State and Observer Patterns

Dr. Isaac Griffith Idaho State University

Outcomes

After today's lecture you will be able to:

- Describe and use the State Pattern
- Describe and use the Observer Pattern

The State Pattern

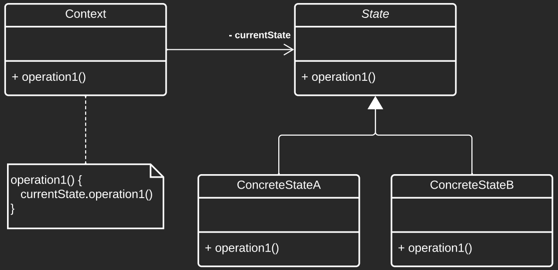
CS 2263

The State Pattern

- **Pattern Intent**
  - We wish to alter an objects behavior when its internal state changes.
  - Thus, allowing the object to appear as if it has changed its class
- **Problem it Solves**
  - We have a system or component, wherein, at any given moment, there is a *finite* number of *states* which the program may be in.
  - While in any unique state, the system/component behaves differently, and the system/component can be switched from one state to another instantaneously.
  - Depending on the current state, the system/component may or may not switch to other states based on an event and a set of rules
  - These rules are called transitions, and they are both *finite* and *predetermined*
- Thus we are solving the problem of modeling a finite state machine.

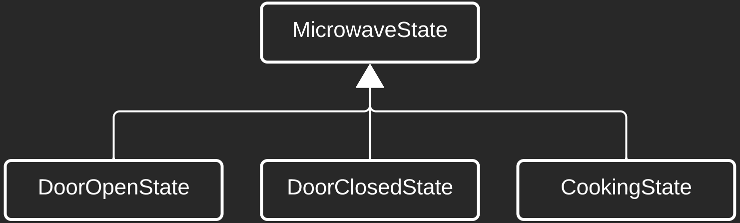
Structure

- The following is the structure of the State Pattern
  - A collection of states, with each state being defined by distinct behavior
  - A set of external inputs to which the system responds
  - A **context** in which the FSM operates
    - needed to provide continuity to the system
    - serves a a facade for the entire system
    - provides the mechanism to transition
    - tracks external entities which need to be notified of internal changes
  - The context has the following attributes
    - A field to tract the current state
    - Mechanism which records the change of state



Creating the Hierarchy

- Due to the fact that we have only a single `Microwave`, it does not make sense to handle the issues identified in the previous lecture as we did with the Library.
- Instead, we want to split the class into two sections
  - The context -> for this we rename `Microwave` to `MicrowaveContext`
  - The state specific components -> We will have one subclass per state



Transitions

- Transition between states implies knowledge of the other states
- Our prior lecture we noted two approaches for describing such transitions
  - Transition table
  - State Transition Diagram
- These correspond to the two approaches for implementation
  - Adjacency Matrices
  - Adjacency Lists
- Both of these approaches are used in constructing a graph data structure

Transitions

Idaho State University | Computer Science

Adjacency Matrix Representation

- We simply provide numeric values for each of the states
- We then construct a matrix
- From our example, we map 0 -> `Open Door`, 1 -> `Close Door`, 2 -> `Press Cook`, 3 -> `Clock Ticks`, 4 -> `Timer Runs Out`
- We can then store the transition table as follows:

```
int[][] transitions = {{1,0,2,0,0},
                       {1,0,1,1,1},
                       {1,2,2,2,0}}
```

- We would then need some method in `Context` to transition

```
public void changeState(int next) {
  currentState = transitions[currentState][next];
  state[currentState].run();
}
```

Adjacency List Representation

- In this approach each state provides a direct reference to the next state
- Each concrete state is a singleton, thus we use the `instance()` method for the ref
- `Context` contains the following transitioning method:

```
public void changeState(MicrowaveState state) {
  currentState = state;
  currentState.run();
}
```

State Classes

MicrowaveState

- + processDoorClose() : void
- + processDoorOpen() : void
- + processCookRequest() : void
- + processClockTick() : void
- + *run() : void*

DoorOpenState

- + processDoorClose() : void
- + run() : void

DoorClosedState

- + processDoorOpen() : void
- + processCookRequest() : void
- + run() : void

CookingState

- + processDoorOpen() : void
- + processCookRequest() : void
- + processClockTick() : void
- + run() : void

MicrowaveContext

Idaho State University | Computer Science

MicrowaveContext

- instance : MicrowaveContext
- timeRemaining: int
- currentState : MicrowaveState
- display : MicrowaveDisplay

```
+ instance() : MicrowaveContext
+ processDoorOpen() : void
+ processDoorClose() : void
+ processCookRequest() : void
+ processClockTick() : void
+ changeState(state : MicrowaveState) : void
+ setTimeRemaining(timeRemaining : int) : void
+ getTimeRemaining() : int
+ getDisplay() : MicrowaveDisplay
```

Implementing State Pattern

MicrowaveState

- We implement this as an abstract class
- Each of its event processing methods are provided a default empty implementation
  - It is expected that these will be overridden by the subclasses (if needed)
- `run()` is abstract and is meant to be invoked when control is transferred to the state.

```java
public abstract class MicrowaveState {
  protected static MicrowaveContext context;
  protected static MicrowaveDisplay display;

  protected MicrowaveState() {
    context = MicrowaveContext.instance();
    display = context.getDisplay();
  }

  public abstract void run();

  public void processDoorClose() {}

  public void processCookRequest() {}

  public void processClockTick() {}
}
```

CookingState


```java
public class CookingState extends MicrowaveState {

  private static CookingState instance;

  private CookingState() { super(); }

  public static MicrowaveState instance() {
    if (instance != null)
      instance = new CookingState();
    return instance;
  }

  public void run() {
    display.turnLightOn();
    context.setTimeRemaining(60);
    display.startCooking();
    display.displayTimeRemaining(
            context.getTimeRemaining());
  }

  public void processClockTick() {
    context.setTimeRemaining(
            context.getTimeRemaining() - 1);
    display.displayTimeRemaining(
            context.getTimeRemaining());
    if (context.getTimeRemaining() == 0) {
      display.notCooking();
      display.turnLightOff();
      context.changeState(
            DoorClosedState.instance());
    }
  }
}
```

Other Updates

- The idea is to move the code from the original `Microwave` implementation into the states
  - What we don't need is the conditionals based on the current state
- The new `MicrowaveContext` class still has the event processing methods
  - They now utilize dynamic binding to simply delegate processing to the `currentState` object
- The `Clock` class can remain mostly unchanged, with exception of updating the code to point to `MicrowaveContext` rather than `Microwave`

State Pattern Features

The State Pattern provides the following features:

- Allows an application to be in one of many states, and for the behavior to depend upon the state of the application
- Each state is represented by a single class, but common functionality across the state's may be placed in an abstract base class
- One instance of each state is created
  - We can utilize a Singleton here in order to avoid unnecessary object creation and deletion
- The Context orchestrates the entire operation and remembers the current state and any shared data
- Only one state is ever active at any time
  - The context delegates the input event to the state that is active
  - Thus only one state ever responds to events
- When the result of an event is a change in state, we can determine the next state to become active. This can be done in one of two ways:
  1. Using a centralized controller containing a transition table (matrix)
  2. The current state is used to determine the next state

State Pattern Advantages

The State Pattern has the following advantages:

1. There is no longer a need to switch on the state in order to decide what action needs to be taken.
   - Using the state pattern we polymorphically choose a method to be executed.
2. New states can be added and old state reused without changing the implementation
3. The code is more cohesive
   - Each state contains code relevant to it and nothing else.
   - Only events that are of interest to the state are processed.

The Observer Pattern

CS 2263

Communication Between Objects

Idaho State University | Computer Science

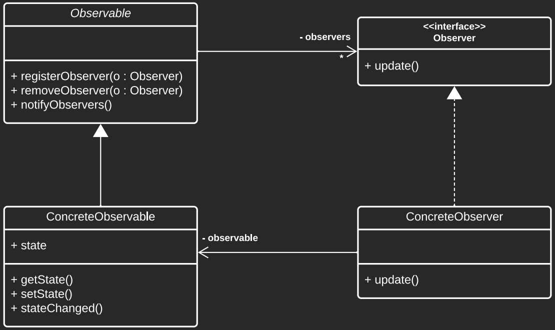
- In the prior lecture we noted two issues with communication in the design
  - The first of which was `Clock` was tightly coupled to `Microwave`
- To handle this particular issue we will look at the **Observer Pattern** which provides for loosely coupled communication

Loosely Coupled Communication

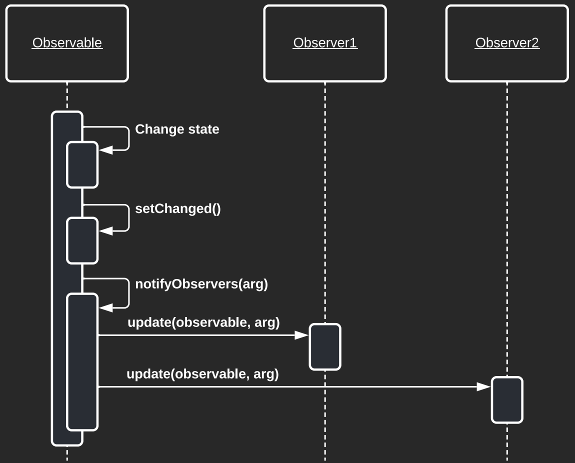
- For loosely coupled communication we must have the following properties
  - The listener is responsible for registering interest
  - All interested listeners share a common interface
    - Preventing the sender from needing to distinguish between listeners
  - Sender has a mechanism for maintaining a collection of listeners
- This is essentially the entire intent of the **Observer Pattern**

Observer Structure

- `Observable` (or `Subject`): a single object, which maintains the list of 'interested observers'
  - Contains the following types of methods
    - Methods to maintain the list of observers (`registerObserver`, `removeObserver`)
    - Methods which inform observers that a change occurred (`notifyObserver`)
    - Methods that provide other miscellaneous information (i.e., `countObservers`)
- `Observers`: provides an interface to be notified when an event occurs. Allows for many different observes without the `Observable` requiring specific knowledge of the different types.
  - provides method `update`



Observer Behavior



Clock Redesign

Changes to Clock

```
public class Clock extends Observable
      implements Runnable {

  private Thread thread = new Thread(this);
  private static Clock instance;

  public enum Events {CLOCK_TICKED_EVENT};

  private Clock() {
    thread.start();
  }

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

```
  public void run() {
    try {
      while (true) {
        Thread.sleep(1000);
        setChanged();
        notifyObservers(Events.CLOCK_TICKED_EVENT);
      } catch (InterruptedException ie) {}
    }
  }
}
```

Changes to MicrowaveContext

```
public class MicrowaveContext implements Observer {
  public void update(Observable source,
                            Object event) {
    // code to process clock tick
  }
  // other attributes as before
}
```


ROAR

Communication with User

Idaho State University | Computer Science

- The second issue with communication was providing better handling of user events
- In the current version, we have a separate `processSomething` methods for each event in `MicrowaveContext`.
  - What we want is to have a single method, `processEvent` or `handleEvent` which passes the event type as an argument.

- So this also means we need an Event type:

```
public enum Events {
  DOOR_CLOSED_EVENT,
  DOOR_OPENED_EVENT,
  COOKING_REQUESTED_EVENT
}
```

- Our `MicrowaveContext` handler is then:

```
public void handleEvent(Object arg) {
  currentState.handle(arg);
}
```

- Of course this implies that we need a `handle(event)` method in our `MicrowaveState` which will need to be overridden by each concrete state

```
public void handle(Object event) {
  if (event.equals(Events.COOKING_REQUESTED_EVENT))
    processCookRequest();
  else if (event.equals(Events.DOOR_OPENED_EVENT))
    processDoorOpen();
  else fi (event.equals(Events.DOOR_CLOSED_EVENT))
    processDoorClose();
}
```

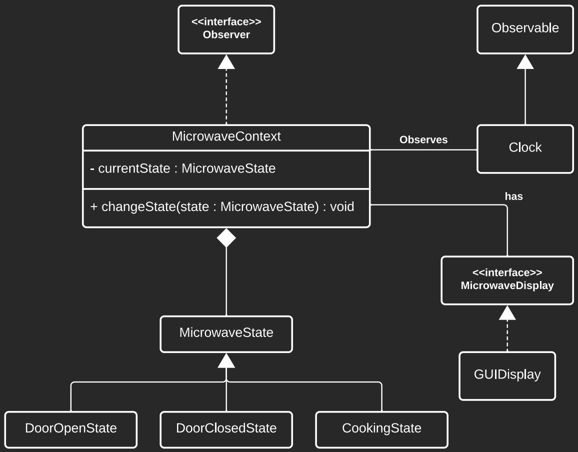
- But now we are switching on event type!

Improved Design

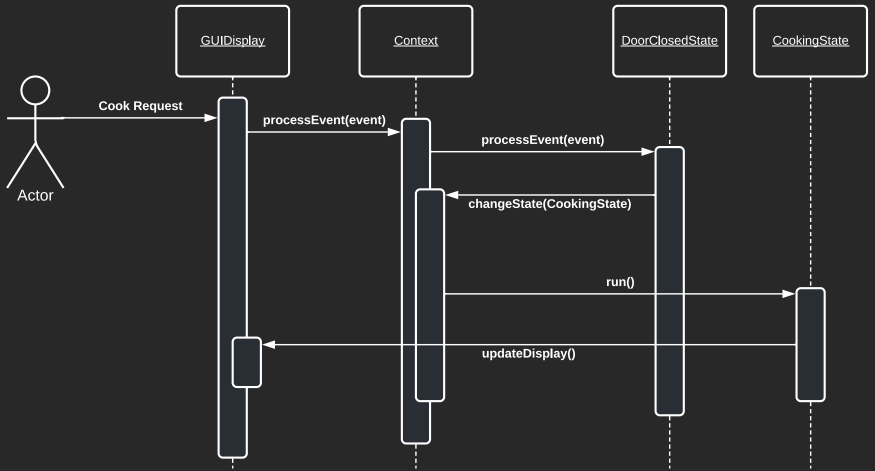
- We will also need to update the `GUIDisplay`

```
public void actionPerformed(ActionEvent event) {
  if (event.getSource().equals(frame.doorCloser)) {
    MicrowaveContext.instance().handleEvent(
          Events.DOOR_CLOSED_EVENT);
  } else if (event.getSource().equals(frame.doorOpener)) {
    MicrowaveContext.instance().handleEvent(
          Events.DOOR_OPENED_EVENT);
  } else if (event.getSource().equals(frame.cookButton)) {
    MicrowaveContext.instance().handleEvent(
          Events.COOKING_REQUESTED_EVENT);
  }
}
```

Improved Design



Improved Design



Consequences of Observer

- The Observer Pattern allows an arbitrary object to be registered as a listener.
- This is a powerful technique but it has several consequences:
  - The is a problem with memory leaks:
    - If the Observable contains a reference to an object, it makes it difficult to know when to release that reference for garbage collection
  - Order of observer notification:
    - The pattern does not specify the order or if there are temporal constraints
    - External mechanisms will be needed to handle these situations
  - Due to the fact that any object may become a listener
    - We can end up invoking unsafe code
  - Listening to several observables can result in very complex `update` methods
  - There will be several issues if we intermix the Observer Pattern and Concurrent/Parallel Processing

For Next Time

- Review Chapter 10.5 - 10.7
- Review this lecture
- Read Chapter 10.8 - 10.10
- Watch Lecture 28

Are there any questions?

ROAR