Welcome

Advance Java Introduction

Copyright Notice

Design Principles With Java

Prakash Badhe

prakash.badhe@vishwasoft.in

About Me : Prakash Badhe

❑Enterprise Technologies Consultant-Trainer for 20+ years
❑Passion for technologies and frameworks
❑Proficient in Java, JavaScript Frameworks and UI libraries
❑Proficient with Enterprise implementation standards
❑Supports agile technical practices with agile development environment.

About you..

- Job Role
- Skill-Sets
- Objectives
- Prior experience with Java, Design.
- Exposure to database applications with SQL commands.

Agenda in short

- Design Principles with java
- Java concurrency
- Java streams
- Java reflection API
- Garbage collector optimization
- GOF Design Patterns
- Database optimization
- SQL Queries

Set UP

- Windows 7 / 10 64 bit
- JDK 1.8
- Eclipse-Jee-Mars
- Adobe PDF Reader
- MySQl 5.1 Database server
- MySql Workbench client 6.1

Application Development With Java

- Understand the requirements.
- Design application architecture.
- Define java classes,interfaces,components one by one.
- Test them one by one.: Unit testing
- Integrate them together.(combine, build and release)
- Test with Integration tests.
- Acceptance Testing in final release stage

Design Guidelines

- Java naming conventions.
- Coding guidelines about variables scope, structure and methods.
- Proper error handling blocks.
- Composition over   inheritance.
- Expose private members properly.
- Avoid duplicate names in side methods.
- Design Principles : Guide about design practices

SOLID Design Principles

- **S**ingle responsibility principle
- **O**pen-closed principle.
- **L**iskov substitution principle
- **I**nterface segregation principle
- **D**ependency inversion principle

SOLID History

- Introduced by Robert C. Martin in his paper *Design Principles and Design Patterns in year 2000.*
- Principles of Object Oriented Design.
- Intended to make software designs more understandable, flexible and maintainable.
- Solves the problems faced by design of object oriented applications.

Case of Multiple Responsibilities

Swiss Knife : Multi-purpose knife with multiple tools.

Versatile, Multipurpose , all in one..

- Should a person be versatile..suitable for any kind of tasks..?
- How it manages then the tasks or updates..?
- Can there be a single java class containing all the different methods (behaviors) defined within it.
- How to make reusable only certain of features of this class ..?
- How to add more behaviors without disturbing existing implementations..?

Problems with Single Versatile Class

- How to make reusable only certain features of this class ..?
- How to add more behaviors without disturbing existing implementations..?
- How to update multiple versions of same behavior ?
- How to make the design maintainable.?
- How to refactor the existing behaviors..?
- What about the performance ..?

Single Responsibility Principle

- Robert C. Martin describes this as **one class should have only one and only responsibility**.
- Accordingly there should be only one reason due to which a class has to be changed.
- It means that a class should have only one task to do.
- This principle is often termed as subjective.

Example Scenario

- DataService class
  - Manage connections to a database
  - Reads some data from database tables
  - Writes the data to a file.
  - Prints some useful information
- The class has multiple reasons to change, and few of them are the modification of file output, new data base adoption.
- When talking about *single principle responsibility*, there are too many reasons for the class to change; hence, it doesn’t fit properly in the single responsibility principle.

More Examples on SR Principle

- An *Automobile* class has start or stop itself but the task of washing it belongs to the *CarWash* class.
- *Book* class has operations to store/update its own name and text. But the task of printing the book must belong to the *BookPrinter* class.
- The BookPrinter class above might print to console or another medium but such dependencies are NOT needed in the Book class

SR Implementation

- Identify/categorize the behaviors you want to implement and isolate from other classes.
- In case of multiple ***unrelated*** behaviors, separate them into different classes.
- Still..needed to access those multiple behaviors..access them in top level *composite* class.
- There should be ONLY one treason for the class to change/update.

The responsibilities of logger

- The log4j logging library for kava applications has different classes with each logging methods (warn, error etc.) and different classes for the purpose of setting the logging levels and so on.

Case study : SR

Implement a EmailSetting class wherein the user can change the settings but before that the user has to be authenticated.

```
public class EmailSetting
{
public void changeEmail(User user)
{
 if(checkAccess(user))
{
  //Allow to change
 }
}
public boolean checkAccess(User user)
{
   //Verify that the user has logged in with valid credentials.
}
}
```

Issues and Solution

- To reuse the checkAccess code at some other place OR to make changes to the way checkAccess is being done.
- In these 2 cases you would end up changing the same class and in the first case you would have to use EmailSetting to check for access as well.
- Solution : Decompose the EmailSetting into UserEmailSetting and SecurityService. And move the checkAccess code into SecurityService.

Responsible : EmailSettings

```
public class  EmailSetting
{
Public void changeEmail(User user)
 {
 if(SecurityService.checkAccess(user))
 {
  //Grant option to update
  }
 }
}
```

Responsible : Security Validator

```
public class SecurityService
{
  public static boolean checkAccess(User user)
  {
    //code to check the access.
  }
}
```

Advantage

- When the Single Responsibility Principle is followed, testing is easier.
- With a single responsibility, the class will have fewer test cases.
- Less functionality also means fewer dependencies to other classes.
- It leads to better code organization since smaller and well-purposed classes are easier to search.

Contradictory Scenarios

- GUI class : With UI related operations like handle,update,chnagetitle,print etc.
- Database class : Read and write operations with sql queries.
- Here the multiple behaviors are categorized in groups as belonging to individual group responsibilities.
- But such kind of practice should be avoided.

Quiz Time

- The UI should display information and allow updates about the user information.
- How to assign the responsibilities ..?

Next Quiz : SR

- User is an entity with properties as id, name, password and salary. The application has to provide the User values storage and retrieval from database.
- To access a resource the User has to first login with id and password to validate.
- New User has to register with new id and password parameters.
- Define set of classes to implement the application with responsibilities specified.

Design Issues

- Product has features related to usage and recycling. The custom products extend the Product and override the features.
- For adding more features to the products like driveAutomatic and driveManual should the base Product be modified ..? Or the custom derived product should add more features.
- The Product may be used in different applications with different versions and implementations.

Open Closed Principle

- Robert C. Martin describes it as *Software components should be open for extension, but closed for modification.*
- The class should be written in such a manner that it performs its jobs without the need to modify it in the future.
- The class should remain closed for modification, but it should have the option to get extended.
- Extending the class includes:
  - Inheriting from the class
  - Overwriting the required behaviors from the class
  - Extending certain behaviors of the class

Example : Open Closed Principle

- Browsers are defined to render the html response from web servers.
- For adding support for displaying PDF documents in browsers, you donot need to modify the browser source code and rebuild it.
- Instead add the extension to the browser that adds new features to display documents.
- Similar to this, other extensions for the browsers are available.

Software components : OC principle

- Typical Database service classes provide basic features for DB connections, DB Query etc.
- The application classes extend it to add new features and re-use existing features such as manage connections, query processing etc.
- The application classes should be designed such a way that whenever developers wants to change the flow of control in specific conditions in application, all they need to is extend the class and override some functions and that’s it.

OC Examples

- The *Action* class in Struts1.X framework is to be extended to have custom processing while re-using the basic functions from the base class.
- The Front Controller **DispatcherServlet** in Spring Web MVC applications, is the central dispatcher for HTTP request handlers/controllers, e.g. for web UI controllers or HTTP-based remote service exporters.; extends from GenericServlet, HttpServlet and FrameworkServlet classes.

Quiz Time

- Why the base classes should be closed for modification..?
- What is extended behavior for children classes ?
- Identify examples of Open Closed principle in your applications.
- For database Entity classes ..i.e. Employee class has variables like id, name, email and salary. Should the database connection logic added into the same Employee class or other service class..? Justify your answer..

Quiz exercise

- Mobile smart phones with different set of features are to be sold online with different set of discounts for each phone. The price discount is based on the features, latest updates and location of users.
- Design the structure of classes with hierarchies and responsibilities clearly defined.

Banking Application Issues

- Defined class SavingAccount with certain features like withdraw, deposit methods.
- Define LoanAccount class extending SavingAccount and overriding the methods.
- The LoanAccount should not allow to withdraw amount.
- If the LoanAccount is used as SavingAccount then, the action for withdraw will fail..it does not represent the base completely!
- They are having different behavior..should have been designed separately with common base as Account.

Another Mismatch..

- Automobile Petrol **Car** has features related to engine for fuel management and exhaustion.
- The New Car **ElectricCar** may have different behavior related to fuel management and exhaustion and hence cannot fit as derived type of Car.
- Both of above should be defined as separate classes and if required can extend from common base type or implement common interface types, but not as base and derived types..

Liskov’s Substitution Principle

- **The Derived types must be completely substitutable for their base types.**
- Applies to interface, abstract and concrete base classes as well.
- This avoids misusing inheritance!.

Liskov’s requirement

- The classes created by extending the base class should be able to fit in application without failure.
- This requires the objects of the subclasses to behave in the same way as the objects of the super-class.
- To avoid mismatch make run time type identification and then cast it to appropriate reference type.

Liskov’s Example cases

- Classic example Square class extends Rectangle and represents the Rectangle.
- Can the Circle act as Rectangle..?
- They can extend or implement common behavior related to **Shape.**

Another case of Circle..

- The **RoundType** is a type of ***ellipse*** but RoudTypes don’t have two foci or major/minor axes.
- Can the RoundType act as Ellipse ..?
- They can extend or implement common behavior related to CurvedObject.

Liskov’s ensure..

- Helps to conform to the “is-a” relationship between the base and derived types.
- Avoids the mismatch/failures at run time.
- Make the design more simple and extendible and maintainable.

Quiz Time

- Define entities as *CarInsurance* and *HouseInsurance, PersonInsurance* and *TruckInsurance* in insurance application with features as startAssurance, depreciateValue and claimAmount.
- Can these defined as chain of inherited/extended features ?
- Can they have common *identity* by way of extension/implementation..?

Problem of multiple expectations…

- In previous case if we define a common interface *InsurnaceProvider* specifying different methods for insurance like insureCar, insureTruck and insureHouse, insurePerson etc.
- To implement *PersonInsurance* we can implement insurePerson but what about other methods like insureCar,insureTruck and insureHouse ..they still have to be implemented..

Interface Segregation Principle

- **Clients should not be forced to implement unnecessary methods which they will not use**.
- It is applicable to interfaces as single responsibility principle holds to classes.
- One interface -> one category of specifics and NOT multiple specified operations.

Case of Segregation and Responsibilities

- To generate report with values from database , files or from URL in different formats like Excel, Pdf, CSV define the set of classes and interfaces to specify the report formats.

Another case of Segregation

- BookPrinter has to print the Book objects to console or other Medias like file or paper.
- Identify the interfaces and implement them with only the features required.

Tight coupling with Dependencies

- The application class *App* is using *OracleService* to manage the connections with OracleDatabase and output the report with PdfWriter class.
- The App class is a composite using OracleService and PdfWriter object references to execute the activities.

OracleService

```
Public Class OracleService {
 private Connection;
Public Connection initialize()
{
  //load the DB driver and initialize the
    connection and return.
}
}
```

PdfWriter

```
Public Class PdfWriter {
Public void write()
 {
  // generate the text to pdf and write it
 }
}
```

Class Application

```
Public Class App  {
 Private OracleService service ;
 Private PdfWriter writer ;
Public App(OracleService obj1, PdfWriter obj2) {
    service= obj1;
  writer = obj2;
 }
Public void readFromDB() {
 //read with OracleService
 }
Public void printed() {
 //write to pdf
 }
}
```

Issues with hard coded dependencies

- How to configure the App class with SqlService (MS SQL Database) and ExcelWriter to generate excel reports from MS SQL Server data records ...?
- Modify the App ..!
- How about existing users using App with OracleService and PdfWriter..?

Dependency Inversion Principle

- **Dependency Inversion refers to the decoupling of software modules/components.**
- Instead of hard coded dependencies specifying the dependent name directly , define the dependencies with abstractions.
- This suggests loose coupling between the parent and dependent classes.

Define abstraction with interfaces

```
Interface DbService {
  public void initialize();
}
Interface Writer {
  public void write();
}
```

Service classes

```
Class OracleService implements DbService {
public void initialize();
}
Class SQLService implements DbService {
public void initialize();
}
```

Writer implementations

```
Class  PDFWriter implements Writer {
 public void write() {//write to pdf}
}
Class  FileWriter implements Writer {
 public void write() {//write to file}
}
```

App class with abstraction

```
Public Class App  {
 Private DbService service ;
 Private  Writer writer ;
Public App(DBService obj1, Writer obj2) {
    service= obj1;
    writer = obj2;
 }
Public void readFromDB() {
 //read with DBService
 service.initialize();
 }
Public void printOutput() {
 //write to pdf
  writer.write();
 }
}
```

App with different dependencies

```
App app1 = new App(OracleServiceOBj,
   PdfWriterObj);
  app1.initilize();
  app1.write();
App app2 = new App(SQLServiceOBj,
   FileWriterObj);
   app2.initilize();
   app2.write();
```

DI Examples

- Spring bean dependency configuration for any type of dependent bean classes into any parent with standard XML configuration as well as with annotations.
- Any RDBMS Database connection injection in Hibernate SessionFactory classes to manage the database interactions from application side.

Case study with design principles

- Employees have properties as id, job, workingHours and salary.
- New employees get registered with the Company and get their id values.
- Existing employees having id values are working in Company office.
- For every working employee salary is calculated based on workingHours values.
- Employees with id values up to 1000 are stored in MySQL database while id values greater than 1000 are stored in PostGreSQL database.
- Apply SOLID principles to design the application architecture