Principles, Mediator, and Factory

Making the Concrete Abstract

Cartoon:

1. "I can't wait to see the changes I asked you to make on the interface.
2. "Our last meeting was two months ago. You must be finished by now.
3. "I haven't started yet.
4. "I had a few questions.
5. "I figured I'd ask you about them the next time we talked.
6. "In the meantime I only did work for people who yelled at me every day.
7. "Micromanagement has a bad reputation, but I'm not too proud to say I need it.
8. "Okay... well, I'm optimistic that you can make those changes for me by next week.
9. "I probably should have asked my questions."
Principles
The Single Responsibility Principle (SRP)
- Each class should be responsible for one thing
- Easy to violate, we start with few classes

Classes should be open for extension but closed for modification (OCP)
- Do not modify classes (overriding methods, etc…)
- Add functionalities using separate classes
- When we need to change some functionality we want to modify a single class

Dependency Inversion Principle (DIP)
- Code to interfaces not implementations

The Don’t Repeat Yourself (DRY) Principle
- Pull out repeated code into new method or class
- Repeated code often result of cut-and-paste programming
Liskov Substitution Principle (LSP)

“objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.”

- Pattern Example
  - Strategy let me create different implementations of an algorithm

- For strategy to work
  - Can replace one algorithm implementation with another
  - So algorithms must not only implement the syntactic interface (signature)
  - They must expose the same behavior

- Example two sorting algorithms
  - Will have different speed/memory usage
  - They will both result in an ordered list
Interface segregation principle (ISP)

“many client-specific interfaces are better than one general-purpose interface.”
Where is ISP in Observer?

- Work with your team and come up with an explanation of how ISP applies to the Observer Pattern.
Interface segregation principle (ISP)

“many client-specific interfaces are better than one general-purpose interface.”

- Pattern Example
  - Observer uses only simple interfaces used for connecting subject/observers

- Concrete classes in observer pattern must do more
  - Subject and observers must do something with the state
  - State of subject changes somehow before notify observers is called
  - Both observers and subject have additional methods (not necessarily implementing another interface)
Your classes should depend on as few objects as possible

- **Rationale**
  - If you change some object you need to update all its users
  - It is actually more complex to understand classes that depend on many objects
Follow well established object-oriented design principles
- Single Responsibility Principle (SRP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface segregation principle (ISP)
- Dependency inversion principle (DIP)

Together they are known with the acronym **SOLID**

Follow additional principles
- Don’t repeat yourself (DRY)
- Principle of Least Knowledge

Follow patterns
- Embed principles
- Ready to use solutions
Hollywood Principle

- “Don’t call us, we’ll call you!”
- Technique to reduce coupling by moving configuration and dependency wiring outside a component
- Difference between Frameworks and Libraries or APIs (Application Programming Interfaces)

Also called: Inversion of Control
Frameworks

A framework defines the skeleton and developers fill in the flash when using it.

```java
public class Main {
    public static void main(String[] args) {
        Human h = new Human(new Walk());
        h.doMove();
    }
}
```

```java
public abstract class Move {
    public abstract void action();
}
```

```java
public class Walk extends Move {
    @Override
    public void action() {
        // TODO Auto-generated method stub
        System.out.println("5 miles per hour - it is slow!");
    }
}
```

The Hook: A hook is where developers can define / extend functions based on their own requirements.
This simple framework allows and requires developers to extend "move" class.

action() method is the only thing developers are able to change.

Inside of the implementation, different "action"s can be programmed to different purposes.

E.g. print "5 miles per hour", of course, you can redefine it as "50 miles per hour".
Come up with one example

- Can you give me an example of the framework calling you in your project?

- Can you give me an example of you calling the API in your project?
“Don’t call us, we’ll call you!”

- Referred to as the “Hollywood Principle”
- Also referred to as Dependency Inversion Principle (DIP)
  - The principle states that high level or low level modules should not depend upon each other, instead they should depend upon abstractions.
  - Specific implementations (object instances) are deferred to a higher level of abstraction of control.
    - Examples:
      - Parent class(es)
      - A Container
Dependency Inversion

- Avoid deriving from concrete classes
- Avoid associating to concrete classes
- Avoid aggregating concrete classes
- Avoid dependencies on concrete components
Applying the Pareto Principle

- To your project as a whole to define sprint backlog, velocity
- To your code in terms of what depth of functionality needs to be there for a given sprint
Design By Variation

- Variations, Changing things in your domain
  - Rules
  - Data, Information, Structure, Types
  - Process, Flow, Flow of Control

- Each variation is a scenario

- Pick a subset of the variations to implement

- Use UML to model the problem domain, solution domain

- Bad design will make you late. Perfect design will make you late.
  - Therefore, make your design good enough
Pareto Principle

The 80-20 Rule

"For many events, roughly 80% of the effects come from 20% of the causes." - Pareto

Therefore 20% of the effort produces 80% of the results but the last 20% of the results consumes 80% of the effort.

www.EndlesslyCurious.com
80/20 Rule – diminishing returns tradeoff

- Returns on engineering investment are not linear
- 80 percent of the benefit for 20% of the effort
- Diminishing returns for effort/investment beyond
  - (last 80% of effort gets only 20% gain)
- Is an $80K car 4x better than a $20K car?
80/20 rule: Project Examples

- Geofencing sounds cool…
  - …but a bit of overkill (Porsche vs. Honda)
  - 20% better than brute force, but more effort than it’s worth

- When we prioritize our Stories, we are invoking the benefits of the 80/20 rule
  - OK, we didn’t get the recurring events done
    - Complicated! Time-consuming!
  - But we still have location-based reminders!

- JUnit is installed, working, and familiar…
  - …and JBehave/Robotium is not
  - Yes, it’s 20% better, but cost you time you didn’t have

- Remember the 80/20 rule, diminishing returns, cost/benefit, tradeoffs, and risk in general
Mediator
Mediator Pattern

Define and object that encapsulate how a set of objects interacts

- It is an important pattern to help creating a loosely coupled system
Mediator Pattern Context

- Many objects that depend on each other
  - Implies that all your classes depend on many other
  - Violates the principle of least knowledge
- Dependency is needed because the behavior distributed in all these objects
  - Cannot simply move all needed behavior in one class
  - Doing this would potentially violate SRP or DRY
  - Use of segregated interfaces could help but not completely solve the problem
Mediator Pattern Objectives

- Reduce dependencies between classes
- Ideally we should replace this dependency network with only one class dependency
- Objects interacts with each other
- A class should only care about its part in the interaction
- Problem with this design: if I change a class I have to fix all other classes!
- Make sure that I limit ripple effects
- Our solution should not violate other principles
Mediator Pattern Solution

- One Mediator class
- All other depend on the mediator for their interactions
- Mediator is aware of interaction goals
- Can expose an interface per class that interacts (SIP)
- If a class changes its behavior, mediator can be modified to make sure other classes don’t see this change
Factory
Factory Pattern: Motivation

- Correctly making objects is complex

- Especially making collection of related objects
  - Parts of a car
  - Look-and-feel of a window: canvas, scrollbar, etc.

- The correct making of objects is not easily centralized in one place
  - Often do it all over code wherever object is needed
  - Violates SRP and DRY

- DP angle is that “new” names and makes a concrete class; should be referring to abstract
  - concrete class has to be named, but we can at least hide that from most of system
  - encapsulating class will violate OCP, but no others will
Factory is the answer

- Actually a collection of patterns
  - Simple factory (idiom)
  - Factory Method
  - Abstract Factory

```java
class Event {
  protected String name;
  protected int firstAllowedDate = Integer.MAX_VALUE; // fail hard if no init

  public Event(int eventsFirstAllowedDate, String eventName) {
    firstAllowedDate = EventsFirstAllowedDate;
    name = eventName;
  }

  protected boolean dateSupported(int dateNumber) {
    return dateNumber >= firstAllowedDate;
  }
}
```

Concrete class; Mystery how to make one
Factory for Dating Events

```java
class Event {
  protected String name;
  protected int firstAllowedDate = Integer.MAX_VALUE; // fail hard if no init
}
```

/*
 * static Factory methods, for convenience and correctness.
 * Note that Date can’t even tell if Event has subclasses.
 */

```java
public static Event makeSeeMovie() { return new Event(1, "SeeMovie"); }

public static Event makeGoToRestaurantEvent() {
  return new Event(1, "GoToRestaurant");
}

public static Event makeOrderFlowers() {
  return new Event(2, "OrderFlowers");
}
```

static “Factory” methods keep Event details local
What’s wrong with 2¢ Factory?

Violates the Open/Closed principle

No delegation to an abstract class. No CLASS. And geez, the methods are STATIC.
Simple Factory (idiom)

```java
public enum DatingEvent { MOVIE, RESTAURANT, FLOWERS }

class /* class name */ {
    public /* method name */(DatingEvent eventEnum) {
        switch (eventEnum) {
            case MOVIE:
                return makeSeeMovie();
            case RESTAURANT:
                return makeGoToRestaurant();
            case FLOWERS:
                return makeOrderFlowers();
        }
    }

    /* static Factory helper methods. */

    protected static Event makeSeeMovie() { return new Event(1, "SeeMovie"); }
    protected static Event makeGoToRestaurantEvent() {
        return new Event(1, "GoToRestaurant");
    }
    protected static Event makeOrderFlowers() {
        return new Event(2, "OrderFlowers");
    }
}
```

How to **make** Events not necessarily part of Event
Comparison – increasingly abstract

class MyDatingClass {

    ...  
    Event event =  
        new Event(2, "OrderFlowers"); // magic constants  

    // concrete Event class (majorly violates OCP)
    Event event = Event.makeOrderFlowers();  

    // abstract class object (passed into constructor)
    Event event = factory.createEvent(FLOWERS);  

Class Diagram for Simple Factory

This is the factory where we create pizzas; it should be the only part of our application that refers to concrete Pizza classes.

This is the product of the factory: pizza!

PizzaStore
- orderPizza()

SimplePizzaFactory
- createPizza()

Pizza
- prepare()
- bake()
- cut()
- box()

CheesePizza

PepperoniPizza

VeggiePizza

ClamPizza

This is the client of the factory. PizzaStore now goes through the SimplePizzaFactory to get instances of pizza.

The create method is often declared statically.

We've defined Pizza as an abstract class with some helpful implementations that can be overridden.

These are our concrete products. Each product needs to implement the Pizza interface* (which in this case means "extend the abstract Pizza class") and be concrete. As long as that's the case, it can be created by the factory and handed back to the client.
What’s wrong with Simple Factory? (Clicker)

Client class composes concrete class

PizzaStore
- orderPizza()

SimplePizzaFactory
- createPizza()

Pizza
- prepare()
- bake()
- cut()
- box()

CheesePizza

VeggiePizza

PepperoniPizza

ClamPizza

The create method is often declared statically.

We’re using as an abstract with some help implementation can be overriden.

These are our concrete products. Each product needs to implement the Pizza interface (which in this case means “extend the abstract Pizza class”) and...

the client of the PizzaStore goes through the SimplePizzaFactory to get needs of pizza.
How do we make a factory more Abstract?

- Simple factory is good but too concrete.
- It violates OCP and DIP
- What happened if we do not know about the concrete class we are going to use when we write our code?
- We already know patterns that can help us.

What pattern should we use?

A: Dependency Injection
B: Strategy
C: Observer
D: Mediator
Design your Abstract Factory

- Work with your team to design an Abstract Factory.
- Apply the pattern to the simple factory and show come up with a Class Diagram for it.
Who uses Abstract Factory?
An example for the Project

- Does anybody use this abstract factory pattern?
- Let’s connect your web app to a database!

Standard API to get a database connection.

This let us find objects defined in the server.

jdbc/TestDB is the name we give to our data source.

Code in your web app servlet

```java
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    InitialContext ctx = null;
    Connection con = null;
    try {
        ctx = new InitialContext();
        DataSource ds = (DataSource) ctx.lookup("java:/comp/env/jdbc/TestDB");
        con = ds.getConnection();
    }
```

First of all, what is a data source?

- **DataSource** is a standard interface of Java (javax.sql.DataSource)
- All database JDBC drivers must implement it
- Your code depends only on the interface
- It has the getConnection() method
- Connection is another interface defined by JDBC that let you interact with the DB
- You can swap the DB implementation (strategy)
XML Configuration (context.xml)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Context>
  <Resource name="jdbc/TestDB"
           type="javax.sql.DataSource"
           auth="Container"
           description="Hyper SQL DB database"
           username="SA" password=""
           driverClassName="org.hsqldb.jdbc.JDBCDriver"
           url="jdbc:hsqldb:TestDB"/>
</Context>
```

- **Name to lookup**: Our `DataSource` interface
- **Connection string specific to the DB implementation**: `jdbc:hsqldb:TestDB`
- **Concrete implementation of the database driver**: `org.hsqldb.jdbc.JDBCDriver`
Webpage Generated

http://localhost:8080/Test_DB/GetDataSource

**DataSource**

org.apache.tomcat.dbcp.dbcp2.BasicDataSource@71c16912

**Connection**

935137402, URL=jdbc:hsqldb:TestDB, UserName=SA, HSQL Database Engine Driver
Alternative Database

- Our code does not change
- We only change the Resource definition in the context.xml file

```
<Resource name="jdbc/TestDB"
   type="javax.sql.DataSource" auth="Container"
   description="Derby SQL DB database"
   username="" password=""
   driverClassName="org.apache.derby.jdbc.EmbeddedDriver"
   url="jdbc:derby:memory:TestDB;create=true"/>
```

Connection string specific to Derby specifies that the DB is stored in memory not on the disk.

Derby implementation of the database driver
**Data Source**

org.apache.tomcat.dbcp.dbcp2.BasicDataSource@6753e65a

**Connection**

745151115, URL=jdbc:derby:memory:TestDB, UserName=APP, Apache Derby Embedded JDBC Driver