Table of Contents

[Design patterns 2](#_Toc470364416)

[Strategy Design pattern 2](#_Toc470364417)

[State Design Pattern 5](#_Toc470364418)

[Difference/Similarity between Strategy and State design patterns. 5](#_Toc470364419)

[Decorator Design pattern 5](#_Toc470364420)

[Adapter Design Pattern 5](#_Toc470364421)

[Singletone Designpattern 5](#_Toc470364422)

[Different latest options of Singletone design pattern. 5](#_Toc470364423)

[Real World Example of Design Patterns 5](#_Toc470364424)

[Spring Related 9](#_Toc470364425)

[DI concepts and how to achieve 9](#_Toc470364426)

[Constructor based dependency injection – different examples. 9](#_Toc470364427)

[Setter based Dependency injection – different examples. 9](#_Toc470364428)

[Static factory based dependency injection. 9](#_Toc470364429)

[Auto wiring concepts and examples. 9](#_Toc470364430)

[Bean scopes 9](#_Toc470364431)

[Custom Scope 10](#_Toc470364432)

[AOP concepts and examples. 12](#_Toc470364433)

[Property file loading and how to use that in java code. 12](#_Toc470364434)

[Message source by property file with example. 12](#_Toc470364435)

[Web service client part using JAXB as marshalling. 12](#_Toc470364436)

[Web service exposer example 12](#_Toc470364437)

[Wiring using annotations, component scan and stereotypes. 12](#_Toc470364438)

[JDBC template to do insert, update etc. example 12](#_Toc470364439)

[DAO using Hibernate Dao Support and generic Dao pattern with ready example 12](#_Toc470364440)

[ProxyFactoryBean to initialize spring bean in runtime without loading application context. 12](#_Toc470364441)

[Event publishing 12](#_Toc470364442)

[Lifecycle callbacks 12](#_Toc470364443)

[Pre and post initiallization. 12](#_Toc470364444)

[Initializing Spring beans for the first layer class where application context has not loaded by framework. 12](#_Toc470364445)

[Design Patterns being used in Spring framework 12](#_Toc470364446)

[General topics 13](#_Toc470364447)

[Eh cache example for configuration to maintain different cache effectively. 13](#_Toc470364448)

[Make asynchronous call and respond as synchronous one. 13](#_Toc470364449)

[Maven profile and classifier to generate different named artifact. 13](#_Toc470364450)

[Process Related 13](#_Toc470364451)

[How to manage defects using HP ALM and what are the lifecycle we followed etc. 13](#_Toc470364452)

[JIRA. What are the process of delivering story etc. 13](#_Toc470364453)

[General day to day activity related to projects like Daily calls, Scrum calls etc. 13](#_Toc470364454)

# Design patterns

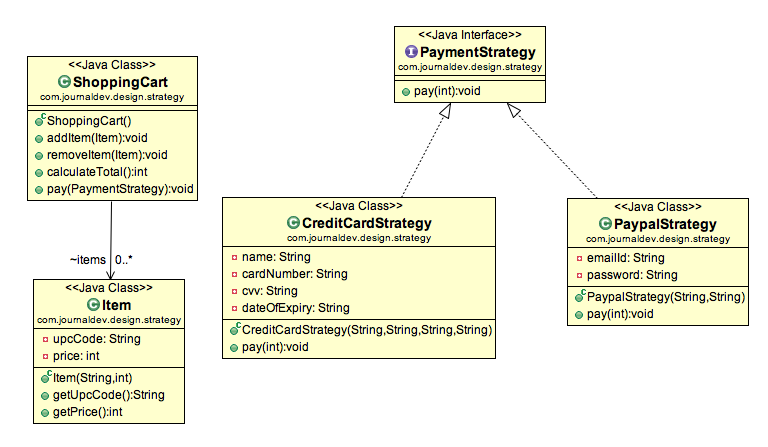
## Strategy Design pattern

Strategy Design Patterns is one the behavioral design pattern we use frequently in programming.

**What it is** – This design pattern is used when we have multiple related algorithms (read Strategies) for a specific task and client can choose any one of the algorithm to do its job and the supplies the real implementation to the actual method and pass it to the common generic method ( called Context).

**Real world example** – Collections.sort(list, comparator) method where client actually passes suitable comparator based on the requirement in runtime to the method and the method is generic to accept any comparator type. Based on the comparator being passed, same collection can be sorted differently.

**Based on Open/closed principle** – like we don’t need to modify the context [closed for modification] and can choose and add any implementation [open for extension]. In real example Colelctions.sort – we don’t need to change the sort method to achieve different sort and supplying different comparator in runtime.



**Code example in java**

**package** strategy;

**public** **interface** PaymentStrategy {

**boolean** pay(**int** value);

}

Strategy Interface

**package** strategy;

**public** **class** CCStrategy **implements** PaymentStrategy {

**public** **boolean** pay(**int** value) {

System.*out*.println("Doing payment with CC of amount " + value);

**return** **true**;

}

@Override

**public** String toString() {

**return** "CCStrategy";

}

}

Credit Card Strategy

**package** strategy;

**public** **class** PaytmStrategy **implements** PaymentStrategy {

**public** **boolean** pay(**int** value) {

System.*out*.println("Doing payment with paytm wallet. amount is " + value);

**return** **true**;

}

@Override

**public** String toString() {

**return** "PaytmStrategy";

}

}

Paytm Strategy

**package** strategy;

**public** **class** BusinessContext {

PaymentStrategy paymentStrategy;

**public** BusinessContext(PaymentStrategy paymentStrategy) {

**this**.paymentStrategy = paymentStrategy;

}

**public** **boolean** buy(**int** quantity, **int** price) {

**int** amountToPay = price \* quantity;

**boolean** result = paymentStrategy.pay(amountToPay);

System.*out*.println("Amount of " + amountToPay + " paid using " +

paymentStrategy);

**return** result;

}

}

Business Context

Client choosing different strategy

**package** strategy;

**public** **class** StrategyTester {

**public** **static** **void** main(String[] args) {

**new** BusinessContext(**new** PaytmStrategy()).buy(5, 50);

System.*out*.println();

**new** BusinessContext(**new** CCStrategy()).buy(6, 50);

}

}

Output –

Doing payment with paytm wallet. amount is 250

Amount of 250 paid using PaytmStrategy

Doing payment with CC of amount 300

Amount of 300 paid using CCStrategy

**Difference between Strategy and State design Pattern in Java?**  
This is an interesting Java design pattern interview questions as both Strategy and State pattern has same structure. If you look at UML class diagram for both pattern they look exactly same, but there intent is totally different. State design pattern is used to define and mange state of object, while Strategy pattern is used to define a set of interchangeable algorithm and let's client to choose one of them. So Strategy pattern is a client driven pattern while Object can manage there state itself.

**When to use Strategy Design Pattern in Java?**  
Strategy pattern in quite useful for implementing set of related algorithms e.g. compression algorithms, filtering strategies etc. Strategy design pattern allows you to create Context classes, which uses Strategy implementation classes for applying business rules. This pattern follow open closed design principle and quite useful in Java. One example of Strategy pattern from JDK itself is a Collections.sort() method and [Comparator interface](http://java67.blogspot.sg/2012/10/how-to-sort-object-in-java-comparator-comparable-example.html), which is a strategy interface and defines strategy for comparing objects. Because of this pattern, we don't need to modify sort() method (closed for modification) to compare any object, at same time we can implement Comparator interface to define new comparing strategy (open for extension).

Points to remember –

1. This pattern is very similar to state pattern. Please take a look in that parent now.

References –

1. <http://java67.blogspot.in/2012/09/top-10-java-design-pattern-interview-question-answer.html>

## State Design Pattern

## Difference/Similarity between Strategy and State design patterns.

## Decorator Design pattern

## Adapter Design Pattern

## Singletone Designpattern

## Different latest options of Singletone design pattern.

## Real World Example of Design Patterns

<https://ageekview.wordpress.com/2011/05/15/a-dive-into-design-patterns-used-in-jdk-%E2%80%93-ii/>

<http://www.briandupreez.net/2010/11/design-patterns-in-jdk.html>

|  |  |  |
| --- | --- | --- |
| Design Pattern | What the design pattern Does | Existing example from JDK or any popular tools |
| Factory. | Factory methods are static methods that return an instance of the native class | LogManager.getLogManager  Logger.getLogger  -java.lang.Proxy#newProxyInstance() -java.lang.Object#toString() -java.lang.Class#newInstance() -java.lang.reflect.Array#newInstance() -java.lang.reflect.Constructor#newInstance() -java.lang.Boolean#valueOf(String)  -java.lang.Class#forName()  loading beans through BeanFactory and Application context |
| Abstract factory: | To provide a contract for creating families of related or dependent objects without having to specify their concrete classes. It enables one to decouple an application from the concrete implementation of an entire framework one is using. This is also found all over the JDK and a lot of frameworks like Spring. | -java.util.Calendar#getInstance()  -java.util.Arrays#asList()  -java.util.ResourceBundle#getBundle()  -java.sql.DriverManager#getConnection()  -java.sql.Connection#createStatement()  -java.sql.Statement#executeQuery()  -java.text.NumberFormat#getInstance()  -javax.xml.transform.TransformerFactory#newInstance()  loading beans through BeanFactory and Application context |
| Builder | Used simplify complex object creation by defining a class whose purpose is to build instances of another class. The builder pattern also allows for the implementation of a Fluent Interface. | -java.lang.StringBuilder#append()  -java.lang.StringBuffer#append()  -java.sql.PreparedStatement  -javax.swing.GroupLayout.Group#addComponent() |
| Prototype | Allows for classes whose instances can create duplicates of themselves. This can be used when creating an instance of a class is very time-consuming or complex in some way, rather than creating new instances, you can make copies of the original instance and modify it. | -java.lang.Object#clone()  -java.lang.Cloneable |
| Singleton | This tries to ensure that there is only a single instance of a class.  Enum Singletone – plase read in this context | -java.lang.Runtime#getRuntime() |
| Chain of responsibility: | Allows for the decoupling between objects by passing a request from one object to the next in a chain until the request is recognized. The objects in the chain are different implementations of the same interface or abstract class. | -java.util.logging.Logger#log() |
| Command | To wrap a command in an object so that it can be stored, passed into methods, and returned like any other object. | -java.lang.Runnable -javax.swing.Action |
| **Interpreter** | This pattern generally describes defining a grammar for that language and using that grammar to interpret statements in that format. | java.util.Pattern -java.text.Normalizer -java.text.Format |
| **Iterator** | To provide a consistent way to sequentially access items in a collection that is independent of and separate from the underlying collection. | -java.util.Iterator  -java.util.Enumeration |
| **Mediator** | Used to reduce the number of direct dependencies between classes by introducing a single object that manages message distribution. | -java.util.Timer -java.util.concurrent.Executor#execute() -java.util.concurrent.ExecutorService#submit() -java.lang.reflect.Method#invoke() |
| **Memento** | This is a snapshot of an object’s state, so that the object can return to its original state without having to reveal it's content. Date does this by actually having a long value internally. | -java.util.Date  -java.io.Serializable |
| **Observer** | Used to provide a way for a component to flexibly broadcast messages to interested receivers. | -java.util.EventListener -javax.servlet.http.HttpSessionBindingListener -javax.servlet.http.HttpSessionAttributeListener -javax.faces.event.PhaseListener  Spring has event handling framework similar to this. |
| **State** | This allows you easily change an object’s behavior at runtime based on internal state. | -java.util.Iterator -javax.faces.lifecycle.LifeCycle#execute() |
| **Strategy** | Is intended to provide a means to define a family of algorithms, encapsulate each one as an object. These can then be flexibly passed in to change the functionality. | -java.util.Comparator#compare() -javax.servlet.http.HttpServlet -javax.servlet.Filter#doFilter() |
| **Template method:** | Allows subclasses to override parts of the method without rewriting it, also allows you to control which operations subclasses are required to override. | -java.util.Collections#sort() -java.io.InputStream#skip()  -java.io.InputStream#read()  -java.util.AbstractList#indexOf() |
| **Visitor** | To provide a maintainable, easy way to perform actions for a family of classes. Visitor centralizes the behaviors and allows them to be modified or extended without changing the classes they operate on. | -javax.lang.model.element.Element and javax.lang.model.element.ElementVisitor -javax.lang.model.type.TypeMirror and javax.lang.model.type.TypeVisitor |
|  |  |  |
|  |  |  |
| Adapter | This is used to convert the programming interface/class into that of anothe | -**java.util.Arrays#asList**()  -javax.swing.JTable(TableModel)  -java.io.InputStreamReader(InputStream)  -java.io.OutputStreamWriter(OutputStream)  -javax.xml.bind.annotation.adapters.XmlAdapter#marshal()  -javax.xml.bind.annotation.adapters.XmlAdapter#unmarshal() |
| Bridge | This decouples an abstraction from the implementation of its abstract operations, so that the abstraction and its implementation can vary independently. | JDBC – it acts as a bridge to the underlying Database. |
| Composite | Lets clients treat individual objects and compositions of objects uniformly. So in other words methods on a type accepting the same type. | -javax.swing.JComponent#add(Component)  -java.awt.Container#add(Component)  **-java.util.Map#putAll(Map)**  **-java.util.List#addAll(Collection)**  **-java.util.Set#addAll(Collection)** |
| Decorator | Attach additional responsibilities to an object dynamically and therefore it is also an alternative to subclassing. Can be seen when creating a type passes in the same type. This is actually used all over the JDK, the more you look the more you find, so the list below is definitely not complete. | -java.io.BufferedInputStream(InputStream)  -java.io.DataInputStream(InputStream)  -java.io.BufferedOutputStream(OutputStream)  -java.util.zip.ZipOutputStream(OutputStream)  -java.util.Collections#checked[List|Map|Set|SortedSet|SortedMap]() |
| Facade | To provide a simplified interface to a group of components, interfaces, abstractions or subsystems. | -java.lang.Class -javax.faces.webapp.FacesServlet  Session Facade – an J2EE pattern which has the similar concepts of keeping one single session bean in front to server all types of methods which will intern call different session beans to do its job. |
| Flyweight | Caching to support large numbers of smaller objects efficiently. | -java.lang.Integer#valueOf(int)  -java.lang.Boolean#valueOf(boolean)  -java.lang.Byte#valueOf(byte)  -java.lang.Character#valueOf(char) |
| Proxy | The Proxy pattern is used to represent with a simpler object an object that is complex or time consuming to create. | -java.lang.reflect.Proxy -RMI  -AOP |
|  |  |  |
|  |  |  |

# Spring Related

## DI concepts and how to achieve

## Constructor based dependency injection – different examples.

## Setter based Dependency injection – different examples.

## Static factory based dependency injection.

## Auto wiring concepts and examples.

## Bean scopes

In spring framework there are mainly below scopes are present. Default scope is Singleton.

|  |  |
| --- | --- |
| Scope | Description |
| Singleton | Default scope- single bean definition, single object in a single container. Means if a bean is defined once, then in a particular spring context otherwise container, single physical object would be created for that spring bean irrespective of how many times it has been referenced from different beans. |
| Prototype | Single bean definition and multiple instances is created per container based on its reference. |
| Request | Single bean definition and only one instance is created per HTTP request. So lifecycle of that instance is within that HTTP request only. Only valid in the context of a web-aware Spring ApplicationContext. |
| session | Like request scope, Single bean definition and lifecycle is within a particular session of HTTP. It is applicable to the context of web-aware spring ApplicationContext. |
| global session | Similar to session scope, but lifecycle is applicable to global scope generally used in portlet application and applicable to the context of web-aware spring ApplicationContext. |
| Application | Scopes a single bean definition to the lifecycle of a ServletContext. Only valid in the context of a web-aware Spring ApplicationContext |

\*\*\*\* So in general – singleton/prototype scope is available in non-web contexts and request/session/global session, application is being used in web application scopes.

**How to declare scope of a spring bean ?**

In spring xml style -

**<bean id**=**"userPreferences" class**=**"com.foo.UserPreferences" scope**=**"session/request etc" />**

In Annotation

**import** org.springframework.context.annotation.Scope;

**import** org.springframework.stereotype.Service;

@Service

@Scope("prototype")

**public** **class** CustomerService

{

String message;

**public** String getMessage() {

**return** message;

}

**public** **void** setMessage(String message) {

**this**.message = message;

}

}

Custom Scope - We can create custom scope also. Please refer spring documentation for detailed understanding.

To write custom scope, we need to implement scope interface and implement all the methods here. To register scope we need to do like this – here we are defining SimpleThreadScope which has been provided with spring like below –

Scope threadScope = **new** SimpleThreadScope();

beanFactory.registerScope(***"thread"***, threadScope);

We can also do the Scope registration declaratively, using the CustomScopeConfigurer class:

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:aop=*"http://www.springframework.org/schema/aop"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop.xsd"*>

<bean class=*"org.springframework.beans.factory.config.CustomScopeConfigurer"*>

<property name=*"scopes"*>

<map>

<entry key=*"thread"*>

<bean

class=*"org.springframework.context.support.SimpleThreadScope"* />

</entry>

</map>

</property>

</bean>

<bean id=*"bar"* class=*"x.y.Bar"* scope=*"thread"*>

<property name=*"name"* value=*"Rick"* />

<aop:scoped-proxy />

</bean>

<bean id=*"foo"* class=*"x.y.Foo"*>

<property name=*"bar"* ref=*"bar"* />

</bean>

</beans>

Reference - <http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#beans-factory-scopes>

## AOP concepts and examples.

## Property file loading and how to use that in java code.

## Message source by property file with example.

## Web service client part using JAXB as marshalling.

## Web service exposer example

## Wiring using annotations, component scan and stereotypes.

## JDBC template to do insert, update etc. example

## DAO using Hibernate Dao Support and generic Dao pattern with ready example

## ProxyFactoryBean to initialize spring bean in runtime without loading application context.

## Event publishing

## Lifecycle callbacks

## Pre and post initiallization.

## Initializing Spring beans for the first layer class where application context has not loaded by framework.

## Design Patterns being used in Spring framework

1. **MVC** - The advantage with Spring MVC is that your controllers are POJOs as opposed to being servlets. This makes for easier testing of controllers.

2. **Front controller** - Spring provides "DispatcherServlet" to ensure an incoming request gets dispatched to your controllers.

3. **View Helper** - Spring has a number of custom JSP tags, and velocity macros, to assist in separating code from presentation in views.

4. **Singleton** - Beans defined in spring config files are singletons by default.

5. **Prototype** - Instance type can be prototype.

6. **Factory** - Used for loading beans through BeanFactory and Application context.

7. **Builder** - Spring provides programmatic means of constructing BeanDefinitions using the builder pattern through Class "BeanDefinitionBuilder".

8. **Template** - Used extensively to deal with boilerplate repeated code (such as closing connections cleanly, etc..). For example JdbcTemplate.

9. **Proxy** - Used in AOP & Remoting.

10. **DI/IOC** - It is central to the whole BeanFactory/ApplicationContext stuff.

# General topics

## Eh cache example for configuration to maintain different cache effectively.

## Make asynchronous call and respond as synchronous one.

## Maven profile and classifier to generate different named artifact.

# Process Related

## How to manage defects using HP ALM and what are the lifecycle we followed etc.

## JIRA. What are the process of delivering story etc.

## General day to day activity related to projects like Daily calls, Scrum calls etc.