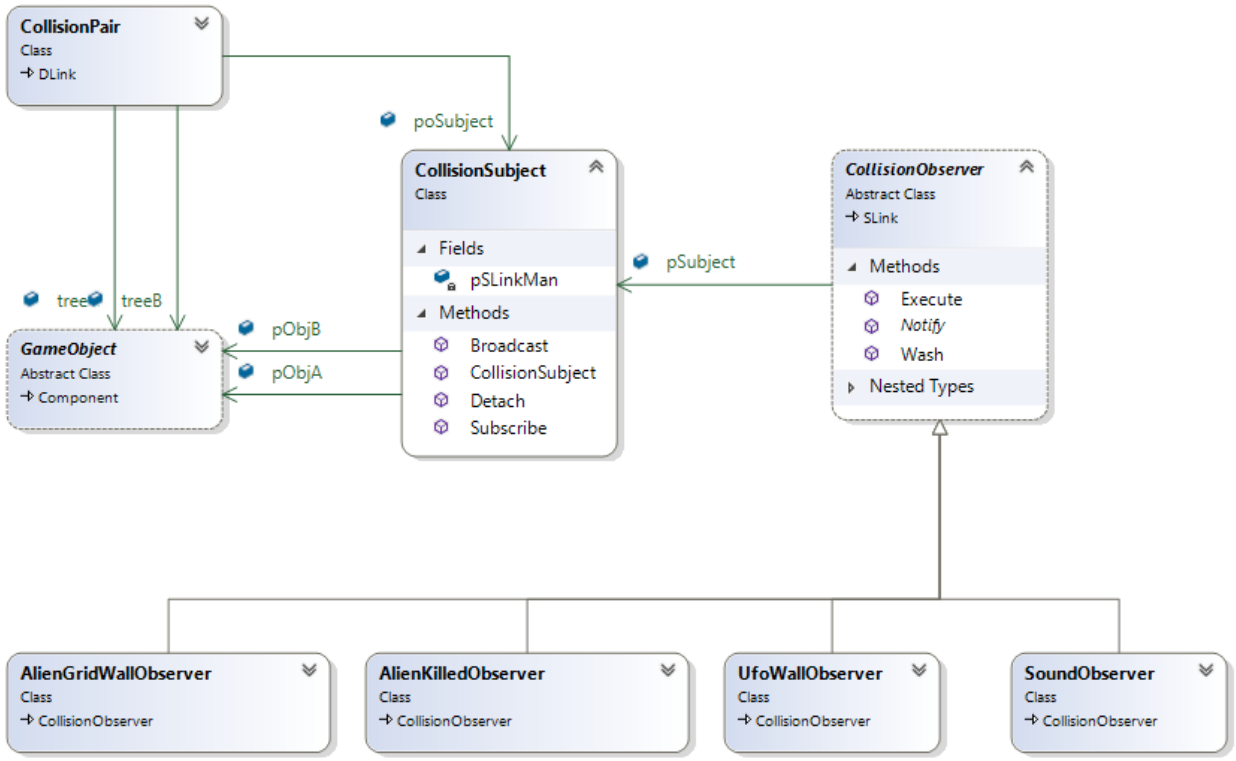
The Observer Pattern adds a layer of indirection between cause and effect. Specifically, the call site of the cause does not need to know what the effect is, how many effects there are, or if there are even any effects at all. The cause is called the Subject and simply executes a Broadcast() method. Any real reaction to this event is delegated to Observer objects. Because these reactions are reified as Observers, they can be placed on a list and executed in sequence. They can also be easily interchanged when the program design changes, and even interchanged at runtime to create more dynamic system.

The Observer is distinguished from the Command Pattern because the Observers subscribed to a particular Subject are persistent. They are not removed from the collection after being executed. This creates a different type of system where the relationship between Subject and Observer can be defined once and persist but is still decoupled in terms of its implementation.

Theory

The Observer brings up the idea that any reference to a single object can be replaced with a reference to many of those objects. Any operation that would be performed on the single object can applied sequentially to each element. No functionality is lost because that collection can always just have one element. However, many other objects can be added to the collection, and the collection can be modified at any point. Every level of indirection has some cost in complexity and performance, so this should only be added where appropriate. However, the Observer pattern has a pretty wide array of use cases because there are many instances where an additional level of indirection between a cause and effect can be added.

With the Observer Pattern, the “state” that is relevant to the event can be stored on the Subject or on the Observer. This depends on the nature of the problem. Sometimes, the Subject needs to provide the Observer with something to work with. In other cases, the Observer does not care at all what is going on with the Subject, they only want to know that something happened. Here, the Observer would just react to the event and source all of the relevant state itself.



Implementation

There are two implementations of the Observer Pattern in Space Invaders. The Collision Observers and Subjects handle collision events, and the Input Observers handle input events. The Collision Pair class holds a Subject that represents the event of that collision occurring. The Collision Observers can be attached to these events with Subscribe(). Using this pattern for the collision system makes sense because the inner workings of collision are generic and do not care about the game mechanics. The game mechanics are specific to the game and do not need to know how the underlying collision system works. So, the Observer Pattern adds the necessary layer of indirection between the low-level system and the game design code.

Using the Observer Pattern for the input system is also useful because specific keys should never be strongly bound to specific in-game events. There are many different control schemes and controller types for interactive applications, and we need to be able to swap out one scheme for any other. This will be easier if we can take the same Input Observers and re-assign them to different Input Subjects.

2.8 Command

Problem

There need to be events in that game that are triggered based on a timer. There are many different types of these events, but they should all use the same timer system. We need to be able to trigger these timed events efficiently without knowing what they are going to do. Once the timer event has been triggered, it should not be triggered again because it is a one-off event. However, a timer event should be able to re-register itself to create a repeating effect.

Solution

We can implement the Command Pattern to define these one-off events as derivations of a Command base class with an Execute() method. The timer system can then contain the general purpose logic of calculating time passed and also a list of commands. It can check the requested time of these events and trigger them when needed by calling Execute(). Then, it will remove the Command object from the set of timer events. The nature of Execute() will be defined in the individual commands and can do anything. They can be added with a trigger time to the heterogeneous collection of commands in the timer system.

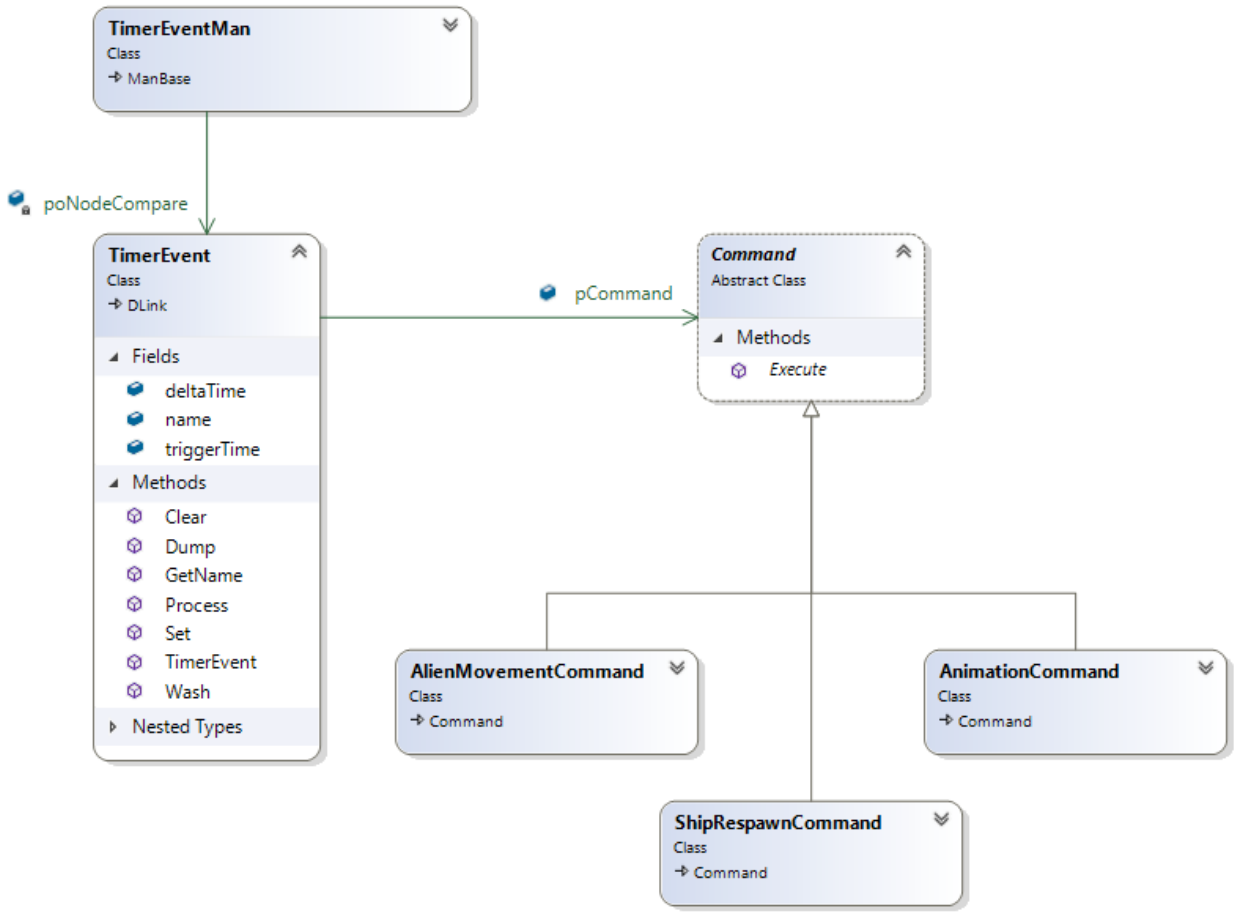
Intent

The intent of the Command Pattern is to reify a unit of work as a Command. Because the work is now a thing, it can be moved around, stored, and executed at any time. In the traditional Command Pattern, the Command is executed once, then discarded. This creates a different type of system than something like the Observer Pattern because the relationship between the cause and effect is ad hoc rather than persistent.

Another aspect of the Command Pattern that is sometimes forgotten is the ability to delay the execution of the work defined in the Command. This allows for creating a hysteresis effect where the Command is delayed and lags behind whatever caused it. This lag creates an opportunity to only execute the Command at the appropriate time. This useful when the Command has potentially disruptive behavior that needs to be executed in controlled environment and at a safe time.

Theory

Reification is the core concept behind what makes the Command Pattern so powerful. Meaning “to make into a thing”, reifying a piece of work as a Command object takes the abstract behavior expressed as code and converts it into a tangible object. The object can be added to a collection, moved around, and even sent over the internet. As a result, the intent of the Command Pattern described above is enabled.



Implementation

The implementation of the Command Pattern for the timer system is shown here. The Timer Event Manager holds Timer Events, which contains trigger time information as well as the Command to execute. The abstract base class for Command has an Execute() method that is overridden by the concrete commands. These include animation, ship respawns, and alien movement. There are many other commands that can be added to the Timer Event Manager because a timed event is a widely needed concept. The Command Pattern perfectly meets this need.

A unique aspect of the timer system is how time in the game only occurs at discrete steps separated by milliseconds. No event is ever triggered at the physically correct time, but is instead triggered on the closest possible frame. The Command Pattern accommodates this simulation by allowing the system to iterate over all events that should be triggered on a given frame.

The Command Pattern is also used in the Death Manager, where commands to kill objects are added throughout the frame. The Death Manager waits until the end of the frame to execute the kill commands. This utilizes the lag effect of the pattern and prevents the removal of objects from disrupting the scene mid-frame.

2.9 Composite

Problem

The scene contains many Game Objects of various types, and there is a semantic hierarchy of objects. For example, the Aliens are organized into a 2 dimensional grid structure and moved as a single unit. Also, the Shields are organized into larger structures that need to be replicated. It would be useful to group objects and perform operations on them collectively. We also want to add groups of groups. This grouping could be useful in optimizing a collision system, where we could reduce the number of collision-checks by having a more hierarchical approach.

Solution

The Composite Pattern can be used to define these groups as Composite classes, and define the real objects inside the groups as Leaf objects. The Composites and Leaves are both derived from a common base Component class. Composites will hold a collection of other Components, which can be either other Composite groups, or real Game Object Leaves. The result is that the semantic scene hierarchy laid out above becomes a literal tree of objects. The objects can then have their collision dimensions be based on this structure to enable a more efficient collisions system. We can also use the tree structure to operate on a subset of objects in a simpler way by implementing an Iterator for the Composite objects.

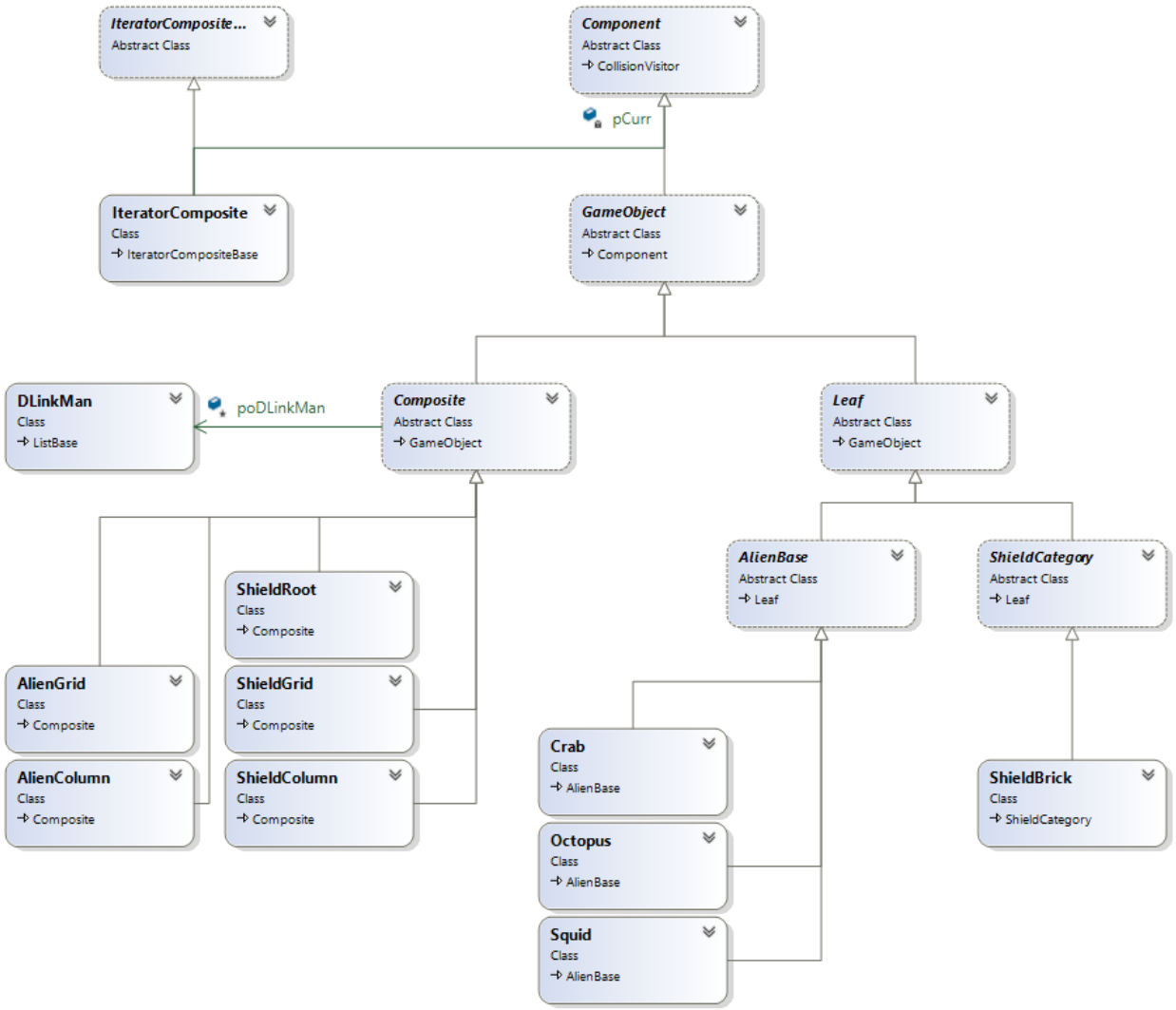
Intent

With the Composite Pattern, the intent is to define a tree structure using object-oriented techniques. This is done by giving Composite objects a collection of other Composite objects or Leaves. Operations performed on parent Composites can be percolated down to the child objects using member recursion. An elegant example of the Composite pattern is a behavior tree, where logical Composite nodes such as sequences and selectors can be combined with game-specific leaf nodes that implement some behavior. From there, a complex tree of behavior can be built up.

Grouping objects into a hierarchy with this pattern is probably the most widely applicable of all the object-oriented techniques. Hierarchy allows high level systems to operate on the surface level root objects without having to worry about all of the complexity that may be built up in the branches.

Theory

Because it is an object-oriented pattern, the Composite Pattern is more than just a tree. The Composite objects can be customized with derivations to have specific behaviors. These specialized Composites may implement new behaviors for how to pass on requests to their children and react to their properties. These nodes can be extremely complex and utilize Strategies, Iterators, and other object-oriented patterns to enhance what can be done with a hierarchy of objects.



Implementation

In Space Invaders, there is a Component base class for all Game Objects in the Composite hierarchy. The Composite Game Objects hold a DLink list of other Components, which can be either other Composites or Leaf objects. Leaf objects are always real Game Objects that typically have some visual representation in the scene. There is a Composite Iterator to iterate over these trees in the correct order.

The Composites in the game derive their size and location from the mathematical union of all of their child objects. They also do not have a real Sprite and use the Null Object Pattern so that nothing is rendered. The collision system uses the derived union-based dimensions of the Composite objects to perform a hierarchical collision check. If a colliding object is outside of the parent Composite, then it can be ruled out from any collision with a child object.

The pattern also makes it easy to locate an object based on its position in the Game Object hierarchy. It gives context to an object in relation to the rest of the objects in the scene.

2.10 Object Pools

Problem

Space Invaders is a real-time application where performance is important. Allocating and deallocating memory can be an expensive operation, and it is frequently unnecessary for objects that could potential be reused. Objects like Sprite Nodes or Game Object Nodes may be frequently destroyed and recreated. We can afford some memory overhead for our system because speed is more important.

Solution

We use Object Pools to control the memory management of asset types and enable the recycling of their memory within the system. The Manager Base class will have an active list and a reserve list. The active list represents the objects that are alive on the manager and have meaningful properties. The reserve list has objects that have been discarded. Rather than deallocating these reserve objects, we hold onto them and clear their properties. Then, when a new object is needed, we return one from the reserve list to be recycled and load it up with useful data.

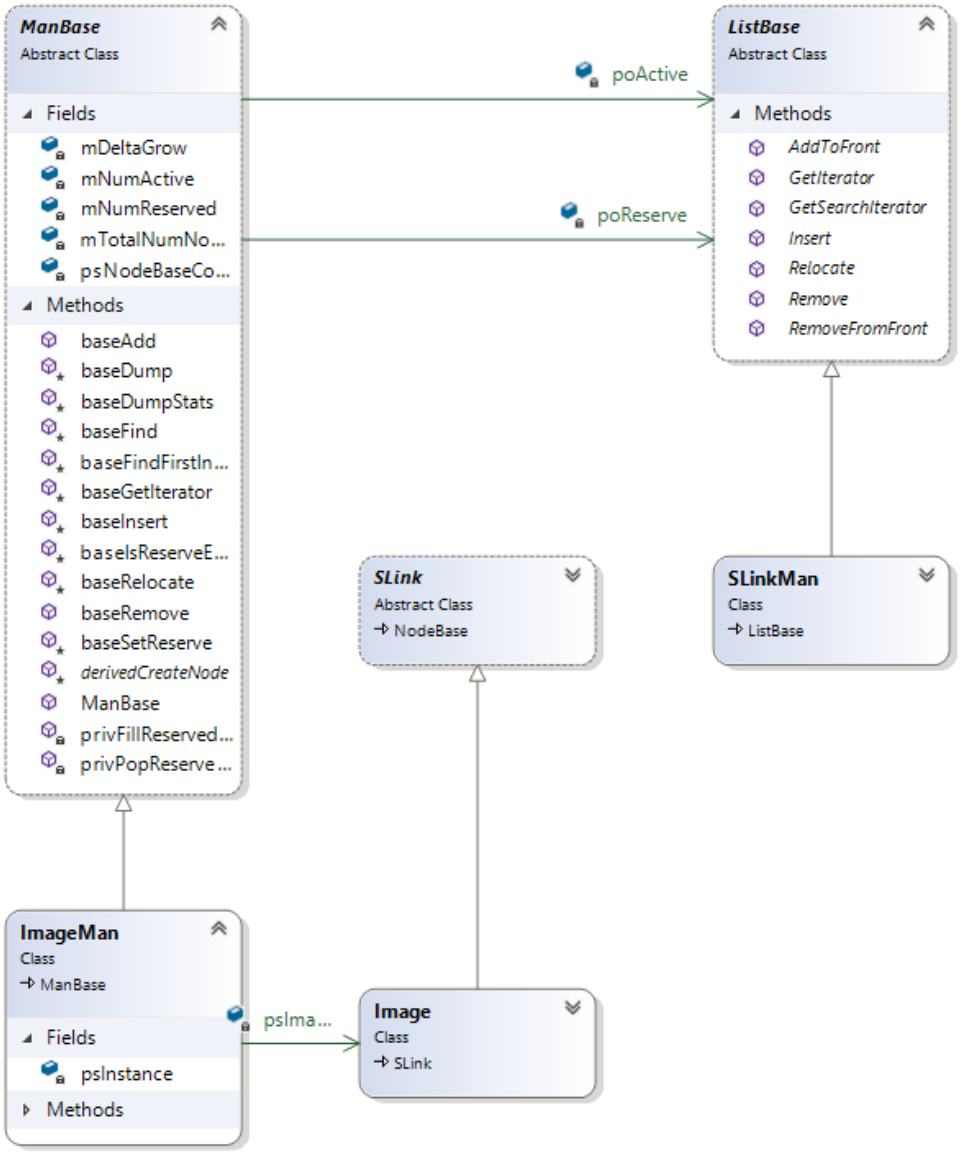
Intent

The intent of the Object Pool is to control our application's relationship with the memory system. The Object Pool is greedy with memory and holds onto objects. This is beneficial because we are betting on the fact that objects of this type will be requested by other parts of the code base at some point in the future. When they do make such a request, we give them a recycled object that has already been allocated. If we run out of reserved objects, we create more and return one. The interface to create a new object is always the same from the client's perspective, and they have no way of knowing whether the object was freshly allocated or recycled.

There are several ways to implement an Object Pool with different levels of sophistication. A simple Object Pool holds onto every reserve object and can be combined with a Factory Pattern to provide the interface for object creation. A more complex Pool might have a garbage collection system that clears out objects.

Theory

By implementing an Object Pool as part of the application architecture, instead of relying on built-in allocation and garbage collection systems, the deallocation and reservation of memory can be more carefully controlled and coordinated to maximize performance. Customized Pools have more information about the system and can be cleverer about how to deal with memory than any general-purpose memory management system. A real time system could combine the delay tactics from the Command Pattern, creational aspects of the Factory Pattern, and memory management of an Object Pool to create a robust object lifetime framework.



Implementation

The Object Pool Pattern is implemented in the Manager Base class so that the behavior can be shared across all manager types in the game. The Manager Base has an active list and reserve list. Items can only ever be added to the active list from the reserve list. If the reserve list is empty, it has to be populated before adding to the active list. The types of the Object Pool collections can be customized by the derived manager class. This system allows all asset types to be pooled within Space Invaders. All long term or re-usable objects are never deallocated and can be recycled quickly during the game loop.

Another area where the Object Pool Pattern is used is in the Ghost Manager, where Game Objects can be pooled. Game Objects are frequently added and removed from the scene and the Ghost Manager can resurrected any removed object.

3 Conclusion

3.1 Commentary

Overall, the design patterns increase the cohesion of systems in the game. The reactions to events are easy to correlate with their causes, whether that is a collision, input, or timer. The system is optimized by various patterns by reducing memory allocations, using hierarchical collision checks, and avoid conditional checks in many instances. The design patterns also reduce the coupling of systems. Large parts of the code base can be refactored without affecting other parts. New mechanics and features can be added easily without having to understand every part of the project.

3.2 Improvements

Assets and Mix-ins

One improvement I would like to explore is refactoring the relationship between asset types and the collections on which they are stored. Currently, the Sprites, Images, Textures, etc derive from a specific Link type and can then be made part of that type of list. The managers have to use the list type that works with the asset they are managing.

I think it would be interesting to have an Asset Base class that all Sprites, Images, Textures, etc derive from. Then, DLink and SLink would become concrete classes with a pointer to an Asset Base. This would decouple the asset type from the Link type and allow any asset to be stored on any type of list. This would be useful for the Sprites, where the Sprite Manager only needs an single-linked list where as the more dynamic Sprite Batch benefits from a double-linked list. The only change would be when an object is dereferenced from the linked list because you need to return the pointer to the asset instead of the Link node itself.

A counterargument could be that this extra level of indirection is too heavy weight for such a low-level part of the system. The mix-in technique also allows the linked list code to be very simple.

Scene Level Managers

Scenes in the game were implemented towards the end of the process and I think they are suboptimal. The way that the singleton managers have to be reset when the scene is changed is confusing. This lead to most of my bugs in the end, where I wasn't sure what was a scene level resource and what was a global resource.

I think this could be improved by making the singletons non-static classes that just exist as components of the scene, and can only be referenced from a reference to the current scene. This would make it more obvious which assets are relative to scene and which assets are global.

Game Mode State

Single player versus multiplayer was also added at the end. It could be refactored as a state machine so that the different behaviors between modes are handled more cleanly. The rest of the game could be further decoupled from the logic of single vs multiplayer.

Unified Game Object Lifecycle

There is a lot of common logic for creating each type of Game Object with only minor differences. I think this process could be generalized so that there is a unified, safe way to create new objects.