Register | Login | Logout | Contact Us

Java-Success.com

Industrial strength Java/JEE Career Companion for those who want to go places

search here … Go

Home | Java FAQs | 600+ Java Q&As | Career | Tutorials | Member | Why?

Can u Debug? | Java 8 ready? | Top X | Productivity Tools | Judging Experience?

Home › Tech Key Areas › 13 Technical Key Areas Interview Q&A › Design Concepts › How to create a well designed Java application?

How to create a well designed Java application?

Posted on August 11, 2014 by Arulkumaran Kumaraswamipillai — No Comments ↓

A software application is built by coupling various classes, modules, and components. Without coupling, you can't build a software system. But, the software applications are always subject to changes and enhancements. So, you need to build your applications such a way that they can not only adapt to growing requirements, but also are easy to maintain and understand. 3 key aspects to achieve this are

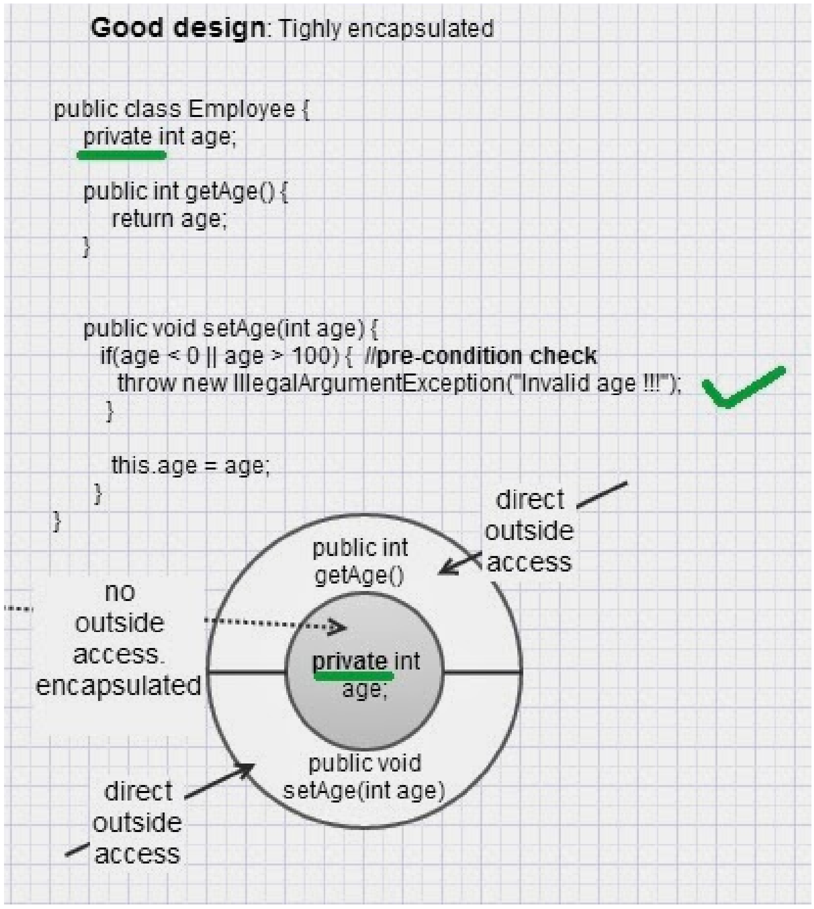
1. Tight **encapsulation**.
2. Loose (or low) **coupling**.
3. High **cohesion**.

The purpose of SOLID design principles and GoF (GangOfFour) design patterns is to achieve the above goal.

Q1. What is encapsulation?

A1. Encapsulation is about hiding the implementation details. This means

1) **Hiding data** by setting member variables with private access level and providing public methods with pre and post condition checks to access the member variables.


Good design: Tighly encapsulated
direct outside access
public int getAge()
no outside access. encapsulated
private int age;
public void setAge(int age)
direct outside access

```
public class Employee {
  private int age;

  public int getAge() {
    return age;
  }


  public void setAge(int age) {
   if(age < 0 || age > 100) {  //pre-condition check
     throw new IllegalArgumentException("Invalid age !!!");
   }

    this.age = age;
  }
}
```

Good design: Tightly Encapsulated

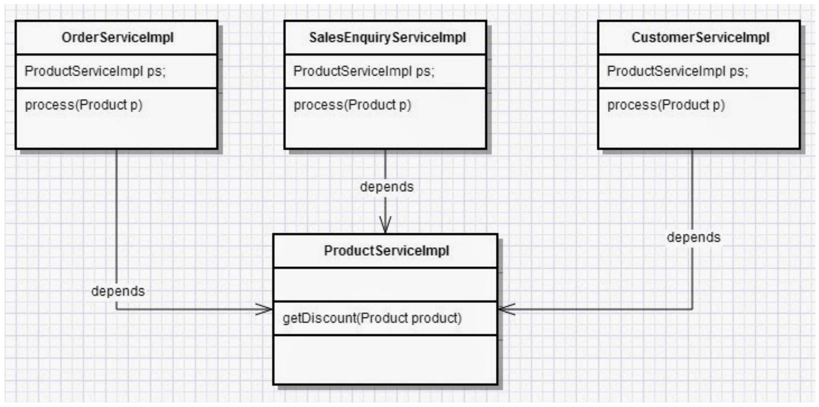
2) **Coding to an interface and not implementation**: Using a given class or module by its interface, and not needing to know any implementation details.

Badly Encapsulated: Directly dependent on *ProductServiceImpl*

♦ Why favor com
08: ♦ Write code
Explain abstracti
How to create a
Top 5 OOPs tips
Top 6 tips to go a
Understanding C
What are good r
GC (2)
Generics (5)
FP (8)
IO (7)
Multithreading (12)
Algorithms (5)
Annotations (2)
Collection and Data
Differences Betwee
Event Driven Progr
Exceptions (2)
Java 7 (2)
Java 8 (24)
JVM (6)
Reactive Programm
Swing & AWT (2)
JEE Interview Q&A (3
Pressed for time? Jav
SQL, XML, UML, JSC
Hadoop & BigData Int
Java Architecture Inte
Scala Interview Q&As
Spring, Hibernate, & I
Testing & Profiling/Sa
Other Interview Q&A f
▶ Free Java Interview

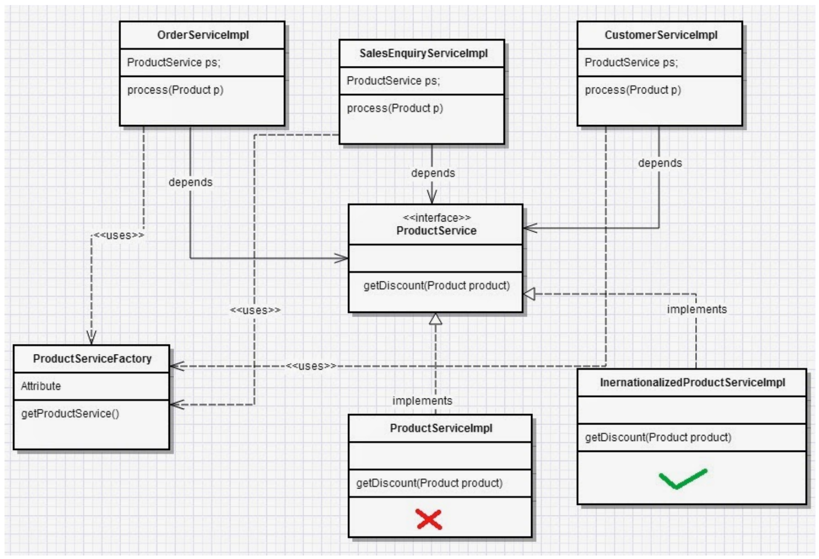
As a Java Architect

Java architecture & design concepts



Loosely Encapsulated

Tightly Encapsulated: Dependent on the interface ProductService. The implementation can change from *ProductServiceImpl* to InternationalizedProductServiceImpl without the callers or clients *OrderServiceImpl*, *SalesEnquiryServiceImpl*, and *CustomerServiceImpl* needing to know about the implementation details.



Tightly Encapsulated

SOLID Principle: LSP (Liskov substitution principle), DIP (Dependency inversion principle), and SRP (Single Responsibility Principle) strive to create tight encapsulation.

Q2. Does tight encapsulation promote loose coupling?
A2. Yes, poor encapsulation can allow tight coupling as it will be possible to access non-encapsulated variables directly

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 I
- ⊞ Low Latency (7)
- ⊞ Memory Managemen
- ⊞ Performance (13)
- ⊞ QoS (8)
- ⊞ Scalability (4)
- ⊞ SDLC (6)
- ⊞ Security (13)
- ⊞ Transaction Managem

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

from another class, making it dependent on the implementation of the class that contains those variables. It is also possible to have poor encapsulation with loose coupling. So, encapsulation and coupling are different concepts, but poor encapsulation makes it possible to create tight coupling.

Q3. What is coupling?
A3. Coupling in software development is defined as the degree to which a module, class, or other construct, is tied directly to others. You can reduce coupling by defining intermediate components (e.g. a factory class or an IoC container like Spring) and interfaces between two pieces of a system. Favoring composition over inheritance can also reduce coupling.

Let’s write some code for the above UML diagram to demonstrated loose coupling with GoF factory design pattern.

Step 1: Define the interfaces for the above diagram

```

public interface OrderService {
    abstract void process(Product p);
}

```

```

import java.math.BigDecimal;

public interface ProductService {
    abstract BigDecimal getDiscount(Product p);
}

```

Step 2: Define the implementation class for ProductService

```

public class ProductServiceImpl implements Produ

  @Override
  public BigDecimal getDiscount(Product p) {
     //logic goes here
     return null;
```

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

```
   }

}

```

Step 3: Factory eliminate need to bind application specific classes into the code. The code only deals with the ProductService interface , and can therefore work with any user-defined concrete implementations like ProductServiceImpl or InternationalizedProductServiceImpl.

```

public final class ProductServiceFactory {

   private ProductServiceFactory(){}

   public static ProductService getProductServic
      ProductService ps = new ProductServiceImpl
      return ps;
   }
}

```

Step 4: The caller or client class who is loosely coupled to ProductService via the ProductServiceFactory.

```

import java.math.BigDecimal;

public class OrderServiceImpl implements OrderSe

   ProductService ps;

   @Override
   public void process(Product p) {
     ps = ProductServiceFactory.getProductServic
     BigDecimal discount = ps.getDiscount(p);
     //logic
   }

}


```

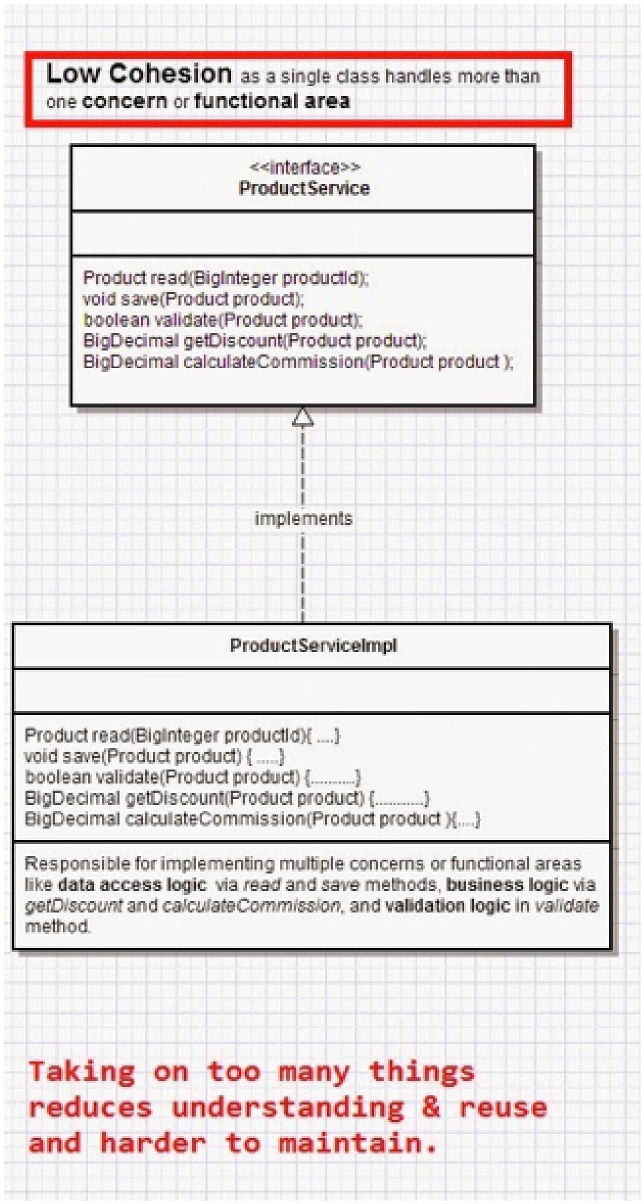
So, if you want to change over from ProductServiceImpl to InternationalizedProductServiceImpl, all you have to do is make a change in the ProductServiceFactory class. An alternative approach to using a factory is to use an IoC container like Spring instead of a factory class to loosely wire the dependencies via dependency injection at run time.

Q4. How do you create loose coupling?
A4. By abstracting many of the implementation needs into various interfaces and introducing the concepts of OCP and DIP, you can create a system that has very low coupling.

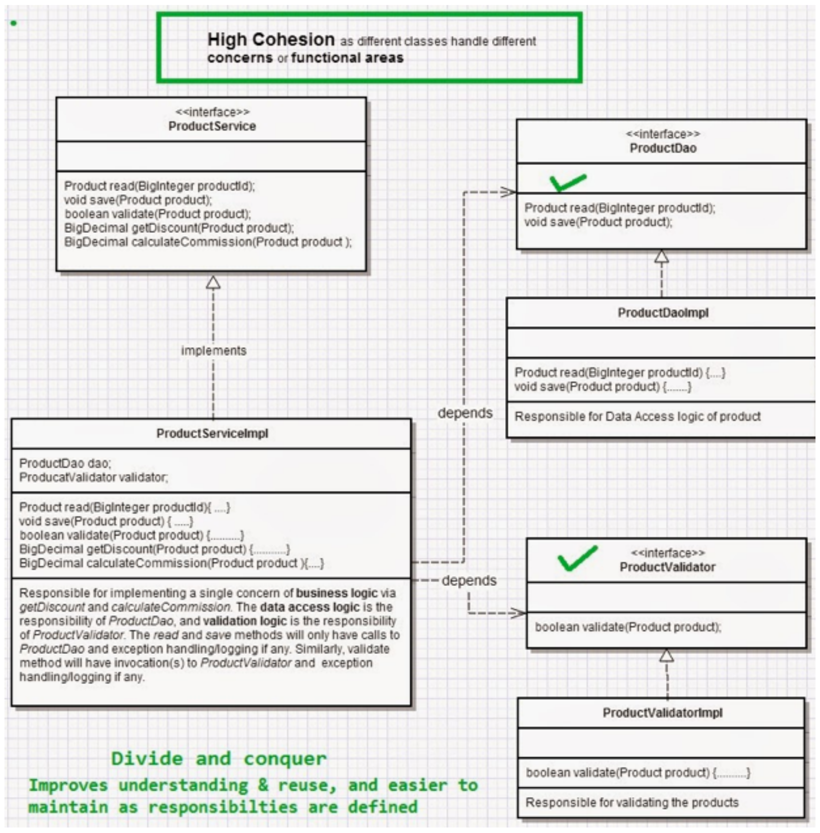
1. Open/Closed Principle (OCP) from the SOLID OO principles promotes loose coupling
2. Favor composition over inheritance for loose coupling
3. Dependency Inversion Principle for loose coupling
4. Event Driven Programming loosely couples producer and consumer

Q5. What is cohesion?
A5. Cohesion is the extent to which two or more parts of a system are related and how they work together to create something more valuable than the individual parts. You don't want a single class to perform all the functions (or concerns) like being a domain object, data access object, validator, and a service class with business logic. To create a more cohesive system from the higher and lower level perspectives, you need to break out the various needs into separate classes.

Coupling happens in between classes or modules, whereas cohesion happens within a class.


Low Cohesion as a single class handles more than one concern or functional area
<<interface>>
ProductService
Product read(BigInteger productId);
void save(Product product);
boolean validate(Product product);
BigDecimal getDiscount(Product product);
BigDecimal calculateCommission(Product product);
implements
ProductServiceImpl
Product read(BigInteger productId){ ....}
void save(Product product) { .....}
boolean validate(Product product) {..........}
BigDecimal getDiscount(Product product) {...........}
BigDecimal calculateCommission(Product product){....}
Responsible for implementing multiple concerns or functional areas like data access logic via read and save methods, business logic via getDiscount and calculateCommission, and validation logic in validate method.
Taking on too many things reduces understanding & reuse and harder to maintain.

Low cohesion as a single class does many things


High Cohesion as different classes handle different concerns or functional areas
<<interface>>
ProductService
Product read(BigInteger productId);
void save(Product product);
boolean validate(Product product);
BigDecimal getDiscount(Product product);
BigDecimal calculateCommission(Product product);
implements
ProductServiceImpl
ProductDao dao;
ProducatValidator validator;
Product read(BigInteger productId){ ....}
void save(Product product) { .....}
boolean validate(Product product) {..........}
BigDecimal getDiscount(Product product) {...........}
BigDecimal calculateCommission(Product product){....}
depends
depends
<<interface>>
ProductDao
Product read(BigInteger productId);
void save(Product product);
ProductDaoImpl
Product read(BigInteger productId) {....}
void save(Product product) {.......}
Responsible for Data Access logic of product
<<interface>>
ProductValidator
boolean validate(Product product);
ProductValidatorImpl
boolean validate(Product product) {.........}
Responsible for validating the products
Divide and conquer
Improves understanding & reuse, and easier to maintain as responsibilties are defined

Responsible for implementing a single concern of **business logic** via *getDiscount* and *calculateCommission*. The **data access logic** is the responsibility of *ProductDao*, and **validation logic** is the responsibility of *ProductValidator*. The *read* and *save* methods will only have calls to *ProductDao* and exception handling/logging if any. Similarly, validate method will have invocation(s) to *ProductValidator* and exception handling/logging if any.

High Cohesion as each class focus on one thing

Q6. How do you create high cohesion?
A6. You can get higher cohesion with a combination of low coupling and SRP (Single Responsibility Principle from SOLID), which allows you to stack a lot of small pieces (e,g. Service, DAO, Validator, etc) together like puzzles to create something larger and more complex.
So, think, tight encapsulation, loose (low) coupling, and high cohesion.

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.

‹ ♦ Why favor composition over inheritance? a must know interview question for Java developers

Top 6 tips to go about writing loosely coupled Java applications ›

Posted in Design Concepts, Judging Experience Interview Q&A, member-paid, OOP

Tags: Architect FAQs

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