DeAngelo Wilson
SE456
Design Pattern Document

Overview

This game is the result of several different systems layered on top of each other and working together. The first of these components is the management of global resources, in order to manage this, we made use of the singleton design pattern as well as object pools for resources such as the *Sprites*, *Images*, *Textures*, *Fonts*, and inactive objects (through the *GhostManager*). By storing these resources in a shared, or globally accessible location we are able to reuse them throughout the different scenes and game modes of the application. With that said, we also make use of the state design pattern to segregate stages of the game into distinct scenes, each of which has resources unique to their situation: including a SelectSceneState, AttractSceneState, PlaySceneState, and GameOverSceneState. We cycle through these different scene states based on certain external events within the current application state. Some of these unique resources each scene has include systems such as the *TimerEventManager*, *CollisionPairManager*, as well as the *SpriteBatchManager*" and *GameObjectNodeManager*; these remaining systems enable a majority of the functionality throughout the game.

Starting with the most primitive, the SpriteBatchManager facilitates the Draw() stage of the game loop, and provides the capability to specify a draw ordering, through the creation of unique *SpriteBatch*-es which maintain a draw priority, dictating the batches draw ordering. By attaching *GameObjects*, or more appropriately their *SpriteProxy* component, to a SpriteBatch we enable the drawing of the Sprite. A similar concept is used to enable a clean Update() stage of all active GameObjects, through the use of the GameObjectNodeManager. In addition to the singleton manager, this functionality uses the composite pattern, seamlessly enabling the updating of groups of objects. So, by attaching a root Composite node to the GameObjectNodeManager, all children, which form a tree structure as a result of the composite pattern, can be updated each frame through a single Update() call to the manager. Along with the GameObjectNodeManager, which manages active GameObjects, there is also the GhostManager, which manages inactive, or recycled, GameObjects. This object pool manager, maintains references to all GameObjects which have been marked for delete and subsequently deactivated,

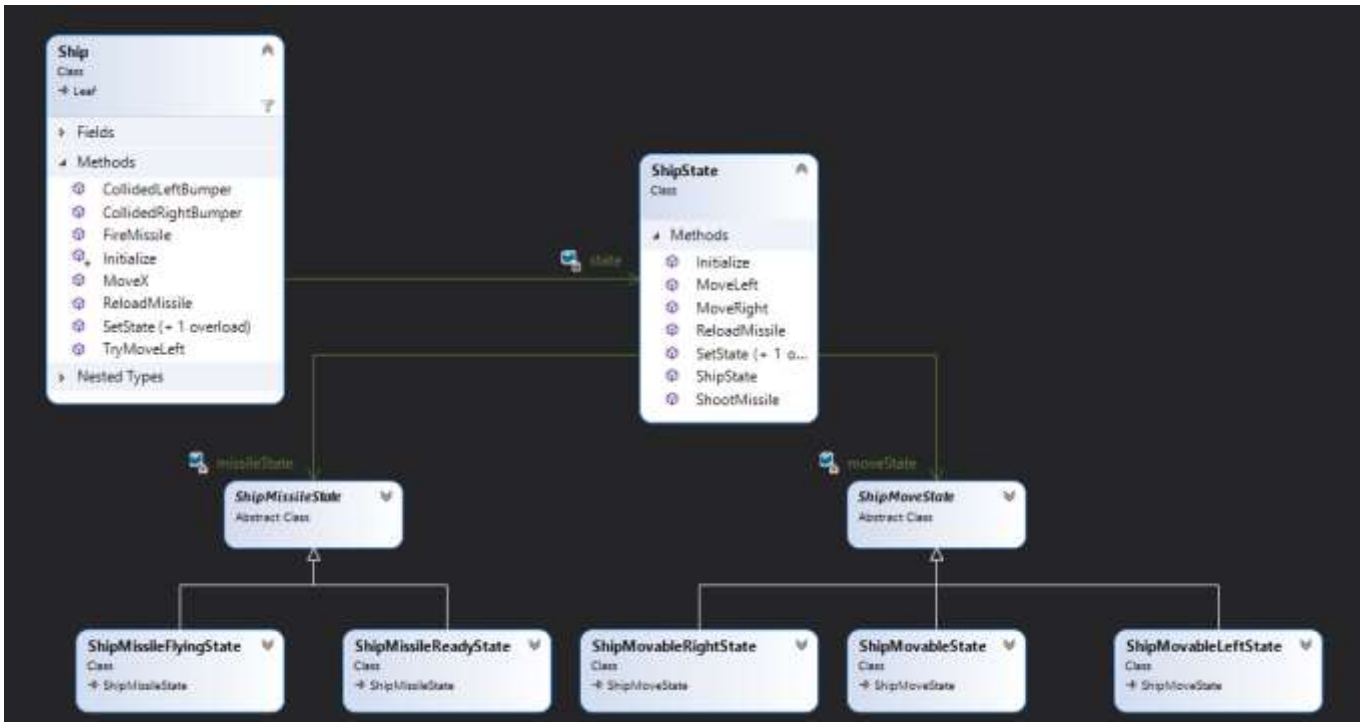
and is then also used for reviving deactivated GameObjects as a way to avoid extraneous construction of GameObjects.

Besides drawing and updating objects, another key component of the game is the collision system which is managed by the CollisionPairManager. Firstly, this system establishes *CollisionPairs*, or 2 composite GameObject types which merit the need for collision checking. After these have been established and attached to the manager, any necessary side effects of a collision event can then be attached to the CollisionPair in the form of an observer. In addition to the observer design pattern, the collision system makes use of the visitor pattern in order to unwrap composite groups to their underlying children which are collidable objects. The TimerEventManager is used extensively throughout the application to enable the execution of commands of events based on a global time. With that, its implementation makes use of both the command design pattern as well as a priority queue to enable an early out processing of *TimerEvents*. The last large system in the game, the *InputManager*, is another shared resource which makes use of the singleton design pattern as well as the observer pattern. Any active state in the game can receive notifications, through the observer design pattern, on the event of relevant, observed keyboard input.

The infrastructure described above facilitated the robust and extensible implementation of a majority of the game mechanics included in the Space Invaders 1978 Arcade game.

DeAngelo Wilson
SE456
Design Pattern Document

State Design Pattern



The State design pattern is a flexible pattern that allows for an object to be given a set of behaviors which it can dynamically, at run-time, alternate between.

The intention of this pattern is for a seamless, from an external viewpoint, changing of behavior based on the current situation, or context, and object is in.

This entails our context, Ship, own through composition a State, Ship::state. This pattern works by the context object, in our case “Ship”, delegating the decision on course of action to it’s State behavioral object. So, for any execution of state dependent behavior, the Ship class, in this example, will ask the State what it should do given the event, for instance: input to move left [Ship::TryMoveLeft()]. So, States derived from the same base State all implement the same set of all behaviors that are dependent on the particular state.

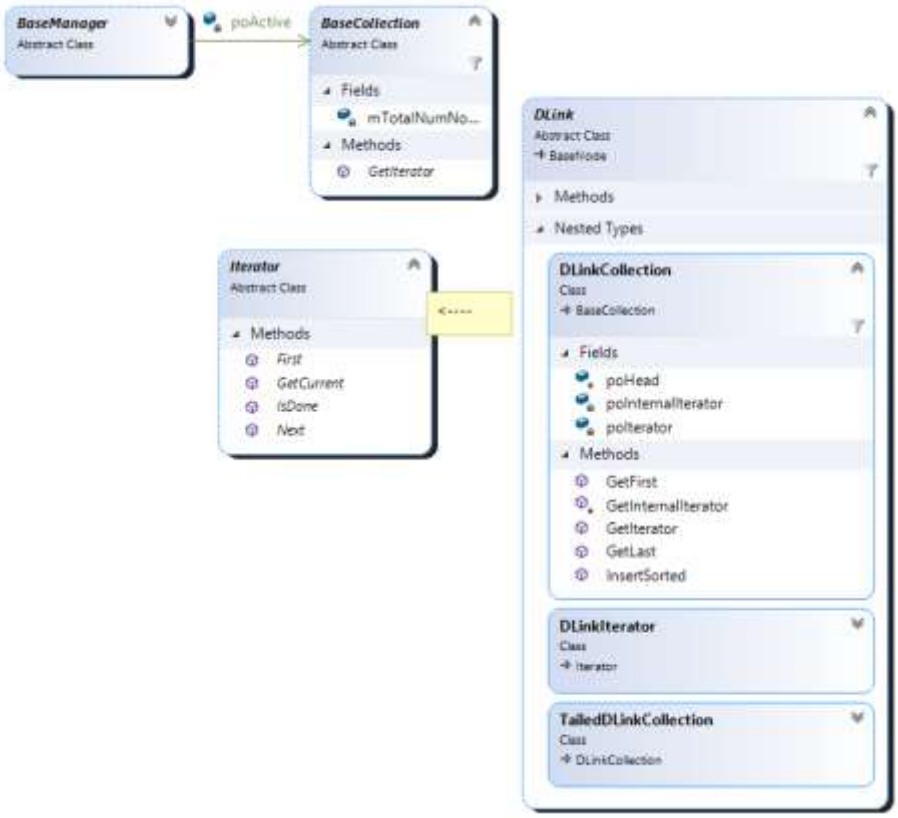
By having our specific States derive from a common abstract base class (i.e. ShipState), we are able to interchange the Ship’s Ship::state on any internal state updates. In order to perform these internal state update we firstly use the States ShipState::Handle() method. This is used when a particular derived State knows exactly, in all cases, the next state to transition to. In

the case where a derived State cannot, without further information, decern the next state to transition to, the state must be set externally; this occurs for the ShipMovableState. In this case, one could transition to either the ShipMovableLeftState or the ShipMovableRightState. Instead of passing additional information to the ShipState::Handle() method which would result in a brittle design, I simply prohibit the use of the Handle() for that state and will use ShipState::SetState() to transition from the ShipMovableState on a event. In particular for this case, on a bumper collision event either callback, Ship::CollidedLeftBumper() or Ship::CollidedRightBumper(), is called.

The problem we solve using the State design pattern is the need for multiple different behaviors dependent on a scene that is constantly changing and new events consistently coming in. So, using this design we are able to avoid the need for a set of conditionals or switch statement which would need to account for all possible states, all squeezed into one or several methods, based on the number of state dependent actions. This solution is better in terms of being more extensible, without need for much modification to existing code, and is optimized as it requires significantly less conditional checks. This State design pattern solution is more extensible due to the nature of having each state specific functionality separated into distinct derived State class files. And it does not require conditional as instead we simply maintain what our current state is at all times, and the State has the functionality for each state dependent action.

DeAngelo Wilson
SE456
Design Pattern Document

Iterator Design Pattern



The Iterator design pattern is a very simple and useful pattern which essentially provides the ability to traverse a collection from beginning to end, while hiding the details of that traversal through the implementation of a general API. The intention of this pattern is to provide a unified, or abstract way to iterate, or traverse nodes in a collection without being forced to release any resources or knowledge about the collection.

This design pattern solves the issue being able to modify or traverse a collection without knowing what type the collection is, for instance in the case of an abstract manager, which is exactly what it is used for in this application. The *BaseManager* class is an abstract base class used for majority of collections and, by forcing a *BaseCollection* to have an iterator we are able to treat each different kind of collection (i.e. doubly-linked list, singly-link list, tree) the same, as

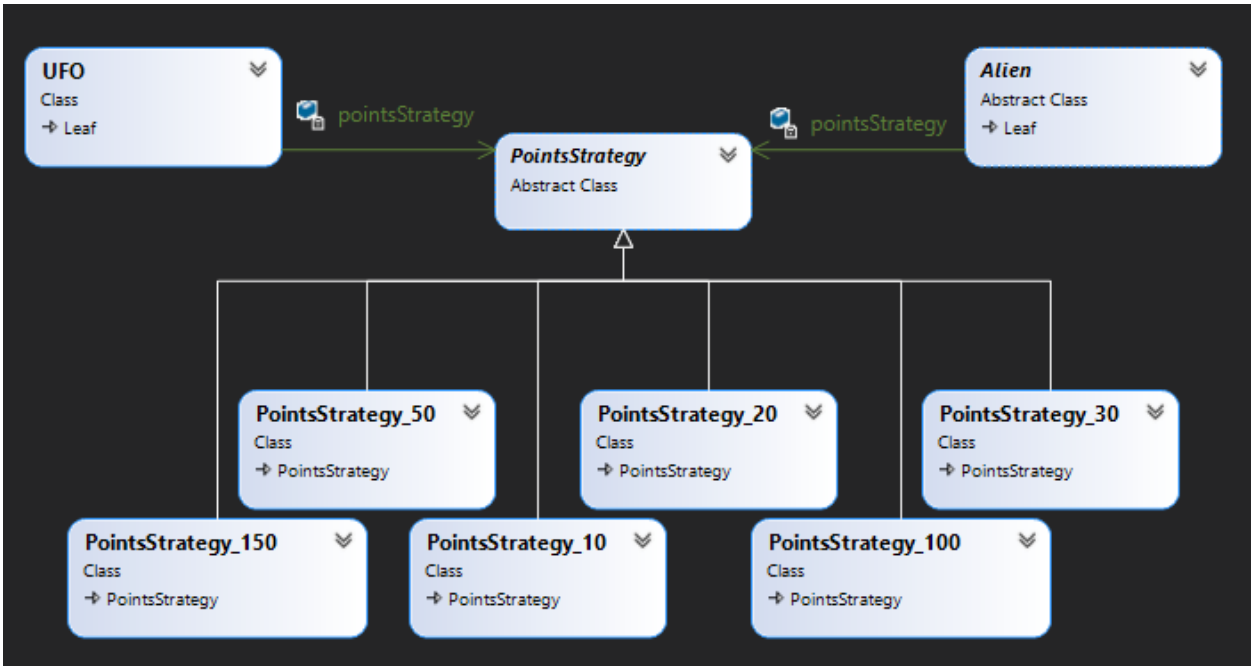
our BaseManager is able to stay completely agnostic from the implementation of a derived BaseCollection.

The common solution to this challenge would be to simply create multiple different abstract BaseManager classes for each different kind of BaseCollection, and each of those manager classes would assume full knowledge of the implementation of its associated BaseCollection. This solution does not allow for any form of reusability among managers and rather forces much code duplication for common functionality of any BaseCollection, such as: adding or removing elements, or iterating and printing elements of a collection.

In this application we use an iterator in two instances, one of which is an internal iterator which simply acts as a wrapper around the logic for iteration, and the other case is in conjunction with the BaseManager, in order to maintain its agnostic viewpoint. The iterator is very clean and packaged which facilitates the ease of execution in either providing or restricting access to iterating the contents of a collection. An interesting trade-off with this implementation of an iterator is that a collection maintains a single external iterator, and as such it only needs to be created once on construction and then can be re-used all through out the lifetime of the application. A constraint this places is that, since there is only one iterator for a collection, every retrieval of the iterator resets its, therefore this is not a suitable design for a multi-threaded environment, but, with consideration, fit the needs of this application perfectly.

DeAngelo Wilson
SE456
Design Pattern Document

Strategy Design Pattern



The Strategy design pattern is a simple pattern which is very closely related to the state pattern, but more rigid in its design and use. The design is essentially just an inheritance hierarchy for a particular set of behaviors, in which the base class is bound through composition to a subject. The intention behind this pattern is to enable the ability to seamlessly change a behavior of a class or instance on construction.

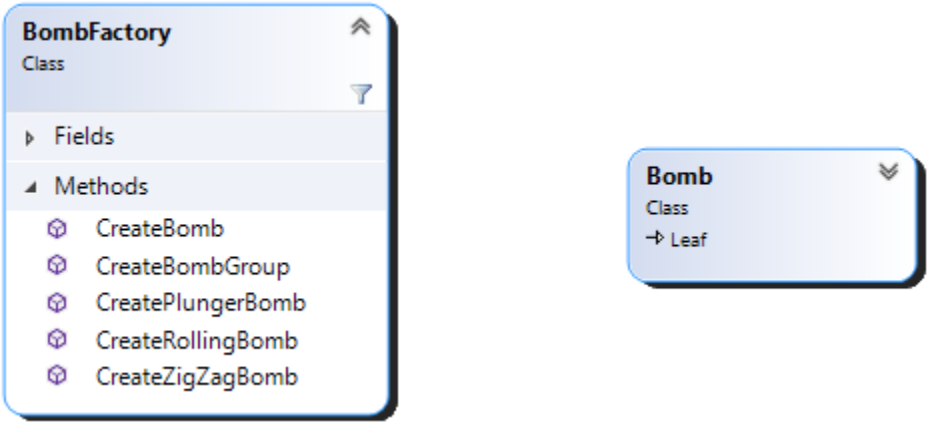
This pattern works by encapsulating all needed functionality for a particular behavior within a separate class, abstracting out to a base class, and providing an API for the subject class, which owns the strategy through composition, to make use.

This design pattern solves the problem of needing to define certain behaviors of a class during run-time and provides a clean way to go about that. The alternatives, or common coding solution to this problem would be to create larger inheritance hierarchies or to provide some sort of flag or unique value on construction which would then be used, through a conditional check, to determine the correct behavior to call. This firstly forces the behavioral logic to reside all within the subject class and also either forces a very fine tuned inheritance structure or additional

conditional checks. The strategy design is better than the common solution for both of those reasons, as is more flexible than the alternative sub-classing solution, is more optimized, and by having particular logic separated into its own class, we are able to create a more robust solution which is more closed to modification, but extremely open to extension.

The use of strategy within this application is restricted entirely to the subject class, as the subject privately composes the strategy, therefore, beside for the construction, the strategy is completely encapsulated within its subject. A *PointsStrategy* is given to a derived Alien on construction and it never changes, while the UFO's strategy changes every initialization in accordance with the mechanics of the SpaceInvaders game. The Strategy pattern can be considered to be fairly restricted in its use case, as it is intended to be constant for a class, and especially so when compared to the state pattern. Although, that is by design and it is a rather flexible design in its own right due to the ease in switching between strategies.

Factory Design Pattern



The Factory design pattern is a creational pattern which encapsulates the logic for instantiating objects. The intention behind this pattern is to provide a way to create variations of a particular common interface.

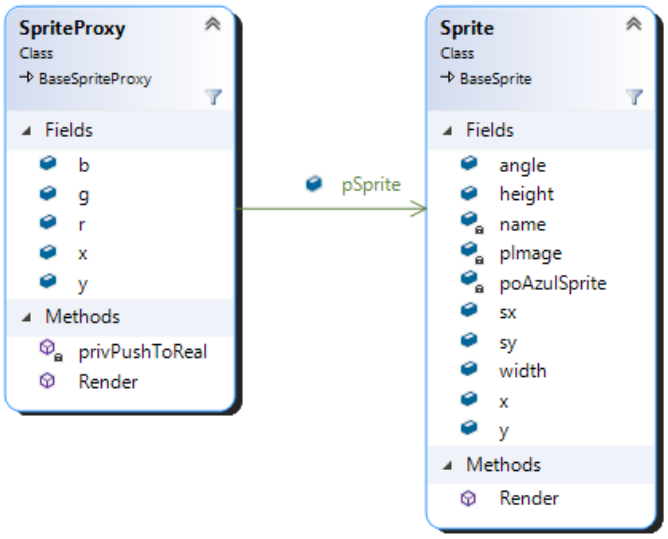
This design works by providing a simple and consistent API that takes in as parameters all necessary data needed to instantiate a particular object type. It does this while hiding specific

details needed for this instantiation and only requiring the varying necessities. The factory design pattern implementation can entail a method or class.

The factory design pattern is fairly simple as it provides a way to create objects, but it benefits is that it is able to hide complex creation as well as streamline repetitive creation of objects, especially in the context of games.

For the context of the alien grid in SpaceInvaders, an alternative implementation to this design pattern could be to instead have the *AlienGrid* hold the logic for initializing itself, creating its columns and aliens, and then moving the creation of the AlienGrid into the aggregating class, which in my case would be the *Player*. This solution is significantly less robust than having all Alien related creational logic in a single class.

Proxy Design Pattern



The Proxy design pattern is a structural pattern, which is incredibly simply in its implementation, but provides a huge benefit in optimization. The Proxy pattern is essentially a wrapper around a resource and its intention is to essentially enable the reuse of that resource.

The Proxy design works by the proxy holding a reference to a resource through aggregation, in our case a *SpriteProxy* maintains a reference to its underlying Sprite resource.

This resource is shared among 1 to many Proxies and is continually updated and overwritten with a proxy's instance data.

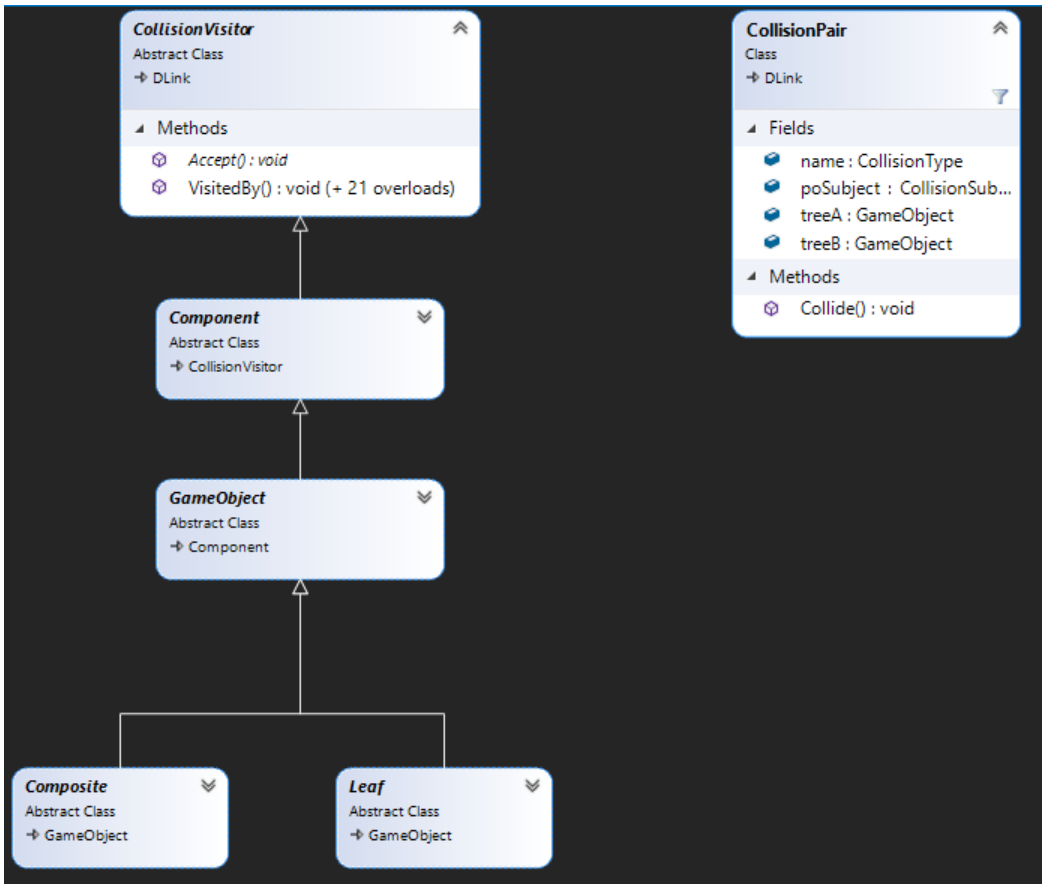
This pattern addresses the issue of duplicate screen images, or sprites. In our context, a particular alien sprite is drawn several times onto the screen with only the x, y position of the sprite changing: this is exactly what the Proxy takes advantage of. Since only a small part of the underlying resource needs to be updated, those members are duplicated into a proxy for that resource and anytime we need to update an instance of the sprite we simply push the instance data of the proxy onto the underlying resource and, for the Sprite, Render it to the screen.

This design pattern is simply an optimization, as it avoids unnecessary construction and destruction of reusable resources (i.e., Sprite) through the use of lightweight placeholders. The alternative to the proxy design implementation would be to simply duplicate the resource for each instance needed, despite very little data changing between each instance. In the case of the aliens in the game, on initialization of the application a sprite would be needed for each of the 11 instances of the "Squid" alien.

In this application, the main use of proxies is in relation to drawing Sprites. The instances of the proxy object, SpriteProxy, are attached to the SpriteBatchManager and when drawing each proxy, we simply push to its underlying Sprite resources and render it, essentially stamping it onto the screen at a particular position and then repeating for each proxy with its instance data.

DeAngelo Wilson
SE456
Design Pattern Document

Visitor Design Pattern



The Visitor design pattern is a behavioral pattern, which allows us to add external functionality to a class. The intention behind its use in our case is primarily for its ability of double dispatch, which allows us to determine the concrete types of the two classes involved in the accept/visit.

The pattern consists of 2 types of methods which are implemented by all classes which are involved in the visitor pattern. The first of which is the Accept() the first dispatch in this pattern, the second is the VisitedBy(ConcreteVisitor v) method, the second dispatch, and as such within this VisitedBy method we are able to have the concrete types of both classes involved.

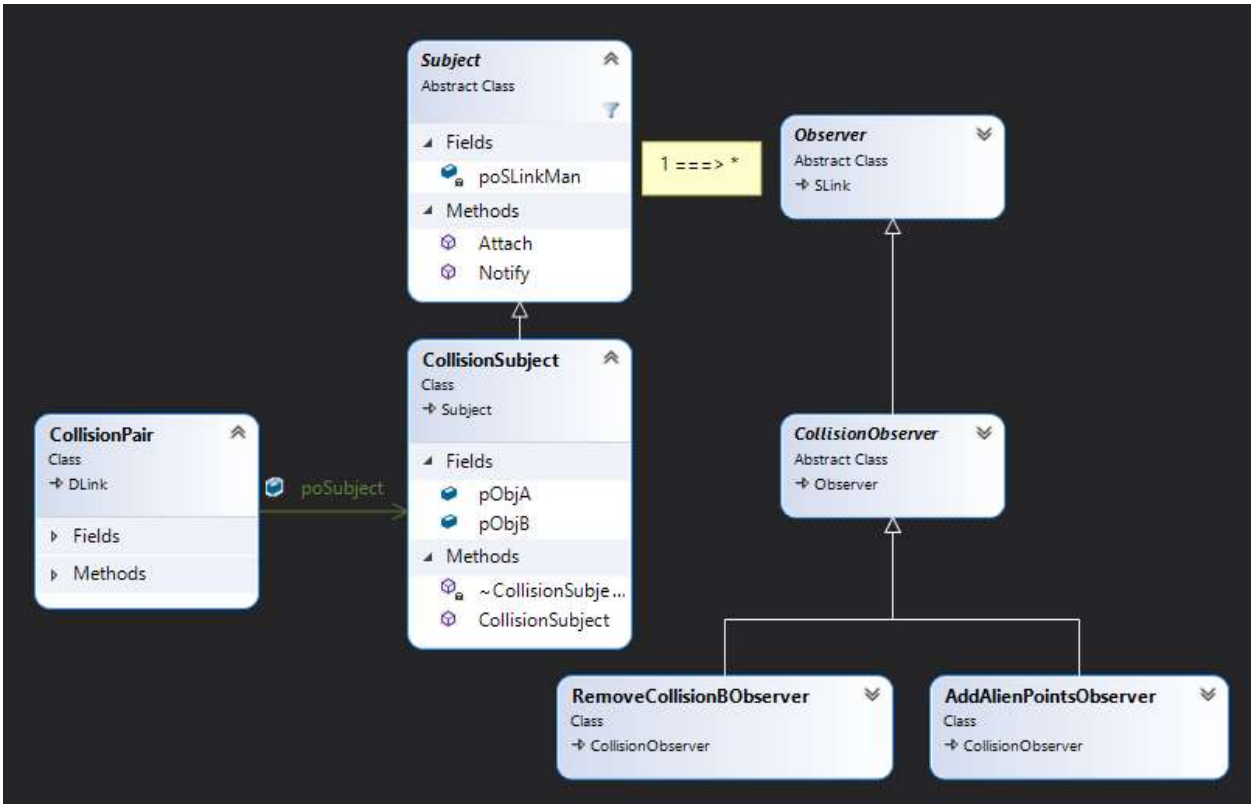
We use this pattern to solve the problem of checking and processing collision events, and it is ideal for this use case for a couple reasons. The first of which is that, in determining collision we need to know the most derived type of the classes involved in the collision, which

the double dispatch nature of the visitor pattern facilitates. This double dispatch also enables seamless unwrapping of our GameObject composite root nodes. The second reason the visitor pattern addresses collision processing cleanly is that, through the use of the pattern and our CollisionPairs, we are able to determine a set ordering for our collision, in which collision object A always accepts Collision Object B. Also, in conjunction with our composite GameObjects and the Visitor pattern we are able to enable an optimized collision checking system, where we are able to "early-out" from the check in majority of cases. The collision pairs are given 2 composite objects, for which their collision rectangle is simply the union of all their children. With this, we are able to see if this general composite GameObject is colliding before moving down the tree and checking for each collidable GameObject. Lastly, this Visitor pattern based collision system works smoothly along with the Observer pattern for triggering all side effects of a collision event.

The common solution to this problem would be to ultimately check, every frame in Update() if a collidable object was intersecting with any other collidable object on the scene, and then for each collision dynamic cast down to see if it is a collision event of interest and process as such. This would require significantly more cycles to process and requires each object to check for collision rather than offloading all collision to a centralized location as it is for the CollisionPair and Visitor implementation in this application. In addition, this common solution requires additional dynamic casting and conditionals based on that type, since it does not take advantage of a use case for double dispatch.

DeAngelo Wilson
SE456
Design Pattern Document

Observer Design Pattern



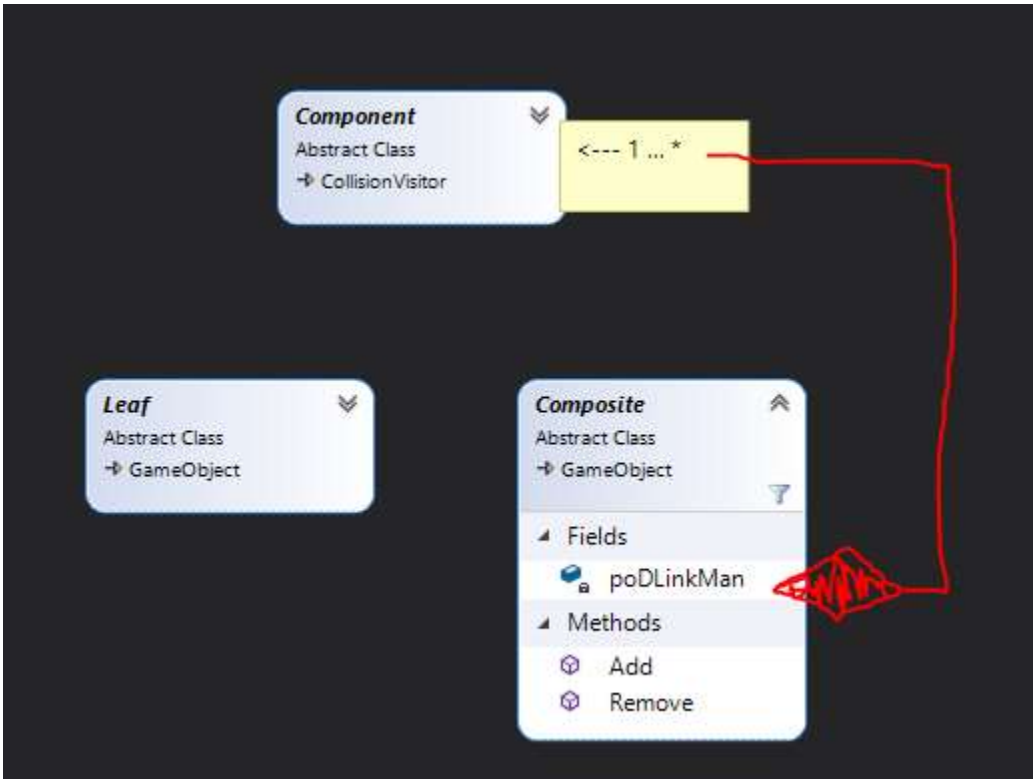
The Observer design pattern is a behavioral pattern that is used to define a one-to-many dependency between a subject and its observer. So, on any observed state change of the subject, each of the observers in its collection will be notified, allowing them to execute any action, unbeknownst to the subject. The observers act very similarly to a command in that they simply have a reference back to their subject context and each derived observer will implements its own particular behavior that needs to occur in response to some event.

The Observer pattern addresses the issue of maintaining encapsulation in spite of dependencies. For instance, despite the fact that the PlaySceneState's life time is determined by the state of the Player's ship, that does not merit either of the two being coupled together. This pattern in used in multiple instances throughout the application, one of which is in the collision system. On any relevant collision event there are certain side-effects that need to occur as a result, these side-effects are wrapped into a observer, which the CollisionPair::CollisionSubject

will trigger a notification to in the event of a collision. We also use the observer for responding to user keyboard input.

An alternative solution to this challenge would be to simply couple the observers and subject together, resulting in a more fragile code base.

Composite Design Pattern



The Composite design pattern is a structural design pattern, as it dictates the composition of objects and describes a way to view a hierarchy. The intent behind this pattern is to compose objects into a tree structure and also allows for treating a collection of objects the same as individual objects.

The pattern is composed of three elements, a base class called *Component*, and 2 derived classes of *Leaf* and *Composite*. The component class provides the shared interface and is used to

allow us to treat leaves and composite objects uniformly. The Composite object is simply a collection of components, while the leaf is an individual component.

The Composite pattern allows for us to establish a grouping, or collection of GameObjects in this context. And we can operate (i.e. Draw() and Update()) these collections without needing to care whether it is a leaf or composite type.

The common solution to this problem would be to essentially just hardcode the grouping, or tree structure, resulting in a much less robust code as it would not be closed to modification and open to extension. If the common solution did make use of a common hierarchy, but still did not fully implement the Composite pattern design, it would alternatively result in additional type checking as without the proper interface setup in the component class, one is unable to treat the different objects uniformly.

Post-Mortem

I have numerous aspirations to implement for this project still, to list a few: I would like to add a demo scene, finish the attract scene to have the squid alien flip the “Y” character, refactor my TimerEventManager to maintain its own local time rather than updating all TimerEvents each time I set it active, refactor my factories to more accurately fit the design pattern as well as integrating the visitor pattern, and lastly, for additional features, I would like to add the coloring overlays depending on which y-value a GameObject is at.

I found the 2-players mode to impose quite a few bugs on my game, but after working through some oversights I was happily surprised with how stable my entire game felt as a whole. I think this is primarily attributable to my design of having a “Player” class and “GameController” to hold each players state and to synchronize the current players and provide access to its members.

I am proud of the way I did my Font/SpriteText system as I feel it is very robust and open to extension, as I was able to easily integrate any needed changes to create TimedCharacterSpriteTexts.

I think if I was to re-implement this project, starting from the when we began with scenes, I would make several changes to the way I maintain my TimerEventManager and create my GameObjects, but it would require me to give much more thought, as I realized I had not thought through everything.

Lastly, I think this project had helped me tremendously in my understanding or awareness of the development lifecycle. With this I mean that the importance of prototyping versus thorough planning out of system, as it is a tough line to walk and found instances in which I saw benefits for prioritizing one over the other.