UI and the State Pattern

Dr. Isaac Griffith Idaho State University

Outcomes

After today's lecture you will be able to:

- How to apply the state pattern to UIs

UIs & State Machines

CS 2263

Eliminating Conditionals

- If you recall from the prior lecture, we shifted the event processing in the state to use the `handle` method.
  - In this method a conditional switches on the type of event based on external inputs

- In order to remove the conditional, we would need to separate out how the event is delivered to the state.
  - i.e., calling a unique method for each event
  - Unfortunately, since there is only one type of context, we are unable to handle this in the context
  - Thus, we need another location in which to process the communication

Eliminating Conditionals

- There are two options for ensuring that only the current state processes an event
  1. Remove the context completely, and make the states fully responsible for system behavior
     - Could use the existing Java event handling system (similar to observer pattern)
     - Listeners would implement appropriate interface
     - Listeners would register to the event source
     - Have one listener per event
  2. Allow the context to act as a switchboard that connects the event to the current state
     - Requires not having full knowledge of the concrete type of state or event
     - Could use a parallel hierarchy of listener types
     - Use a 2 step process: (i) communicate event from source to context (ii) context invokes appropriate method of a listener on the current state

Using Java Events

- Here our listeners receive events directly from the source

- For each event type, there is a specific ‘manager’ (i.e., `DoorCloseManager`) which notifies all the listeners

- The Java Event system requires that we implement classes for:
  - Events
  - Event Sources
  - Listener Interfaces

The Event Classes

- We create these classes by extending `EventObject`
  - for which the only work is the definition of a constructor

- The following is an example for the `DoorCloseEvent`

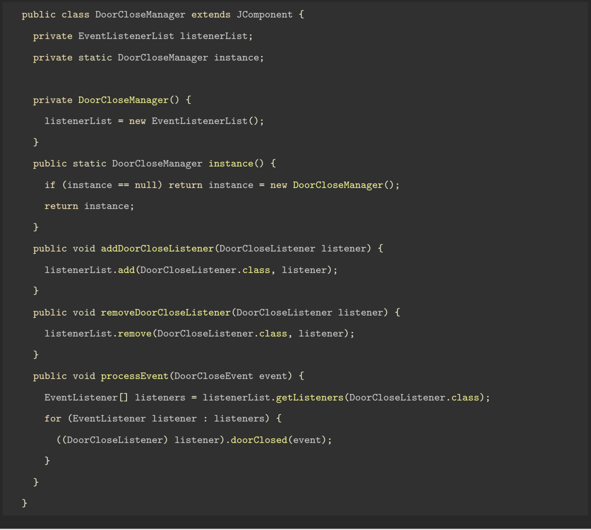
```java
public class DoorCloseEvent extends EventObject {

  public DoorCloseEvent(Object source) {
    super(source);
  }

}
```

The Event Sources

- In the `Microwave` system, our events are generated by button clicks.
- For our purposes, the `GUIDisplay` is the event source.
  - It should provide capabilities to register/de-register listeners
  - It should provide capability to notify listeners of an event
  - Doing this causes unnecessary entanglement of responsibilities
- Thus, we outsource the event management to a manager
  - For example `DoorCloseManager`



```java
public class DoorCloseManager extends JComponent {
  private EventListenerList listenerList;
  private static DoorCloseManager instance;

  private DoorCloseManager() {
    listenerList = new EventListenerList();
  }
  public static DoorCloseManager instance() {
    if (instance == null) return instance = new DoorCloseManager();
    return instance;
  }
  public void addDoorCloseListener(DoorCloseListener listener) {
    listenerList.add(DoorCloseListener.class, listener);
  }
  public void removeDoorCloseListener(DoorCloseListener listener) {
    listenerList.remove(DoorCloseListener.class, listener);
  }
  public void processEvent(DoorCloseEvent event) {
    EventListener[] listeners = listenerList.getListeners(DoorCloseListener.class);
    for (EventListener listener : listeners) {
      ((DoorCloseListener) listener).doorClosed(event);
    }
  }
}
```

Updating The GUI

- We then invoke the event constructors upon a button click:

```java
public void actionPerformed(ActionEvent evt) {
  if (evt.getSource().equals(doorCloser)) {
    DoorCloseManager.instance().processEvent(new DoorCloseEvent(this));
  }
  // code for other events
}
```

The Event Listeners

- All the listeners (i.e., `MicrowaveState` subclasses) must implement a corresponding event listener

- Each state then implements its required listeners (and does the housekeeping as well)
    - this includes registering and de-registering in the run method:

Listener Interface Example

```
public interface DoorCloseListener extends EventListener {
  void doorClosed(DoorCloseListener evt);
}
```

Listener Implementation Example

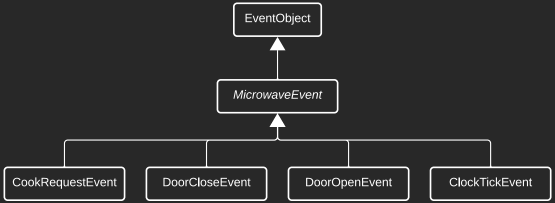
```
public class DoorOpenState extends MicrowaveState implements DoorCloseListener {
  // other fields and methods
  public void run() {
    display.stopCooking();
    display.openDoor();
    display.turnLightOn();
    display.displayTimeRemaining(context.getTimeRemaining());
    DoorCloseManager.instance().addDoorCloseListener(this);
  }

  public void doorClosed(DoorCloseEvent evt) {
    DoorCloseManager.instance().removeDoorCloseListener(this);
    context.changeCurrentState(DoorCloseState.instance());
  }
}
```

Using Context as a Switchboard

- A second approach, useful for FSMs where communication must go through a facade is an event structure which utilizes the context
  - The downside is that the context must be aware of all the event types :(
  - But, this has the following features
    - Context has a single `handleEvent` method
    - States take care of implementing listener methods they need

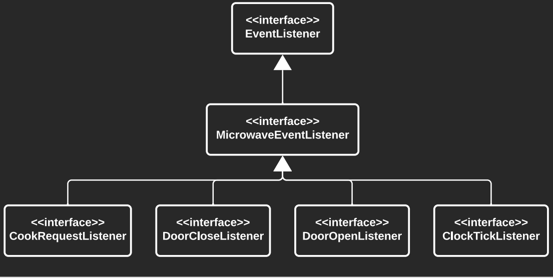
- This type of system has the following components
  - **Event Hierarchy:**
    - Abstract Base Event Class (`MicrowaveEvent`) which extends `EventObject`
    - Concrete classes extending the base class for each event



Using Context as a Switchboard

- A second approach, useful for FSMs where communication must go through a facade is an event structure which utilizes the context
  - The downside is that the context must be aware of all the event types :(
  - But, this has the following features
    - Context has a single `handleEvent` method
    - States take care of implementing listener methods they need

- This type of system has the following components
  - **Listener Hierarchy:**
    - Abstract Base Listener Interface (`MicrowaveEventListener`) which extends `EventListener` interface (used for `MicrowaveState`)
    - Specialized Listener interfaces for each event type

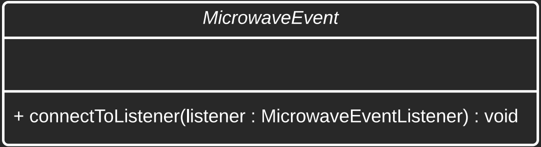


<<interface>> EventListener
<<interface>> MicrowaveEventListener
<<interface>> CookRequestListener
<<interface>> DoorCloseListener
<<interface>> DoorOpenListener
<<interface>> ClockTickListener

Using Context as a Switchboard

- A second approach, useful for FSMs where communication must go through a facade is an event structure which utilizes the context
  - The downside is that the context must be aware of all the event types :(
  - But, this has the following features
    - Context has a single `handleEvent` method
    - States take care of implementing listener methods they need

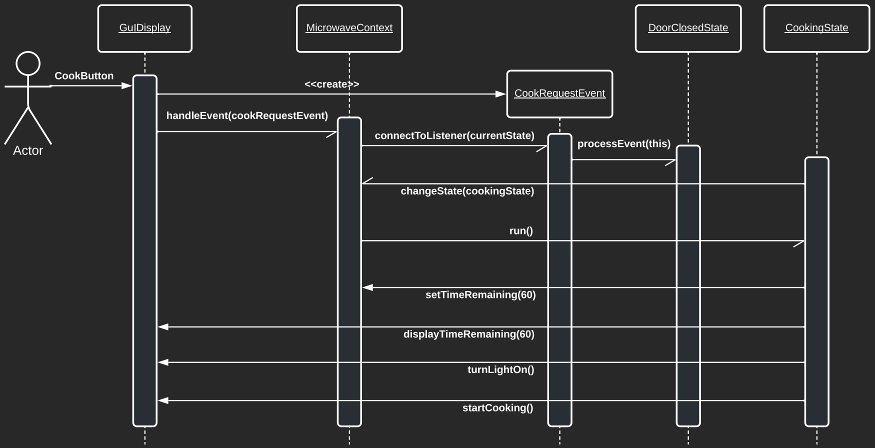
- This type of system has the following components
  - **Method to Connect Listeners:**
    - Abstract Event class (`MicrowaveEvent`) has `connectToListener` which subclass implement to connect listeners (called from the context)
  - **Method to notify listeners:**
    - found in the switchboard
  - **ConcreteListeners:**
    - The state classes that implement the listener interfaces



MicrowaveEvent

+ connectToListener(listener : MicrowaveEventListener) : void

Example Operation



Implementation

MicrowaveEvent

```java
import java.util.*;

public abstract class MicrowaveEvent extends EventObject {

  public MicrowaveEvent(Object object) {
    super(object);
  }

  public abstract void connectToListener(
                  MicrowaveEventListener listener);
}
```

Example: CookRequestEvent

```java
public class CookRequestEvent extends MicrowaveEvent {

  public CookRequestEvent(MicrowaveDisplay display) {
    super(display);
  }

  public void connectToListener(MicrowaveEventListener listener) {
    try {
      ((CookRequestListener) listener).processEvent(this);
    } catch (ClassCastException cce) {
      // message
    }
  }
}
```

Implementation

EventListener

```
public interface MicrowaveEventListener {
  void processEvent(MicrowaveEvent event);
}

public interface CookRequestListener extends
        MicrowaveStateListener {

  @Overrides
  void processEvent(CookRequestEvent event);
}
```

handleEvent method

```
public void handleEvent(MicrowaveEvent event) {
  try {
    event.connectToListener((MicrowaveEventListener) currentState);
  } catch (ClassCastException cce) {
    currentState.logException(cce);
  }
}
```

CookingState

```
import java.util.*;
public class CookingState extends MicrowaveState implements DoorOpenListener, CookRequestListener,
                                                            ClockTickListener {
  // other methods
  @Overrides
  public void run() {
    context.setTimeRemaining(60);
    display.displayTimeRemaining(context.getTimeRemaining());
    display.turnLightOn();
    display.startCooking();
  }
  @Overrides
  public void processEvent(DoorOpenEvent event) {
    context.changeState(DoorOpenState.instance());
  }
  @Overrides
  public void processEvent(CookRequestEvent event) {
    context.setTimeRemaining(context.getTimeRemaining() + 60);
    display.displayTimeRemaining(context.getTimeRemaining());
  }
}
```

Eliminating Conditionals

- Unfortunately, `GUIDisplay` still has conditional code, that we need to get rid of in the `actionPerformed` method

- It is typically too tedious (or even impossible) to remove all conditional process in UIs but the following is a general approach
  - Essentially, we will construct a Button hierarchy for each event generating button in our UI
  - We then implement the specific `handleEvent` in that button
  - We then reduce the `actionPerformed` method to a single line

Eliminating Conditionals

Our Abstract Button

```java
import javax.swing.*;

public abstract class GUIButton extends JButton {
  public GUIButton(String string) {
    super(string);
  }

  public abstract void inform(MicrowaveContext context,
                              MicrowaveDisplay display);
}
```

actionPerformed Update

```java
public void actionPerformed(ActionEvent event) {
  ((GUIButton) event.getSource())
        .inform(MicrowaveContext.instance(), this);
}
```

Concrete Button Example

```java
public class CookButton extends GUIButton {
  public CookButton(String string) {
    super(string);
  }

  public void inform(MicrowaveContext context, MicrowaveDisplay source) {
    context.handleEvent(new CookRequestEvent(source));
  }
}
```

State Pattern & GUI

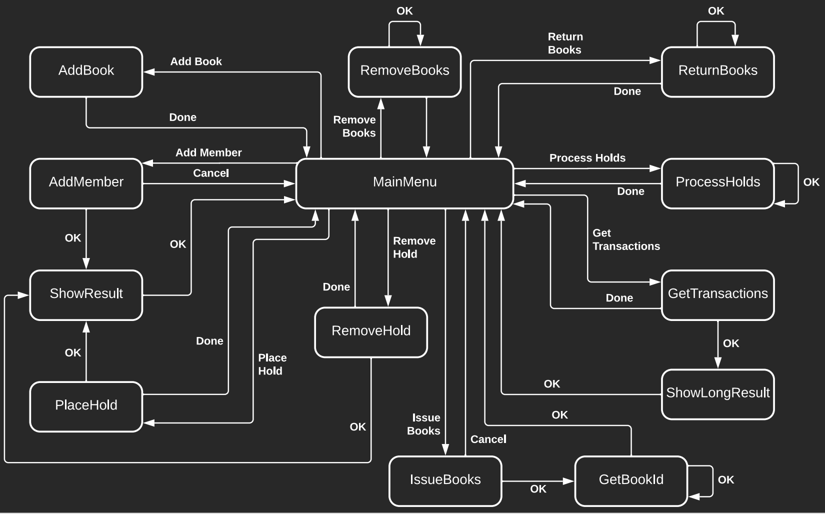
- Often we can think of the way a UI operates as a FSM
- Inputs can range from clicking a button, selecting an item from a list box, even the dragging and dropping of a file, etc.
- The UI responds by updating the current window, change to another window, displaying a popup message, etc.
  - Each of these changes reflect changes in the UI's state

State Pattern & GUI

- Throughout this course we discussed the Library example, which had a simple menu driven UI
  - Here we will discuss how to model this UI (not the requirements) as a state machine which includes:
    - A main menu
    - Adding Books
    - Adding members
    - Issuing books
    - Returning and removing books
    - Placing and removing holds
    - Printing transactions
    - Processing holds

The Library State Diagram

Idaho State University | Computer Science



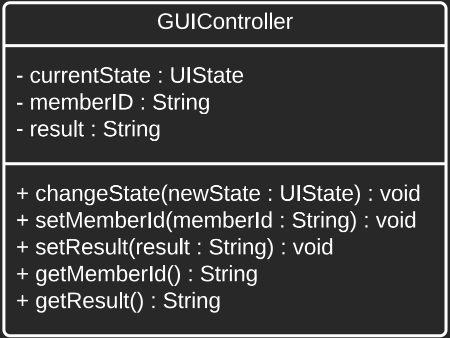
ROAR

The Context

- Each of the states from the prior diagram will implement the `UIState` interface:

```
public interface UIState {
  void handle(Object event);
  void run();
}
```

- Which operates similar to the `MicrowaveState`
- The main menu will be presented as a Dialog (using `JDialog` which extends `JFrame`)
- Finally, the context of the state pattern used here is the `GUIController` class
  - Data shared between states (i.e., `memberId`'s and `bookId`'s are transferred from a state to the Context (via getters and setters) for later access by following states)



GUIController

- currentState : UIState
- memberID : String
- result : String

+ changeState(newState : UIState) : void
+ setMemberId(memberId : String) : void
+ setResult(result : String) : void
+ getMemberId() : String
+ getResult() : String

For Next Time

- Review Chapter 10.8 - 10.10
- Review this lecture
- Read Chapter 11.1 - 11.4
- Watch Lecture 30

Are there any questions?