SOFTWARE DESIGN
COSC 4353/6353

Dr. Raj Singh

OUTLINE

Most Used Design Patterns

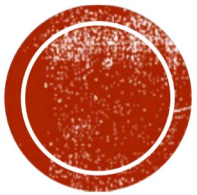
Deep Dive

Examples

Where to go next?

MOST IMPORTANT DESIGN PATTERNS

Pattern	Category	Notes
Singleton	Creational	limit creation of a class to only one object
Factory	Creational	objects are created by calling a factory method instead of a constructor
Builder	Creational	creating complex types can be simplified by using the builder pattern
Adapter	Structural	allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class
Facade	Structural	create a simplified interface of an existing interface to ease usage for common tasks
Observer	Behavioral	when one object changes state, all its dependents are notified
Chain of Responsibility	Behavioral	delegates commands to a chain of processing objects



DEEP DIVE

SINGLETON

This is the most used pattern

A lot of framework already implement this pattern, such as:

Spring (via @ApplicationScoped)
EJBs (using @Singleton)

Ensure a class has only one instance and provide a global point to access it

The concept is sometimes generalized to systems that operate more efficiently when only one object exists

The term comes from the mathematical concept of a singleton

SINGLETON EXAMPLE

Problem: You need an object that needs to be instantiated once.

The class needs to declare a private constructor to prevent people to instantiate it from outside the class.

The method getInstance() assures that only one instance of this class is created at runtime.

```
public class SingletonExample {
 private static SingletonExample instance = null;
 private SingletonExample() { }
 public static SingletonExample getInstance() {
  if(instance == null) {
   instance = new SingletonExample();
  }
 return instance;
 }
}
```

FACTORY

Problem: How can an object be created so that subclasses can redefine which class to instantiate?

The Factory design pattern describes how to solve such problems:

- Define a separate operation (*factory method*) for creating an object.
- Create an object by calling a *factory method*.

This enables writing of subclasses to change the way an object is created

FACTORY EXAMPLE

The MazeGame uses Rooms but it puts the responsibility of creating Rooms to its subclasses which create the concrete classes.

The regular game mode could use this template method.

```
public abstract class Room {
  abstract void connect ( Room room );
}
public class MagicRoom extends Room {
  public void connect ( Room room ) { }
}
public class OrdinaryRoom extends Room {
  public void connect ( Room room ) { }
}
public abstract class MazeGame {
  private final List<Room> rooms = new ArrayList <> ( );
  Public MazeGame () {
    Room room1 = makeRoom( );
    Room room2 = makeRoom( );
    room1.connect( room2 );
    rooms.add( room1 );
    rooms.add( room2 );
   }
  abstract protected Room makeRoom( );
}
```

BUILDER

Problem: Some objects require lots of parameters to be created

In this case, either using the constructor to create this object or using the setters will make code ugly and hard to understand

The builder pattern can help us in this case

The intent of the Builder design pattern is to separate the construction of a complex object from its representation	By doing so the same construction process can create different representations

BUILDER EXAMPLE

A product can have many types.

```
public class Product {
  private String id;
  private String name;
  private String description;
  private Double value;
  private Product(Builder builder) {
    setId(builder.id);
    setName(builder.name);
    setDescription(builder.description);
    setValue(builder.value);
  }
  // Getter and Setter methods for all attributes
  . . .
}
```

BUILDER EXAMPLE

Builder class builds the product

```
public static final class Builder {
 private String id;
 private String name;
 private String description;
 private Double value;
 private Builder() { }
 public Builder id(String id) { this.id = id; return this; }
 public Builder name(String name) { this.name = name; return this; }
 public Builder description(String description) {
  this.description = description; return this;
 }
 public Builder value(Double value) { this.value = value; return this; }
 public Product build() { return new Product(this); }
}
```

ADAPTER

Problem: Often an existing class can't be reused because its interface doesn't conform to the interface clients require

The key idea is to work through a separate adapter that adapts the interface of an existing class without changing it

The adapter design pattern describes how to solve such problems:

Define a separate adapter class that converts the (incompatible) interface of a class (adaptee) into another interface (target) clients require.	Work through an adapter to work with (reuse) classes that do not have the required interface.

ADAPTER EXAMPLE

Charging phones

```
interface LightningPhone {
 void recharge();
 void useLightning();
}
interface MicroUsbPhone {
 void recharge();
 void useMicroUsb();
}
class Iphone implements LightningPhone {
 private boolean connector;
 @Override
 public void useLightning() {
  connector = true;
 }
 @Override
 public void recharge() {
  if (connector) {
   System.out.println("Recharge started");
  } else {
   System.out.println("Connect Lightning first");
  }
 }
}
 . . .
```

ADAPTER EXAMPLE ...

Charging phones

```
class Android implements MicroUsbPhone {
 private boolean connector;

 @Override
 public void useMicroUsb() {
  connector = true;
  System.out.println("MicroUsb connected");
 }

 @Override
 public void recharge() {
  if (connector) {
   System.out.println("Recharge started");
  } else {
   System.out.println("Connect MicroUsb first");
  }
 }
}
. . .
```

ADAPTER EXAMPLE ...

Charging phones

```
class LightningToMicroUsbAdapter implements MicroUsbPhone {
 private final LightningPhone lightningPhone;

 public LightningToMicroUsbAdapter(LightningPhone lightningPhone) {
  this.lightningPhone = lightningPhone;
 }

 @Override
 public void useMicroUsb() {
  System.out.println("MicroUsb connected");
  lightningPhone.useLightning();
 }

 @Override
 public void recharge() {
  lightningPhone.recharge();
 }
}
. . .
```

ADAPTER EXAMPLE ...

Charging phones

```
public class AdapterDemo {
 static void rechargeMicroUsbPhone(MicroUsbPhone phone) {
  phone.useMicroUsb();
  phone.recharge();
 }
 static void rechargeLightningPhone(LightningPhone phone) {
  phone.useLightning();
  phone.recharge();
 }
 public static void main(String[] args) {
  Android android = new Android();
  Iphone iPhone = new Iphone();

  System.out.println("Recharging android with MicroUsb");
  rechargeMicroUsbPhone(android);
  System.out.println("Recharging iPhone with Lightning");
  rechargeLightningPhone(iPhone);
  System.out.println("Recharging iPhone with MicroUsb");
  rechargeMicroUsbPhone(new LightningToMicroUsbAdapter(iPhone));
 }
}
```

FACADE

Problem: Clients that access a complex subsystem directly refer to many different objects having different interfaces, which makes the clients hard to implement, change, test, and reuse

Facade enables to work through an object to minimize the dependencies on a subsystem

The Facade design pattern describes how to solve such problems:

implements a simple interface in terms of (by delegating to) the interfaces in the subsystem	may perform additional functionality before/after forwarding a request

FACADE EXAMPLE

How a client ("you") interacts with a facade (the "computer") to a complex system (internal computer parts, like CPU and HardDrive)

```
/* Complex parts */

class CPU {
 public void freeze() { ... }
 public void jump(long position) { ... }
 public void execute() { ... }
}

class HardDrive {
 public byte[] read(long lba, int size) { ... }
}

class Memory {
 public void load(long position, byte[] data) { ... }
}
. . .
```

FACADE EXAMPLE ...

How a client ("you") interacts with a facade (the "computer") to a complex system (internal computer parts, like CPU and HardDrive)

```
/* Facade */
class ComputerFacade {
 private final CPU processor;
 private final Memory ram;
 private final HardDrive hd;

 public ComputerFacade() {
  this.processor = new CPU();
  this.ram = new Memory();
  this.hd = new HardDrive();
 }

 public void start() {
  processor.freeze();
  ram.load(BOOT_ADDRESS, hd.read(BOOT_SECTOR, SECTOR_SIZE));
  processor.jump(BOOT_ADDRESS);
  processor.execute();
 }
}
/* Client */
class You {
 public static void main(String[] args) {
  ComputerFacade computer = new ComputerFacade();
  computer.start();
 }
}
```

OBSERVER

Problem: Tightly coupled objects are hard to implement, change, test, and reuse because they refer to many different objects with different interfaces

Observer solves following problems:

A one-to-many dependency between objects should be defined without making the objects tightly coupled.	It should be ensured that when one object changes state an open-ended number of dependent objects are updated automatically.	It should be possible that one object can notify an open-ended number of other objects

The Observer design pattern describes how to solve such problems:

Define Subject and Observer objects	When a subject changes state, all registered observers are notified and updated automatically

OBSERVER EXAMPLE

This example takes keyboard input and treats each input line as an event. When a string is supplied from System.in, the method notifyObservers is called, that notifies all observers of the event's occurrence, in the form of an invocation of their 'update' methods.

```
class EventSource {
 public interface Observer {
  void update(String event);
 }
 private final List<Observer> observers = new ArrayList<>();
 private void notifyObservers(String event) {
  observers.forEach(observer -> observer.update(event));
 }
 public void addObserver(Observer observer) {
  observers.add(observer);
 }
 public void scanSystemIn() {
  var scanner = new Scanner(System.in);
  while (scanner.hasNextLine()) {
   var line = scanner.nextLine();
   notifyObservers(line);
  }
 }
}
public class ObserverDemo {
 public static void main(String[] args) {
  System.out.println("Enter Text: ");
  var eventSource = new EventSource();
  eventSource.addObserver(event -> {
  System.out.println("Received response: " + event);
  });
  eventSource.scanSystemIn();
 }
}
```

CHAIN OF RESPONSIBILITY

Problem: Implementing a request directly within the class that sends the request is inflexible because it couples the class to a particular receiver and makes it impossible to support multiple receivers.

Chain of Responsibility solves following problems:

Coupling the sender of a request to its receiver should be avoided.	In addition, it should be possible that more than one receiver can handle a request

The Chain of Responsibility design pattern describes how to solve such problems:

Enable to send a request to a chain of receivers without having to know which one handles the request	The request gets passed along the chain until a receiver handles the request. The sender of a request is no longer coupled to a particular receiver

CHAIN OF RESPONSIBILITY EXAMPLE

A logger is created using a chain of loggers, each one configured with different log levels.

```
public interface Logger {
 public enum LogLevel {
  INFO, DEBUG, WARNING, ERROR, FUNCTIONAL_MESSAGE, FUNCTIONAL_ERROR;
  public static LogLevel[] all() { return values(); }
 }
 abstract void message(String msg, LogLevel severity);
 default Logger appendNext(Logger nextLogger) {
  return (msg, severity) -> {
  message(msg, severity);
  nextLogger.message(msg, severity); };
 }
 static Logger logger(LogLevel[] levels, Consumer<String> writeMessage) {
  EnumSet<LogLevel> set = EnumSet.copyOf(Arrays.asList(levels));
  return (msg, severity) -> {
   if (set.contains(severity)) { writeMessage.accept(msg); }
  };
 }
 static Logger consoleLogger(LogLevel... levels) {
  return logger(levels, msg -> System.err.println("Writing to console: " + msg));
 }
 static Logger emailLogger(LogLevel... levels) {
  return logger(levels, msg -> System.err.println("Sending via email: " + msg));
 }
 static Logger fileLogger(LogLevel... levels) {
  return logger(levels, msg -> System.err.println("Writing to Log File: " + msg));
 }
}
```

CHAIN OF RESPONSIBILITY EXAMPLE

A logger is created using a chain of loggers, each one configured with different log levels.

```
public class ChainOfResponsibiltyDemo {
 public static void main(String[] args) {
  // Build an immutable chain of responsibility
  Logger logger = consoleLogger(LogLevel.all())
  .appendNext(emailLogger(LogLevel.FUNCTIONAL_MESSAGE, LogLevel.FUNCTIONAL_ERROR))
  .appendNext(fileLogger(LogLevel.WARNING, LogLevel.ERROR));

  // Handled by consoleLogger since the console has a loglevel of all
  logger.message("Entering function ProcessOrder().", LogLevel.DEBUG);
  logger.message("Order record retrieved.", LogLevel.INFO);

  // Handled by consoleLogger and fileLogger since filelogger implements Warning & Error
  logger.message("Customer Address details missing in Branch DataBase.", LogLevel.WARNING);
  logger.message("Customer Address details missing in Organization DataBase.", LogLevel.ERROR);

  // Handled by consoleLogger and emailLogger as it implements functional error
  logger.message("Unable to Process Order ORD1 Dated D1 For Customer C1.", LogLevel.FUNCTIONAL_ERROR);

  // Handled by consoleLogger and emailLogger
  logger.message("Order Dispatched.", LogLevel.FUNCTIONAL_MESSAGE);
 }
}
```

HOMEWORK

Review class notes.

Additional reading:

Examples of Design Patterns

Start a discussion on Google Groups to clarify your doubts.