State Pattern

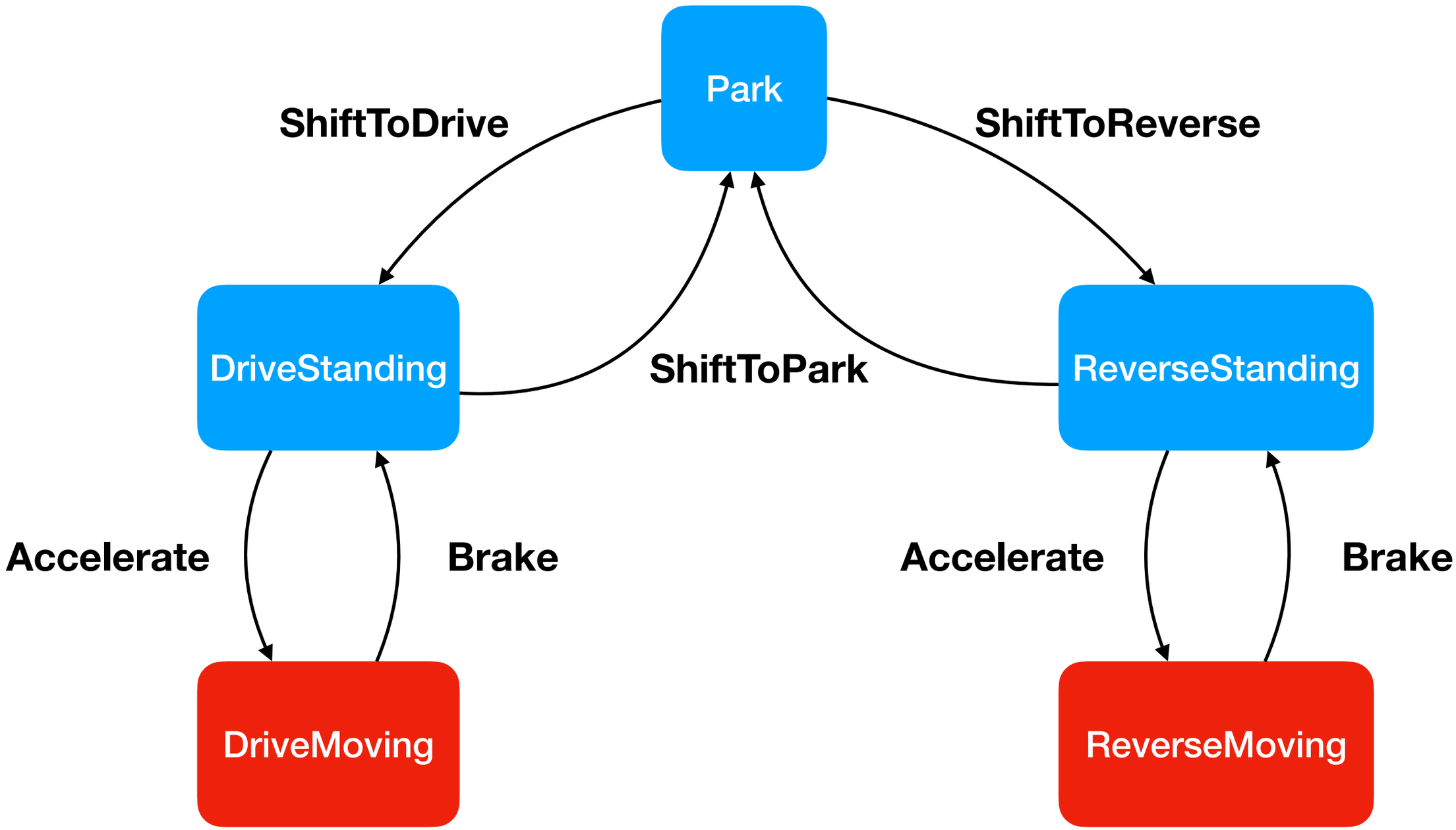
Lecture Question

- Simulate a Car without using control flow (ie. Use the state pattern)
- In a package named oop.car, create a Class named Car with no constructor parameters
- The Car must contain the following methods as its API:
  - shiftToDrive(): Unit
  - shiftToPark(): Unit
  - shiftToReverse(): Unit
  - accelerate(): Unit
  - brake(): Unit
  - velocity(): Int
- In the tests package, write a test suite named TestCar that will test all the functionality on the spec sheet
  - Note: Only call the API methods while testing. Other classes/methods/ variables you create will not exist in the grader submissions

Car Spec Sheet

- Car is initially in Park
- Initial velocity is 0
- When the Car is in Park:
  - Accelerating and braking have no effect
  - The car can shift into drive or reverse
- When the car is in Drive:
  - Calling accelerate will **increase** the velocity of the car by 10
  - Calling brake will completely stop the car (velocity of 0)
  - The car cannot shift into Reverse
  - The car cannot shift into Park while moving
- When the car is in Reverse:
  - Calling accelerate will **decrease** the velocity of the car by 5 (negative velocity)
  - Calling brake will completely stop the car (velocity of 0)
  - The car cannot shift into Drive
  - The car cannot shift into Park while moving

Car State Diagram



State Pattern - Closing Thoughts

State pattern trade-offs

- **Pros**
  - Organizes code when a single class can have very different behavior in different circumstances
  - Each implemented method is only concerned with the reaction to 1 event (API call) in 1 state
  - Easy to change or add new behavior after the state pattern is setup
- **Cons**
  - Can add complexity if there are only a few states or if behavior does not change significantly across states
  - Spreading the behavior for 1 class across many classes can look complex and require clicking through many files to understand all the behavior

State Pattern - Closing Thoughts

- Do not use the state pattern everywhere
  - Decide if a class is complex enough to benefit from this pattern before applying it

- The state pattern in this class
  - I have to force you to use it by removing control flow (Not realistic)
  - Used to reinforce your understanding of **inheritance** and **polymorphism**
  - Used as an example of a design pattern that can help organize your code
- When you're not forced to use this pattern
  - Weight the pros and cons to decide when it is the best approach

Live Coding

Live Coding

- No control flow
- Ping Pong
- API:
  - ballLanding()
  - ballLandingOutOfPlay()
  - PlayerOneHitsBall()
  - PlayerTwoHitsBall()
- PingPong state (not in state object)
  - Player scores

Spec Sheet

- Ball can be on either side of the table or on the floor
- If the ball lands out of play
  - The last person to hit the ball (the other person gets a point)
- Ball must bounce once before you can hit it
  - If you hit it before bounce, other player gets a point
- Ball can't bounce twice
  - If it does, the last person to hit it gets a point
- Track score

Lecture Question

- Simulate a Car without using control flow (ie. Use the state pattern)
- In a package named oop.car, create a Class named Car with no constructor parameters
- The Car must contain the following methods as its API:
  - shiftToDrive(): Unit
  - shiftToPark(): Unit
  - shiftToReverse(): Unit
  - accelerate(): Unit
  - brake(): Unit
  - velocity(): Int
- In the tests package, write a test suite named TestCar that will test all the functionality on the spec sheet
  - Note: Only call the API methods while testing. Other classes/methods/ variables you create will not exist in the grader submissions

Car Spec Sheet

- Car is initially in Park
- Initial velocity is 0
- When the Car is in Park:
  - Accelerating and braking have no effect
  - The car can shift into drive or reverse
- When the car is in Drive:
  - Calling accelerate will **increase** the velocity of the car by 10
  - Calling brake will completely stop the car (velocity of 0)
  - The car cannot shift into Reverse
  - The car cannot shift into Park while moving
- When the car is in Reverse:
  - Calling accelerate will **decrease** the velocity of the car by 5 (negative velocity)
  - Calling brake will completely stop the car (velocity of 0)
  - The car cannot shift into Drive
  - The car cannot shift into Park while moving

Live Coding

- No control flow
- Make an Elevator
  - 3 floors
- API:
  - callButton(Int)
  - floorButton(Int)
  - doorOpen()
  - doorClose()

Spec Sheet

- If someone hits the call button
  - Calls the elevator to that floor
  - Overrides previous call
- Push a floor button
  - Move to that floor
- Can't move while door is open
  - Still gets the call to a new floor

Live Coding

- No control flow
- Is this person willing to go outside?
- Weather can:
  - warmUp()
  - coolDown()
  - startPrecipitating()
  - stopPrecipitating()

Spec Sheet

- When it's raining
  - Only if wearing a raincoat
- When it's snowing
  - No
- When it's Warm
  - Yes
- When it's cold
  - Only if wearing a winter coat

Live Coding

- No control flow
- Write a Eevee class
- API
  - ballThrown()
  - battle()
  - feedWaterStone()
  - feedFireStone()
  - feedThunderStone()

Spec Sheet

- Initial state == Wild and full health and base evolution
- battling the pokemon damages it making it easier to catch
  - damaging twice guarantees a catch
- Catching a Pokemon tames it
- After being caught it can't be caught again
- After being caught, battle is a trainer battle
- After caught it can evolve