Application of Software Architecture Patterns for 2D Video Games Design Report

March 14, 2021

Abstract

Today's programming is changing at a rapid pace and therefore it is a necessity that the practices in which programmers use to organize code needs to be analyzed. Knowing how the code behind big systems are organized, referred to as software architecture, is essential in allowing programmers to write clean code that is easily extendable. This paper will present detailed designs of common programming patterns that make up the software architecture behind video games. My hope is that through this paper, a deeper understanding of modern software architecture can be gained and be applied for multiple facets of programming. To demonstrate modern software architecture, I will be using the Unity Game Engine and the Drawio online diagram software to model and develop three core systems for a 2D video game. The design of the software architecture behind my video game is included in this paper.

1 Introduction

With the birth of computers and the technology behind them took place not too long ago. This makes programming as a concept very young in comparison to other more long-standing concepts, like math or science. Likewise, this makes programming still very much in early development and therefore subject to many changes at a rapid pace. It is this rapid change that has brought about the present-day situation programmers are faced with. We are dealing with more high-level problems than ever before. These problems require hundreds if not thousands of lines of code that can easily form monolithic classes that are impossible to work with. This pressing issue is resolved with a solution that has risen in recent years, that being software architecture.

Software architecture is the ways in which code is structured to define different components and their relationships between each other that make up a system. Every program written has some form of structure in it, even if it is just putting everything in main(). When it comes to dealing with high level problems though, shoving everything into main() is not an option. High level problems require a lot of code to be solved. Simply ignoring software architecture while creating a solution to a high-level problem will result in monolithic classes that are hundreds if not thousands of lines long. As these monolithic classes increase in size, making a change in them becomes the equivalent of walking through a minefield. The codebase becomes a fragile entangled mess, where every section of code can affect any other section. Making a change to something like this often causes unforeseen side effects elsewhere. Then the programmer needs to spend countless hours searching and fixing these issues. This forms a blockade for the programmer, unable to make good progress.

However, good software architecture allows us to bypass this issue by compartmentalizing sections of code and defining which sections know about other sections of code. This minimizes the knowledge that one section of code needs to know about another section of code, decoupling them. When decoupled, a section of code can be changed without affecting the other sections that it communicates with. This decoupling allows us to reason sections of code independently from one another, in turn minimizing the knowledge the human needs in order to make a change without breaking anything. It creates a codebase that whenever you make a change, it is as if the entire program was crafted in anticipation of it. Good software architecture is like a puzzle where you add a piece that fits perfectly without disturbing any other piece. This is especially useful when it comes to building systems that require a lot of communication with its different components.

This is why I have chosen to make a prototype for a 2D video game to demonstrate the application of modern software architecture. Video games have many underlying cogs that make up its whole. It requires a lot of different components, each performing their own simple function. These components then need to communicate with each in order to make up the systems that make a video game what it is. This makes the development of a prototype for a 2D video game a prime subject for the use of modern software architecture.

2 Solution Approach

My solution approach first starts with me picking a core system of a 2D video game that I would like to try my hand at developing. Video games often take years to develop and often take a team to work on many aspects of them. Replicating every system behind a video game would be completely out of the scope of this project. As a result, I will focus on aspects of a video game that are crucial. Things such as music, sound effects, and animation can be considered decorations and not vital to the core gameplay of video games. By selecting a few core systems that are responsible for gameplay, I can scale the project down to a manageable size.

The core systems that I have selected to work on for this project are: A *player controller*, a *HUD* (heads-up-display), and *enemies*. The player controller is what gives the user their main method of interacting with the game world. The HUD is how the user receives important information about the game, like their current health. The enemies are the obstacles in the game that the user must overcome. All three come together to make the core of a video game and are what I am working to recreate.

With the project being at a manageable scale, I then do research about how player controllers, HUDs, and enemies to develop them with good software architecture.

3 Analysis Approach

The Big Three

After breaking up my project into three distinct systems, I begin to do research on the way they work at a high level and break them down into several distinct components.

The player can be broken down into input, colliders, movement, and properties. The HUD can be broken down into a UI element keeping track of health. The enemies can be broken down into responding to the player's position, movement, and properties.

1. The Player Controller

Now that I have broken these systems down into these different components, it's time to define their relationships between each other. The player controller first needs to respond to the user's input, which prompts the player to move. Upon this request from the user, the colliders serve to check whether or not the player is in a physical space where it allows for the desired movement. If the desired movement is allowed, the appropriate form of movement is selected from the movement component (walking, jumping, etc). From the movement component, the degree of the desired movement is decided by the player's properties. The player properties contain values for the player's movement to use such as the speed of walking or jump height.

Ex: In the case of the player jumping, a collider checks whether or not the player is touching the ground and therefore whether or not the player can jump. If the player were to receive a jump input from the user while it the player is grounded, then the player enacts a jump movement from the movement component. The extent of the jump movement is decided by the player properties on how much force is behind the jump.

2. The HUD

The HUD keeps tracks of important elements related to the player and work to display that information to the user. If something were to happen to the player that warrants a change to one of the player's elements, then the HUD would update its UI element to display the change to the user.

Ex: In the case of when the player takes damage, the HUD detects that a change to the health value of the player has occurred. The HUD then displays the change to the user by updating its UI element for displaying the player's health.

3. The Enemy

The enemy needs be able to move around and does so using a similar system of having a movement component in which a specific movement is selected followed by the degree in which the movement is done from the properties component. The enemy must also be able to detect the player using colliders. When the enemy does detect the player, the enemy must enact an attack against the player. This involves a combination of movement and colliders that the enemy uses towards the player.

Ex: The enemy is wandering around, using the movement component reading from the properties to decide how fast it moves. A collider detects the player once the player gets close to the enemy. The enemy then uses its combination of movement and colliders to move and attack the player as its behavior sees fit.

Now that all the relationships have been identified, I investigated common software architecture patterns to see which ones would be the best fit for the systems that I have modeled for my video

game. This involved researching online for good sources of design patterns. After much investigation I decided to rely on the software engineering book Design Patters: Elements of Reusable Object-Oriented Software (1994) by Erich Gama, Richard Helm, Ralph Johnson, John Vlissides, and Grady Booch. This book is largely where I learned the software design patterns that I decided to use in my project.

4 The Patterns and their Application

The Component Pattern

The component pattern is best used for a single entity that spans multiple domains. Each domain is encapsulated in its own class, making up the components we will use. Meanwhile, the entity itself becomes a container that simply brings all the components together. This allows us to avoid the scenario of a single monolithic class that becomes a disorganized mess of combined domains. Instead, we will have several smaller classes who only have their own distinct roles. Then we have a main class that contains a reference of every defined smaller class needed for the system. With this pattern, we can start to decouple a lot of the design we have previously defined. While this pattern is relatively simple, this allows us to prepare our code to be well organized and have a more intuitive process whenever a change needs to be made.

The Component Pattern Application

For our player we have a main class simply named “Player”. This will act as our container for all the components we will be defining. Next, our “Player” class needs all its components. We create an input, collider, movement, and property classes that all have their own distinct functions. These components will each have a method to be played in the main Player script that does their own type of functions. Then we create a reference for each of our now defined components in the main Player class. Now we have a way to access all the relative information needed to control the Player without having all the logic entangled with one another. This implementation is demonstrated in figure 1 below.

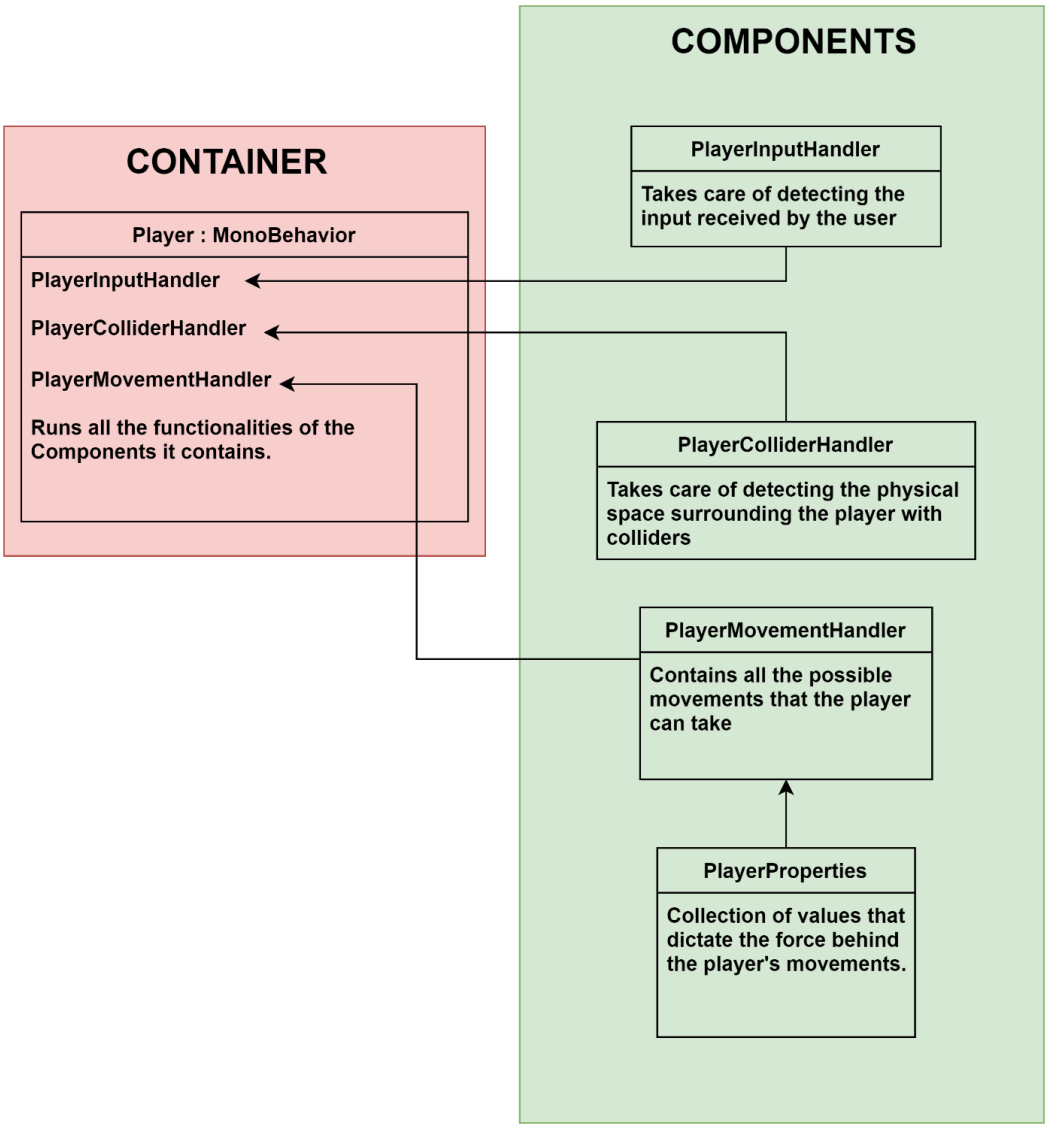

Figure 1: Design of Component Pattern applied to Player Controller

For our Enemy we also have a main class simply name “Enemy”. Just like in our main class for our Player, the “Enemy” class will be our container for all of the Enemy’s components. These include enemy detection, movement, and properties. Continuing the pattern, the main class will contain a reference to each of these classes for the same reasons mentioned for the Player. This implementation is demonstrated in figure 2 below.

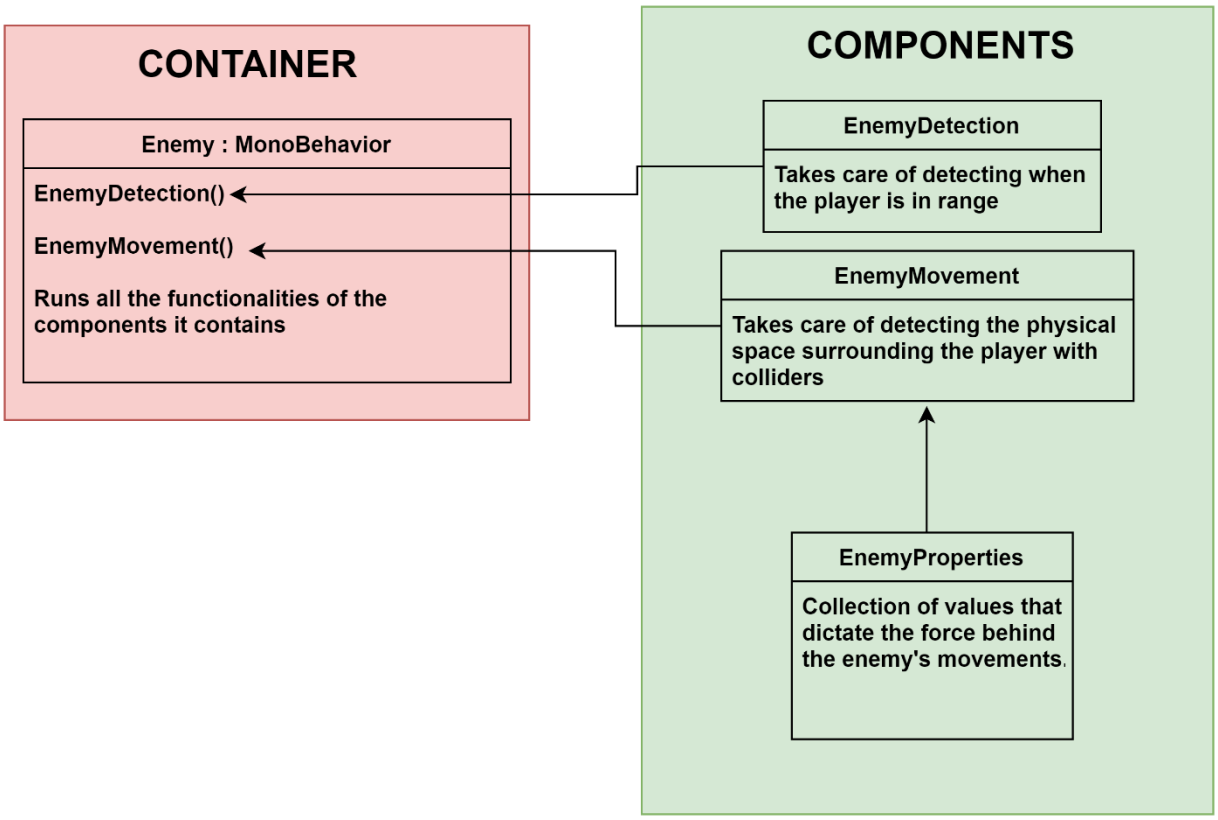

Figure 2: Design of Component Pattern applied to Enemy

The State Pattern

Next is the state pattern. It is best used for an entity whose behavior can change in response to a series of inputs overtime or in response to another entity. These behaviors must be easy to categorize into their own distinct classes. Normally, we use if and else blocks all in the main class in order to give an entity its behaviors. However, this becomes a nightmare to manage the more behaviors you define. Since all the behaviors and data would be located in one class, all behaviors can interact with other behaviors. This often causes conflicts between behaviors, producing unexpected results that are hard to fix. The state pattern allows us to bypass this issue by encapsulating each behavior along with any necessary data into its own distinct class. We call the encapsulated behavior a state. This allows us to easily tell the entity what to do simply by switching the state that it is currently in. This also means it so that the entity can only be in one state at any given time, further making it easier to work with as you do not have to account for any unforeseen code affecting the current behavior you are trying to define (i.e. a car cannot be accelerating and stepping brake pedal at the same time).

The State Pattern Application

Both our player controller and enemies will be using this pattern as both are good candidates for this design pattern. In the case of our player, its behavior changes in accordance to the user's input. When the user inputs a walk input, our player walks. When the user inputs a jump, our player jumps

and so on. We can easily define and divide the player's distinct behaviors just by running through what actions the player can perform in response to the user's input. We then make those behaviors into their own state classes and swap to whichever state corresponds to the user's input. Meanwhile, our enemies need to be able to respond to the player's location. When the player gets near the enemy, it should switch from wandering to attacking the player. We use the state pattern to encapsulate these distinct behaviors into their own classes and switch between these states in accordance to the player's position. This encapsulation is demonstrated in figure 3 below.

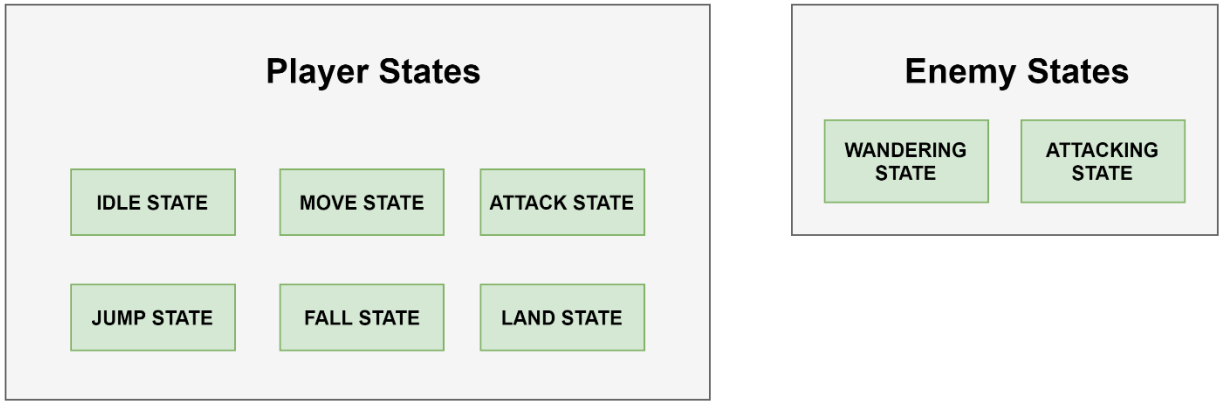

Figure 3: The encapsulation of behaviors of the Player & Enemy into states

After all the encapsulation of behaviors into states, there is still several major responsibilities that need to be addressed for this system to function. How to perform the state itself, and how to transition between states. These responsibilities are handled by a StateMachine that is represented as a simple class. My StateMachine keeps track of the current state that the entity is in. This is so that it knows what behavior to perform and what conditions it should be looking for in accordance to the current state. When the current proper conditions are met, the StateMachine handles transitioning between states by overriding the current state. Once the new state has been taken in as the current state, the player/enemy now executes the new defined behavior. This produces the result of transitioning between the player's/enemy's state. The StateMachine's functions are demonstrated in figure 4 below.

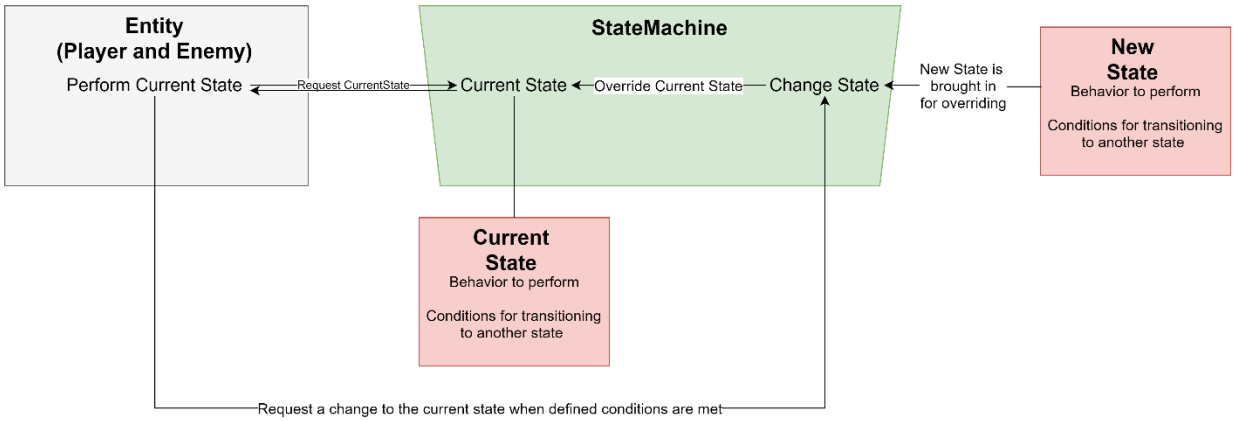

Figure 4: The StateMachine's function in relation with the player/enemy and states

This only leaves the question, how do the states contain the behaviors, and conditions? This is answered when we look at how the states themselves are implemented in detail. For my project, I have all states inherit from a parent class simply named States. Here I define all the basic functionalities all the states in my project will have. Each state has an Enter, Exit, UpdateLogic, and UpdatePhysics methods. The Enter method is used for any initialization needed for the entity starting to perform state's defined behavior. The Exit method is used for any cleanup upon the entity finishing the execution of a state's defined behavior. These two methods work in conjunction whenever the StateMachine decides to transition to another state in order to make the transition a smoother process. The UpdatePhysics method is where our states will be performing their defined actions when being executed. Finally, the UpdateLogic method is where a state will be checking their defined conditions for switching to another state (i.e. user input). The implementation of my states can be seen in figure 5 below.

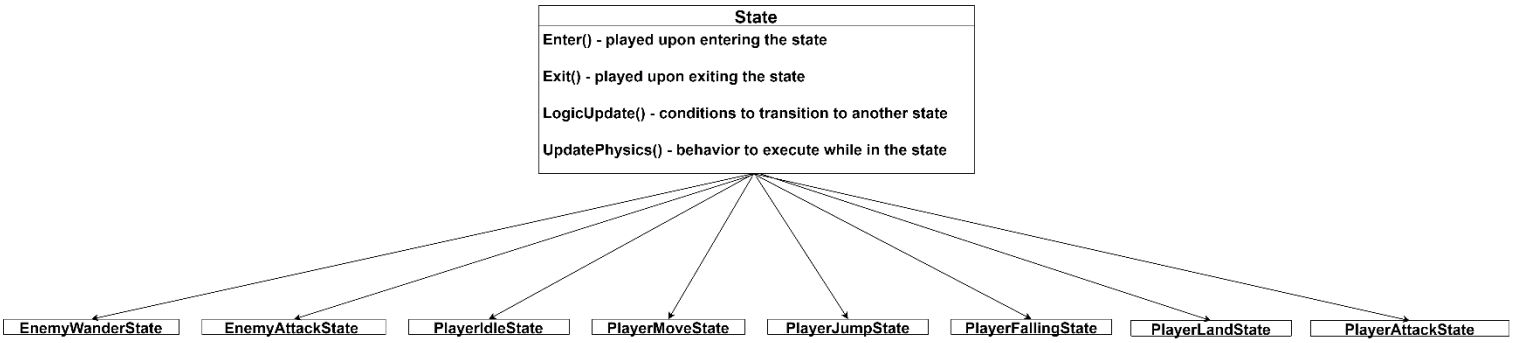

Figure 5: Detailing the implementation of all states

Now with the encapsulation of states, state machine, and the structure of the states defined can we design the State Patterns for both the Player and Enemy that detail the relationships between their states.

The following State Pattern demonstrates the relationships between the states defined for the Player. See Figure 6 below.

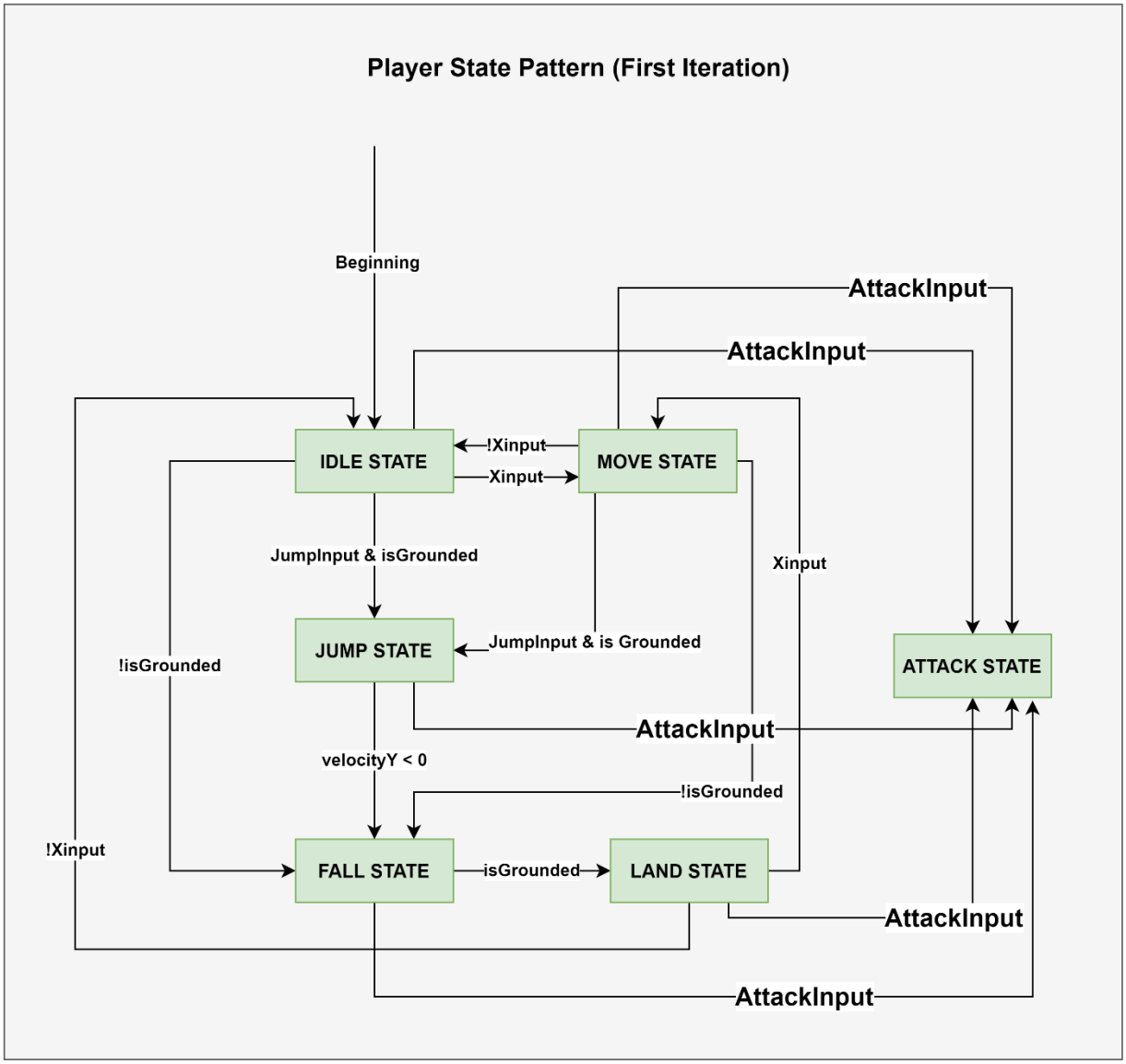

Figure 6: Player's defined states and their relationships between each other

The following Enemy State Pattern demonstrates the relationships between the states defined for the Player. See Figure 7 below.

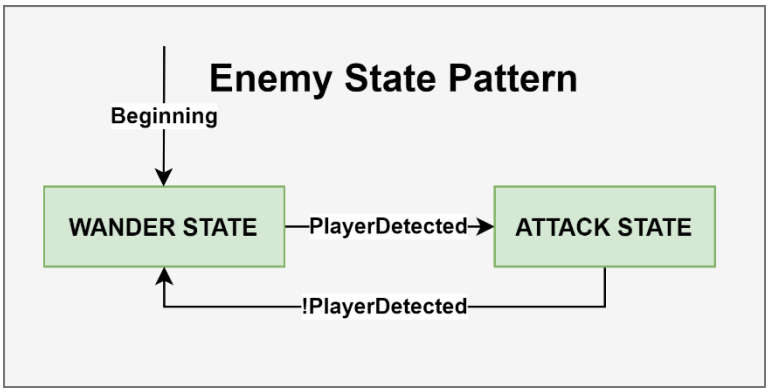

Figure 7: Enemy's defined states and their relationship between each other

However, as we can see, the Player State Pattern is extremely cluttered. There is a lot of similarities between certain states that have common transition conditions. As a result, there is a lot of unwanted repetition. Luckily, we can alleviate a lot of this repetition by taking our Player State Pattern a step further. Instead of using the basic State Pattern, we can introduce a Hierarchical State Pattern. In this more advanced State Pattern, a state can have a superstate (making itself a substate), whose sole purpose is to define common transition conditions between a group of substates. This can be implemented by using inheritance. Now a substate will have its own unique transition condition and should this condition not be met, then the substate will roll up the chain of superstate to the transition condition it has in common with other substates. The redesign of the Player's states can be seen in Figure 8 below.

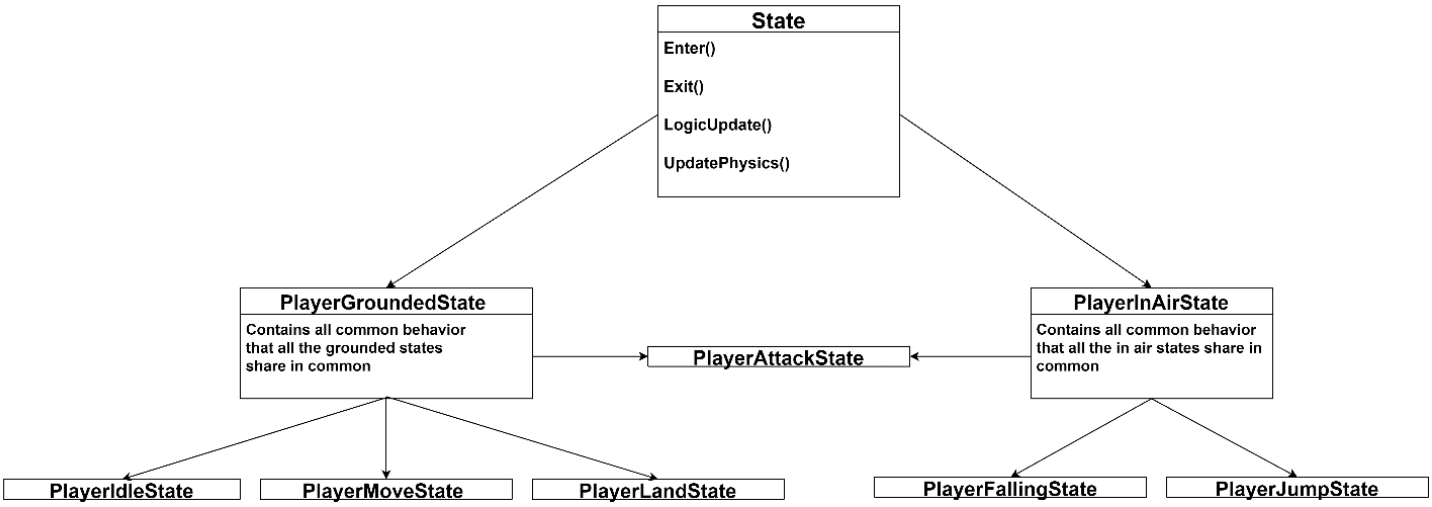

Figure 8: Redefining the Player's states under the context of a Hierarchal State Pattern

Now a re-design can be made for our Player State Pattern that takes advantage of the new superstates by categorizing our substates under them appropriately. This way, the superstates can take care of any common behaviors that their housed substates. This saves the repetition of having to specify the same transition for several states. Below is the final iteration of my Player State Pattern now implementing a Hierarchical State Pattern, see Figure 9 below.

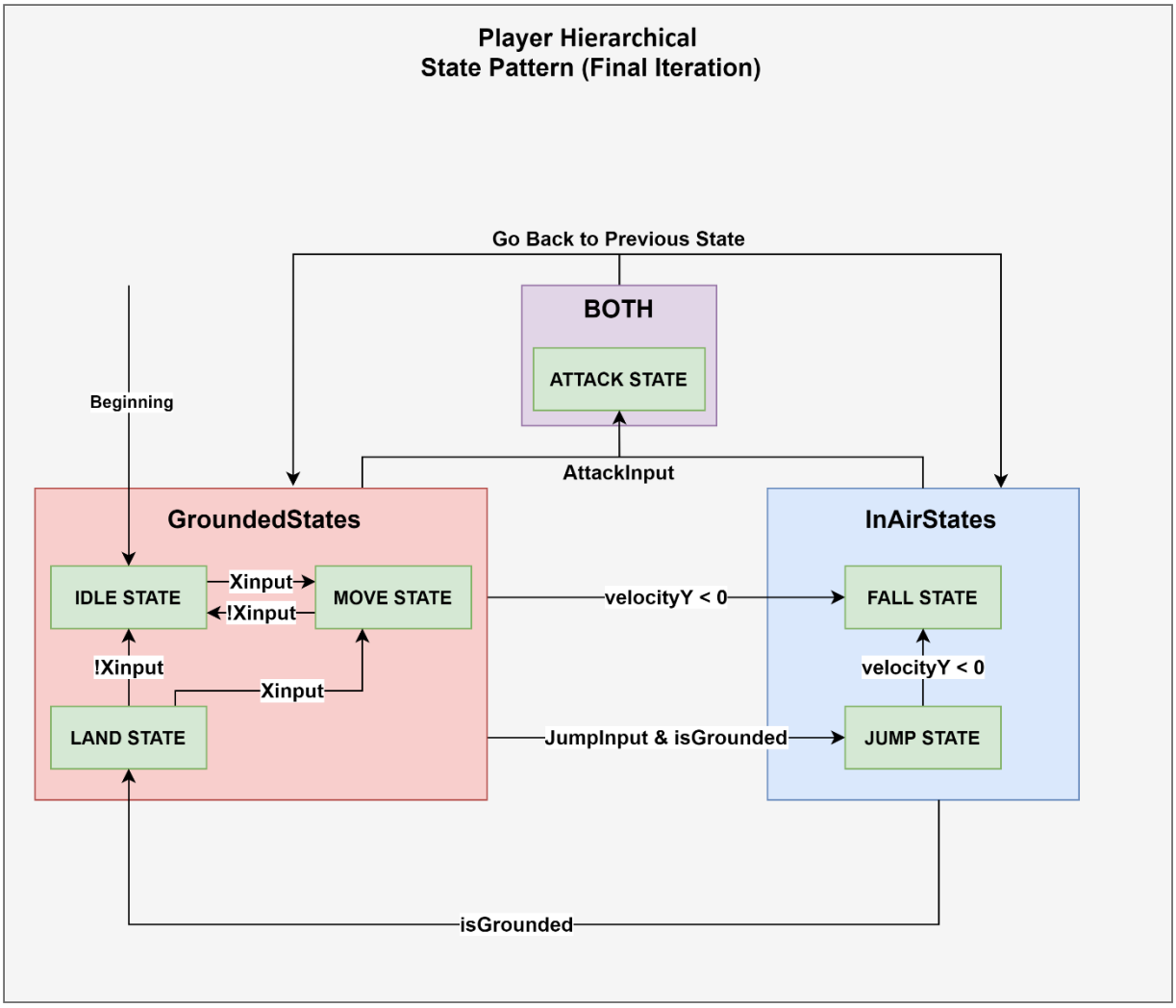

Figure 9: Redesign of the Player State Pattern to a Hierarchal State Pattern with superstates

The Observer Pattern

The observer pattern is best used when an object needs to receive an update on another object's change and respond to said change accordingly. While this is technically achievable normally without this pattern, the brute force method would tightly couple a lot of classes together in ways that do not make sense. Tightly coupling classes means that one class cannot function without the other and vice versa. This can cause a lot of underlying dependencies that do not make sense. The observer pattern helps to avoid this issue by allowing us to reduce the amount of knowledge a class needs to know about another class whenever the first class needs to carry out a task in response to a change in the other class.

In the observer pattern, we define the nosy class that is interested in the change within another class as the observer. The class that has the change itself is defined as the subject. Whenever the change happens within the subject, it fires off a notification that the change has occurred. The subject itself does not know who is listening to the notification, it just fires it without caring. The observer, who unbeknown to the subject, was listening for a notification and then carries out its defined task in response to it.

The Observer Pattern Application:

For my project, my HUD needs to know about the Player's current health to be able to display it to the user. This relationship is where I will be using the observer pattern. If it were brute forced, the HUD would know about the Player, but the Player would develop a dependency on the HUD in order to function. This would not make sense and cause issues, so I use the observer pattern to avoid this.

My HUD will function as the observer, who is looking for a notification that the Player's health has changed. Meanwhile, my Player will be the subject who fires off a notification whenever its health has changed for unbeknown observers. My HUD will respond to the notification and enact the change in health. This way, my HUD correctly displays the Player's current health with no tight dependencies.

See Figure 10 below for the design of the observer pattern between the HUD and Player.

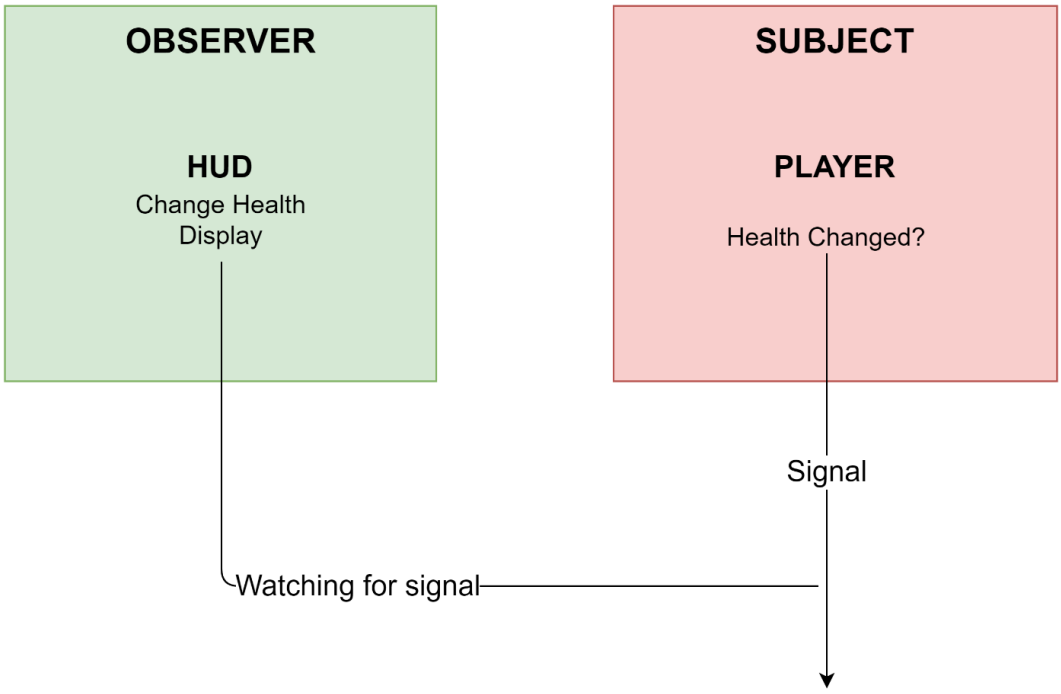

Figure 10: The Observer Pattern applied to the HUD and Player

Choices of Technology

For this project, I used Draw.io for modeling my designs and the Unity Engine to implement my designs.

Drawio is a free online diagram software that made the process of creating and iterating my designs easy. The tools it provided such as labelling shapes, color coding, drawing arrows, and labelling arrows made the process of creating my designs intuitive. The reason why I chose this over other diagram software is that some of the features it has are exclusive to programming. Features such a creating classes and the methods within them are what made this software a much more desirable choice over alternatives. Drawio served to help me better visualize my implementations while I was working on them.

The Unity Engine is a popular game engine used for the development of video games. I chose the Unity Engine because I needed to quickly test my designs as well as iterate on them without worrying about lower level functionalities. Basic things such as a sprite are renderer, methods for collision detection, and a debug console are all taken care of by Unity itself. If I were to use anything less than a game engine, my focus would have been torn away from the creation of my 3 core systems. Unity also has the advantage of being the most popular game engine for indie game developers. This naturally leads to a large community that are experienced with working in Unity who can provide a lot of online assistance should something go wrong with Unity. This huge online resource saved me time whenever I encountered a problem specific to Unity. Unity allowed me to focus on the higher-level functionalities that make up this project and quickly iterate them.

Conclusion

The underlying focus of this project is the importance of good software architecture when dealing with complex problems. There are many young programmers in recent times that have the unhealthy tendency of wanting to immediately code up a solution whenever they receive a problem. In that same light, there are also those who choose to focus on memorizing small specific things like syntax. However, before focusing on that, the bigger picture must be taken into deep consideration. Otherwise the base you build your code upon will quickly crumble under the weight of everything you build on top of it.

Building systems for video games is exactly one of these complex problems where you must make sure to plan ahead, make designs, and do research on software architecture patterns to make sure the foundation of your code is strong. Video games are inherently made up of many parts that must communicate with each other. These two factors are what make video games a prime candidate for demonstrating the use of software architecture patterns as you can easily make code into an unmanageable entangled mess. Software architecture patterns deals with the organization of code by provides many techniques in which you can keep sections code encapsulated while at the same time having them communicate with one another, resulting in clean code. By choosing to make the core systems of a 2D video game (player controller, enemy, and HUD) along with selected software architecture patterns (component pattern, state pattern, and observer pattern) I was able to create a detailed walkthrough of this essential design process , demonstrating to young programmers the importance of software architecture and the process that they must go through to improve the quality of their solutions for complex problems.