CMSC 132: OBJECT-ORIENTED PROGRAMMING II

State Design Pattern / Dynamic Systems

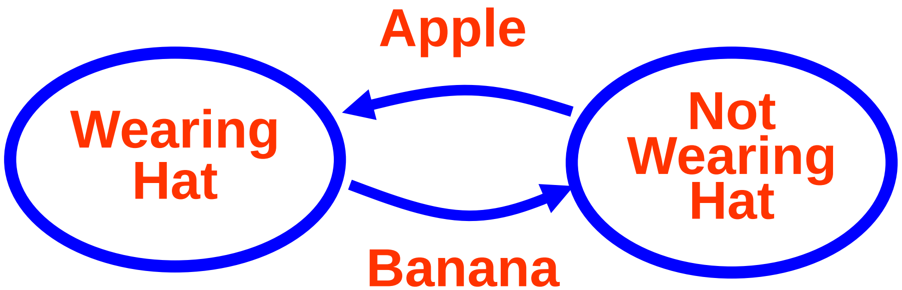
Department of Computer Science
University of Maryland, College Park

State Pattern

- **Definition**
  - Represent change in an object's behavior using its member classes
- **Where to use & benefits**
  - Control states without many if-else statements
  - Represent states using classes
  - Every state has to act in a similar manner
  - Simplify and clarify the program
- **Example**
  - States representing finite state machine (FSM)
  - Original
    - Each method chooses action depending on state
    - Behavior may be confusing, state is implicit
  - Using pattern
    - State interface defines list of actions for state
    - Define inner classes implementing State interface
    - Finite state machine instantiates each state and tracks its current state
    - Current state used to choose action
- **Example:** StateCode

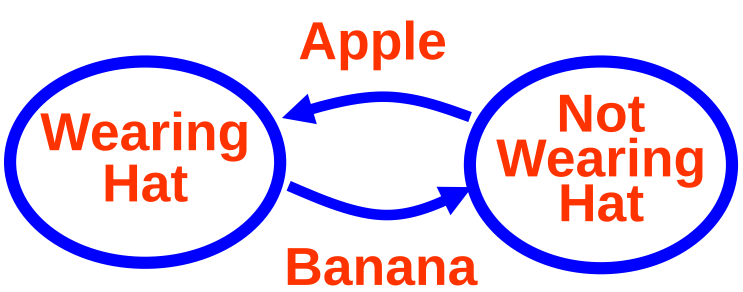
State Example – Original Code

```
public class FickleFruitVendor {
    boolean wearingHat;
    boolean isHatOn() { return wearingHat; }
    String requestFruit() {
        if (wearingHat) {
            wearingHat = false;
            return "Banana";
        } else {
            wearingHat = true;
            return "Apple";
        }
    }
}
```



State Example

```
public interface State {
    boolean isHatOn();
    String requestFruit();
}

public class FickleFruitVendor {
    State wearingHat = new WearingHat();
    State notWearingHat = new
   NotWearingHat();

    // track current state of Vendor
    State currentState = wearingHat;

    // behavior depends on current state
    public boolean isHatOn() {
        return currentState.isHatOn();
    }
    public String requestFruit() {
        return currentState.requestFruit();
    }

    // Inner class
    public class WearingHat implements State {
        boolean isHatOn() { return true; }
        String requestFruit() {
            // change state
            currentState = notWearingHat;
            return "Banana";
        }
    }

    // Inner class
    public class NotWearingHat implements State {
        boolean isHatOn() { return false; }
        String requestFruit() {
            // change state
            currentState = wearingHat;
            return "Apple";
        }
    }
} // End of FickleFruitVendor class
```



Dynamic Systems

- **Dynamic Systems:** Systems that change dynamically over time. Such systems arise naturally when writing programs involving **graphical user interfaces** (video games, interactive graphics). Some issues:
  - How does the system respond to external events or stimuli? Called **reactive** or **event-driven** systems.
  - **State transition**: Most dynamic systems are defined in terms of information called its **state**.
    - What are the **possible states** the system can be in?
    - What sorts of **state transitions** are possible, and under what circumstances do transitions occur?
    - What **actions** are performed in each state?

Dynamic Systems

- **Examples**:
  - **DVD Player/Recorder**: Behavior to remote control commands varies depending on the operating state: recording, playback, idle.
  - **Figure drawing program**: (e.g. Paint) The meaning of mouse actions depends on the drawing state: line, curve, ellipse, rectangle, polygon.

  - **Video game**: The meaning of user inputs depends on the current context in which the game is operating.
  - **Digital watch**: Has various modes (clock, stop watch, timer) and the meaning of buttons varies with the mode.
- How do we **design programs** for such event-driven systems?

State Transition Systems

- These systems have a number of elements in common:
  - **Events**: Inputs/Stimuli come in the form of events (rather than traditional text prompt + text input).
  - **State**: The behavior depends on **internal information** (which the user cannot see) called the system's **state** or **context**.
  - **Transitions**: Events can cause changes in the context and other state information.
  - **Actions**: Actions (which the user may or may not see) are performed in response to each event/transition.
  - **(Spontaneous actions)**: Some actions take place without any user input. (Example: animation in a video game.) These can be modeled as responses to system-generated events, like timer events.

Calculator

- Let us consider the case of a simple **interactive calculator**.
  - **Events**: occur when user hits the keys.
  - **State**: Operands, memory, internal state of the computation (more about this later).
  - **Actions**: Perform calculations, update the display.
- What **internal state** information is needed?
- **Example**: " **3 4 + 5 6 =** "
  When the "**=**" is processed, the calculator has saved the following information internally:
  - **First operand**: "**34**" (call this **v1**)
  - **Operator**: "**+**" (call this **op**)
  - **Second operand**: "**56**" (call this **v2**)
- It must also know **which operand** it is reading, first or second.

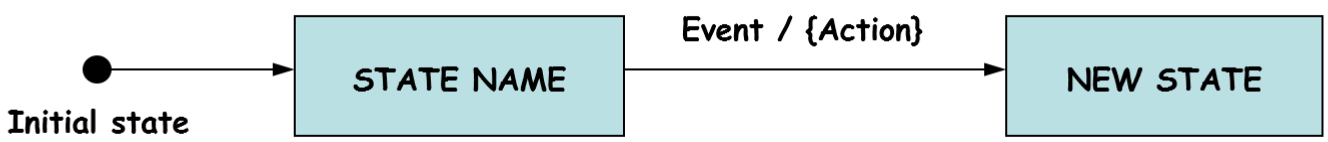
Calculator

- **Calculator**: Has three **states**, or **contexts**:
  Reading-First-Operand (**RFO**): reading digits for the first operand.
  Reading-Second-Operand (**RSO**): reading digits for the second operand.
  Error (**ERR**): An error occurs (e.g., invalid operand or divide by 0).
- **Example**:

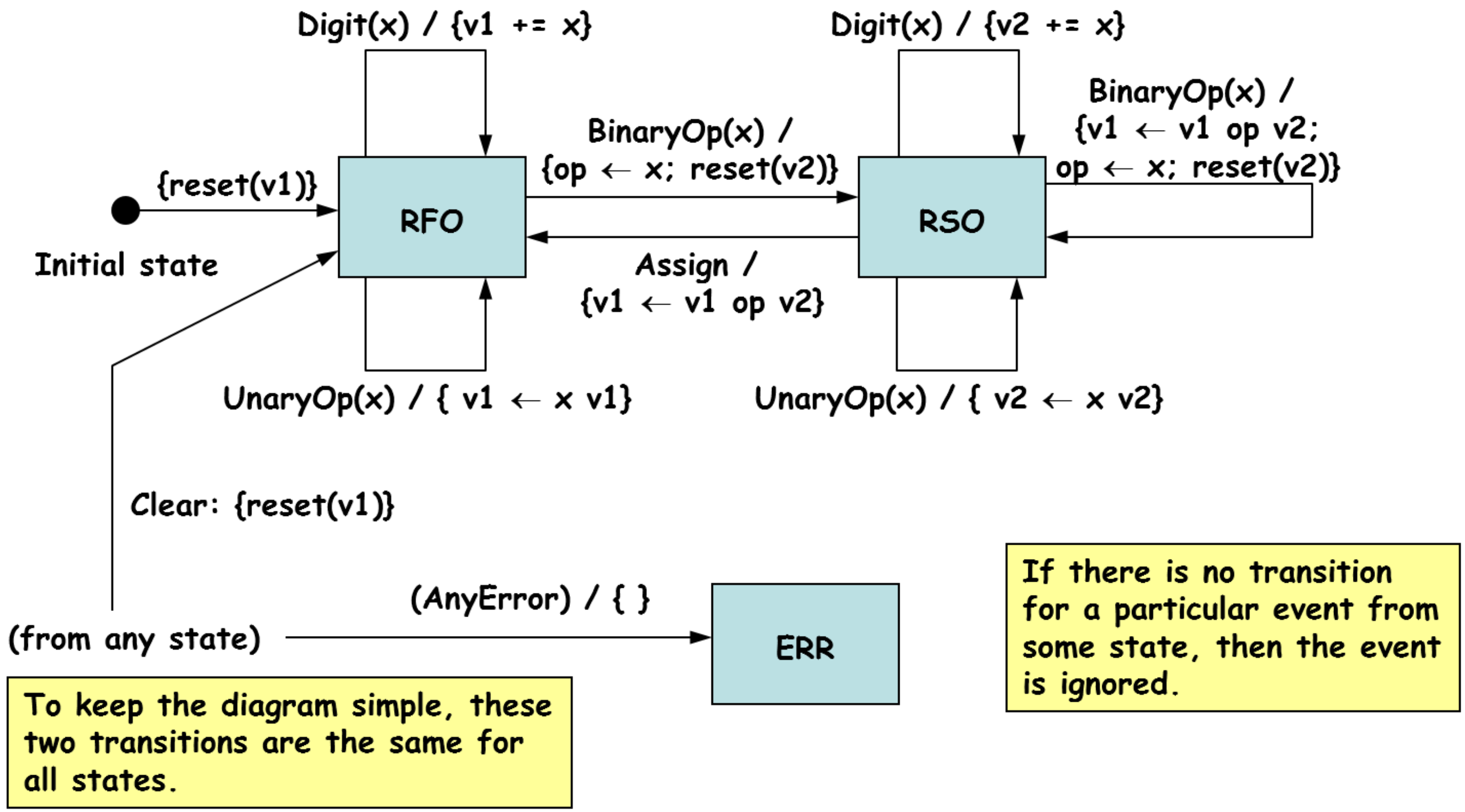
Input:	Context:	Action:	Display:
(init)	RFO	`reset(v1)`	0
3	RFO	`v1 += "3"`	3
4	RFO	`v1 += "4"`	34
+/-	RFO	`v1 ← procUnary: "34", "+/-"`	-34
+	RSO	`op ← "+"; reset(v2)`	-34
5	RSO	`v2 += "5"`	5
6	RSO	`v2 += "6"`	56
*	RSO	`v1 ← procBinary: "-34", "+", "56"`	22
		`reset(v2)`	
2	RSO	`v2 += "2"`	2
1/x	RSO	`v2 ← procUnary: "2", "1/x"`	0.5
=	RFO	`v1 ← procBinary: "22", "*", "0.5"`	11

State-Transition Diagram

- How does the calculator know what operation to perform with each event? This is based on its state, or context (RFO, RSO, ERR).
- We can describe the behavior using a **state-transition diagram**.
  - **Nodes**: represent possible **states** the system can be in. A black circle is the **initial or starting state**.
  - **Arcs** or **Edges**: represent possible **transitions**. Each is labeled with a pair "**Event/Action**" where:
    - **Event**: event that triggers the transition.
    - **Action**: action/computation performed as a result of the event.



(Simplified) State-Transition Diagram



Programming State-Transition Diagrams

- You can use **if-the-else** and/or **switch** statements to control the processing.
- **Example**:

```
if ( event == X ) {      // some event X encountered
  switch ( state ) {
  case STATE1:
    // processing for event X in state 1
    break;
  case STATE2:
    // processing for event X in state 2
    break;
  }
}  else if ( event == Y ) {        // event Y encountered
  // same thing
} // etc…
```

- You can use the **state design pattern**