State Pattern

State Pattern

- Used when a class naturally has multiple "states" with different behavior
- Class has a state variable containing the current state
- Defer behavior to the state
    - Avoids conditionals in behavior methods
    - Decisions made by changing state
- Decisions made on type (Polymorphism) not value (Conditionals)
- Modularizes code
    - More, but smaller, pieces of code
- Easy to add new features without breaking tested features

State Pattern

- State represented by an abstract class (or trait, interface)
  - Defines the methods that can be called
- Extend the state class for each concrete state
  - One class for each possible state of the class
- Each state will have a reference to the object to which it is attached
  - Use this reference access other state variables
  - Use this reference to change state

Tic-Tac-Toe

- Let's build a GUI for a tic-tac-toe game
    - We'll exclude checking for a winner and restarting the game

```
class Game {

  var playerTurn = "X"

  val movesMade = Array(
    Array(" ", " ", " "),
    Array(" ", " ", " "),
    Array(" ", " ", " ")
  )

  def buttonClicked(x:Int, y:Int): Unit = {
    if(this.movesMade(y)(x) == " "){
      this.movesMade(y)(x) = this.playerTurn
      this.playerTurn = if(this.playerTurn == "X") "O" else "X"
    }
  }

}
```

```
class MoveAction(game: Game, button: Button, i:Int, j:Int) extends EventHandler[ActionEvent] {
  override def handle(event: ActionEvent): Unit = {
    game.buttonClicked(i, j)
    button.text.value = game.movesMade(j)(i)
  }
}
```

Tic-Tac-Toe

- Build a 3x3 grid for all the squares
- Use nested if statements to decide how to react to a click

```scala
class Game {

  var playerTurn = "X"

  val movesMade = Array(
    Array(" ", " ", " "),
    Array(" ", " ", " "),
    Array(" ", " ", " ")
  )

  def buttonClicked(x:Int, y:Int): Unit = {
    if(this.movesMade(y)(x) == " "){
      this.movesMade(y)(x) = this.playerTurn
      this.playerTurn = if(this.playerTurn == "X") "O" else "X"
    }
  }

}
```

```scala
class MoveAction(game: Game, button: Button, i:Int, j:Int) extends EventHandler[ActionEvent] {
  override def handle(event: ActionEvent): Unit = {
    game.buttonClicked(i, j)
    button.text.value = game.movesMade(j)(i)
  }
}
```

Tic-Tac-Toe

- Not bad at all for a simple game like this
  - State design pattern will be overcomplicated in this example
  - Used as a smaller example

```scala
class Game {

  var playerTurn = "X"

  val movesMade = Array(
    Array(" ", " ", " "),
    Array(" ", " ", " "),
    Array(" ", " ", " ")
  )

  def buttonClicked(x:Int, y:Int): Unit = {
    if(this.movesMade(y)(x) == " "){
      this.movesMade(y)(x) = this.playerTurn
      this.playerTurn = if(this.playerTurn == "X") "O" else "X"
    }
  }

}
```

```scala
class MoveAction(game: Game, button: Button, i:Int, j:Int) extends EventHandler[ActionEvent] {
  override def handle(event: ActionEvent): Unit = {
    game.buttonClicked(i, j)
    button.text.value = game.movesMade(j)(i)
  }
}
```

Tic-Tac-Toe with State

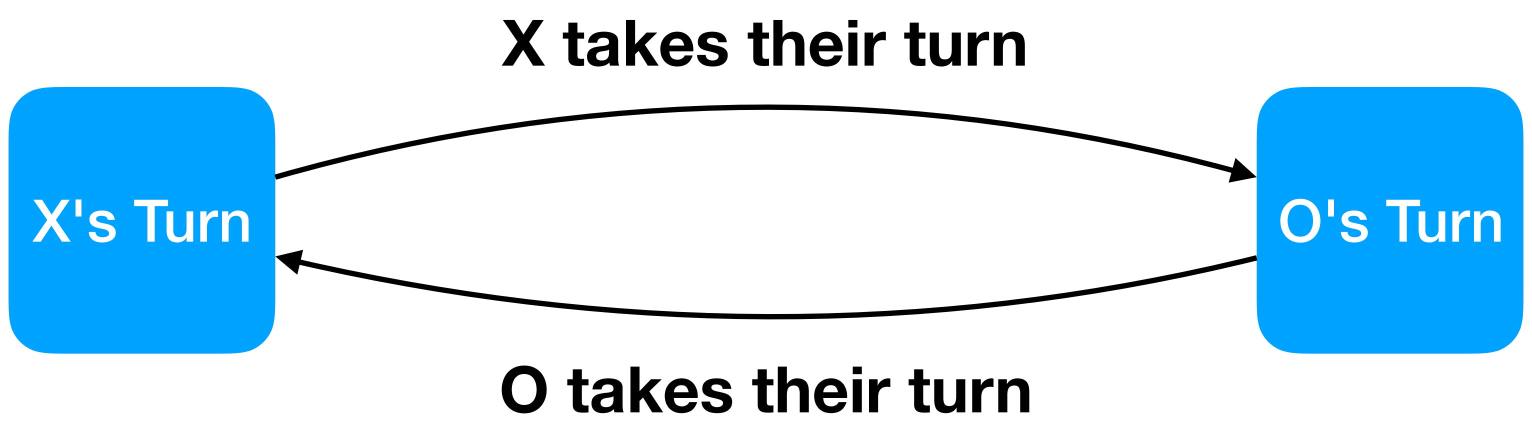
- Defer to state for functionality

```scala
class Game {

  var gameState: GameState = new GameStateXTurn(this)

  val boardSize = 3

  val squares: Array[Square] = (
    for {i <- 0 until boardSize
         j <- 0 until boardSize} yield {
      new Square(this, i, j)
    }).toArray


  def playerTurn(): String = {
    this.gameState.playerTurn()
  }

  def takeTurn(): Unit = {
    this.gameState.takeTurn()
  }

}
```

```scala
class Square(game: Game, val x:Int, val y:Int) extends EventHandler[ActionEvent]  {

  var state: SquareState = new SquareStateEmpty(game, this)

  override def handle(event: ActionEvent): Unit = {
    this.state.clicked()
  }

}
```

Tic-Tac-Toe with State

- Game can be in two different states depending whose turn it is
- Switch between these sates as each player takes their turn
- State returns the symbol for the current player



Tic-Tac-Toe with State

- Game state changes depending on who's turn it is

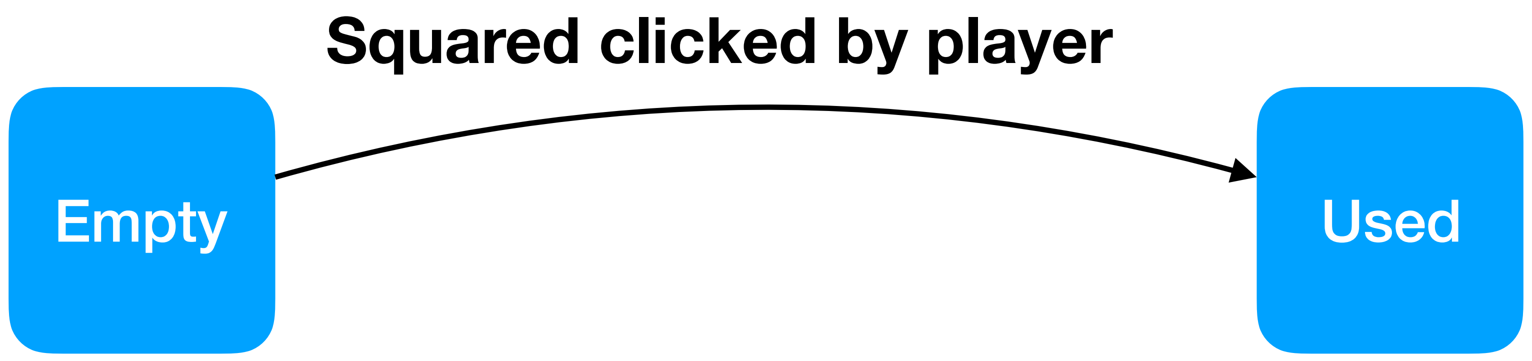
```scala
trait GameState {

  def playerTurn(): String
  def takeTurn(): Unit

}
```

```scala
class GameStateXTurn(game: Game) extends GameState {

  override def playerTurn(): String = "X"

  override def takeTurn(): Unit = game.gameState = new GameStateOTurn(game)

}
```

```scala
class GameStateOTurn(game: Game) extends GameState {

  override def playerTurn(): String = "O"

  override def takeTurn(): Unit = game.gameState = new GameStateXTurn(game)

}
```

Tic-Tac-Toe with State

- Square can either be empty or used
- Empty
  - Initial state of a square
  - When clicked, sate becomes used
- Used
  - Remembers the player who used it
  - No action when clicked again



Tic-Tac-Toe with State

- Square can either be empty or used

```scala
trait SquareState {

  var occupant: String
  def clicked(): Unit

}
```

```scala
class SquareStateEmpty(game:Game, square: Square) extends SquareState {

  override var occupant: String = " "

  override def clicked(): Unit = {
    square.state = new SquareStateUsed(game, game.playerTurn())
    game.takeTurn()
  }

}
```

```scala
class SquareStateUsed(game:Game, occupiedBy: String) extends SquareState {

  override var occupant: String = occupiedBy

  override def clicked(): Unit = {}

}
```

Tic-Tac-Toe with State

- Squares start in empty state
- Defer to state for functionality when clicked
- A few extra lines to update the GUI
  - Could be handled in GUI itself

```
class Square(game: Game, val x:Int, val y:Int) extends EventHandler[ActionEvent]  {

  var button:Button = _  // set by GUI
  var state: SquareState = new SquareStateEmpty(game, this)

  override def handle(event: ActionEvent): Unit = {
    this.state.clicked()
    button.text.value = this.state.occupant
  }

}
```

Tic-Tac-Toe with State

- With states we created 6 new classes
- Spread out the functionality and decisions
  - Each class has very little logic and is easy to reason about
  - All classes combined create a more complex program
- The trade-offs
  - No nested if statement
  - More classes and complex project structure
- For tic-tac-toe the trade-offs might not be worth using the pattern
  - For more complex programs the trade-off can payoff

Lecture Question

Create a model for a vending machine. In a package named vending create a class named VendingMachine with these 4 methods (This defines the API of the model)

- insertCoin(Int):Unit
    - User inserts a coin with an integer representing the number of cents the coin was worth
- coinReturnPressed(): Int
    - Returns all the coins to the user (You don't have to actually return the coins, but must update the state and state variables to reflect this). The return value is the number of cents returned to the user
- beverageButtonPressed(): Boolean
    - Returns true is enough coins are in the machine to afford a beverage costing 200 cents and returns all additional coins to the user (you don't have to actually return anything for the coins, but the total value of all coins in the machine should be set to 0 when a beverage is purchased)
    - returns false if the user cannot afford a beverage and does not return coins
- destroy():Unit
    - Destroys the machine. After this method is called the functionality of the machine is broken (coin returns always returns 0; beverage is never dispensed)

* This question will be open until midnight

Lecture Question

- *You are encouraged to complete this using the state design pattern to gain practice for the calculator homework, but the grader only checks your functionality of the 4 methods and not your approach
- You will need at least 1 if statement to complete this question