♥ Design principles interview questions & answers for Java developers

Posted on September 15, 2014 by Arulkumaran Kumaraswamipillai — 1 Comment ↓

Q1. What are the SOLID design principles?

A1. **SOLID** is an abbreviation for 5 design principles.

SRP	Single Responsibility Principle	A class should only have a single purpose (i.e. cohesive), and all its methods should work together to achieve this goal.
OCP	Open Close Principle	You should be able to extend a class' behavior, without modifying it.
LSP	Liskov Substitution Principle	Derived classes must be substitutable for their base classes. Derived classes must have the same intent, but different implementation. Another way to look at this principle is to think of design by contract. A sub class should honor the contracts made by its parent classes.
ISP	Interface Segregation Principle	Make fine grained interfaces that are client specific. While SRP addresses high cohesion in the class level, ISP promotes high cohesion in the interface level.
DIP	Dependency Inversion Principle	Depend on abstractions, not on implementations. • Higher level modules should not depend directly on lower level modules. Both should depend on abstractions (interfaces or abstract classes). • Abstractions should not depend on implementations. The implementations should depend on abstractions.

SOLID design principles

SRP (Single Responsibility Principle)

If you have a class with calculation logic, validation logic, data access logic and display logic all mixed up then it violates the SRP principle. This makes it difficult to change one part without breaking others. Mixing responsibilities also makes the class harder to understand, harder to test, and increases the risk of duplicating logic.

OCP (Open Closed Principle)

means you extend the behavior of a class without modifying the class but by adding new classes. A typical sign of violating the OCP is having large if/else or switch statements.

LSP (Liskov Substitution Principle)

As a Java Architect

Java architecture & design concepts

Mathematically, a square **is a** rectangle. But in OOD, but when modelled in code, it violates the LSP. A square has only *setWidth(double width)* method, whereas a rectangle has both *setWidth(double width)* and *setLength(double length)* methods. What should setWidth(double width) do when called on a Square? Should it set the height as well? What if you have a reference to it via its parent class, Rectangle?

The best way to reduce LSP violations is to keep very aware of the LSP whenever using **inheritance**, and consider avoiding the problem by favoring **composition** (i.e has-a) where appropriate over inheritance (i.e. is-a).

ISP (Interface Segregation Principle)

simply means keep your interfaces small and cohesive. If you have a fat interface with lots of abstract methods, then you are imposing a huge implementation burden on any class that wants to adhere to that contract. Worse still is that there is a tendency for class to only provide valid implementations for a subset of a fat interface, which defeats the benefits of having an interface. What is the point in having a bulky contract that implementers don't adhere to?

DIP (Dependency Inversion Principle)

The DIP says that if a class has dependencies on other classes, it should rely on the dependencies' interfaces rather than their concrete types. Code to interface, not implementation.

Q2. Can you explain why design principles are only a guide?
A2. It is bad to over design. Design is all about trade offs. You will find some principles contradict with each other. For example, OCP requires interface inheritance, but LSP discourages inheritance and favors composition. Over use of SRP can lead to too many fine grained classes. So, design principles help you have a balance by asking the right

interview Q&As with diagrams | What should be a typical Java EE architecture?

Senior Java developers must have a good handle on

open all | close all
⊞ Best Practice (6)
⊞ Coding (26)
⊞ Concurrency (6)
⊞ Design Concepts (7)
⊞ Design Patterns (11)
⊞ Exception Handling (3
⊞ Java Debugging (21)
⊞ Judging Experience In
⊞ Low Latency (7)
⊞ Memory Managemen
⊞ Performance (13)
⊞ QoS (8)
⊞ Scalability (4)
⊞ SDLC (6)
⊞ Security (13)
⊞ Transaction Managen

80+ step by step Java Tutorials

open all | close all
⊞ Setting up Tutorial (6)
⊞ Tutorial - Diagnosis (2
⊞ Akka Tutorial (9)
⊞ Core Java Tutorials (2
⊞ Hadoop & Spark Tuto

questions. Do you really need a huge interface explosion due while adhering to the OCP and DIP? Do you really need fine granularity of objects or better to have coarse granularity?

Q3. Is there anything wrong with the following class design? If yes, can the design be improved?

```
1  import javax.management.RuntimeErrorException;
2  import org.apache.commons.lang.StringUtils;
3
4  public class MathOperation {
5
6   public int operate(int input1, int input2, Stri
7
8     if(StringUtils.isEmpty(operator)){
9        throw new IllegalArgumentException("Invali
10    }
11
12    if(operator.equalsIgnoreCase("+")){
13       return input1 + input2;
14    }
15    else if(operator.equalsIgnoreCase("*")){
16       return input1 * input2;
17    } else {
18       throw new RuntimeException("unsupported op
19    }
20  }
21 }
22
```

A3. It's not a good idea to try to anticipate changes in requirements ahead of time, but you should focus on writing code that is well written enough so that it's easy to change. This means, you should strive to write code that doesn't have to be changed every time the requirements change. This is what the Open/Closed principle is. According to GoF design pattern authors "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification". Spring framework promotes this principle.

In the above example, you can anticipate more operators like "-" (subtraction) and division (/) to be supported in the future and the class "MathOperation" is not closed for modification. When you need to support operators "-" and "%" you need to add 2 more "else if" statements. Whenever you see large if/else or switch statements, you need to think if "Open/Closed" design principle is more suited.

JEE Tutorials (19)
Scala Tutorials (1)
Spring & HIbernate Tu
Tools Tutorials (19)
Other Tutorials (45)

Preparing for Java written & coding tests

open all | close all
♦ Complete the given
Can you write code? (
Converting from A to I
Designing your classe
Java Data Structures
Passing the unit tests
What is wrong with th
Writing Code Home A
Written Test Core Jav
Written Test JEE (1)

How good are your...to go places?

open all | close all
Career Making Know-
Job Hunting & Resum

Let's open for extension and close for modifications

In the rewritten example below, the classes *AddOperation* and *MultiplyOperation* are closed for modification, but open for extension by allowing you to add new classes like *SubtractOperation* and *DivisionOperation* by implementing the *Operation* interface.

Define the interface Operation.

```
1 public interface Operation {
2         abstract int operate(int input1, int input
3 }
4
```

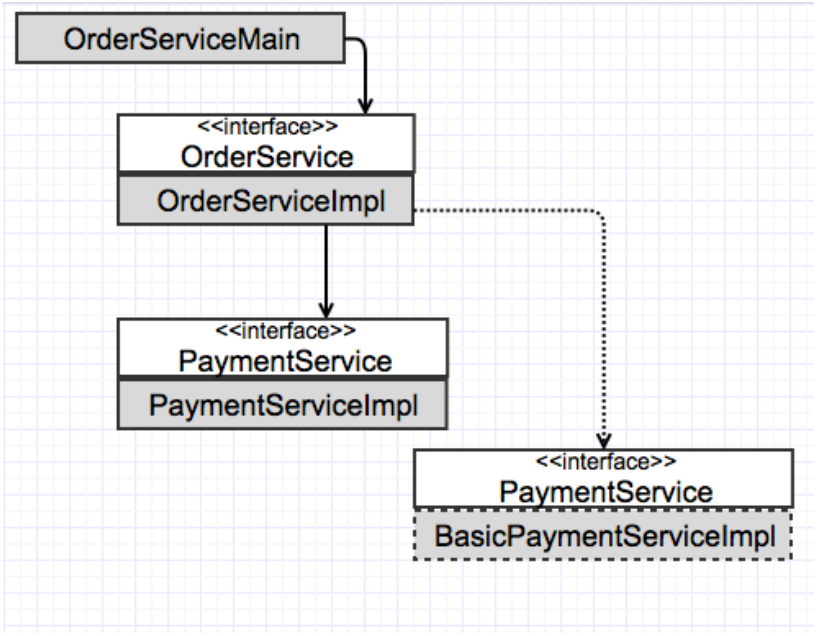
Define the implementations

```
1 public class AddOperation implements Operation {
2
3   @Override
4   public int operate(int input1, int input2) {
5      return input1 + input2;
6   }
7
8 }
9
```

```
1 public class MultiplyOperation implements Operati
2
3   @Override
4   public int operate(int input1, int input2) {
5     return input1 * input2;
6   }
7 }
8
```

Q4. Is DIP related to Dependency Injection (DI)?
A4. Dependency Inversion Principle (**DIP)** is a design principle which is in some ways related to the Dependency Injection (DI) pattern. The idea of DIP is that higher layers of your application should not directly depend on lower layers. Dependency Inversion Principle does not imply Dependency Injection. This principle doesn't say anything about how higher la yers know what lower layer to use. This could be done as shown below by coding to interface using a factory

pattern or through Dependency Injection by using an IoC container like Spring framework, JEE CDI, Guice, etc.



Loosely coupled by coding to interface

Q5. Is there anything wrong with the following class design?

```
1  import java.sql.Connection;
2  import java.sql.SQLException;
3  import org.apache.commons.lang.StringUtils;
4
5  public class Employee {
6
7      private Integer id;
8      private String name;
9      private Connection con = null;
10
11     //getters and setters for above attributes g
12
13     public boolean validate( ) {
14         return id != null && id > 0 && StringUti
15     }
16
17     public void saveEmployee() throws SQLExcepti
18         //save Employee to database using SQL
19         //goes here ...
20     }
21
22     public boolean validateEmployee( )  {
23          //logic to validate an employee
24     }
25 }
26
```

A5. Violates the SRP. The above class has multiple responsibilities like being a domain object, data access object and a validator.

This principle is based on **cohesion**. Cohesion is a measure of how strongly a class focuses on its responsibilities. It is of the following two types:

High cohesion: This means that a class is designed to carry on a specific and precise task. Using high cohesion, methods are easier to understand, as they perform a single task.

Low cohesion: This means that a class is designed to carry on various tasks. Using low cohesion, methods are difficult to understand and maintain.

Hence the above code suffers from **low cohesion**. The above code can be improved as shown below. The Employee class is re-factored to have only a single responsibility of uniquely identifying an employee.

```
1 public class Employee {
2
3     private Integer id;
4     private String name;
5
6     //getters, setters, equals(..), hashCode(), a
7 }
8
9
```

The responsibility of interacting with the database is shifted to a data access object (i.e. DAO) class. The data access object class takes an employee object or any of its attributes as input.

```
1 public interface EmployeeDao {
2    public void saveEmployee(Employee emp);
3 }
```

Provide an implementation for the above interface. Finally, the responsibility of validating an employee is re-factored to a separate class.

```
1 public interface Validator {
2     public boolean validate(Employee emp);
3 }
```

Q6. What is your understanding of some of the following most talked terms like Dependency Inversion Principle (**DIP**), Dependency Injection (**DI**) and Inversion of Control (**IoC)**?
A6. Dependency Inversion Principle (DIP) is a design principle which is in some ways related to the Dependency Injection (DI) pattern. The idea of DIP is that higher layers of your application should not directly depend on lower layers. Dependency Inversion Principle does not imply Dependency Injection. This principle doesn't say anything about how higher layers know what lower layer to use. This could be done as shown in the above example, or through Dependency Injection by using an IoC container like Spring framework, Pico container, Guice, or Apache HiveMind.

Dependency Injection (DI) is a pattern of injecting a class's dependencies into it at runtime. This is achieved by defining the dependencies as interfaces, and then injecting in a concrete class implementing that interface to the constructor. This allows you to swap in different implementations without having to modify the main class. The Dependency Injection pattern also promotes high cohesion by promoting the Single Responsibility Principle (SRP), since your dependencies are individual objects which perform discrete specialized tasks like data access (via DAOs) and business services (via Service and Delegate classes) .

The **Inversion of Control Container (IoC)** is a container that supports Dependency Injection. In this you use a central container like Spring framework, Guice, or HiveMind, which defines what concrete classes should be used for what dependencies through out your application. This brings in an added flexibility through looser coupling, and it makes it much easier to change what dependencies are used on the fly.

The real power of DI and IoC is realized in its ability to replace the compile time binding of the relationships between classes with **binding those relationships at runtime**. For

example, in Seam framework, you can have a real and mock implementation of an interface, and at runtime decide which one to use based on a property, presence of another file, or some precedence values. This is incredibly useful if you think you may need to modify the way your application behaves in different scenarios. Another real benefit of DI and IoC is that it makes your code easier to unit test. There are other benefits like promoting looser coupling without any proliferation of factory and singleton design patterns, follows a consistent approach for lesser experienced developers to follow, etc. These benefits can come in at the cost of the added complexity to your application and has to be carefully manged by using them only at the right places where the real benefits are realized, and not just using them because many others are using them.

Note: The **CDI** (Contexts and Dependency Injection) is an attempt at describing a true standard on Dependency Injection. CDI is a part of the Java EE 6 stack, meaning an application running in a Java EE 6 compatible container can leverage CDI out-of-the-box. Weld is the reference implementation of CDI.

Q7. Can you list some of the key principles to remember while designing your classes?
A7.

1) Favor composition over inheritance.
2) Don't Repeat Yourself (DRY principle). No code or logic duplication.
3) Find what varies and encapsulate it.
4) Code to an interface, and not to an implementation.
5) Strive for loose coupling and high cohesion.
6) Keep It Simple and Stupid (KISS).
7) You Aren't Gonna Need It (YAGNI): Don't add functionality until you need it.
8) Design by contract.

Use design principles and patterns, but use them judiciously. Design for current requirements without anticipating future requirements, and over complicating things. A well designed

(i.e. loosely coupled and more cohesive) system can easily lend itself to future changes, whilst keeping the system less complex.

Q8. What is design by contract?
A8. The design by contract ensures code quality by enforcing that the services offered by classes and interfaces adhere to unambiguous contracts. The design by contract is very useful for designing classes and interfaces. These contracts are like checking for an empty or null string, checking for negative numbers, checking for a particular range, etc.

In general the contracts checked are:

- **Pre-conditions**: These are checked when execution enters a method.
- **Post-conditions:** These are checked when execution exits a method.
- **Class invariants:** These must hold at every observable point.

Contracts are inherited, just like methods. When a method is overridden by a subclass, the subclass may specify its own contracts. This means a subclass may “weaken the pre-condition and strengthen the post-condition”.

```
1  import java.math.BigDecimal;
2
3  public class Funds {
4
5      private BigDecimal result = null;
6      //...
7
8      public BigDecimal calcRate(double rate) {
9          //pre-condition check
10         if (rate <= BigDecimal.ZERO.doubleValue(
11                         || rate > BigDecimal.TEN
12             throw new IllegalArgumentException(
13
14         }
15
16         // ... logic to calculate the result
17
18         //post-condition check
19         if (this.result == null
20                 || this.result.doubleValue( ) <=
21                         .doubleValue( )) {
22             throw new RuntimeException("Invalid
```

```
23          }
24          return result;
25      }
26
27      //...
28  }
```

Firstly, in the above code snippet the parameter "*rate*" is an **invariant** that must hold true. The Funds object's calculation of the result could be incorrect if an invalid "rate" is passed as an argument by its caller. A **precondition** validation is performed on the "rate" to ensure that the caller fulfills the requirements. Secondly, the implementation of the logic to calculate the result could be defective. So a **post-condition** validation is performed to verify the correctness of internal logic.

There are a number of frameworks that support "**design by contract**" like OVal, Jass, etc. The OVal is a validation framework that lets you place the conditions in different ways like annotations, XML, POJO, or scripting languages like Groovy, JavaScript, OGNL (Object Graph Navigation Language – an expression language for getting and setting properties of Java objects), etc and also partly implements programming by contract.

Popular Posts

♦ 11 Spring boot interview questions & answers

861 views

♦ Q11-Q23: Top 50+ Core on Java OOP Interview Questions & Answers

829 views

18 Java scenarios based interview Questions and Answers

448 views

001A: ♦ 7+ Java integration styles & patterns interview questions & answers

407 views

♦ 7 Java debugging interview questions & answers

311 views

♦ 10 ERD (Entity-Relationship Diagrams) Interview Questions and Answers

303 views

01b: ♦ 13 Spring basics Q8 – Q13 interview questions & answers

294 views

01: ♦ 15 Ice breaker questions asked 90% of the time in Java job interviews with hints

288 views

♦ Q24-Q36: Top 50+ Core on Java classes, interfaces and generics interview questions & answers

263 views

8 Git Source control system interview questions & answers

215 views

Bio | Latest Posts

Arulkumaran Kumaraswamipillai

Mechanical Eng to freelance Java developer in 3 yrs. Contracting since 2003, and attended 150+ Java job interviews, and often got 4 - 7 job offers to choose from. It pays to prepare. So, published Java interview Q&A books via Amazon.com in 2005, and sold 35,000+ copies. Books are outdated and replaced with this subscription based site.

About Arulkumaran Kumaraswamipillai

Mechanical Eng to freelance Java developer in 3 yrs. Contracting since 2003, and attended 150+ Java job interviews, and often got 4 - 7 job offers to choose from. It pays to prepare. So, published Java interview Q&A books via Amazon.com in 2005, and sold 35,000+ copies. Books are outdated and replaced with this subscription based site.

‹ JNDI and LDAP interview Q&A QoS interview Q&A ›

Posted in Design Concepts, OOP

Tags: Architect FAQs

One comment on "♥ Design principles interview questions & answers for Java developers"

anjaneyareddym says:

March 17, 2015 at 6:36 pm

Well understanding on SOLID design principle.

Reply

Leave a Reply

Logged in as geethika. Log out?

Comment

Post Comment

Empowers you to open more doors, and fast-track

Technical Know Hows

☀ Java generics in no time ☀ Top 6 tips to transforming your thinking from OOP to FP ☀ How does a HashMap internally work? What is a hashing function?
☀ 10+ Java String class interview Q&As ☀ Java auto un/boxing benefits & caveats ☀ Top 11 slacknesses that can come back and bite you as an experienced Java developer or architect

Non-Technical Know Hows

☀ 6 Aspects that can motivate you to fast-track your career & go places ☀ Are you reinventing yourself as a Java developer? ☀ 8 tips to safeguard your Java career against offshoring ☀ My top 5 career mistakes

Prepare to succeed

☀ Turn readers of your Java CV go from “Blah blah” to “Wow”? ☀ How to prepare for Java job interviews? ☀ 16 Technical Key Areas ☀ How to choose from multiple Java job offers?

Select Category ▼

© Disclaimer

 ↑