Recitation 3

Syed Raza (raza@mit.edu)

Fall 2010
State Machines

Why use State Machines?

- Often, problems are vague (e.g. “design a multipart downloader”), whereas solutions have to consider low-level details (e.g. Java code).

- We use State Machines (and other models) to *bridge the gap* between problems and solutions.
  - State Machines capture the essence of the problem at hand, so they are good for exploring corner cases and refining specifications *upfront*.
  - State Machines hide exact implementation details, so they still leave room for different design choices in solutions.

- State Machines concisely communicate designs – your peers do not need to read through your code.

- When properly used, State Machines help decompose complex behaviors into much simpler models.
State Machines

States

- States represent different *fundamental modes* within which an object can operate.

A **Light Bulb** can be “ON” or “OFF”. In one state, the bulb emits light/heat energy (and consumes power) while in the other it does nothing.
State Machines

Events

- Events represent transitions that change the state of an object, and usually signify actions.
- Events can be external (e.g. someone pushes a button) or internal (e.g. an object reaches its expiration date).

A Light Bulb can change states because of a single action: Toggle Switch. Here, the event is external to the Light bulb, because it represents a person pressing a button.
State Machines

Initial State

- It is important to mark the initial state for your state machine, using an arrow – don’t forget it on the quiz!

A Light Bulb starts in the “OFF” state by default.
State Machines

Input & Output Notation

- Inputs and Outputs can be marked on events, and a slash ("/") is used to separate them.

Inputs/Outputs are not very informative in this example. In lecture, we saw a case where inputs/outputs were helpful in explaining the transitions better. Here, a Toggle Switch takes the key pressed as input, and outputs a ‘click’ sound.
State Machines

Important Points

- Remember not to confuse states and events: it is easy to mix them up.
  - A state is a distinct mode of operation of an object, and an event is an action that changes state.

- Pick the right level of detail to be able to formulate a state machine that helps with your problem.

![Diagram of Toggle Switch state machine with states ON and OFF and event 'press(button) / 'click']
State Machines

Example 1: Door

- **States:**
  - OPENED, CLOSED+LOCKED, CLOSED+UNLOCKED

- **Events:**
  - open, close, lock, unlock(key)
State Machines

Example 1: Door

- **States:**
  - OPENED, CLOSED+LOCKED, CLOSED+UNLOCKED

- **Events:**
  - open, close, lock, unlock(key)

We can parameterize the same event to explain different transitions e.g. unlock(right key) & unlock(right key).
State Machines

Example 1: Door
- States:
  OPENED, CLOSED+LOCKED, CLOSED+UNLOCKED
- Events:
  open, close, lock, unlock(key)

It is important to consider all transitions (especially illegal transitions) from all states – this ensures your code is robust.
State Machines

Example 1: Door

- States:
  - OPENED, CLOSED+LOCKED, CLOSED+UNLOCKED
- Events:
  - open, close, lock, unlock(key)

Even in this trivial example, it is important that one can’t lock an opened door. In real life, many people get locked out of rooms because they accidentally lock an opened door, and later close it!
State Machines

Example 2: Call Waiting Protocol

- We wish to model *incoming* calls for a phone.
- When there is an incoming call, the phone starts ringing, and we can choose to accept or reject the call.
- When we are on the phone with someone, we can hang-up on them (or the call may get dropped).
- If we are on the phone with someone, and there is another incoming call, we can keep the other person waiting till the conversation is over, reject the new incoming call, or switch between the two calls.
- We can only have one call waiting/on-hold at any given time.
State Machines

Example 2: Call Waiting Protocol

- States:
  - IDLE, RINGING, CONVERSING

- Events:
  - incoming, accept, reject, hang-up, call-drop

It can be beneficial to draw state-machines iteratively: start from some simple states and see if you need to add any more states/transition.

Here we first model a simple phone (without call waiting).
Example 2: Call Waiting Protocol

- States:
  IDLE, RINGING, CONVERSING

- Events:
  incoming, accept, reject, hang-up, call-drop, switch

Here, to handle an incoming/waiting call when we are already conversing, we add a new state.

We add a new event for when we switch (or swap) the call on hold, and add new transitions.
Example 2: Call Waiting Protocol

- **States:**
  - IDLE, RINGING, CONVERSING

- **Events:**
  - incoming, accept, reject, hang-up, call-drop, switch

There is a possible edge-case that we have to worry about in our marked transition.

Should the phone start ringing every time I hang up from the conversing + call waiting state?
State Machines

Example 2: Call Waiting Protocol

States:
IDLE, RINGING, CONVERSING

Events:
incoming, accept, reject, hang-up, call-drop, switch

Should the phone start ringing every time I hang up from the conversing + call waiting state?
No. Whether the phone should ring depends on whether I switched between calls. If I did switch, I accepted both calls, so the phone should not ring.
State Machines

Example 2: Call Waiting Protocol

- States:
  - IDLE, RINGING, CONVERSING

- Events:
  - incoming, accept, reject, hang-up, call-drop, switch

There was a subtle bug / design choice. Drawing a state machine allowed me to identify it before any coding occurred. It is easier to fix my design before coding begins, and no code is lost.
Implementing State Machines

Example 2: Call Waiting Protocol

- We will consider two different ways to implement the state machine we just designed:
  - State Machine as Object/Class
  - State as Object/Class
Implementing State Machines

State Machine as Object/Class

```java
public class Phone {
    private static enum PhoneState {
        IDLE, RINGING, CONVERSING, CALL_WAITING
    }

    private PhoneState state;
    private String incomingNumber;
    private String conversingNumber;
    private String waitingNumber;

    public Phone() {
    }
}
```

A Java enumeration is a good way to define some constant values that a particular type ranges between.

Here, a PhoneState type can be one of 4 constants.
Implementing State Machines

State Machine as Object/Class

```java
public class Phone {
    private static enum PhoneState {
        IDLE, RINGING, CONVERSING, CALL_WAITING
    };

    private PhoneState state;
    private String incomingNumber;
    private String conversingNumber;
    private String waitingNumber;

    public Phone() {
        // Some internal state of the State Machine is stored in instance fields.
    }
}
```
Implementing State Machines

State Machine as Object/Class

```java
public class Phone {
    private static enum PhoneState {
        IDLE, RINGING, CONVERSING, CALL_WAITING
    }

    private PhoneState state;
    private String incomingNumber;
    private String conversingNumber;
    private String waitingNumber;

    public Phone() {
        this.state = PhoneState.IDLE;
    }
}
```

The constructor sets the initial state, and default values for any other instance fields.
Implementing State Machines

State Machine as Object/Class

```java
public class Phone {
    /* Other Fields/Methods are Omitted */
    private PhoneState state;
    private String incomingNumber;

    public void incomingCall(String phoneNumber) {
        switch (this.state) {
            case IDLE:
                this.state = PhoneState.RINGING;
                this.incomingNumber = phoneNumber;
                break; /* Make Phone Ring and Display # */
            case CONVERSING:
                this.state = PhoneState.CALL_WAITING;
                this.waitingNumber = phoneNumber;
                break; /* Show Call Waiting and Display # */
            default: // other cases
                /* This should not happen */
                break;
        }
    }
}
```

The switch statement is important for making code cleaner. Do not forget the break after each case.
Implementing State Machines

**State Machine as Object/Class**

```java
public class Phone {
    /* Other Fields/Methods are Omitted */
    private PhoneState state;
    private String incomingNumber;

    public void incomingCall(String phoneNumber) {
        /* Switch Statement */
    }

    public void reject() {
        /* Switch Statement */
    }

    public void hangup() {
        /* Switch Statement */
    }
}
```

Events in this example correspond to instance methods in the State Machine class.

This may not always be the case e.g. you could have a single `read()` method which is called repeatedly. States are accessed and updated inside the method (e.g. Project 1).
Implementing State Machines

State as Object/Class

```java
public interface PhoneState {

    public PhoneState incomingCall(String phoneNumber);

    public PhoneState accept();

    public PhoneState reject();

    public PhoneState hangup();

    public PhoneState switchCalls();

}
```

In the second implementation strategy, each state is represented through a common interface.

The interface must define methods that correspond to each event.

The return type for each event represents the new state if a particular event occurs from the current state.
Implementing State Machines

State as Object/Class

```java
public class IdlePhoneState implements PhoneState {

    @Override
    public PhoneState incomingCall(String phoneNumber) {
        return new RingingPhoneState(phoneNumber); // new state
    }

    @Override
    public PhoneState accept() {
        return this; // no state transition
    }

    @Override
    public PhoneState reject() {
        return this; // no state transition
    }
}
```

There is a class for each state of the Phone state machine e.g. IdlePhoneState, or RingingPhoneState.

Each class implements the common interface `PhoneState`.

The return-type represents the new/next state from an event.
## Implementing State Machines

<table>
<thead>
<tr>
<th>State Machine as Object/Class</th>
<th>State As Object/Class</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Advantage:</strong> Only one object is created for a single state machine.</td>
<td><strong>Disadvantage:</strong> Can cause too many object creations/deletions, leading to performance degradation. A new object is created for each transition.</td>
</tr>
<tr>
<td><strong>Advantage:</strong> All the code is in one place, and can be elegantly written using simple <code>switch</code> statements.</td>
<td><strong>Advantage:</strong> If each state does very different things, it makes logical sense to separate the code into different classes and avoid extremely large/complex switch statements.</td>
</tr>
<tr>
<td><strong>Disadvantage:</strong> Must have all code in the same place (same class/methods), even if different states have completely different behaviors.</td>
<td><strong>Advantage:</strong> Can also reuse code, if states are very similar, via inheritance from an abstract class.</td>
</tr>
<tr>
<td><strong>Advantage:</strong> Forces you to think about, and handle all possible transitions from all states, because an interface requires all transitions to be implemented.</td>
<td></td>
</tr>
</tbody>
</table>
Next Thursday

Why use State Machines?

- Decoupling, Module Dependency Diagrams
- Abstract Classes versus Interfaces
- Designing for Testability