Finite State Machines

Dr. Isaac Griffith Idaho State University

Outcomes

Idaho State University | Computer Science

After today's lecture you will be able to:

- Understand Finite State Machine basics and how to use them to model a system.
- Understand when to select between Use Cases and State Machines for system modeling

Finite State Machines

CS 2263

Introduction

- Often times a use case based approach will not suffice
  - Typical instances of this occur when we are dealing with devices or systems which interact with external entities
  - Examples include:
    - Complex interactive user interfaces
    - Device controllers (i.e., microwave controller)
- We will look at two cases across this week
  - A microwave controller where the behavior of the microwave is based on a response to a user's actions and the current state of the microwave
  - A UI which must present different screens at different stages of the interaction.
    - Which screen is shown is dependent on the kind of input the application is requesting at that instant, and the UI changes as a response to that input.
- Let's start with a Simple Example

Simple Example

- We will consider a microwave controller that is governed by the following rules
  - It has a door, a light, a power-tube, a button, a timer, and a display
  - When the oven is not in use and the door is closed
    - the light and the power tube are off and the display is blank
  - When the door is open
    - the light stays on
  - If the button is pushed when the door is closed and the oven is not operating
    - the oven is activated for one minute
  - When the oven is activated,
    - the light and the power-tube are turned on
  - If the button is pushed when the oven is operating,
    - one minute is added to the timer
  - If the door is opened when the oven is operating
    - the power-tube is turned off
  - When the cooking time is completed
    - the power-tube and light are turned off
  - Pushing the button when the door is open has no effect

Simple Example

- Attempting to model this system with use cases, runs into some difficulties
- Consider the following scenarios:
  1. open door -> place food in oven -> close door -> push button -> wait for cooking to finish -> open door -> remove food -> close door
  2. open door -> place food in oven -> close door -> push button -> wait for cooking to finish -> open door -> remove food and stir -> place food in oven -> close door -> push button -> wait for cooking to finish -> open door -> remove food -> close door
- This represents a **continual sequence of events** for which there is no standard process that can characterize it.
- Furthermore, it should be easy to see that events are state dependent.
- Luckily, we have a means by which we can model this type of system -> **Finite State Machines (FSM)**

Finite State Modeling

- A FSM is defined by:
  - A set of states
  - A set of input symbols
  - A set of transitions
- Each transition is defined by a 4-tuple (s_i, s_f, I, O), where:
  - s_i is the initial state
  - s_f is the final state
  - I is the input that triggers the transition
  - O is the associated output (if any)

Microwave as an FSM

- An initial examination of the microwave yields the following possible states:
  1. Microwave is idle and the door is closed
  2. Microwave is idle and the door is open
  3. Microwave is in operation
  4. Microwave is interrupted by the door being opened
  5. Microwave has completed cooking

- We have the following events which cause the microwave to change state:
  - door is opened (external)
  - door is closed (external)
  - button is pushed (external)
  - clock ticks (internal)
  - timer runs out (internal)

Microwave as an FSM

Idaho State University | Computer Science

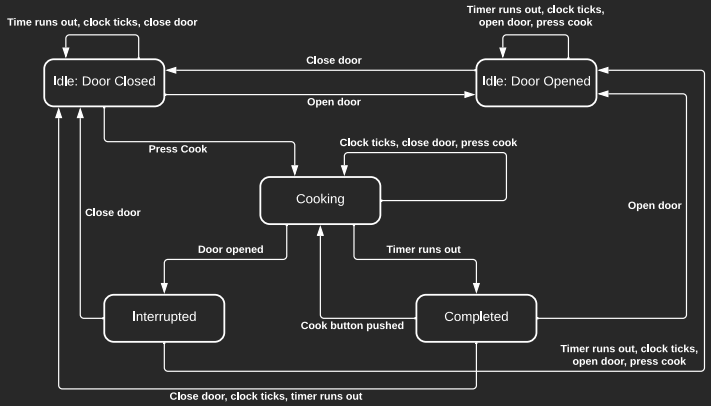
- This analysis leads to the following table, which describes all the actions that correspond to each (state, event) pair

	Open door	Close door	Press cook	Clock ticks	Timer runs out
Idle; Door closed	Idle; Door open	idle; Door closed	Cooking	Idle; Door closed	Idle; Door closed
Idle; Door open	Idle; Door open	Idle; Door closed	Idle; Door open	Idle; Door open	Idle; Door open
Cooking	Interrupted	Cooking	Cooking	Cooking	Idle; Door closed
Interrupted	Idle; Door open	Idle; Door closed	Idle; Door open	Idle; Door open	Idle; Door open
Completed	Idle; Door open	Idle; Door closed	Cooking	Idle; Door closed	Idle; Door closed

- This table can be used as follows
  - When the microwave is in the Cooking state (row 4, column 1) if the door is opened (row 1, column 2), the cell formed by row 4 and column 2 shows that the microwave enters the `Interrupted` state

Microwave as an FSM

- Using the table from the previous slide, we can construct the following UML State Transition Diagram:



Microwave as an FSM

Idaho State University | Computer Science

- If we review the prior table we can note that the behavior for states `Idle; Door Closed` and `Completed` are identical
  - We can do not need different states to distinguish between `Idle; Door Closed` and `Completed`
- Additionally, `Door Open` and `Interrupted` are indistinguishable as well, and thus we can simply combine these into `Door Open`

	Open door	Close door	Press cook	Clock ticks	Timer runs out
Idle; Door closed	Idle; Door open	idle; Door closed	Cooking	Idle; Door closed	Idle; Door closed
Idle; Door open	Idle; Door open	Idle; Door closed	Idle; Door open	Idle; Door open	Idle; Door open
Cooking	Interrupted	Cooking	Cooking	Cooking	Idle; Door closed
Interrupted	Idle; Door open	Idle; Door closed	Idle; Door open	Idle; Door open	Idle; Door open
Completed	Idle; Door open	Idle; Door closed	Cooking	Idle; Door closed	Idle; Door closed

Microwave as an FSM

- Thus from these changes we arrive at the following minimized transition table

	Open door	Close door	Press Cook	Clock ticks	Timer runs out
Door closed	Door open	Door closed	Cooking	Door closed	Door closed
Door open	Door open	Door closed	Door open	Door open	Door open
Cooking	Door open	Cooking	Cooking	Cooking	Door closed

Use Cases vs. FSM

Idaho State University | Computer Science

Q: Under what conditions should we use FSMs, and under what conditions do we employ use cases?

- If we consider the Library system.
  - We can note that at the start of each use case a set of preconditions hold.
  - The final output of a given use-case then depends upon which set of preconditions were true
  - Additionally, for any one transaction with the system there is one “most-common” outcome and other secondary outcomes
  - If we modeled this using an FSM it would be exceptionally complex with an unmanageable and possibly unbounded set of states
    - but when we model how a user interacts with this system, we have a finitely bounded set of interactions
  - Thus, use-case modeling is appropriate and FSMs are inappropriate

Use Cases vs. FSM

Idaho State University | Computer Science

Q: Under what conditions should we use FSMs, and under what conditions do we employ use cases?

- If we consider the Microwave example.
  - We note that we have a possibly unbounded set of ways in which the user can interact with the system.
  - But from the specification, we can see that we are only interested in how the system responds to input.
  - Furthermore, the nature of the response is driven by the current state of the system.
  - Finally, we also know that we have a finite (and small) set of states that the system may be in at any time.
  - Thus, use-case modeling is inappropriate but FSMs are appropriate

First Solution

CS 2263

Completing the Analysis

Idaho State University | Computer Science

- We now need to identify the conceptual classes.
- Just as before we start by identifying the nouns:
  - `Microwave`
  - `Powertube`
  - `Light`
  - `Display`
  - `Door`
  - `CookButton`
  - etc.
- Both `Display` and `CookButton` will be part of the GUI
- And, since this is a simulation, there is no real `Powertube`, `Light`, or `Door` so we will model these by displaying a message
- Additionally, we will use a `Clock` to handle tracking cooking time, counting “ticks” every second

The Components

- **Microwave**
  - Tracks the state of the oven and turns on/off the power-tube and light.
  - Events: Open/closing door, pushing button, timer runs out
- **GUI Display**
  - Has components for user input and displays information to simulate operation
  - There are 4 displays:
    - Cooking indicator
    - Door indicator
    - Time indicator (shows 0 if idle)
    - Light indicator
- **Clock**
  - generates clock tick at regular 1 second intervals

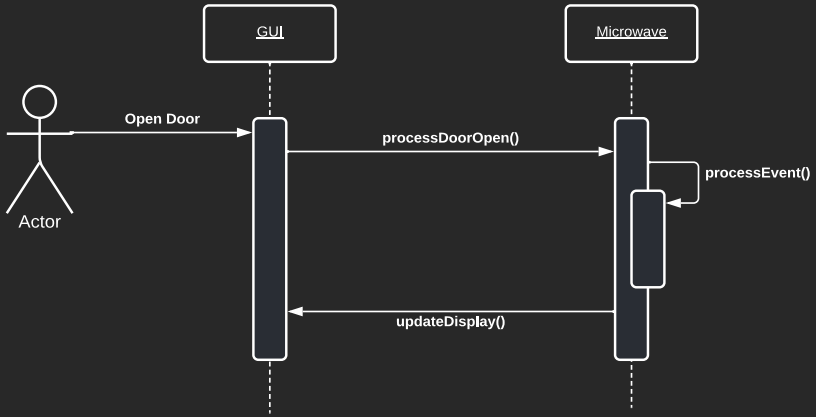
Designing the System

- Using the conceptual classes, our next task is to determine how to distribute responsibilities across the system
  - That is how the events will be processed

- There are two kinds of events to process:
  - **User inputs**: these are recognized by the GUI
  - **Clock ticks**: these originate in `Clock`

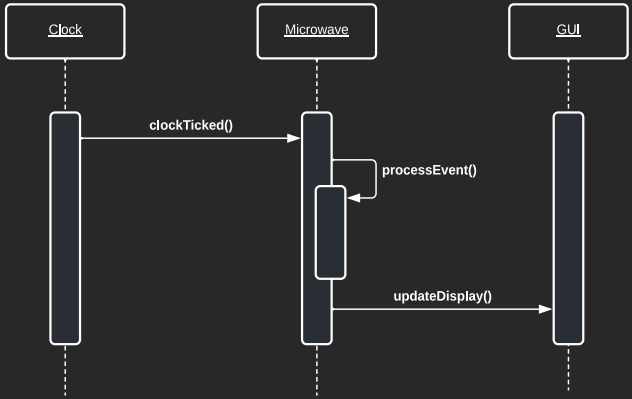
- To specify this we will use sequence diagrams

Opening a Door



- Here we assert that there is a separate method in `Microwave` for each type of event

Clock Ticks



- Note this sequence is not initiated by an actor.

Remaining Events

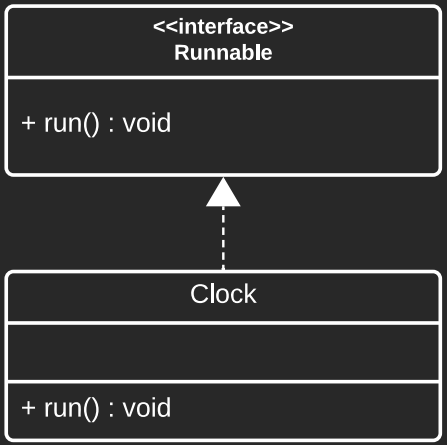
- The remaining events operate similar to Opening a door
- We assume that the `Microwave` class has methods to handle each of the events
- We also assume that the `GUI` has appropriate methods to update the displays which will be invoked with the appropriate parameters
  - The `GUI` will be required to provide two types of methods
    - Methods that handle user input
    - Methods that are invoked by `Microwave` to update the display
- Additionally, `Microwave` is to be a singleton
  - To mimic an FSM, it will have a variable `currentState` which tracks the state the microwave is in

Implementation Classes

CS 2263

Clock Class

- This is the simplest



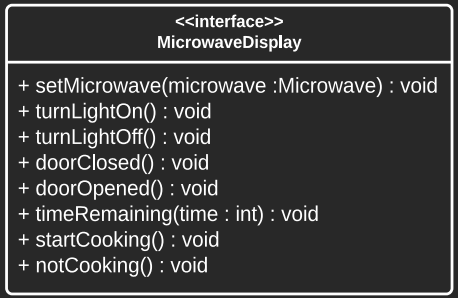

```
public class Clock implements Runnable {
  private static Microwave microwave;
  public Clock() {
    microwave = Microwave.instance();
    new Thread(this).start();
  }

  public void run() {
    try {
      while (true) {
        Thread.sleep(1000);
        microwave.clockTicked();
      }
    } catch (InterruptedException e) {}
  }
}
```

Display Class

- `GUIDisplay` implements `MicrowaveDisplay`
- To handle input, it creates a `JFrame` with a `JButton` for each kind of operation
  - open door
  - close door
  - cook
- When run it appears as shown:

Display Class

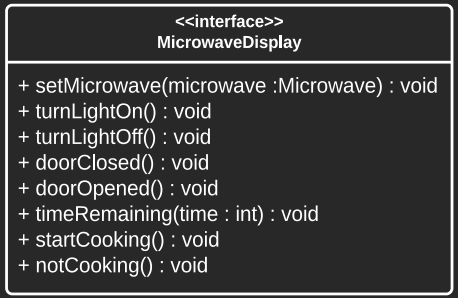
Idaho State University | Computer Science




```
public class GUIDisplay extends JFrame
      implements ActionListener, MicrowaveDisplay {
  private Microwave microwave;
  private JButton doorCloser =
            new JButton("close door");
  private JButton doorOpener =
            new JButton("open door");
  private JButton cookButton =
            new JButton("cook");
  private JLabel doorStatus =
            new JLabel("Door Closed");
  private JLabel timerValue =
            new JLabel("            ");
  private JLabel lightStatus =
            new JLabel("Light Off");
  private JLabel cookingStatus =
            new JLabel("Not cooking");
```

Display Class

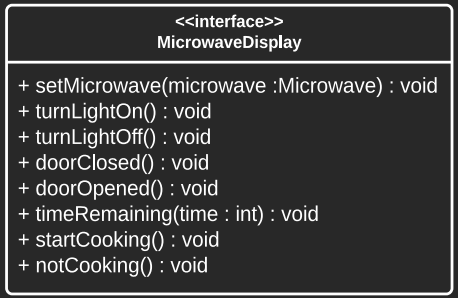
Idaho State University | Computer Science




```
public GUIDisplay() {
  super("Microwave");
  addWindowListener(new WindowAdapter()) {
    public void windowClosing(WindowEvent event) {
      System.exit(0);
    }
  }
  getContentPane().setLayout(new FlowLayout());
  Container c = getContentPane();
  c.add(doorStatus);
  c.add(lightStatus);
  c.add(timerValue);
  c.add(cookingStatus);
  c.add(doorCloser);
  c.add(doorOpener);
  c.add(cookButton);
  doorCloser.addActionListener(this);
  doorOpener.addActionListener(this);
  cookButton.addActionListener(this);
  pack();
  setVisible(true);
}
```

Display Class

Idaho State University | Computer Science



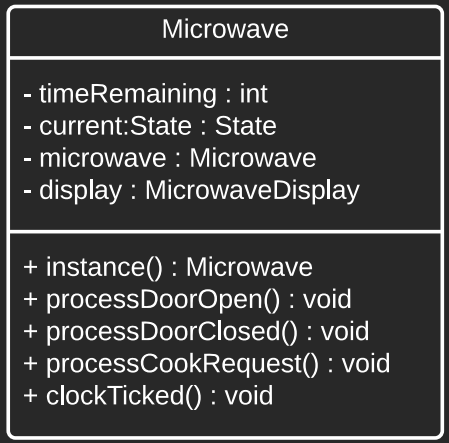

```
  public void actionPerformed(ActionEvent event) {
    if (event.getSource().equals(doorCloser))
      microwave.processDoorClosed();
    else if (event.getSource().equals(doorOpener))
      microwave.processDoorOpen();
    else if (event.getSource().equals(cookButton))
      microwave.processCookRequest();
  }

  public void setMicrowave(Microwave microwave) {
    this.microwave = microwave;
  }

  public void turnLightOn() {
    lightStatus.setText("Light On");
  }

  // other methods
}
```

Microwave Class

Idaho State University | Computer Science

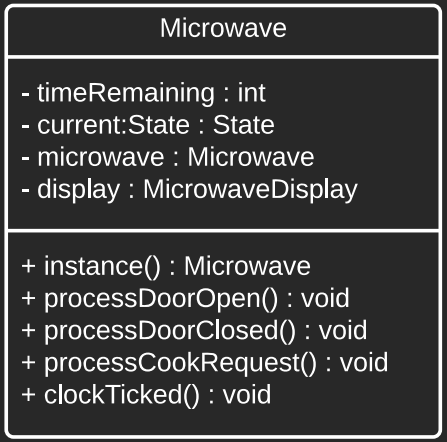
- This class maintains the variables for tracking remaining cooking time and current state
- This is also where an instance of `MicrowaveDisplay` is constructed
- We will also place the `main(...)` method here

Microwave Class



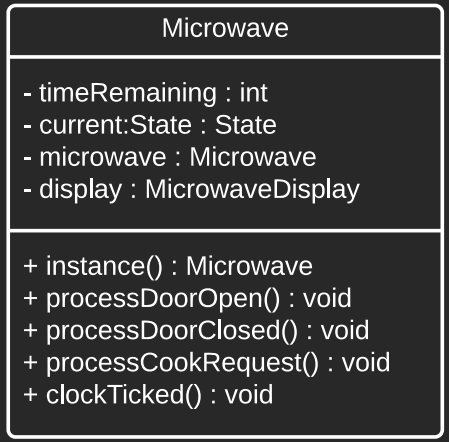

```
public class Microwave {
  public enum States { DOOR_CLOSED_STATE, DOOR_OPENED_STATE, COOKING_S
  private int timeRemaining;
  private States currentState;
  private static Microwave instance;
  private MicrowaveDisplay display;

  private Microwave() {
    currentState = States.DOOR_CLOSED_STATE;
    timeRemaing = 0;
    display = new GUIDisplay();
    display.setMicrowave(this);
    display.timeRemaining(timeRemaining);
    display.turnLightOff();
    display.doorClosed();
    display.notCooking();
  }
```

Microwave Class




```
public static Microwave instance() {
  if (instance == null) {
    return instance = new Microwave();
  }

  return instance;
}

public void clockTicked() {
  if (currentState == States.COOKING_STATE) {
    timeRemaining--;
    display.timeRemaining(timeRemaining);
    if (timeRemaining == 0) {
      currentState = States.DOOR_CLOSED_STATE;
      display.notCooking();
      display.turnLightOff();
    }
  }
}
```

Microwave Class




```
  public void processCookRequest() {
    if (currentState == States.DOOR_CLOSED_STATE) {
      currentState = States.COOKING_STATE;
      display.startCooking();
      display.turnLightOn();
      timeRemaining = 60;
      display.timeRemaining(timeRemaining);
    } else if (currentState == States.COOKING_STATE) {
      timeRemaining += 60;
      display.timeRemaining(timeRemaining);
    }
  }

  public void processDoorOpen() {}

  public void processDoorClosed() {}

  public static void main(String[] args) {
    new Clock();
  }
}
```

Critique of the Design

- So what have we done so far?
  - We were presented with a problem, for which we presented an object-oriented solution
- What is the next task?
  - We need to critically examine our design
- What will we find?
  - That there are two primary flaws
    1. Extreme Complexity in `Microwave`
    2. Issues in the Communication Between Objects

`Microwave` Complexity

Idaho State University | Computer Science

- `Microwave` is a large class which handle all of the states and events
- Although at this point it does not seem too complex
- If we extrapolate this approach to a larger system, we can begin to see the problem
  - The conditionals for each method (selecting on the states) add in the unwanted complexity
  - To address this in previous lectures, we constructed an inheritance hierarchy and extracted out the branches into their own subclasses
  - This allowed us to facilitate reuse and to work in a true OO way
- We can so something similar, but it will take a bit more work than the more straight forward approach taken earlier
  - This will be facilitated by the **State Pattern**

Object Communication

- Our design has two primary communication contexts
  - Events specific to the microwave (i.e., close door button)
  - Events general to any application (i.e., click ticks)

- For both types, `Microwave` is the interested listener
  - The GUI catches events and forwards them to `Microwave`
  - This involves deep coupling between `Microwave` and the `GUI`, which hurts reuse
  - Furthermore, as the system evolves we will need to add new events, thus leading to a violation of OCP

Object Communication

- The clock poses another issue
  - Currently it is tightly coupled to the `Microwave` class, and even starts off the entire system
  - Yet, this seems to be a concept that any time-dependent system could possibly want to use
  - Thus, we would expect that we could instantiate a clock whenever it is needed

- Both of these issues stem from how we process events.
  - That is we process the events at the **event generator**
  - But to better facilitate reuse, we should process the events at the **event listener**

Object Communication

- Our approach is poor for the following reasons:
  1. If the generator handles events, it must also manage the listeners, which makes the generator vulnerable to changes in the listener classes
  2. Responsibility current rests with the generator to register the listeners, but this implies that the generator "knows" about all interested listeners.
     - We would rather move this responsibility to the listener, allows any interested party to register as they see fit.
  3. The set of listeners cannot change dynamically
     - We do not have the flexibility to change listeners, but if the listeners could register/de-register themselves this would be solved

- All of this is due to having **tightly coupled communication** between the generator and the listeners.

- To get around this we have two standard approaches:
  - the **Observer Pattern**
  - **Event-Driven Communication** (also sometimes called reactive programming)

For Next Time

- Review Chapter 10.1 - 10.4
- Review this lecture
- Read Chapter 10.5 - 10.7
- Watch Lecture 28

Are there any questions?