Table of Contents

[Design patterns 2](#_Toc435800190)

[Strategy Design pattern 2](#_Toc435800191)

[State Design Pattern 5](#_Toc435800192)

[Difference/Similarity between Strategy and State design patterns. 5](#_Toc435800193)

[Decorator Design pattern 5](#_Toc435800194)

[Adapter Design Pattern 5](#_Toc435800195)

[Singletone Designpattern 5](#_Toc435800196)

[Different latest options of Singletone design pattern. 5](#_Toc435800197)

[Spring Related 5](#_Toc435800198)

[DI concepts and how to achieve 5](#_Toc435800199)

[Constructor based dependency injection – different examples. 5](#_Toc435800200)

[Setter based Dependency injection – different examples. 5](#_Toc435800201)

[Static factory based dependency injection. 5](#_Toc435800202)

[Auto wiring concepts and examples. 5](#_Toc435800203)

[Bean scopes 5](#_Toc435800204)

[Custom Scope 7](#_Toc435800205)

[AOP concepts and examples. 8](#_Toc435800206)

[Property file loading and how to use that in java code. 8](#_Toc435800207)

[Message source by property file with example. 8](#_Toc435800208)

[Web service client part using JAXB as marshalling. 8](#_Toc435800209)

[Web service exposer example 8](#_Toc435800210)

[Wiring using annotations, component scan and stereotypes. 8](#_Toc435800211)

[JDBC template to do insert, update etc. example 8](#_Toc435800212)

[DAO using Hibernate Dao Support and generic Dao pattern with ready example 8](#_Toc435800213)

[ProxyFactoryBean to initialize spring bean in runtime without loading application context. 8](#_Toc435800214)

[Event publishing 8](#_Toc435800215)

[Lifecycle callbacks 8](#_Toc435800216)

[Pre and post initiallization. 8](#_Toc435800217)

[Initializing Spring beans for the first layer class where application context has not loaded by framework. 8](#_Toc435800218)

[General topics 8](#_Toc435800219)

[Eh cache example for configuration to maintain different cache effectively. 8](#_Toc435800220)

[Make asynchronous call and respond as synchronous one. 8](#_Toc435800221)

[Maven profile and classifier to generate different named artifact. 8](#_Toc435800222)

[Process Related 8](#_Toc435800223)

[How to manage defects using HP ALM and what are the lifecycle we followed etc. 8](#_Toc435800224)

[JIRA. What are the process of delivering story etc. 8](#_Toc435800225)

[General day to day activity related to projects like Daily calls, Scrum calls etc. 8](#_Toc435800226)

# Design patterns

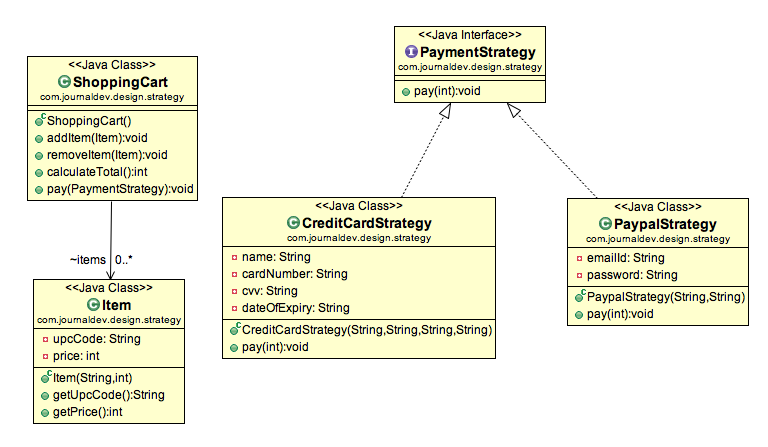
## Strategy Design pattern

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 1

**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.

# 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.

# 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.