Creating a Character State Component

By: Justin Mackenzie

The purpose of this tutorial is to hopefully provide you with some knowledge in software development and software engineering principles. In this tutorial, we will walk through the whole process of developing a piece of software. We will go from a plain set of features to producing a piece of software that implements those features. We will develop a simple component that represents the state of a character in a video game. Hopefully after this tutorial, you will be able to implement your own state component for any object you may create in your game, database application or any type of software.

Features

Every useful piece of software fulfills some purpose; our component will be no different. Typically, you either come up with or someone else gives you a set of features you want to implement. Here are the features that I came up with for our Character State Component:

- Character can move in all directions. (front, back and both sides)
- Character has the ability to sprint.
- Character has the ability to crouch.
- Character has the ability to jump.
- Character cannot sprint while crouching and vice versa.
- Jumping will cancel sprinting or crouching.
- Cannot sprint, crouch or jump during a jump.
- The character will have a limited amount of energy to be used on sprinting and other activities.
- Energy will be regenerated fastest when idle, at a slower rate when moving and not at all while sprinting or jumping.
- Sprinting will increase movement speed.
- Crouching will decrease movement speed.

Let's leave it at that for now; we don't want to bite off more than we can chew. Now that we have our set of features, it is time to create requirements from these features.

Requirements

I personally like to represent the functional requirements of a system in the form of use cases. Looking at the above features, I see four functions that this component provides:

- Move
- Sprint
- Crouch
- Jump

The other features are not functional requirements, but more so specifications for the functional requirements listed above.

Use Case Model

Firstly, let's think of the actors in our system. This character state component is being designed to be used for any character in a game; this could be a player-controlled character or even a computer-controlled character. With the above information, the actor could be any number of roles, from a player to an artificial intelligence (AI) script. We will call our actor "CharacterController" and will represent any entity that is controlling our character. For each functional requirement listed above, we will make a use case that describes the specifications of that functional requirement. For use cases, I like to use the following set of specifications to describe a use case: use case name, participating actors, flow of events, entry conditions, exit conditions and quality requirements. Here are the use case specifications that I came up with:

Table 1: Move Use Case

Use Case Name	Move
Participating Actors	Initiated by CharacterController
Flow of Events	1. The CharacterController activates the "Move" function during a game session. 2. The system sets the character state to "Moving" and changes the attributes accordingly.
Entry Condition	• CharacterController currently in a game session.
Exit Conditions	• The use case successfully completes a flow of events.
Quality Requirements	• System must respond within 0.02 seconds.

Table 2: Sprint Use Case

Use Case Name	Sprint
Participating Actors	Initiated by CharacterController
Flow of Events	1. The CharacterController activates the "Sprint" function during a game session. 2. The system checks to see if the "Sprint" function is available at that time. It does this by checking if the character is not jumping and has sufficient energy. a. If it is available, change the character's state to "Sprinting" and change attributes accordingly. b. If it is not available, do nothing and exit use case.
Entry Condition	• CharacterController currently in a game session.
Exit Conditions	• The use case successfully completes a flow of events.
Quality Requirements	• System must respond within 0.02 seconds.

Table 3: Jump Use Case

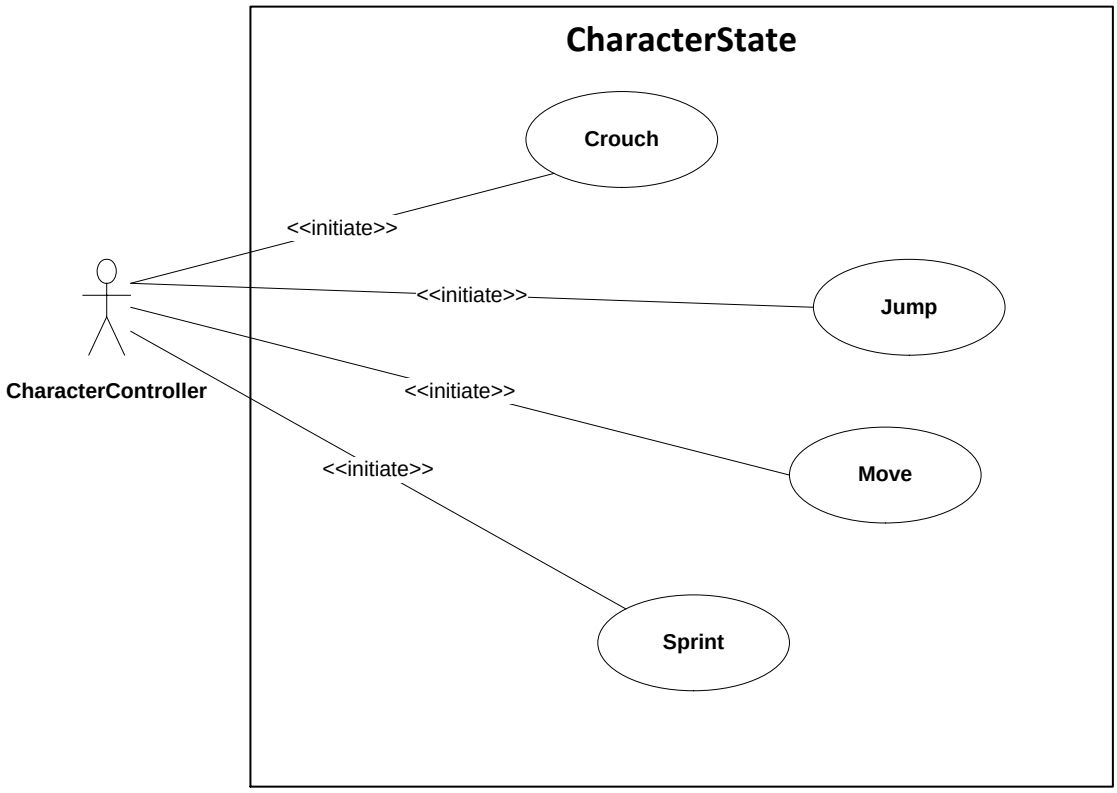
Use Case Name	Jump
Participating Actors	Initiated by CharacterController
Flow of Events	1. The CharacterController activates the "Jump" function during a game session. 2. The system checks to see if the "Jump" function is available at that time. It does this by checking if the character is already not jumping. a. If it is available, change the character's state to "Jumping" and change attributes accordingly. b. If it is not available, do nothing and exit use case.
Entry Condition	• CharacterController currently in a game session.
Exit Conditions	• The use case successfully completes a flow of events.
Quality Requirements	• System must respond within 0.02 seconds.

Table 4: Crouch Use Case

Use Case Name	Crouch
Participating Actors	Initiated by CharacterController
Flow of Events	1. The CharacterController activates the "Crouch" function during a game session. 2. The system checks to see if the "Crouch" function is available at that time. It does this by checking if the character is not jumping. a. If it is available, the system checks to see if the character is already crouching. i. If the character's state is already "Crouching", change state to "Standing" and change attributes accordingly. ii. If the character's state is "Standing", change state to "Crouching" and change attributes accordingly. b. If it is not available, do nothing and exit use case.
Entry Condition	• CharacterController currently in a game session.
Exit Conditions	• The use case successfully completes a flow of events.
Quality Requirements	• System must respond within 0.02 seconds.

Here is a use case diagram to help illustrate the use cases specified above.

Figure 1: Character State Use Case Diagram



Now that we have the basic functionality of the system listed out in the form of use cases, it's time to analyze these use cases and create a design to show how we will meet the features that we listed out earlier.

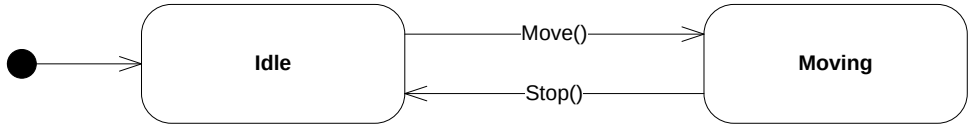
Analysis and Design

Once you get to this step, you can go in a number of different directions. You could start making sequence diagrams for the above use cases, you could make class diagrams, but I will start off by creating a state chart that represents the character movement state.

State Chart for Character Movement

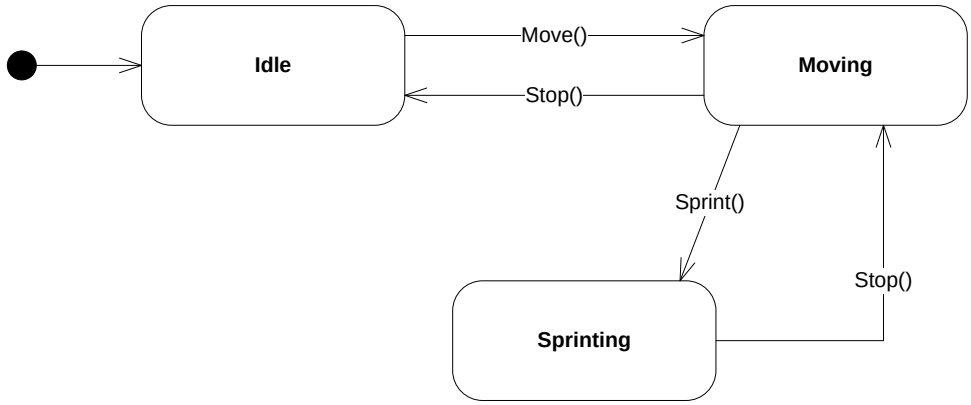
I will look at each use case listed above and create a state chart from each one. First let's look at the "Move" use case. In the flow of events, it states that when the actor activates the move function, the state will change to moving. We can make the following state chart from that description and a little though from us.

Figure 2: Movement State Chart



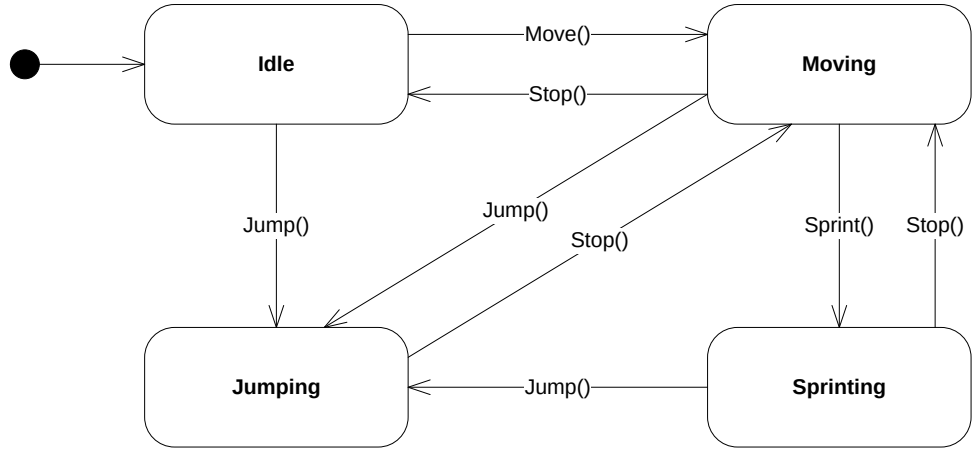
We start in the "Idle" state. When a "Move()" command is sent, the state is changed to "Moving." We can get back to the "Idle" state by sending a "Stop()" command. Next we look at the "Sprint" use case and can see that we can simply add the "Sprinting" state to the state chart in Figure 2.

Figure 3: Movement State Chart with Sprinting



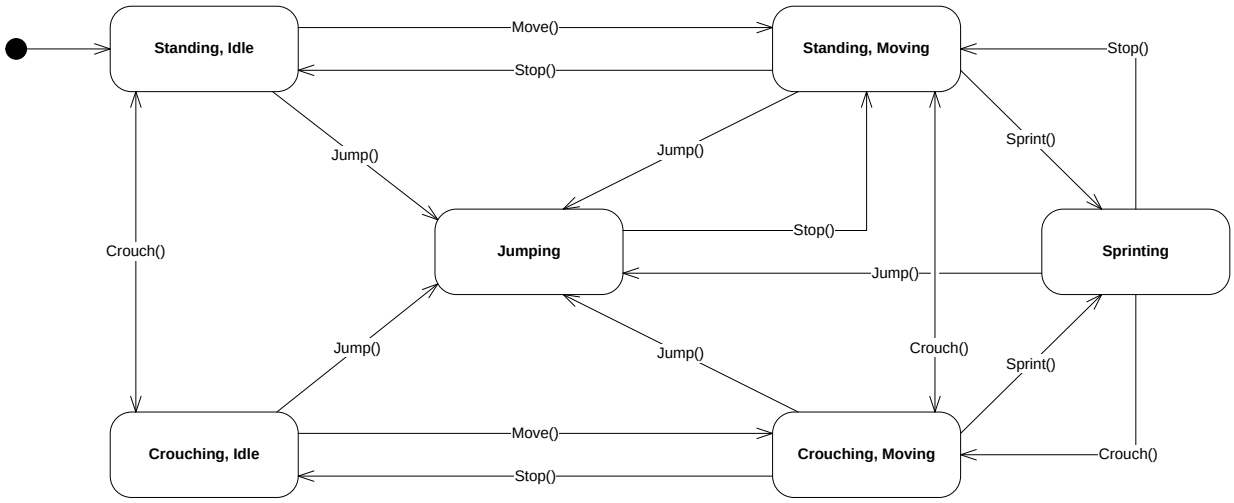
In this state chart I assume that you cannot go from idle to sprinting and vice versa. This state chart models the movement of the character nicely, but we are not done yet. Next, we look at the "Jump" use case and see that we need a "Jumping" state. Here is the state chart with jumping incorporated.

Figure 4: Movement State Chart with Jumping



As you can see, you can jump from any state and after the jump is complete, a Stop() message is sent and the character returns to moving. Next we look at the "Crouching" use case. This is where things will get a bit tricky because you can be crouching and moving along with crouching and idle. Since crouching decreases movement speed and may need its own animation, we will need to split "Moving" into "Standing, Moving" and "Crouching, Moving" and also split "Idle" into "Standing, Idle" and "Crouching, Idle". We can derive this state chart from the idea just listed.

Figure 5: Movement State Chart

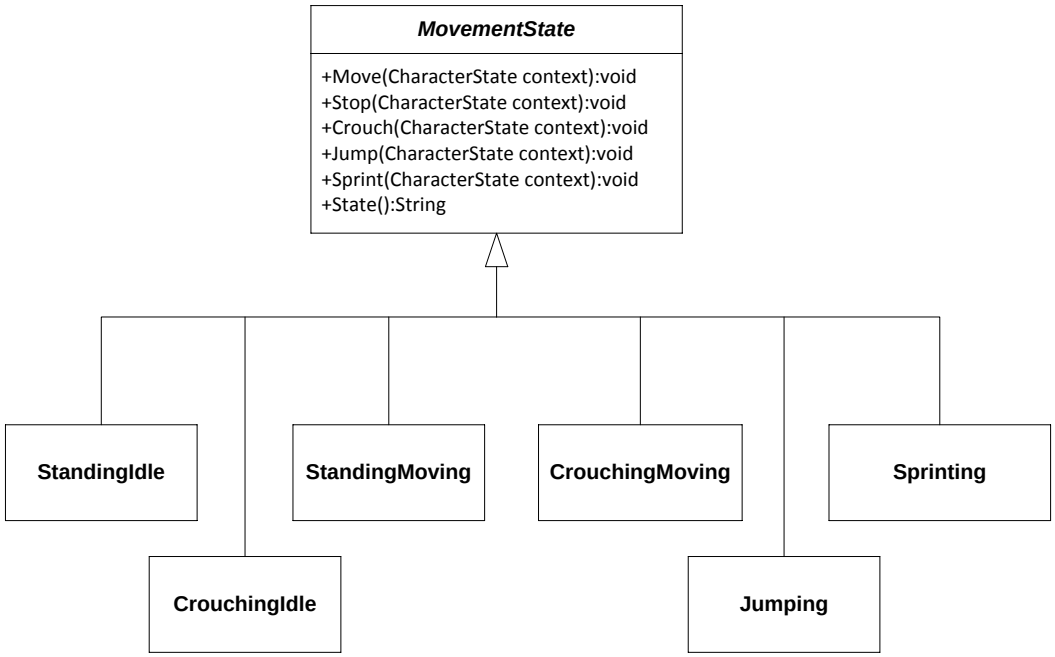


This is our final state chart. Keep in mind that any inputs that are shown for a given state, for example "Sprint()" in the "Standing, Idle" state; will be ignored. Creating a state chart is nice and all, but how do we take this state chart and turn it into something that help us accomplish what we want to do? Luckily, there is a design pattern that is made for this situation and its name is, well what do you know, the State Pattern.

Class Diagram for Character Movement

The basic idea behind the State pattern is to have each state in a state chart represented by its own class. All these classes will implement an interface that will contain methods which will act as inputs to the state chart. When we apply the State pattern to our state chart, this is the resulting class diagram:

Figure 6: Movement State Class Diagram



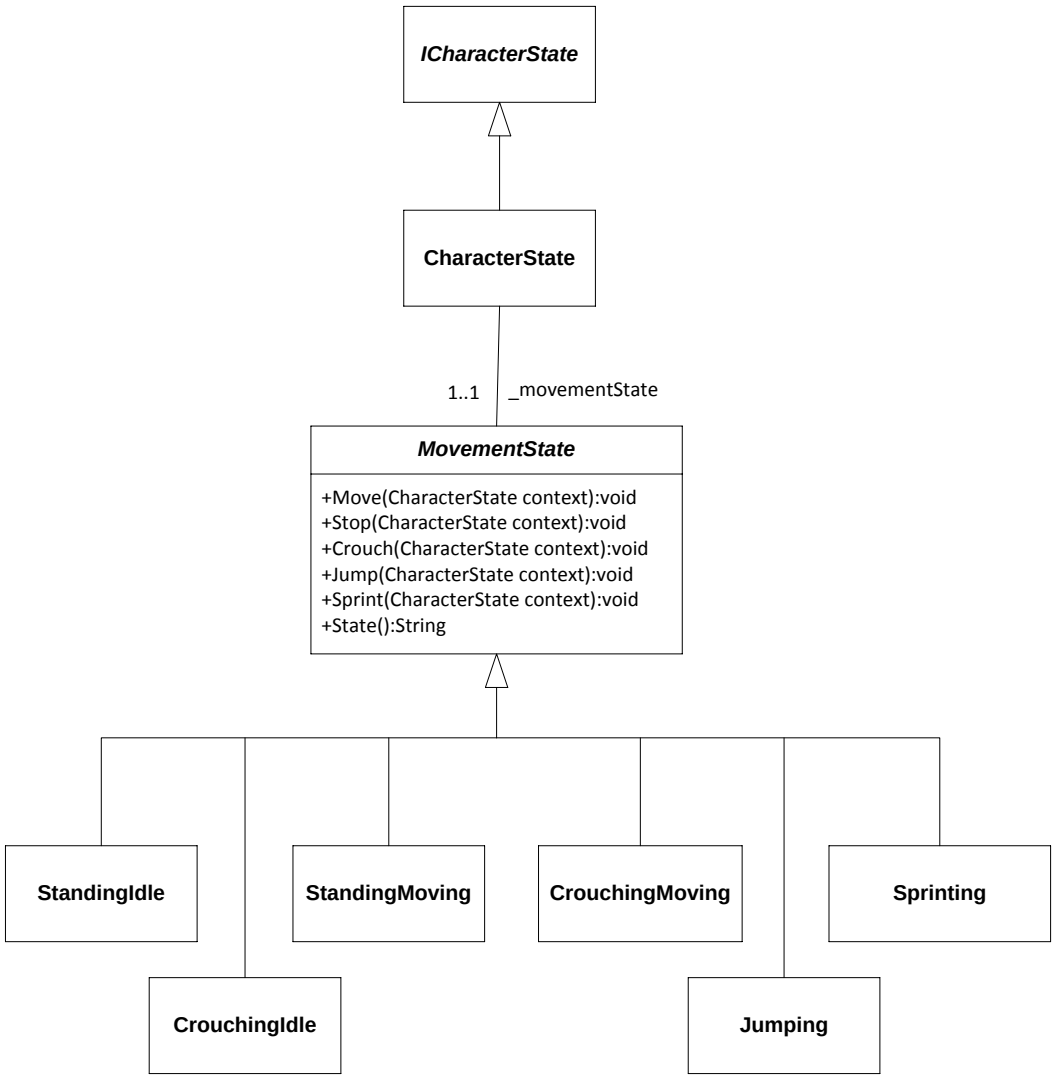
As you can see, each one of our states is represented by its own class and they all implement an interface called *MovementState*. The set of methods of the *MovementState* interface are the set inputs of our state chart, with the addition of a "State():String" method that will allow us to know what state we are currently in. The beauty of the State pattern is that no variables are needed. The entire solution shown above has no variables and is just composed of methods. The methods themselves are not hard to implement either. When you want to transition from one state to another, just create the new desired state. Here is the "Sprint()" method in the "StandingMoving" class:

```
/* A sprint is performed. */
public void Sprint(CharacterState context)
{
	context.SetMovementState(new Sprinting());
}
```

Class Diagram for Character State

Now that we have our movement state portion of our component complete, we need to move onto the next part. We should protect our movement state from other components and provide a simple set of services to our character state component that other components will use. We also want to reduce coupling between our component and other components that may be used with it. To achieve these goals that were just listed, we should use the façade pattern. The façade pattern provides a single interface for other components to access our component and will shield the inner parts of our component from other components. Applying the façade pattern to our existent design gives us the following class diagram:

Figure 7: Character State Class Diagram

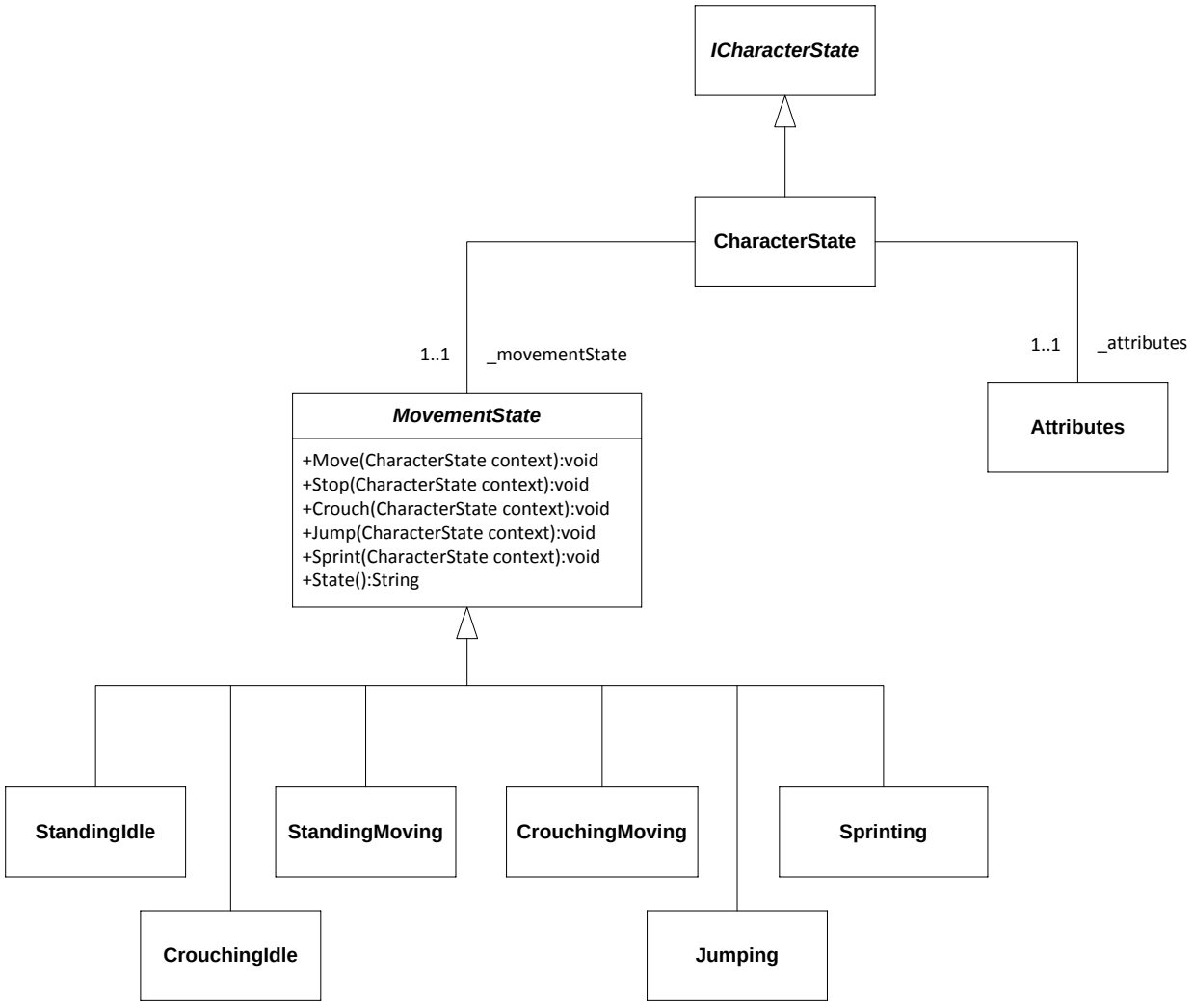


What this design achieves is reduced coupling with other components. If in the future I find that my implementation of the Character State component can be improved, I can simply create a new Character State class and plug it into the interface being used by other components. Also, making

changes in my movement state implementation will not affect other components, because they are only using the *ICharacterState* interface to communicate with this component.

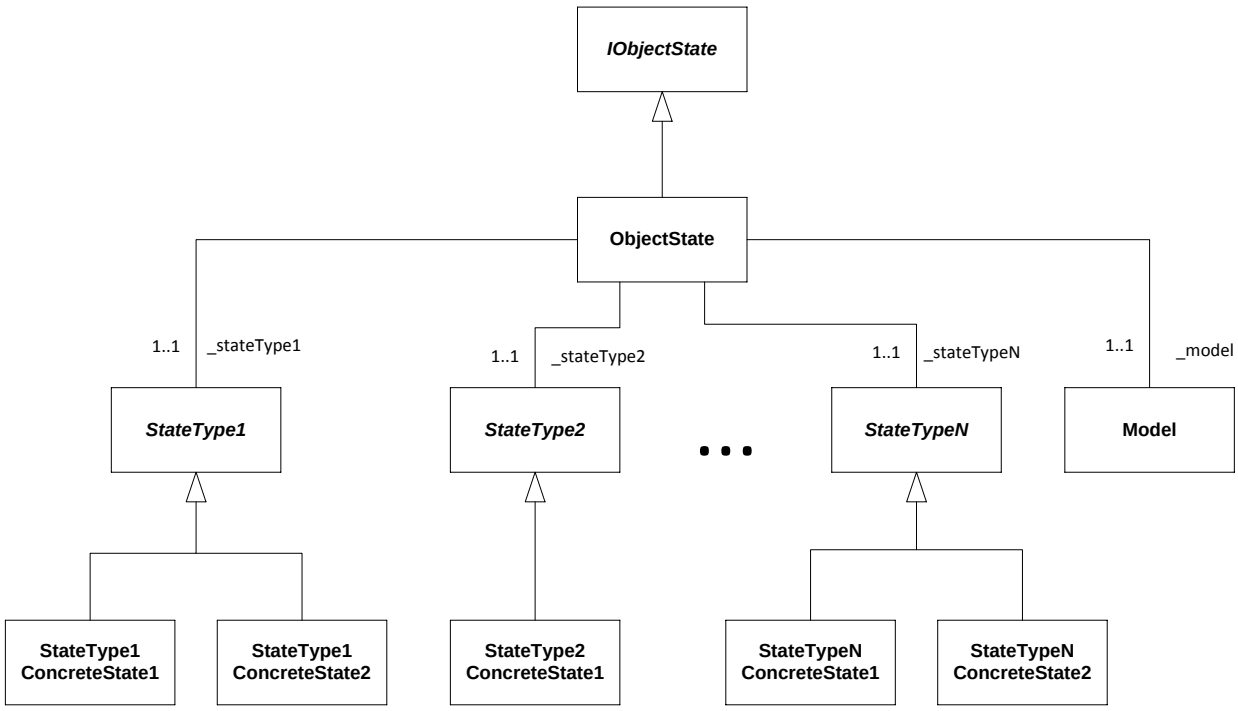
Our design is coming along quite nicely, but we are not quite done yet. Reading the use cases above, we see that we need an "Attributes" class to contain all of the attributes of our character. This is a good idea from a design perspective because it will allow us to keep all of the attributes of a character in one class that can be used by a number of different components. To achieve this, simply add an attributes class to our CharacterState façade class. The resulting class diagram is shown below.

Figure 8: Character State Class Diagram with Attributes



Looking at our use cases, this is all we need for the structure of our component. The beauty of this design is that you can support many types of states with your character state component. You can simply repeat the steps we used to create the movement state structure to create a new type of state. You can use this pattern to model the state of any object. The pattern shown below is the abstract version of what I am talking about.

Figure 9: General Object State Component Pattern



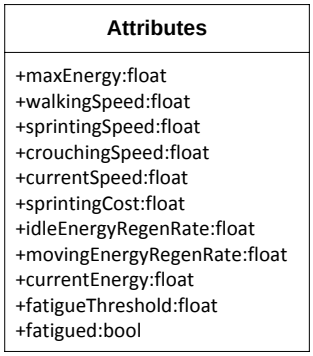
The model represents the data structure of the object. This pattern can actually support many models if you wish.

Now that we have the structure of our component complete, it is time to add the necessary methods and fields to our classes and interfaces. The *MovementState* interface and its classes that implement it are already done. For the attributes class, let's list out the fields we need by looking at the features we listed above and our use cases. Here are the fields for the attribute class:

- Max Energy (Energy value when character has 100% energy. Type: Float)
- Walking Speed (Movement speed when walking. Type: Float)
- Sprinting Speed (Movement speed when sprinting. Type: Float)
- Crouching Speed (Movement speed when crouching. Type: Float)
- Current Speed (Current moving speed. Type: Float)
- Sprint Cost (How much energy is subtracted for one second of sprinting. Type: Float)
- Idle Energy Regeneration Rate (How much energy regenerates per second when idle. Type: Float)
- Moving Energy Regeneration Rate (How much energy regenerates per second when moving. Type: Float)
- Current Energy (Current energy value for the character. Type: Float)
- Fatigue Threshold (After reaching 0 energy and while regenerating energy, the point at which character is no longer fatigued. Type: Float)
- Fatigued (Determines if the character is fatigued or not. Type: Boolean)

Adding these fields to the Attributes class will result in the following class diagram for the Attributes class:

Figure 10: Attributes Class Diagram



Next we need to add the methods and fields to our *ICharacterState* interface and our CharacterState façade class. We need to add the following methods to our interface:

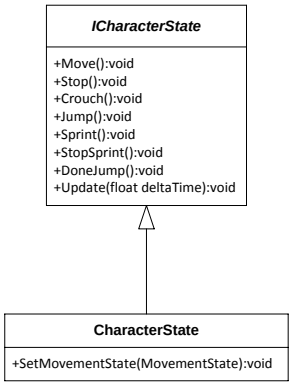
- Move (Alerts component that the character has moved.)
- Stop (Alerts component that the character has stopped.)
- Sprint (Alerts component that the character has begun to sprint.)
- Crouch (Alerts component that the character has crouched or un-crouched.)
- Jump (Alerts component that the character has jumped.)
- JumpDone (Alerts component that the character has landed and is longer jumping.)
- StopSprint (Alerts component that sprint is done.)
- Update (Component updates all of its states; this method is to be called every frame.)

The following methods are needed for the façade class (It also implements all the above methods in the interface):

- SetMovementState (Sets the movement state to the given state.)

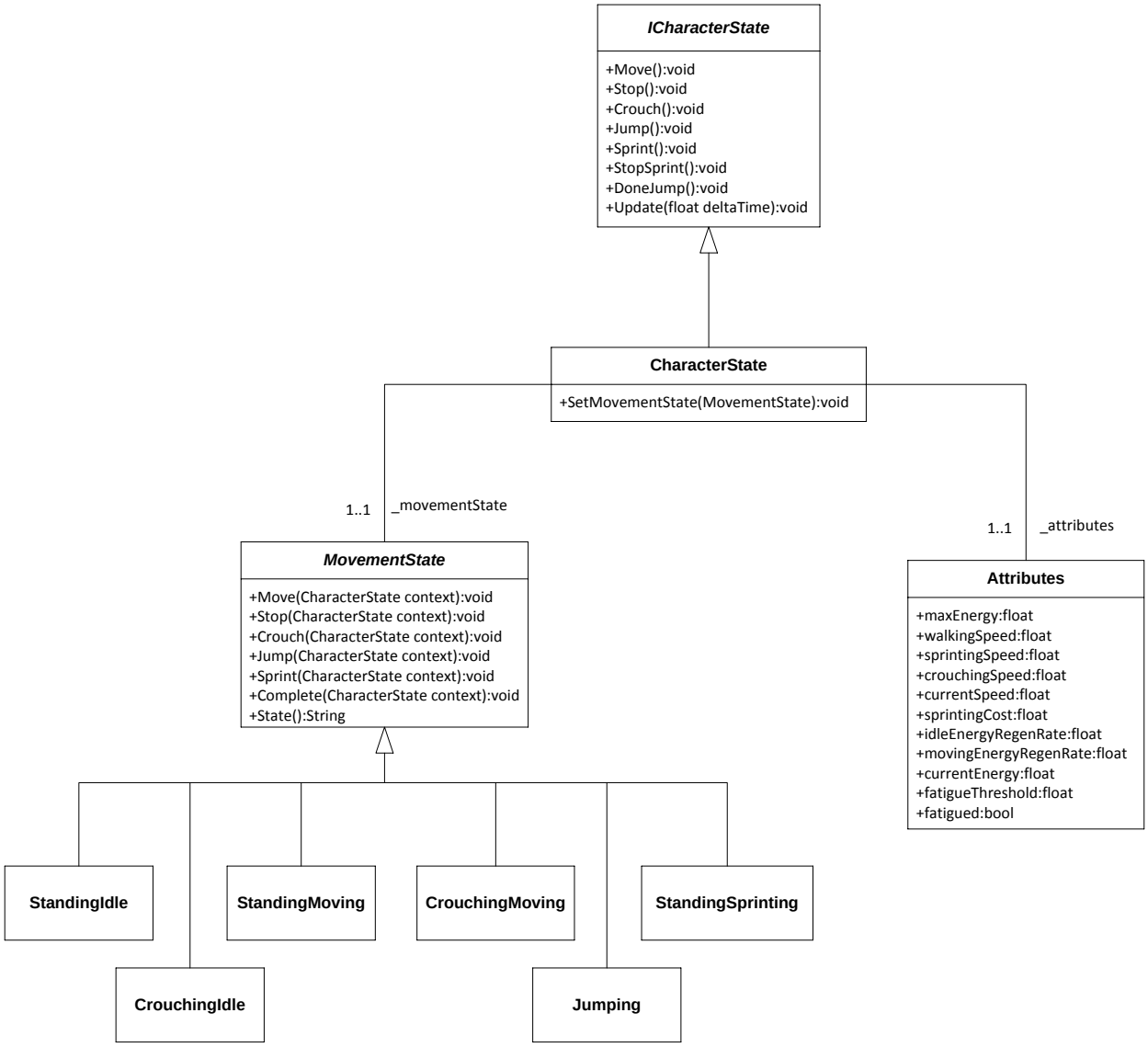
Below is the resulting class diagram:

Figure 11: Character State Class Diagram



When we put it all together, the final product is shown in figure 12.

Figure 12: Final Character State Class Diagram



Now we spent all this time on design, but what did we get out of it? Well we have a structure to follow and implementing it will be simple. Now we have to implement this design. You can implement this design in almost any programming language. I have chosen to implement this in C# because I am using this component in my game that I am developing using the Unity3d game engine.

Implementation

I have posted my implementation of this design in C# in the CharacterStateComponent repository on gitHub. (https://github.com/jmac1908/CharacterStateComponent)