COMP 3550

7.3 — OBSERVER & CHAIN OF RESPONSIBILITY PATTERNS

Week 7: Design Patterns & Architecture

OBSERVER PATTERN

IF YOU COULD LIKE, COMMENT, SUBSCRIBE, HIT THE NOTIFICATION BELL AND FOLLOW ME ON OTHER SOCIAL MEDIAS
THAT WOULD BE GREAT
imgflip.com

Definition:

- A pattern where observers subscribe to a subject.
- When the subject’s state changes, it notifies all observers automatically.
- Used to decouple the source of a change from the code that responds to it.

OBSERVER PATTERN

Commonly seen with buttons (Action Listeners)

```
button.addActionListener(e -> showDialog());
```

OBSERVER PATTERN

Making your own?

```
interface Observer {
    void update(String message);
}

class Subject {
    void register(Observer o);
    void detach(Observer o);
    void notifyObservers(String message);
}
```

Flow:

- Observers register with the subject
- Subject tracks them in a list
- When notifyObservers() is called, each one gets the update

OBSERVER PATTERN IN THE WILD

Game Engines / UI Systems

- Event Listeners:
  - Player presses a key → multiple systems react (move character, play sound, trigger animation)
  - onKeyDown(), onMouseClick() in Unity, Unreal, JavaFX, etc.
- Health system:
  - When a player's health changes, UI bars, damage effects, and audio cues are all notified.

Logging Frameworks

- A system triggers a log event → multiple loggers receive it:
  - Console logger
  - File logger
  - Remote logger (e.g., Logstash, Graylog)
- All are observing the same log subject.

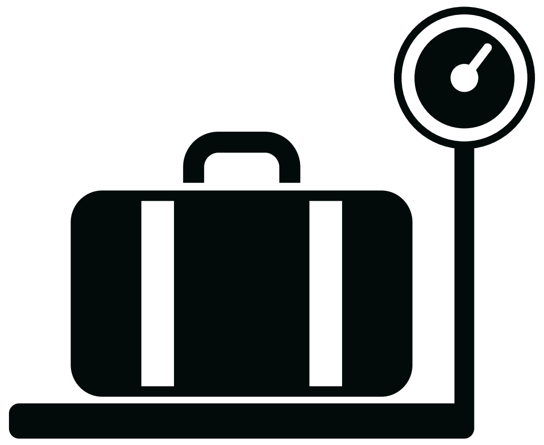
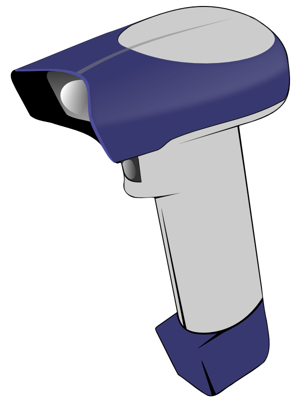
CHAIN OF RESPONSIBILITY

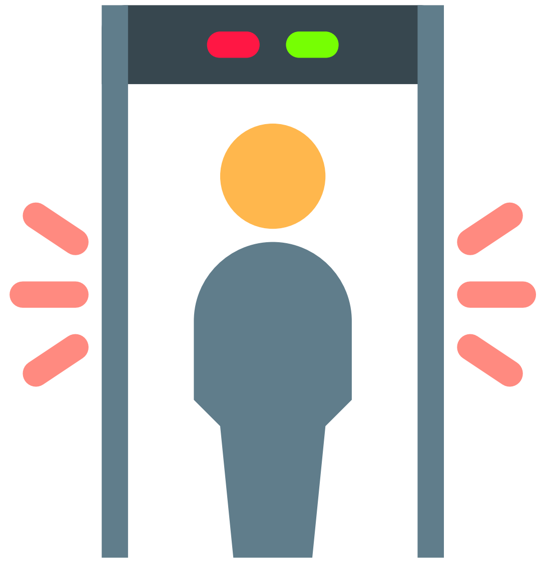
Definition:

- Lets you pass a request through a chain of handlers, where each one:
  - Can handle it,
  - Or pass it to the next in line.
- The sender doesn't need to know who will ultimately handle the request.

Common Use Case:

- Middleware pipelines
  - Example: HTTP request flow
  - → Authentication → Validation → Business Logic → Logging



CHAIN OF RESPONSIBILITY

```java
abstract class Handler {
    protected Handler next;

    public Handler setNext(Handler next) {
        this.next = next;
        return next;
    }

    public abstract void handle(Request req);
}

class AuthHandler extends Handler {
    public void handle(Request req) {
        if (!req.isAuthenticated()) {
            System.out.println("Unauthorized!");
            return;
        }
        next.handle(req);
    }
}

class ValidationHandler extends Handler {
    public void handle(Request req) {
        if (!req.isValid()) {
            System.out.println("Invalid data!");
            return;
        }
        next.handle(req);
    }
}

class ExecutionHandler extends Handler {
    public void handle(Request req) {
        System.out.println("Request executed!");
    }
}
```

```java
// Setup chain
Handler chain = new AuthHandler();
chain.setNext(new ValidationHandler())
     .setNext(new ExecutionHandler());

chain.handle(new Request());
```

CHAIN OF RESPONSIBILITY

```java
abstract class Handler {
    protected Handler next;

    public Handler setNext(Handler next) {
        this.next = next;
        return next;
    }

    public abstract void handle(Request req);
}

class AuthHandler extends Handler {
    public void handle(Request req) {
        if (!req.isAuthenticated()) {
            System.out.println("Unauthorized!");
            return;
        }
        next.handle(req);
    }
}

class ValidationHandler extends Handler {
    public void handle(Request req) {
        if (!req.isValid()) {
            System.out.println("Invalid data!");
            return;
        }
        next.handle(req);
    }
}

class ExecutionHandler extends Handler {
    public void handle(Request req) {
        System.out.println("Request executed!");
    }
}
```

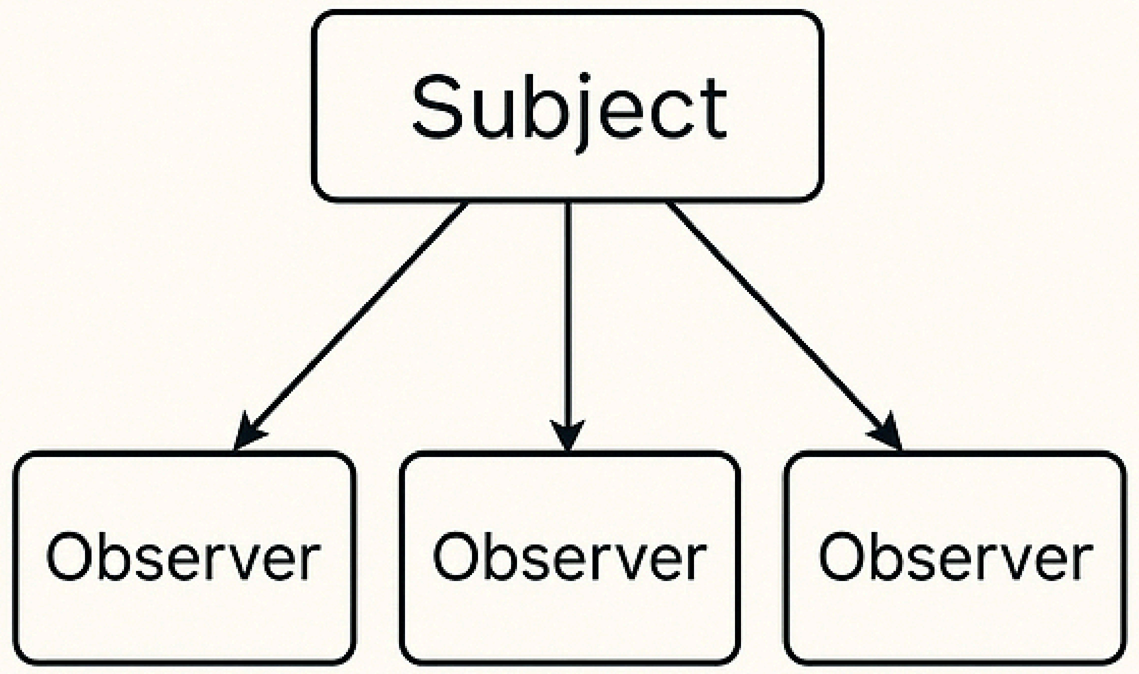
```java
// Setup chain
Handler chain = new AuthHandler();
chain.setNext(new ValidationHandler())
     .setNext(new ExecutionHandler());

chain.handle(new Request());
```

Clarifying CoR and LoD:

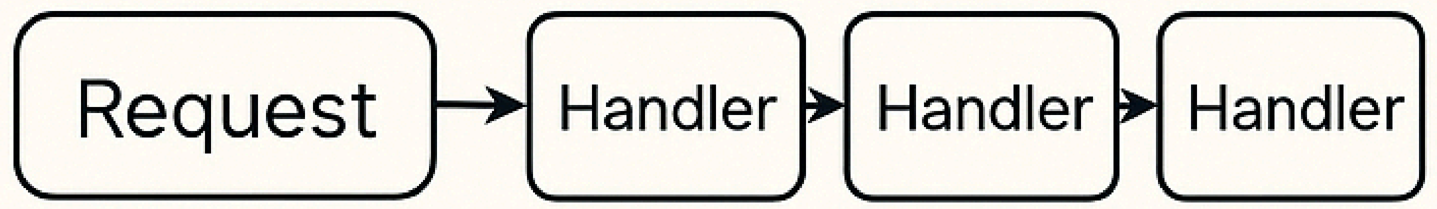
Chain of Responsibility (CoR) is a design pattern for delegating who handles a request.
Law of Demeter (LoD) is a design principle for reducing how much a class knows about others.

CoR chooses who handles a request. LoD limits how you ask for things. They both delegate, but for very different reasons.

Observer Pattern



Chain of Responsibility Pattern



REFACTOR THIS NOTIFICATION USING EITHER PATTERN DISCUSSED

```
public class OrderService {
    public void completeOrder(Order order) {
        // Step 1: Send confirmation email
        EmailService.sendConfirmation(order.getEmail());

        // Step 2: Update inventory
        InventorySystem.decreaseStock(order.getItemId(),
order.getQuantity());

        // Step 3: Log the transaction
        Logger.log("Order completed for: " + order.getEmail());

        System.out.println("Order complete.");
    }
}
```

Testing Reminder

Don’t forget to test your code before and after the refactor to make sure it still works correctly and you didn’t break anything!