ECE 448/528
Application Software Design

Lecture 10. Event-Driven Programming and The Observer Pattern

Spring 2025

Won-Jae Yi, Ph.D.

Department of Electrical and Computer Engineering
Illinois Institute of Technology

Event-Driven Programming

Event-Driven Applications

- Events: actions from various sources.
  - For example, from external environments like user inputs and network communications.
- An event-driven application runs infinitely, waiting for events to happen and then reacting to them.
  - Cause more events to happen, either processed by the application itself or sent to the external environment.
- Our IoT simulator is event-driven.
  - Timer events cause plugs to measure their power consumption.
    - Remember that power consumption is random! in this project.
  - HTTP requests may cause plugs to change their state. Plugs further react by sending back their current state via HTTP responses.
- The browser is event-driven.
  - User inputs cause the browser to send HTTP requests.
  - HTTP responses cause the browser to update its display.

Event-Driven Programming

- A programming paradigm for event-driven applications.
- General architecture: a dead loop of
  - Waiting for an event to happen, e.g. a timer, a user input, or a message from the network.
  - Allow the application to react to the event, via IoC or callbacks.
  - Allow the application to trigger more events and to send those events to their destinations.
- Generally, we rely on an event-driven framework or language to take care of the loop and the events.
  - So, the developer only needs to focus on event handling.

Challenges in Event Handling

- For event handling, developers need to specify
  - What events are of interest?
  - How to process an event?
  - Where newly created events should be sent to?
- These are three separate responsibilities.
  - Don’t overlook the complexity when there are more than a few objects producing and consuming events.
  - What class design should we have to support them?

Event Processing and Threading

- It is quite appealing to accelerate event processing by using multiple threads.
- However, as we need to update objects during event processing, accessing the same object from multiple threads without protection may result in racing conditions.
- **Method 1**: use locks, e.g. synchronized methods, to protect objects.
  - This is used by our Java code.
- **Method 2**: rely on thread confinement provided by the framework or the language, which guarantees events delivered to the same object are processed in the same thread.
  - This is how JavaScript works.

The Observer Pattern

Design Patterns

- Common OOD/OOP practices to solve software design problems
- Learn design experiences from experts
  - Design patterns are solutions that are applied routinely.
  - Design patterns are independent of programming languages.
- Facilitate effective communication among designers
  - Design patterns serve as a shared technical language for designers.
  - They help quickly grasp the structure of complex software.
  - They enable others to easily understand your design concepts.

The Observer Pattern

- A design pattern to decompose event processing and event delivery.
- **EventSource**: the class that produces events
  - As a consequence of processing events.
- **Observer**: an interface providing an abstraction of event delivery.

Example: MLB Scoreboard

- **Subject.java**: *interface* for register, unregister, notifyObserver
- **ScoreGetter.java**: *implements* Subject (register, unregister, notifyObserver), sets scores which will notifyObserver.
  - This is the *EventSource* that generates events (state changes) and notifies the observers.
- **Observer.java**: *interface* for update scores
- **ScoreObserver.java**: *implements* Observer (update), keeps count of Observers
- **GetScores.java**: main(). Utilizes the above.

The Design Problem

```
public class PlugSim {
   ...
   synchronized public void switchOn() {
      on = true;
   }
   ...
}
```

- How should we update `PlugSim` so that we may notify others that the switch is on?
  - If this is an actual smart plug, we may need to set the GPIO connecting to the relay to 1.
  - We may also need to notify others across the network for home automation.
- So, `PlugSim` is the EventSource in the observer pattern.

The Observer Interface

```
public class PlugSim {
  ...
  public static interface Observer {
    void update(String name, String key, String value);
  }
  ...
}
```

- As the EventSource decides what events are produced, the `Observer` interface is defined as an inner/nested interface.
- A `static` inner interface depends on the `PlugSim` class but not any `PlugSim` object.
- The `update` method indicates an event.
  - `name`: name of the plug.
  - `key/value`: details of the event, e.g. `"state"/"on"`, `"power"/"100"`.

Observers Management

```
public class PlugSim {
  ...
  private final ArrayList<Observer> observers =newArrayList<>();
  synchronized public void addObserver(Observer observer) {
    observers.add(observer);
    observer.update(name, "state", on? "on": "off");
    observer.update(name, "power", String.format("%.3f", power));
  }
  ...
}
```

- There could be multiple observers, the EventSource should use a container to manage them.
- An `addObserver` method adds an observer.
  - The EventSource should send events now so the observer could initialize itself without waiting.
  - Be careful with multi-threading.
- But how does an `Observer` process an event?

Inversion of Control

```
public class PlugSim {
  ...
  synchronized public void switchOn() {
    updateState(true);
  }
  protected void updateState(boolean o) {
    on = o;
    logger.info("Plug {}: state {}", name, on? "on": "off");
    for (Observer observer: observers) {
      observer.update(name, "state", on? "on": "off");
    }
  }
  ...
}
```

- This is IoC.
  - We don't care how an `Observer` processes an event.
  - Need to make sure the observers are notified whenever necessary – via the `updateState` method.
- `protected` methods cannot be accessed out of this and derived classes.
  - We don't need a lock as long as `public` methods are locked.