State Pattern

Jumper Example

Lecture Question

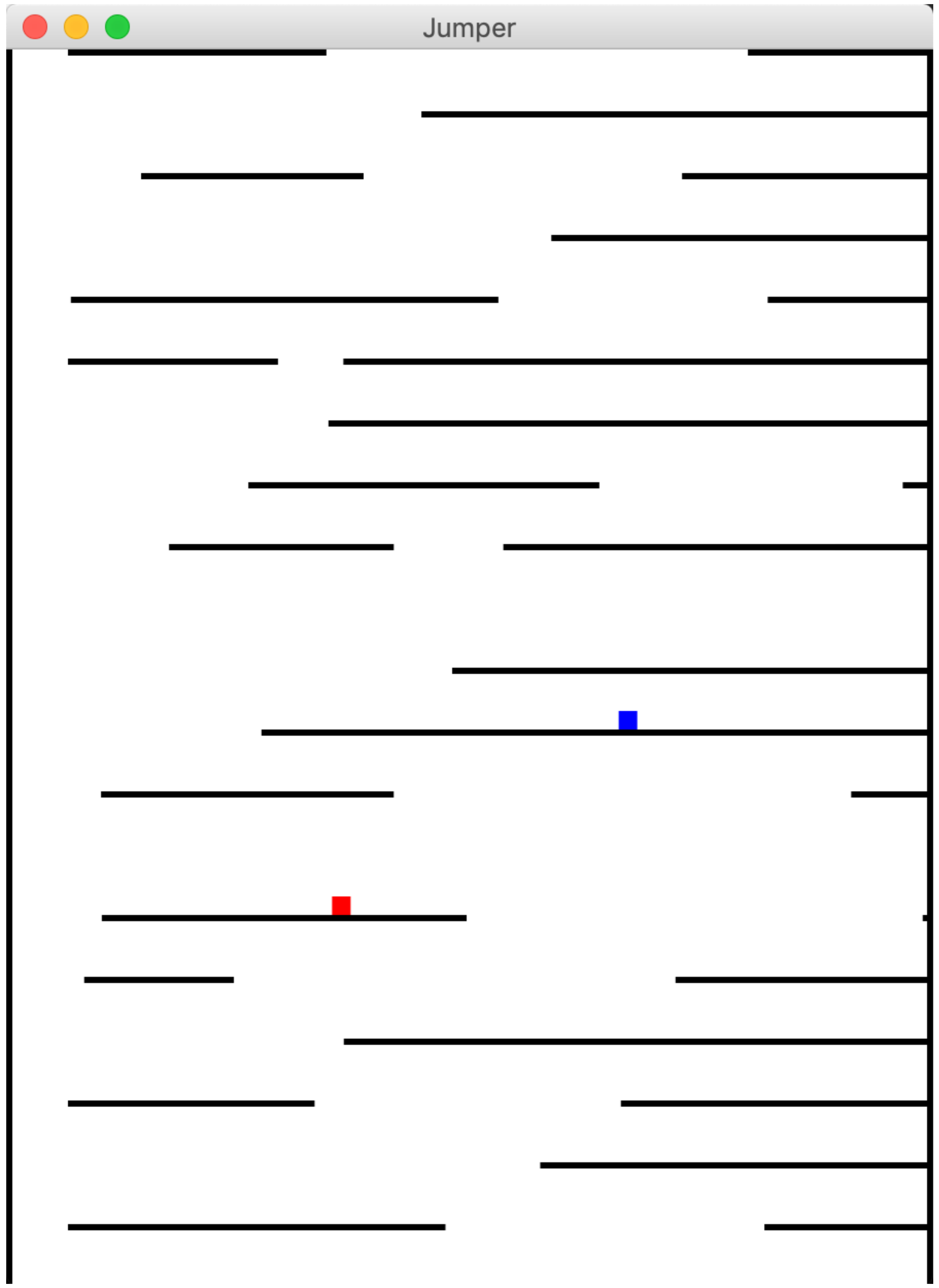
- Simulate a TV without using control flow (ie. Use the state pattern)
- In a package named oop.tv, create a Class named TV with no constructor parameters
- The TV class must contain the following methods as its API:
    - volumeUp(): Unit
    - volumeDown(): Unit
    - mute(): Unit
    - power(): Unit
    - currentVolume(): Int
- In the tests package, write a test suite named TestTV that will test all the functionality on the spec sheet
    - Note: Only call the API methods while testing. Other methods/ variables you create will not exist in the grader submissions

TV Spec Sheet

- TV is initially off when created
- Initial volume is 5
- When the TV is off:
    - Volume up/down and mute buttons do nothing
    - Current volume is 0
- The power button turns the TV on/off
- Volume up button increases volume by 1 up to a maximum volume of 10
- Volume down button decreases volume by 1 down to minimum volume of 0
- Pressing the mute button mutes the TV
- When the TV is muted:
    - Current volume is 0
    - Pressing the mute, volume up, or volume down buttons will unmute the TV and restore the volume to the pre-mute volume (Do not in/decrease the volume)
- When turning the TV back on, the volume should return to its value when the TV was last on
    - When the TV is first turned on the volume will be 5
- If the TV was turned off while muted, when it is turned back on it should not be muted

Jumper

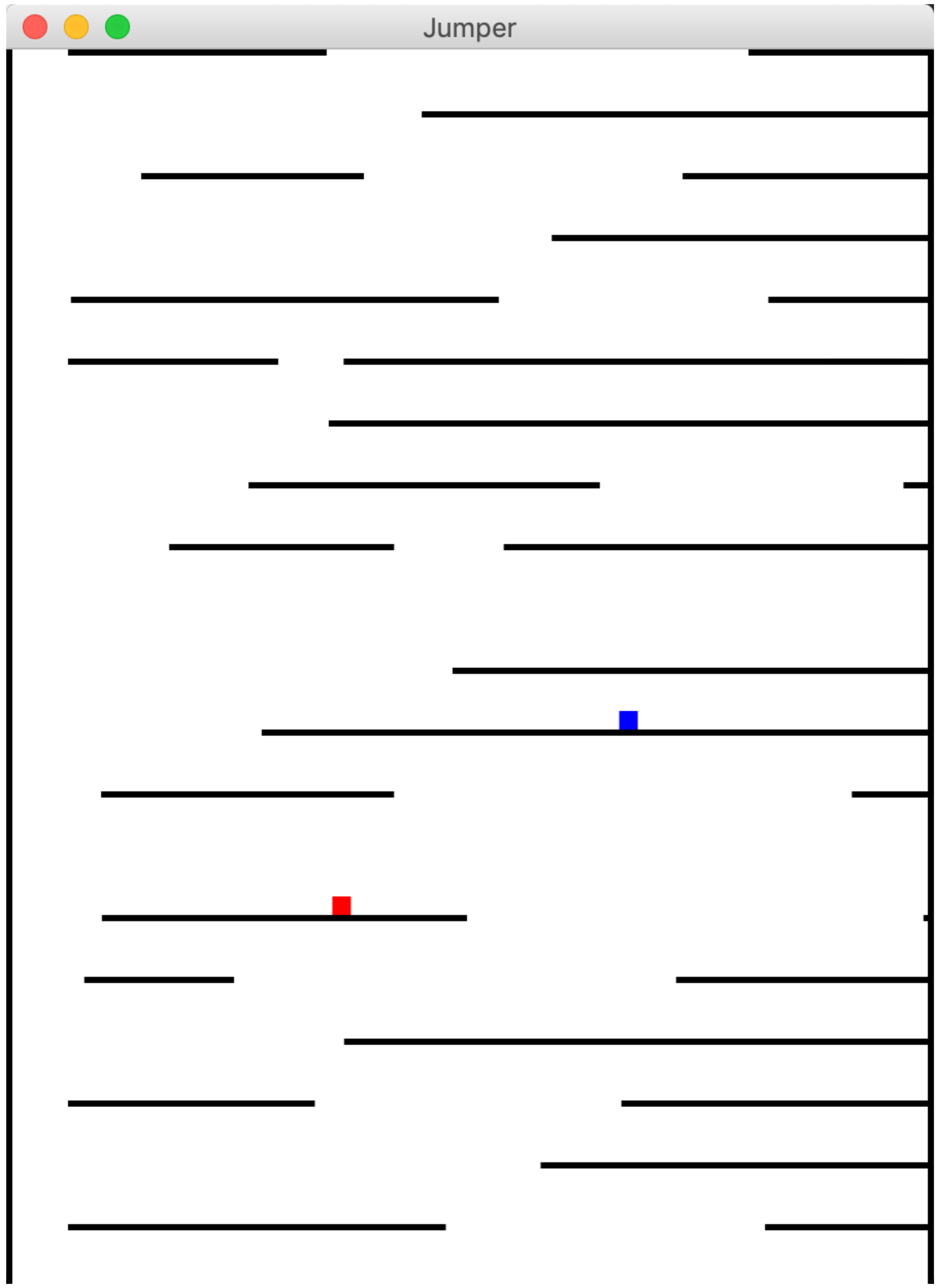
- 2 Player vertical scrolling platform
- Screen scrolls up as the players climb the platforms
- The bottom of the screen is game over
- **Goal**: Climb faster than the other player



Jumper

We've seen how physics was added to the game

- Platforms/Wall extend StaticObject
- Players extend DynamicObject
- Fully compatible with the Physics Engine HW



Jumper - Physics

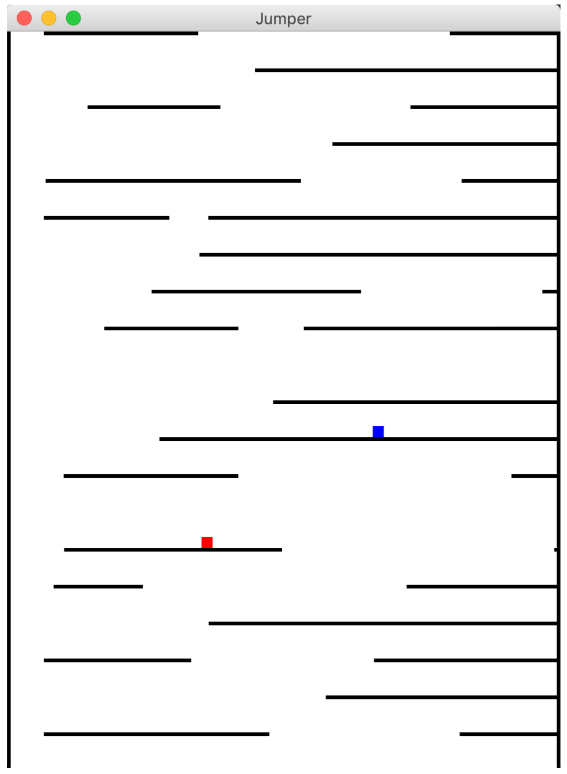
Walls and Platforms extend StaticObject

- Add behavior after collision with player

```
class JumperObject(location: PhysicsVector, dimensions: PhysicsVector)
extends StaticObject(location, dimensions){
  val objectID: Int = JumperObject.nextID
  JumperObject.nextID += 1
}
```

```
class Platform(location: PhysicsVector, dimensions: PhysicsVector) extends
JumperObject(location, dimensions) {

  override def collideWithDynamicObject(otherObject: DynamicObject, face: Integer): Unit = {

   if (face == Face.top) {
      otherObject.velocity.z = 0.0
      otherObject.location.z = this.location.z + this.dimensions.z
      otherObject.onGround()
    }

  }

}
```

```
class Wall(location: PhysicsVector, dimensions: PhysicsVector) extends JumperObject(location,
dimensions){

  override def collideWithDynamicObject(otherObject: DynamicObject, face: Integer): Unit = {
    if(face == Face.negativeX){
      otherObject.velocity.x = 0.0
      otherObject.location.x = this.location.x - otherObject.dimensions.x
    }else if(face == Face.positiveX){
      otherObject.velocity.x = 0.0
      otherObject.location.x = this.location.x + this.dimensions.x
    }
  }

}
```

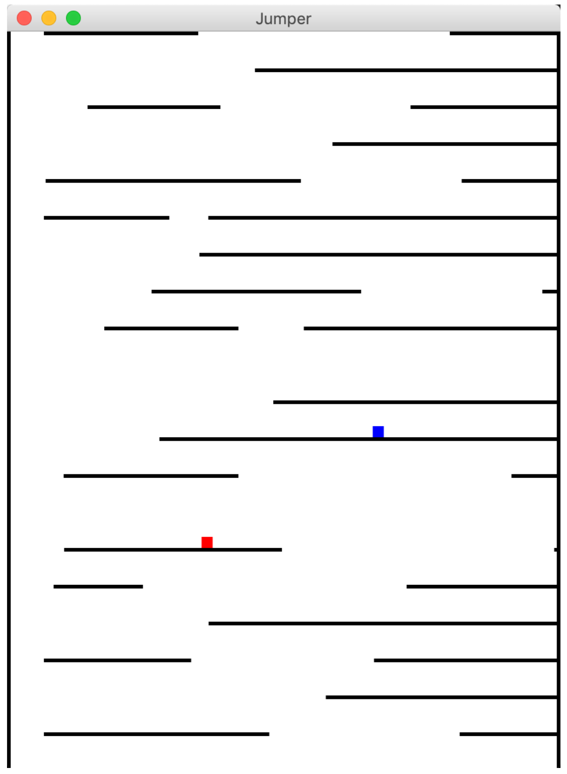


Jumper - Physics

Players extend DynamicObject

- Physics engine applies since all objects in our game are StaticObjects or DynamicObject
- The Player class will set its own velocity based on user inputs
    - Velocities are updated by gravity and collisions
    - User inputs are effectively the "intended" velocity
- How does the Player set its velocity?

```
class Player(playerLocation: PhysicsVector,
             playerDimensions: PhysicsVector
            ) extends DynamicObject(playerLocation, playerDimensions) {

  // ...

}
```



Jumper - Player

How does the Player set its velocity?

- User inputs
- States! <-- Good stuff

Only 3 inputs to control each player

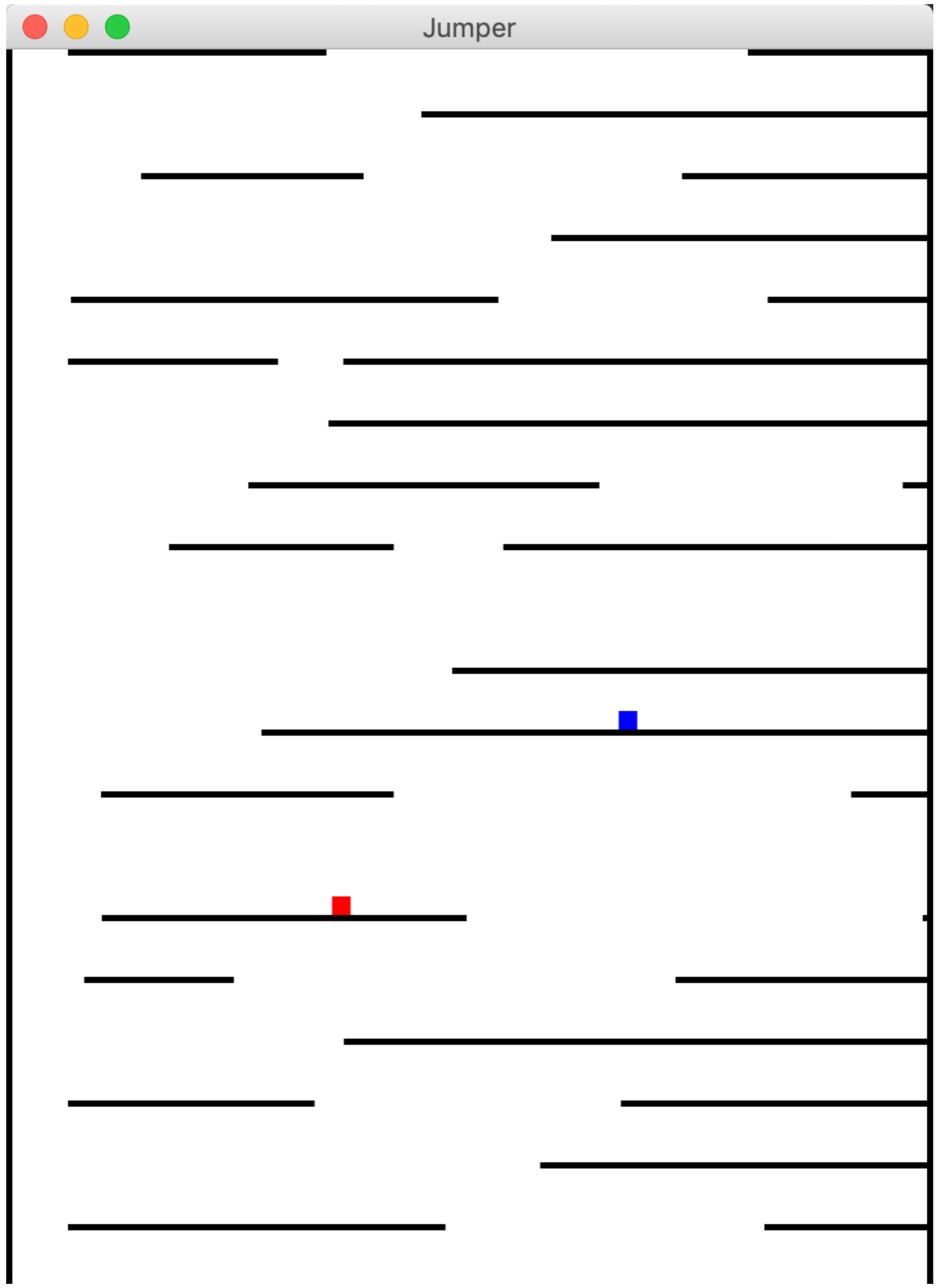
- Left button
- Right button
- Jump button

Player 1:

- a, d, w

Player 2:

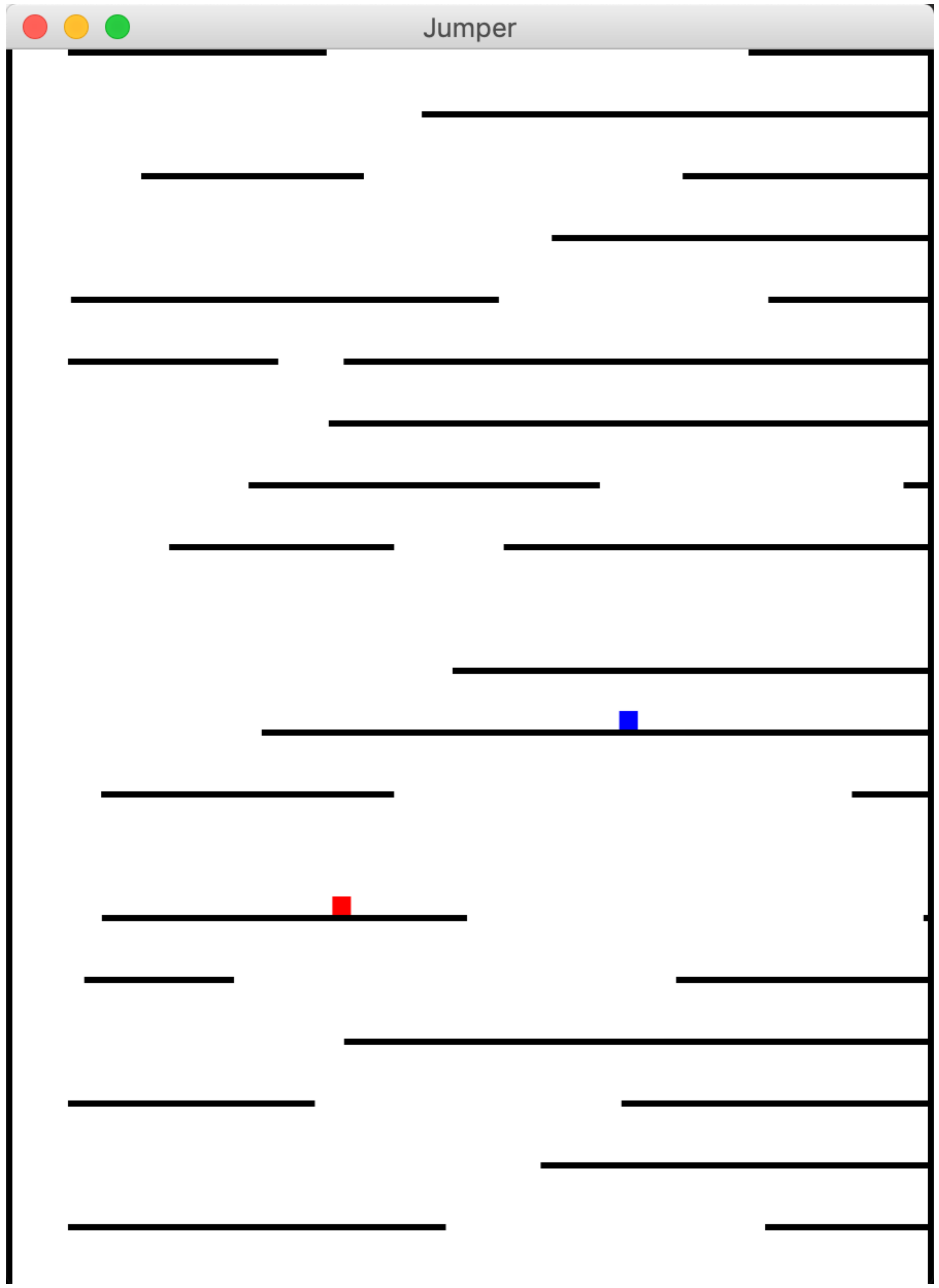
- Left, right, up arrows



Jumper Player Behavior

Each player should

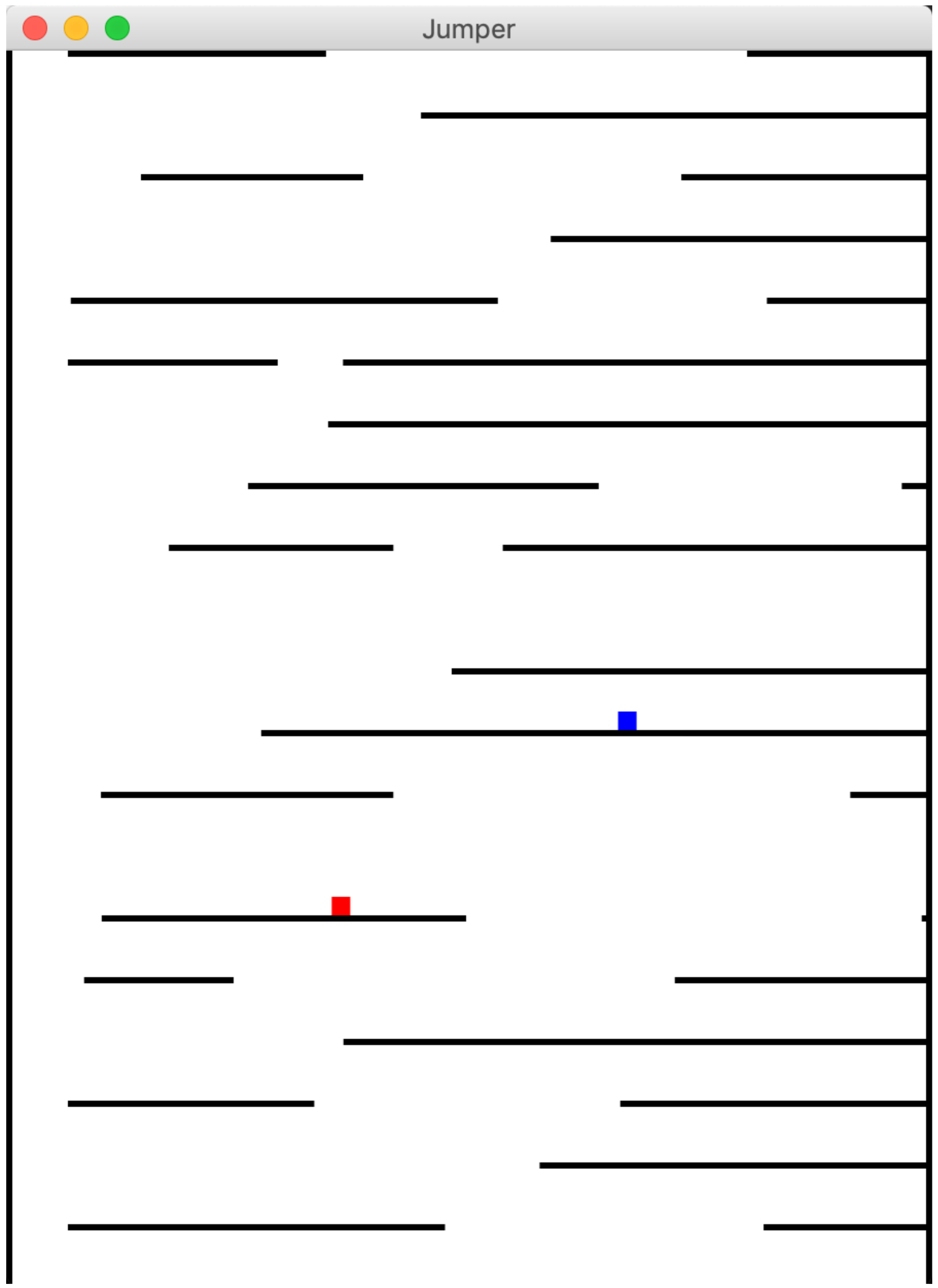
- Walk left and right when keys are pressed
- Jump when jump is pressed
- Jump higher if walking instead of standing still
- Jump at different heights based on how long the jump button is held after a jump
- Move left and right slower while in the air if the direction is changed
- Jump through platforms while jumping up
- Land on platforms while falling down
- Fall if walked off a ledge
- Block all inputs if the bottom of the screen is reached



Player behavior

We could write all this behavior without the state pattern

- Code will likely be hard to follow
- Difficult to add new features



Jumper Player Behavior

Each player should

- Walk left and right when keys are pressed
- Jump when jump is pressed
- Jump higher if walking instead of standing still
- Jump at different heights based on how long the jump button is held after a jump
- Move left and right slower while in the air if the direction is changed
- Jump through platforms while jumping up
- Land on platforms while falling down
- Fall if walked off a ledge
- Block all inputs if the bottom of the screen is reached

How to implement these features?

- Write your API
    - What methods will change behavior depending on the current state of the object
    - These methods define your API and are declared in the state abstract class
- Decide what states should exist
    - Any situation where the behavior is different should be a new state
- Determine the transitions between states

Jumper Player Behavior

Each player should

- Walk left and right **when keys are pressed**
- Jump **when jump is pressed**
- Jump higher if walking instead of standing still
- Jump at different heights based on **how long the jump button is held** after a jump
- Move left and right slower while in the air **if the direction is changed**
- Jump through platforms while jumping up
- **Land on platforms** while falling down
- Fall if **walked off a ledge**
- Block **all inputs** if the bottom of the screen is reached

How to implement these features?

- Write your API
  - What methods will change behavior depending on the current state of the object

API:

- left/right/jump pressed or released
  - 6 methods
- Land on a platform

Jumper Player Behavior

Each player should

- **Walk** left and right when keys are pressed
- **Jump** when jump is pressed
- Jump higher if **walking** instead of **standing** still
- **Jump** at different heights based on how long the jump button is held **after a jump**
- Move left and right slower while **in the air** if the direction is changed
- Jump through platforms while **jumping up**
- Land on platforms while **falling down**
- **Fall** if **walked** off a ledge
- Block all inputs if the **bottom of the screen is reached**

How to implement these features?

- Decide what states should exist

States:

- Standing
- Walking
- Jumping/Rising
- Falling
- Dead (Bellow Screen)

Jumper Player Behavior

Each player should

- **Walk left and right when keys are pressed**
- **Jump when jump is pressed**
- Jump higher if walking instead of standing still
- Jump at different heights based on how long the jump button is held after a jump
- Move left and right slower while in the air if the direction is changed
- Jump through platforms while jumping up
- **Land** on platforms while falling down
- **Fall if walked off a ledge**
- **Block all inputs if the bottom of the screen is reached**

How to implement these features?

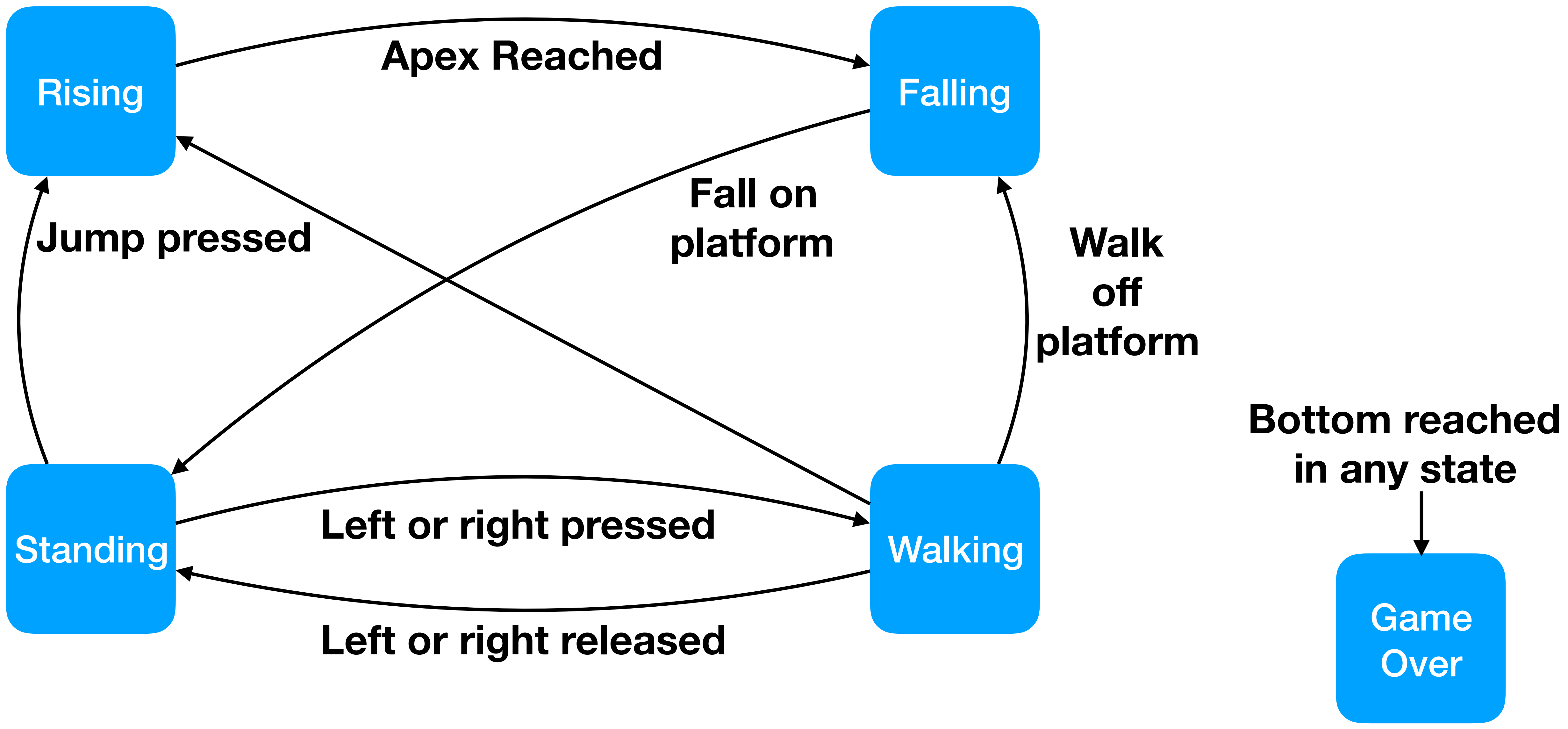
- Determine the transitions between states

State Transitions:

- Standing -> Walking
  - left/right pressed
- Walking -> Standing
  - left/right released
- Walking/Standing -> Jumping
  - Jump pressed
- Falling -> Standing
  - Land on a platform
- Walking -> Falling
  - Walk off a platform
- Jumping -> Falling
  - Apex of jump reached
- Any -> GameOver
  - Reach the bottom of the screen

Jumper Player Behavior

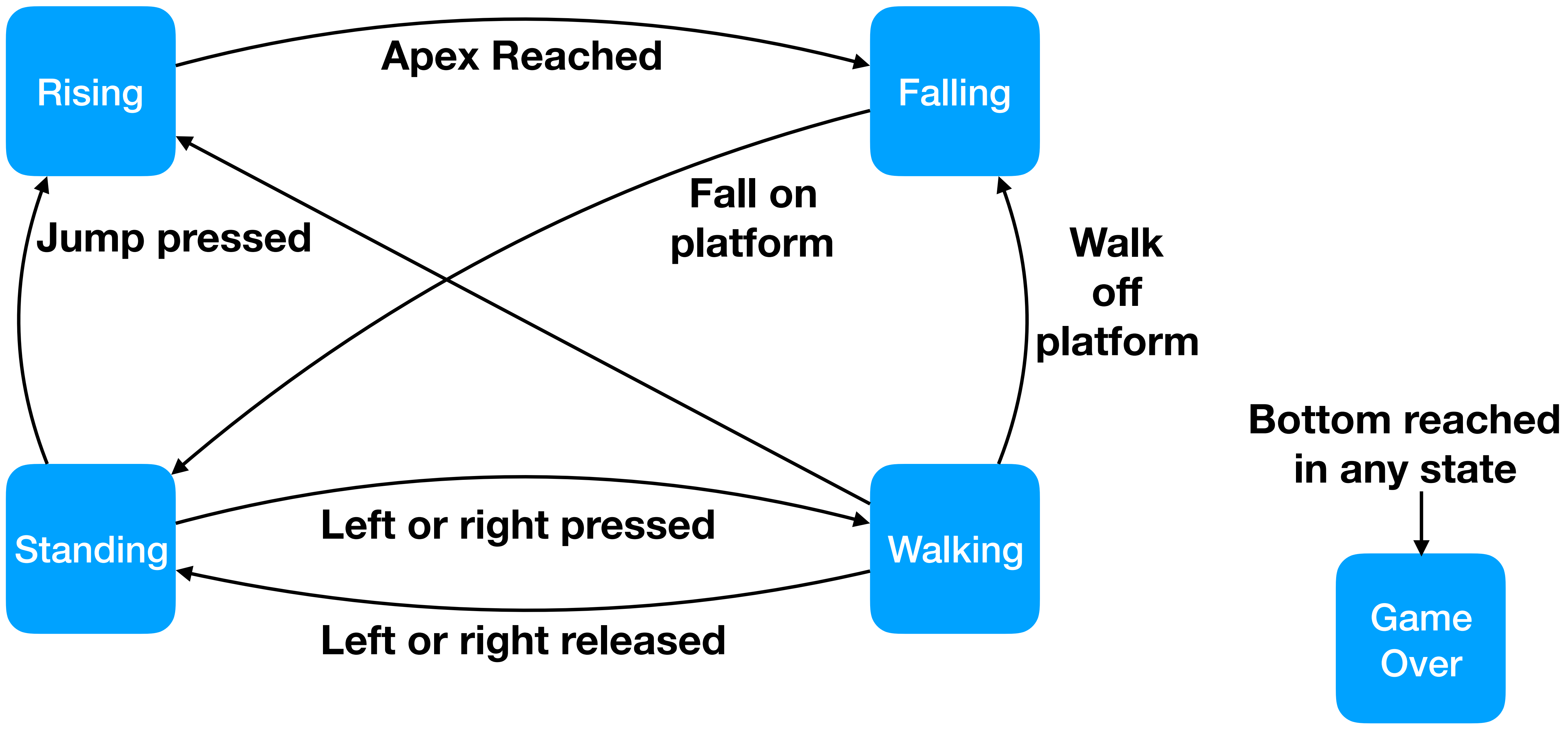
Let's visualize the states and transitions in a state diagram



Jumper Player Behavior

For each state implement the API methods with the desired behavior in that state

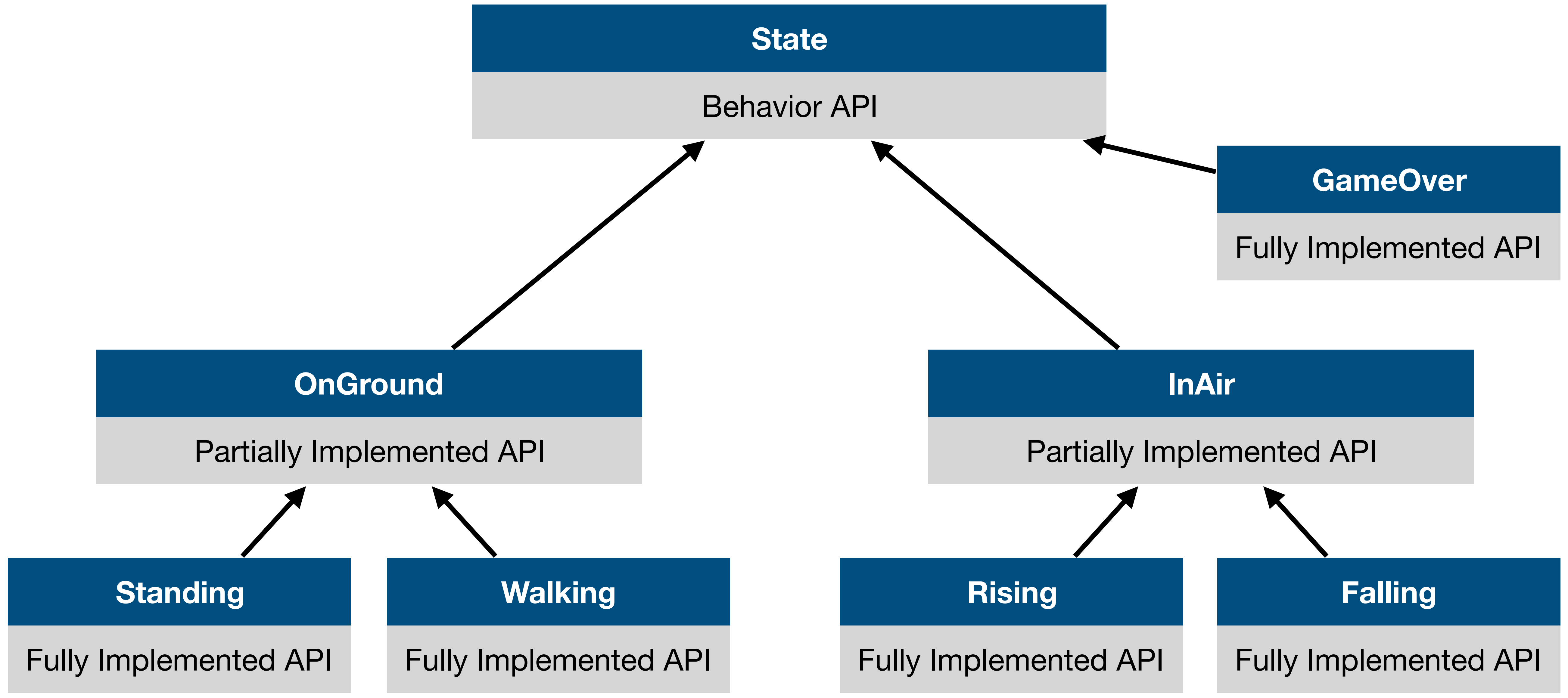
- Add default behavior in the state subclass



Jumper Player Behavior

Use inheritance to limit duplicate code

- Factor out common behavior between states into new classes



Adding Functionality

Task: Add a double jump to Jumper

- How can we add a double jump?
  - Players can jump 1 additional time while in the air
- With poor design
  - This could be extremely difficult!
  - May required modifying a significant amount of existing code
- With our state pattern
  - No problem at all

Adding Functionality

Task: Add a double jump to Jumper

- Add functionality to existing states
    - Rising and Falling states now react to the jump button by jumping again (Set velocity.z to the jump velocity)
- We'll add new states
    - RisingAfterDoubleJump/FallingAfterDoubleJump
    - Extend Rising/Falling respectively
    - Override the jump button press to do nothing
- Update state transitions
    - Press jump from Rising/Falling transitions to the respective AfterDoubleJump state
    - Reaching the apex in RisingAfterDoubleJump transitions to FallingAfterDoubleJump (Not Falling)

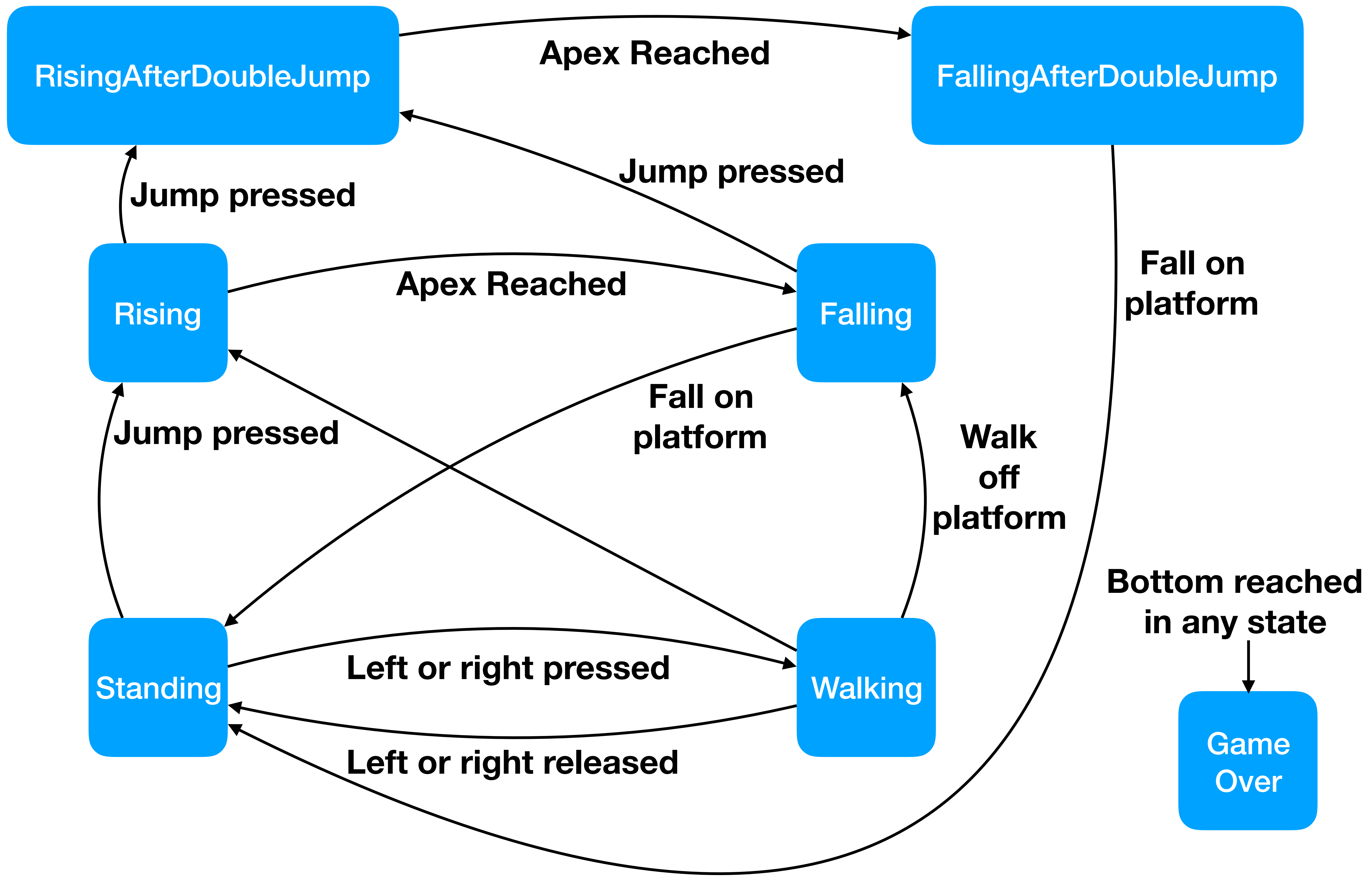
Adding Functionality

Task: Add a double jump to Jumper

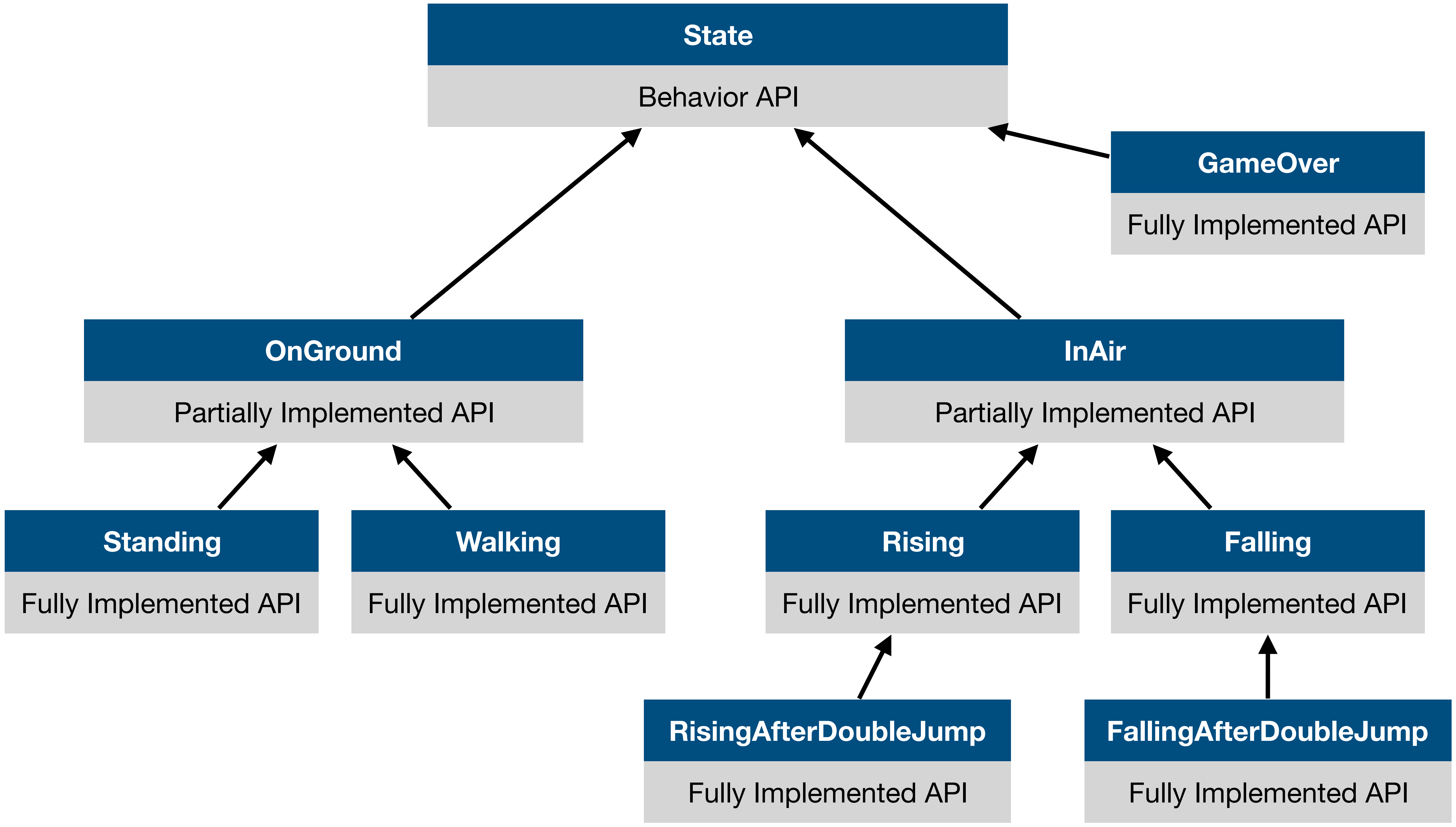
- This task could have been completed with a boolean flag instead of using new states
- If this approach is used for many features the code will be harder to maintain
- **More to the point**: What if your professor says you can't use control flow, but you have a situation where a button should only work once?
    - Try adding more states

```
var usedDoubleJump = false

override def jumpPressed(): Unit = {
  if(!this.usedDoubleJump) {
    player.velocity.z = player.standingJumpVelocity
    this.usedDoubleJump = true
  }
}
```

Jumper Player Behavior



Jumper Player Behavior



Lecture Question

- Simulate a TV without using control flow (ie. Use the state pattern)
- In a package named oop.tv, create a Class named TV with no constructor parameters
- The TV class must contain the following methods as its API:
    - volumeUp(): Unit
    - volumeDown(): Unit
    - mute(): Unit
    - power(): Unit
    - currentVolume(): Int
- In the tests package, write a test suite named TestTV that will test all the functionality on the spec sheet
    - Note: Only call the API methods while testing. Other methods/ variables you create will not exist in the grader submissions

TV Spec Sheet

- TV is initially off when created
- Initial volume is 5
- When the TV is off:
    - Volume up/down and mute buttons do nothing
    - Current volume is 0
- The power button turns the TV on/off
- Volume up button increases volume by 1 up to a maximum volume of 10
- Volume down button decreases volume by 1 down to minimum volume of 0
- Pressing the mute button mutes the TV
- When the TV is muted:
    - Current volume is 0
    - Pressing the mute, volume up, or volume down buttons will unmute the TV and restore the volume to the pre-mute volume (Do not in/decrease the volume)
- When turning the TV back on, the volume should return to its value when the TV was last on
    - When the TV is first turned on the volume will be 5
- If the TV was turned off while muted, when it is turned back on it should not be muted

Lecture Question

Hints/Suggestions

- You can use Math.min and Math.max to prevent the volume from being negative or greater than 10 without using a conditional
    - Ex. volume = Math.max(0, volume-1) will prevent negative volumes
- You can store the volume in a variable in the TV class so it will persist as the state changes (ex. Turning the TV off will not change this variable, the off state just won't access the variable and will return 0 instead when currentVolume is called)