## Design Principles

* 1. Open Close Principle
  2. Dependency Inversion Principle
  3. Interface Segregation Principle
  4. Single Responsibility Principle
  5. Liskov's Substitution Principle

## Creational Patterns

* 1. Singleton
  2. Factory
  3. Factory Method
  4. Abstract Factory
  5. Builder
  6. Prototype
  7. Object Pool

## Behavioral Patterns

* 1. Chain of Responsibility
  2. Command
  3. Interpreter
  4. Iterator
  5. Mediator
  6. Memento
  7. Observer
  8. Strategy
  9. Template Method
  10. Visitor
  11. Null Object

## Structural Patterns

* 1. Adapter
  2. Bridge
  3. Composite
  4. Decorator
  5. Flyweight
  6. Proxy

Design Pattern Books

## What are Software Design Principles?

Software design principles represent a set of guidelines that helps us to avoid having a bad design. The design principles are associated to Robert Martin who gathered them in "Agile Software Development: Principles, Patterns, and Practices". According to Robert Martin there are 3 important characteristics of a bad design that should be avoided:

* Rigidity - It is hard to change because every change affects too many other parts of the system.
* Fragility - When you make a change, unexpected parts of the system break.
* Immobility - It is hard to reuse in another application because it cannot be disentangled from the current application.

## [Open Close Principle](http://www.oodesign.com/open-close-principle.html)

* *Software entities like classes, modules and functions should be****open for extension****but****closed for modifications.***

OPC is a generic principle. You can consider it when writing your classes to make sure that when you need to extend their behavior you don�t have to change the class but to extend it. The same principle can be applied for modules, packages, libraries. If you have a library containing a set of classes there are many reasons for which you�ll prefer to extend it without changing the code that was already written (backward compatibility, regression testing, �). This is why we have to make sure our modules follow Open Closed Principle.

When referring to the classes Open Close Principle can be ensured by use of Abstract Classes and concrete classes for implementing their behavior. This will enforce having Concrete Classes extending Abstract Classes instead of changing them. Some particular cases of this are Template Pattern and Strategy Pattern.

## [Dependency Inversion Principle](http://www.oodesign.com/dependency-inversion-principle.html)

* *High-level modules should not depend on low-level modules. Both should depend on abstractions.*
* *Abstractions should not depend on details. Details should depend on abstractions.*

Dependency Inversion Principle states that we should decouple high level modules from low level modules, introducing an abstraction layer between the high level classes and low level classes. Further more it inverts the dependency: instead of writing our abstractions based on details, the we should write the details based on abstractions.

Dependency Inversion or Inversion of Control are better know terms referring to the way in which the dependencies are realized. In the classical way when a software module(class, framework, �) need some other module, it initializes and holds a direct reference to it. This will make the 2 modules tight coupled. In order to decouple them the first module will provide a hook(a property, parameter, �) and an external module controlling the dependencies will inject the reference to the second one.

By applying the Dependency Inversion the modules can be easily changed by other modules just changing the dependency module. Factories and Abstract Factories can be used as dependency frameworks, but there are specialized frameworks for that, known as Inversion of Control Container.

## [Interface Segregation Principle](http://www.oodesign.com/interface-segregation-principle.html)

* *Clients should not be forced to depend upon interfaces that they don't use.*

This principle teaches us to take care how we write our interfaces. When we write our interfaces we should take care to add only methods that should be there. If we add methods that should not be there the classes implementing the interface will have to implement those methods as well. For example if we create an interface called Worker and add a method lunch break, all the workers will have to implement it. What if the worker is a robot?

As a conclusion Interfaces containing methods that are not specific to it are called polluted or fat interfaces. We should avoid them.

## [Single Responsibility Principle](http://www.oodesign.com/single-responsibility-principle.html)

* *A class should have only one reason to change.*

In this context a responsibility is considered to be one reason to change. This principle states that if we have 2 reasons to change for a class, we have to split the functionality in two classes. Each class will handle only one responsibility and on future if we need to make one change we are going to make it in the class which handle it. When we need to make a change in a class having more responsibilities the change might affect the other functionality of the classes.

Single Responsibility Principle was introduced Tom DeMarco in his book Structured Analysis and Systems Specification, 1979. Robert Martin reinterpreted the concept and defined the responsibility as a reason to change.

## [Liskov's Substitution Principle](http://www.oodesign.com/liskov-s-substitution-principle.html)

* *Derived types must be completely substitutable for their base types.*

This principle is just an extension of the Open Close Principle in terms of behavior meaning that we must make sure that new derived classes are extending the base classes without changing their behavior. The new derived classes should be able to replace the base classes without any change in the code.

Liskov's Substitution Principle was introduced by Barbara Liskov in a 1987 Conference on Object Oriented Programming Systems Languages and Applications, in [Data abstraction and hierarchy](http://portal.acm.org/citation.cfm?id=62141)

## Creational Design Patterns (from <http://www.oodesign.com/> )

## Singleton –

Ensure that only one instance of a class is created and Provide a global access point to the object.

### Common Usage

There are many common situations when singleton pattern is used:

- Logger Classes

- Configuration Classes

- Accessing resources in shared mode

- Other design patterns implemented as Singletons: Factories and Abstract Factories, Builder, Prototype

### When to Use:

Singleton pattern should be used when we must ensure that only one instance of a class is created and when the instance must be available through all the code. A special care should be taken in multi-threading environments when multiple threads must access the same resources through the same singleton object.

## Factory

(Simplified version of Factory Method) - Creates objects without exposing the instantiation logic to the client and Refers to the newly created object through a common interface.

### When to Use

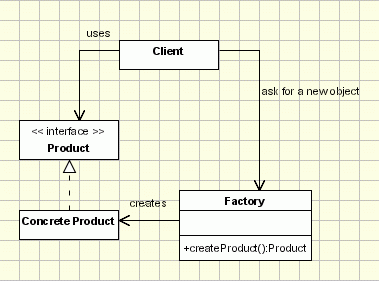
Factory pattern should be used when: - a framework delegate the creation of objects derived from a common superclass to the factory - we need flexibility in adding new types of objects that must be created by the class

### Common Usage

Along with singleton pattern the factory is one of the most used patterns. Almost any application has some factories. Here are a some examples in java:

- factories providing an xml parser: javax.xml.parsers.DocumentBuilderFactory or javax.xml.parsers.SAXParserFactory

- java.net.URLConnection - allows users to decide which protocol to use



## Factory Method

- Defines an interface for creating objects, but let subclasses to decide which class to instantiate and Refers to the newly created object through a common interface.

When to Use , Common Usage

Click to zoom

Abstract Factory Look & Feel Example

Abstract Pattern Example

## Abstract Factory

- Offers the interface for creating a family of related objects, without explicitly specifying their classes.

### When to Use

Abstract Factory should be used when:

* A system should be configured with one of multiple families of products
* A system should be independent of how its products are created, composed and represented
* Products from the same family should be used all together, products from different families ahould not be used togheter and this constraint must be ensured.
* Only the product interfaces are revealed, the implementations remains hidden to the clients.

### Common Usage

Examples of abstract factories:

* java.awt.Toolkit - the abstract superclass of all actual implementations of the Abstract Window Toolkit. Subclasses of Toolkit are used to bind the various components to particular native toolkit implementations(Java AWT).
* javax.swing.LookAndFeel - an abstract swing factory to swithct between several look and feel for the components displayed(Java Swing).
* java.sql.Connection - an abstract factory which create Statements, PreparedStatements, CallableStatements,... for each database flavor.

## Builder

- Defines an instance for creating an object but letting subclasses decide which class to instantiate and Allows a finer control over the construction process.

Example: Text Converter in Java

Click to zoom

Highslide JS

## Prototype

- Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Click to zoom

Object Pool Database Example

Object Pool Example

Object Pool - reuses and shares objects that are expensive to create..

When to Use , Common Usage , Sourcecode: Database Connection Pool in Java

## Behavioral Design Patterns:

## Chain of Responsibility

It avoids attaching the sender of a request to its receiver, giving this way other objects the possibility of handling the request too.

- The objects become parts of a chain and the request is sent from one object to another across the chain until one of the objects will handle it.

Sourcecode:

Click to zoom

Command Pattern Restaurant Example

Command Pattern Example

## Command

- Encapsulate a request in an object, Allows the parameterization of clients with different requests and Allows saving the requests in a queue.

Sourcecode: Buying/Selling stocks in Java

Click to zoom

Interpreter Pattern

## Interpreter

- Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language / Map a domain to a language, the language to a grammar, and the grammar to a hierarchical object-oriented design

Sourcecode: Romans Numerals Converter in Java

Click to zoom

## Iterator Pattern

Iterator - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Sourcecode: Java Iterator

Click to zoom

Mediator Pattern

Mediator - Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Sourcecode:

Click to zoom

Observer Pattern News Publisher Example

Highslide JS

Observer - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Sourcecode: News Publisher in Java

Click to zoom

Strategy Pattern Robot Example

Highslide JS

Strategy - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Sourcecode: Robot Application in Java

Click to zoom

Template Method Travel Example

Highslide JS

Template Method - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses / Template Method lets subclasses redefine certain steps of an algorithm without letting them to change the algorithm's structure.

Sourcecode: Travel Agency Application in Java

Click to zoom

Visitor Pattern Customers Example

Visitor Pattern Example

Visitor - Represents an operation to be performed on the elements of an object structure / Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Sourcecode: Customers Report Example

Click to zoom

Null Object Pattern

Null Object - Provide an object as a surrogate for the lack of an object of a given type. / The Null Object Pattern provides intelligent do nothing behavior, hiding the details from its collaborators.

Sourcecode:

## Structural Design Patterns:

## [Adapter](http://www.oodesign.com/adapter-pattern.html)

 - Convert the interface of a class into another interface clients expect. / Adapter lets classes work together, that could not otherwise because of incompatible interfaces.

Click to zoom  
[Bridge Pattern](http://www.oodesign.com/images/structural/bridge-pattern.png)Click to zoom  
[Bridge Pattern](http://www.oodesign.com/images/structural/bridge-pattern-example.png)

## [Bridge](http://www.oodesign.com/bridge-pattern.html)

 - Compose objects into tree structures to represent part-whole hierarchies. / Composite lets clients treat individual objects and compositions of objects uniformly.

Sourcecode: [Object Persistence Api in Java](http://www.oodesign.com/bridge-pattern-object-persistence-api-example-java-sourcecode.html)

Click to zoom  
[Composite Pattern](http://www.oodesign.com/images/structural/composite-pattern.png)Shapes Example  
[Composite Pattern Example](http://www.oodesign.com/images/structural/composite-pattern-example.png)

## [Composite](http://www.oodesign.com/composite-pattern.html)

Compose objects into tree structures to represent part-whole hierarchies. / Composite lets clients treat individual objects and compositions of objects uniformly.

Sourcecode: [Shapes Example in Java](http://www.oodesign.com/composite-pattern-shapes-example-java-sourcecode.html)

Click to zoom  
[Decorator Pattern](http://www.oodesign.com/images/structural/decorator-pattern.png)GUI Example  
[Decorator Pattern Example](http://www.oodesign.com/images/structural/decorator-pattern-example.png)

## [Decorator](http://www.oodesign.com/decorator-pattern.html)

- add additional responsibilities dynamically to an object.

Sourcecode: [Gui Application Example](http://www.oodesign.com/decorator-pattern-gui-example-java-sourcecode.html)

Click to zoom  
[Flyweight Pattern](http://www.oodesign.com/images/structural/flyweight-pattern.png)Wargame Example  
[Flyweight Pattern Example](http://www.oodesign.com/images/structural/flyweight-pattern-example.png)

## [Flyweight](http://www.oodesign.com/flyweight-pattern.html)

- use sharing to support a large number of objects that have part of their internal state in common where the other part of state can vary.

Sourcecode: [Java Wargame Example](http://www.oodesign.com/flyweight-pattern-wargame-example-java-sourcecode.html)

Click to zoom  
[Memento Pattern](http://www.oodesign.com/images/structural/memento-pattern.png)Calculator Example  
[Memento Pattern Example](http://www.oodesign.com/images/structural/memento-pattern-example.png)

## [Memento](http://www.oodesign.com/memento-pattern.html)

- capture the internal state of an object without violating encapsulation and thus providing a mean for restoring the object into initial state when needed.

Source Code: [Calculator Example in Java](http://www.oodesign.com/memento-pattern-calculator-example-java-sourcecode.html)

Click to zoom  
[Proxy Pattern](http://www.oodesign.com/images/structural/proxy-pattern.png)Image Viewer  
[Proxy Pattern Example](http://www.oodesign.com/images/structural/proxy-pattern-example.png)

## [Proxy](http://www.oodesign.com/proxy-pattern.html)

provide a “Placeholder” for an object to control references to it. -

Sourcecode: [Proxy Pattern in Java](http://www.oodesign.com/proxy-pattern.html)----------------------

## [Decorator design Pattern in Java with Example Java Tutorial](http://javarevisited.blogspot.in/2011/11/decorator-design-pattern-java-example.html)

I was thinking to write on **decorator design pattern in Java** when I first wrote [10 interview questions on Singleton Pattern in Java](http://javarevisited.blogspot.com/2011/03/10-interview-questions-on-singleton.html). Since design pattern is quite important while building software and it’s equally important on any Core Java Interview, It’s always good to have clear understanding of various design patterns in Java. In this article we will explore and learn **Decorator Design pattern in Java** which is a prominent core Java design pattern and you can see lot of its example in JDK itself. JDK use decorator pattern in IO package where it has decorated Reader and Writer Classes for various scenario, for example BufferedReader and BufferedWriter are example of decorator design pattern in Java. From design perspective its also good idea to learn how existing things work inside JDK itself for example [How HashMap works in Java](http://javarevisited.blogspot.com/2011/02/how-hashmap-works-in-java.html) or [How SubString method work in Java](http://javarevisited.blogspot.com/2011/10/how-substring-in-java-works.html), that will give you some idea of things you need to keep in mind while designing your Class or interface in Java. Now let’s Move on to **Decorator pattern in Java**.

## Java Decorator Design Pattern

In this Java tutorial we will see:

What is decorator pattern in Java?

When to use decorator pattern in Java?

How to use decorator pattern in Java?

Example of decorator design pattern

Advantage and Disadvantage of decorator pattern in Java

### What is decorator design pattern in Java?

          Decorator design pattern is used to **enhance the functionality of a particular object at run-time** or dynamically.

          At the same time **other instance of same class will not be affected by this** so individual object gets the new behavior.

          Basically we wrap the original object through decorator object.

          Decorator design pattern is based on abstract classes and we derive concrete implementation from that classes,

          It’s a structural design pattern and most widely used.

I prefer to answer *What is decorator design pattern* in point format just to stress on important point like this pattern operator at individual object level. This question also asked in many [Core Java interviews in Investment banks](http://javarevisited.blogspot.com/2011/04/top-20-core-java-interview-questions.html)

### Problem which is solved by Decorator Pattern:

[decorator design pattern java example code](http://javarevisited.blogspot.com/2011/09/generics-java-example-tutorial.html)Now the question is why this pattern has came into existence what is the problem with existing system, so the answer is if anyone wants to add some functionality to individual object or change the state of particular object at run time it is not possible what the possible is we can provide the specific behavior to all the object of that class at design time by the help of inheritance or using subclass, but **Decorator pattern** makes possible that we provide individual object of same class a specific behavior or state at run time. This doesn’t affect other object of same [Class in Java](http://javarevisited.blogspot.com/2011/10/class-in-java-programming-general.html).

**When to use Decorator pattern in Java**

          When sub classing is become impractical and we need large number of different possibilities to make independent object or we can say we have number of combination for an object.

          Secondly when we want to add functionality to individual object not to all object at run-time we use decorator design pattern.

**Code Example of decorator design pattern:**

To better understand concept of decorator design pattern let see a code example using Decorator Pattern in Java. You can also look inside JDK and find what are classes and packages which are using decorator pattern.

// Component on Decorator design pattern

**public** **abstract** **class** Currency **{**  
 String description = "Unknown currency";

**public** String getCurrencyDescription**()** **{**  
  **return** description;  
 **}**

 public abs**tract** **double** cost**(double** value**)**;

**}**

// Concrete Component

**public** **class**Rupee **extends** Currency**{**

**double**value**;**

**public** Rupee**()** **{**  
  description = "indian rupees";  
 **}**

 public double cost(double v){  
  value=v;

  returnvalue;  
 **}**

**}**

//Another Concrete Component

public class Dollar extends Currency{

**double**value**;**

 public Dollar () {  
  description = "Dollar”;  
 }

**public** **double** cost**(double v){**

**value=v;**

**return** value;

**}**

**}**

// Decorator

**public** **abstract** **class** Decorator **extends** Currency**{**

**public** **abstract** String getDescription**()**;

**}**

// Concrete Decorator

**public** **class** USDDecorator **extends** Decorator**{**

 Currency currency;

**public** USDDecorator**(**Currency currency**){**  
  **this**.currency = currency;  
 **}**

**public** String getDescription**(){**  
  **return** currency.getDescription**()**+" ,its US Dollar";  
 **}**

**}**

//Another Concrete Decorator

**public** **class** SGDDecorator **extends** Decorator**{**  
 Currency currency;

**public** SGDDecorator**(**Currency currency**){**  
  **this**.currency = currency;  
 **}**

**public** String getDescription**(){**  
  **return** currency.getDescription**()**+" ,its singapore Dollar";  
 **}**

**}**

Now its time to check currency.

**public** **class** CurrencyCheck **{**

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

  // without adding decorators

  Currency curr = **new** Dollar**()**;

  System.out.println**(**curr.getDescription**()** +" dollar. "+curr.cost**(2.0))**;

  //adding decorators

  Currency curr2 = **new** USDDecorator**(new** Dollar()**)**;

  System.out.println**(**curr2.getDescription**()** +" dollar. "+curr2.cost**(4.0))**;

Currency curr3 = **new** SGDDecorator**(new** Dollar()**)**;

  System.out.println**(**curr3.getDescription**()** +" dollar. "+curr3.cost**(4.0))**;

}

**Explanation of the code**:

We can understand this in following term;

1.      **Component Interface**: In our example Currency interface is component which used on its own or we need decorator for that.

2.      **Concrete Component: it** implements Component and we add new behavior to this object at dynamically. Dollar and Rupee are the concrete implementation of currency.

3.      **Decorator: Decorator** contains a HAS a Relationship in simple word we can say it has a instance variable that holds reference for component they implement same component which they are going to decorate. Here a Decorator is an abstract class which extends the currency.

4.      **Concrete Decorator:** it’s an implementation of Decorator So USD Dollar and SGD Dollar are the implementation of Decorator contains instance variable for component interface or the thing which they are going to decorate.

**Advantage of Decorator design Pattern in Java**

In brief we see what the main advantages of using decorator design patterns are.

1.      Decorator Pattern is flexible than inheritance because inheritance add responsibilities at compile time and it will add at run-time.

2.      Decorator pattern enhance or modify the object functionality

**Disadvantage**

Main disadvantage of using Decorator Pattern in Java is that the code maintenance can be a problem as it provides a lot of similar kind of small objects (each decorator).

That’s all on **decorator design pattern in Java**. To get mastery on decorator pattern I suggest looking inside JDK library itself and finding what classes are decorated, why they are decorated. Also think of scenario where inheritance is impractical and you look more flexibility and try to **use decorator pattern in Java** there.

**Some more Interesting tutorials:**

## Design Pattern Used by Java Itself:

Creational patterns : Abstract factory (recognizable by creational methods returning the factory itself which in turn can be used to create another abstract/interface type)

javax.xml.parsers.DocumentBuilderFactory#newInstance()

javax.xml.transform.TransformerFactory#newInstance()

javax.xml.xpath.XPathFactory#newInstance()

Builder (recognizable by creational methods returning the instance itself)

java.lang.StringBuilder#append() (unsynchronized)

java.lang.StringBuffer#append() (synchronized)

java.nio.ByteBuffer#put() (also on CharBuffer, ShortBuffer, IntBuffer, LongBuffer, FloatBuffer and DoubleBuffer)

javax.swing.GroupLayout.Group#addComponent()

All implementations of java.lang.Appendable

Factory method (recognizeable by creational methods returning an implementation of an abstract/interface type)

java.util.Calendar#getInstance()

java.util.ResourceBundle#getBundle()

java.text.NumberFormat#getInstance()

java.nio.charset.Charset#forName()

java.net.URLStreamHandlerFactory#createURLStreamHandler(String) (Returns singleton object per protocol)

Prototype (recognizeable by creational methods returning a different instance of itself with the same properties)

java.lang.Object#clone() (the class has to implement java.lang.Cloneable)

Singleton (recognizeable by creational methods returning the same instance (usually of itself) everytime)

java.lang.Runtime#getRuntime()

java.awt.Desktop#getDesktop()

java.lang.System#getSecurityManager()

Structural patterns

Adapter (recognizeable by creational methods taking an instance of different abstract/interface type and returning an implementation of own/another abstract/interface type which decorates/overrides the given instance)

java.util.Arrays#asList()

java.io.InputStreamReader(InputStream) (returns a Reader)

java.io.OutputStreamWriter(OutputStream) (returns a Writer)

javax.xml.bind.annotation.adapters.XmlAdapter#marshal() and #unmarshal()

Bridge (recognizeable by creational methods taking an instance of different abstract/interface type and returning an implementation of own abstract/interface type which delegates/uses the given instance)

None comes to mind yet. A fictive example would be new LinkedHashMap(LinkedHashSet<K>, List<V>) which returns an unmodifiable linked map which doesn't clone the items, but uses them. The java.util.Collections#newSetFromMap() and singletonXXX() methods however comes close.

Composite (recognizeable by behavioral methods taking an instance of same abstract/interface type into a tree structure)

java.awt.Container#add(Component) (practically all over Swing thus)

javax.faces.component.UIComponent#getChildren() (practically all over JSF UI thus)

Decorator (recognizeable by creational methods taking an instance of same abstract/interface type which adds additional behaviour)

All subclasses of java.io.InputStream, OutputStream, Reader and Writer have a constructor taking an instance of same type.

java.util.Collections, the checkedXXX(), synchronizedXXX() and unmodifiableXXX() methods.

javax.servlet.http.HttpServletRequestWrapper and HttpServletResponseWrapper

Facade (recognizeable by behavioral methods which internally uses instances of different independent abstract/interface types)

javax.faces.context.FacesContext, it internally uses among others the abstract/interface types LifeCycle, ViewHandler, NavigationHandler and many more without that the enduser has to worry about it (which are however overrideable by injection).

javax.faces.context.ExternalContext, which internally uses ServletContext, HttpSession, HttpServletRequest, HttpServletResponse, etc.

Flyweight (recognizeable by creational methods returning a cached instance, a bit the "multiton" idea)

java.lang.Integer#valueOf(int) (also on Boolean, Byte, Character, Short, Long and BigDecimal)

Proxy (recognizeable by creational methods which returns an implementation of given abstract/interface type which in turn delegates/uses a different implementation of given abstract/interface type)

java.lang.reflect.Proxy

java.rmi.\*, the whole API actually.

The Wikipedia example is IMHO a bit poor, lazy loading has actually completely nothing to do with the proxy pattern at all.

Behavioral patterns

Chain of responsibility (recognizeable by behavioral methods which (indirectly) invokes the same method in another implementation of same abstract/interface type in a queue)

java.util.logging.Logger#log()

javax.servlet.Filter#doFilter()

Command (recognizeable by behavioral methods in an abstract/interface type which invokes a method in an implementation of a different abstract/interface type which has been encapsulated by the command implementation during its creation)

All implementations of java.lang.Runnable

All implementations of javax.swing.Action

Interpreter (recognizeable by behavioral methods returning a structurally different instance/type of the given instance/type; note that parsing/formatting is not part of the pattern, determining the pattern and how to apply it is)

java.util.Pattern

java.text.Normalizer

All subclasses of java.text.Format

All subclasses of javax.el.ELResolver

Iterator (recognizeable by behavioral methods sequentially returning instances of a different type from a queue)

All implementations of java.util.Iterator (thus among others also java.util.Scanner!).

All implementations of java.util.Enumeration

Mediator (recognizeable by behavioral methods taking an instance of different abstract/interface type (usually using the command pattern) which delegates/uses the given instance)

java.util.Timer (all scheduleXXX() methods)

java.util.concurrent.Executor#execute()

java.util.concurrent.ExecutorService (the invokeXXX() and submit() methods)

java.util.concurrent.ScheduledExecutorService (all scheduleXXX() methods)

java.lang.reflect.Method#invoke()

Memento (recognizeable by behavioral methods which internally changes the state of the whole instance)

java.util.Date (the setter methods do that, Date is internally represented by a long value)

All implementations of java.io.Serializable

All implementations of javax.faces.component.StateHolder

Observer (or Publish/Subscribe) (recognizeable by behavioral methods which invokes a method on an instance of another abstract/interface type, depending on own state)

java.util.Observer/java.util.Observable (rarely used in real world though)

All implementations of java.util.EventListener (practically all over Swing thus)

javax.servlet.http.HttpSessionBindingListener

javax.servlet.http.HttpSessionAttributeListener

javax.faces.event.PhaseListener

State (recognizeable by behavioral methods which changes its behaviour depending on the instance's state which can be controlled externally)

javax.faces.lifecycle.LifeCycle#execute() (controlled by FacesServlet, the behaviour is dependent on current phase (state) of JSF lifecycle)

Strategy (recognizeable by behavioral methods in an abstract/interface type which invokes a method in an implementation of a different abstract/interface type which has been passed-in as method argument into the strategy implementation)

java.util.Comparator#compare(), executed by among others Collections#sort().

javax.servlet.http.HttpServlet, the service() and all doXXX() methods take HttpServletRequest and HttpServletResponse and the implementor has to process them (and not to get hold of them as instance variables!).

javax.servlet.Filter#doFilter()

Template method (recognizeable by behavioral methods which already have a "default" behaviour definied by an abstract type)

All non-abstract methods of java.io.InputStream, java.io.OutputStream, java.io.Reader and java.io.Writer.

All non-abstract methods of java.util.AbstractList, java.util.AbstractSet and java.util.AbstractMap.

javax.servlet.http.HttpServlet, all the doXXX() methods by default sends a HTTP 405 "Method Not Allowed" error to the response. You're free to implement none or any of them.

Visitor (recognizeable by two different abstract/interface types which has methods definied which takes each the other abstract/interface type; the one actually calls the method of the other and the other executes the desired strategy on it)

javax.lang.model.element.AnnotationValue and AnnotationValueVisitor

javax.lang.model.element.Element and ElementVisitor

javax.lang.model.type.TypeMirror and TypeVisitor

java.nio.file.FileVisitor and SimpleFileVisitor

share

edited Dec 22 '15 at 14:37

community wiki

31 revs, 9 users 79%

BalusC

4

impressive.. :) +1. javax.lang.model.element defines visitors ;) I'm not quite sure whether doXXX and doFilter are "strategies". – Bozho Apr 26 '10 at 13:14

4

This related blog post just appeared briandupreez.net/2010/11/design-patterns-in-jdk.html – Bozho Nov 23 '10 at 19:09

7

The mentioned builders e.g. StrinbgBuilder are all not an example for the Builder-Pattern. It is a very common mistake however to consider them as builders (so you are not really to blame ^\_^) – Angel O'Sphere May 25 '11 at 13:41

9

@BalusC: Object.toString() can hardly be considered to be a factory method; the class relationship is right but the intention is wrong. It's hard to draw the line of course, but any method creating and returning another object can't be called a factory method. Maybe you can say that purpose of toString isn't to create a string but the return info about the receiver, therefore it is not a factory method. – Lii Oct 20 '12 at 15:40

19

@BalusC, I have a question to ask you. Did you read the WHOLE source code of Java and JSF? – Tapas Bose Jan 9 '13 at 21:39

show 13 more comments

up vote

68

down vote

Observer pattern throughout whole swing (Observable, Observer)

MVC also in swing

Adapter pattern: InputStreamReader and OutputStreamWriter NOTE: ContainerAdapter, ComponentAdapter, FocusAdapter, KeyAdapter, MouseAdapter are not adapters; they are actually Null Objects. Poor naming choice by Sun.

Decorator pattern (BufferedInputStream can decorate other streams such as FilterInputStream)

AbstractFactory Pattern for the AWT Toolkit and the Swing pluggable look-and-feel classes

java.lang.Runtime#getRuntime() is Singleton

ButtonGroup for Mediator pattern

Action, AbstractAction may be used for different visual represntations to execute same code -> Command pattern

Interned Strings or CellRender in JTable for Flyweight Pattern (Also think about various pools - Thread pools, connection pools, EJB object pools - Flyweight is really about management of shared resources)

The Java 1.0 event model is an example of Chain of Responsibility, as are Servlet Filters.

Iterator pattern in Collections Framework

Nested containers in AWT/Swing use the Composite pattern

Layout Managers in AWT/Swing are an example of Strategy

and many more I guess

share

edited Oct 5 '12 at 3:00

community wiki

7 revs, 3 users 69%

jitter

32

java.lang.Math (6th one) is not singleton, you don't have an instance to start with, everything is static. That's not singleton – Ion Todirel Apr 25 '10 at 5:11

1

Thanks for the tip on MouseAdapter. I found this exaplanation: stackoverflow.com/questions/9244185/… – Lincoln May 20 '15 at 14:24

add a comment

up vote

36

down vote

Flyweight is used with some values of Byte, Short, Integer, Long and String.

Facade is used in many place but the most obvious is Scripting interfaces.

Singleton - java.lang.Runtime comes to mind.

Abstract Factory - Also Scripting and JDBC API.

Command - TextComponent's Undo/Redo.

Interpreter - RegEx (java.util.regex.) and SQL (java.sql.) API.

Prototype - Not 100% sure if this count, but I thinkg clone() method can be used for this purpose.