# Design Patterns

## Creational

Creational patterns are ones that create objects for you, rather than having you instantiate objects directly. This gives your program more flexibility in deciding which objects need to be created for a given case.

1. **Factory Method** – creates objects without specifying the exact class to create. Define an interface for creating an object, but let the classes that implement the interface decide which class to instantiate. The Factory method lets a class defer instantiation to subclasses.

****

|  |
| --- |
| **interface** Pet {  **public** String speak();  }  **class** Dog **implements** Pet {  **public** String speak() {  **return** "Bark bark...";  }  }  **class** Duck **implements** Pet {  **public** String speak() {  **return** "Quack quack...";  }  }  **class** PetFactory {  **public** Pet getPet(String petType) {  Pet pet = **null**;  // based on logic factory instantiates an object  **if** ("bark".equals(petType))  pet = **new** Dog();  **else** **if** ("quack".equals(petType))  pet = **new** Duck();  **return** pet;  }  }  **public** **class** FactoryMethodExample {  **public** **static** **void** main(String[] args) {  //creating the factory  PetFactory petFactory = **new** PetFactory();    //factory instantiates an object  Pet pet = petFactory.getPet("bark");    //you don't know which object factory created  System.*out*.println(pet.speak());  }  } |

### [Factory method](http://en.wikipedia.org/wiki/Factory_method_pattern) (recognizeable by creational methods returning an implementation of an abstract/interface type)

* [java.util.Calendar#getInstance()](http://docs.oracle.com/javase/6/docs/api/java/util/Calendar.html#getInstance%28%29)
* [java.util.ResourceBundle#getBundle()](http://docs.oracle.com/javase/6/docs/api/java/util/ResourceBundle.html#getBundle%28java.lang.String%29)
* [java.text.NumberFormat#getInstance()](http://docs.oracle.com/javase/6/docs/api/java/text/NumberFormat.html#getInstance%28%29)
* [java.nio.charset.Charset#forName()](http://docs.oracle.com/javase/6/docs/api/java/nio/charset/Charset.html#forName%28java.lang.String%29)
* [java.net.URLStreamHandlerFactory#createURLStreamHandler(String)](http://docs.oracle.com/javase/6/docs/api/java/net/URLStreamHandlerFactory.html) (Returns singleton object per protocol)

1. **Abstract Factory** – groups object **factories** that have a common theme.

In the example below, Sea/Land factories are grouped as AnimalFactory.

Note: Shark and Elephant classes may not be implementing a common interface but may still be created by factories that can be grouped by a common abstract factory interface.

****

|  |
| --- |
| **interface** Animal {  **public** **void** breathe();  }  **interface** AnimalFactory {  **public** Animal createAnimal();  }  **class** SeaFactory **implements** AnimalFactory {  **public** Animal createAnimal() {  **return** **new** Shark();  }  }  **class** LandFactory **implements** AnimalFactory {  **public** Animal createAnimal() {  **return** **new** Elephant();  }  }  **class** Shark **implements** Animal {  **public** **void** breathe() {  System.*out*.println("I breathe in water! He he!");  }  }  **class** Elephant **implements** Animal {  **public** **void** breathe() {  System.*out*.println("I breathe with my lungs. Its easy!");  }  }  /\*\*  \* Given an animal factory, creates an animal.  \*  \* **@author** Watsh  \*  \*/  **class** Wonderland {  **public** Wonderland(AnimalFactory factory) {  Animal animal = factory.createAnimal();  animal.breathe();  }  }  /\*\*  \* Test abstract factory pattern.  \*  \* **@author** Watsh  \*  \*/  **public** **class** AbstractFactoryExample {  **public** **static** **void** main(String[] args) {  **new** Wonderland(*createAnimalFactory*("water"));  }  **public** **static** AnimalFactory createAnimalFactory(String type) {  **if** ("water".equals(type))  **return** **new** SeaFactory();  **else**  **return** **new** LandFactory();  }  } |

### [Abstract factory](http://en.wikipedia.org/wiki/Abstract_factory_pattern) (recognizeable by creational methods returning the factory itself which in turn can be used to create another abstract/interface type)

* [javax.xml.parsers.DocumentBuilderFactory#newInstance()](http://docs.oracle.com/javase/6/docs/api/javax/xml/parsers/DocumentBuilderFactory.html#newInstance%28%29)
* [javax.xml.transform.TransformerFactory#newInstance()](http://docs.oracle.com/javase/6/docs/api/javax/xml/transform/TransformerFactory.html#newInstance%28%29)
* [javax.xml.xpath.XPathFactory#newInstance()](http://docs.oracle.com/javase/6/docs/api/javax/xml/xpath/XPathFactory.html#newInstance%28%29)

1. **Builder -** Separate the construction of a complex object from its representation.

If there are optional properties in a class then we will have to write several different constructors that can take different combinations of those optional properties. Instead if we use Builder pattern we only set the optional properties through their setter methods - thus keeping the code cleaner.

In the example below, NutritionalFacts class:

1. has a static Builder class which takes the mandatory properties in its constructor,
2. there are setter methods in Builder for each of the optional properties.
3. The setter methods return this instance – for fluent API.
4. NutritionalFacts constructor takes Builder and is private.



|  |
| --- |
| **class** NutritionalFacts {  **private** **int** sodium; // mandatory property  // optional properties  **private** **int** fat;  **private** **int** carbo;  /\*\*  \* A nested static class.  \*  \*/  **static** **class** Builder {  **private** **int** sodium;  **private** **int** fat;  **private** **int** carbo;  /\*\*  \* Use constructor param for mandatory properties.  \*  \* **@param** s  \*/  **public** Builder(**int** s) {  **this**.sodium = s;  }  /\*  \* Have methods for each property that return this.  \*  \* @param f  \* @return  \*/  **public** Builder fat(**int** f) {  **this**.fat = f;  **return** **this**;  }  **public** Builder carbo(**int** c) {  **this**.carbo = c;  **return** **this**;  }  **public** NutritionalFacts build() {  **return** **new** NutritionalFacts(**this**);  }  }  **private** NutritionalFacts(Builder b) {  **this**.sodium = b.sodium;  **this**.fat = b.fat;  **this**.carbo = b.carbo;  }  @Override  **public** String toString() {  **return** "NutritionalFacts [sodium=" + sodium + ", fat=" + fat  + ", carbo=" + carbo + "]";  }      }  /\*\*  \* Builder test.  \*  \* **@author** Watsh  \*  \*/  **public** **class** BuilderExample {  **public** **static** **void** main(String[] args) {  NutritionalFacts n = **new** NutritionalFacts.Builder(10).carbo(23).fat(1)  .build();  System.*out*.println(n);  }  } |

### [Builder](http://en.wikipedia.org/wiki/Builder_pattern) (recognizeable by creational methods returning the instance itself)

* [java.lang.StringBuilder#append()](http://docs.oracle.com/javase/6/docs/api/java/lang/StringBuilder.html#append%28boolean%29) (unsynchronized)
* [java.lang.StringBuffer#append()](http://docs.oracle.com/javase/6/docs/api/java/lang/StringBuffer.html#append%28boolean%29) (synchronized)
* [java.nio.ByteBuffer#put()](http://docs.oracle.com/javase/6/docs/api/java/nio/ByteBuffer.html#put%28byte%29) (also on [CharBuffer](http://docs.oracle.com/javase/6/docs/api/java/nio/CharBuffer.html#put%28char%29), [ShortBuffer](http://docs.oracle.com/javase/6/docs/api/java/nio/ShortBuffer.html#put%28short%29), [IntBuffer](http://docs.oracle.com/javase/6/docs/api/java/nio/IntBuffer.html#put%28int%29), [LongBuffer](http://docs.oracle.com/javase/6/docs/api/java/nio/LongBuffer.html#put%28long%29),[FloatBuffer](http://docs.oracle.com/javase/6/docs/api/java/nio/FloatBuffer.html#put%28float%29) and [DoubleBuffer](http://docs.oracle.com/javase/6/docs/api/java/nio/DoubleBuffer.html#put%28double%29))
* [javax.swing.GroupLayout.Group#addComponent()](http://docs.oracle.com/javase/6/docs/api/javax/swing/GroupLayout.Group.html#addComponent%28java.awt.Component%29)
* All implementations of [java.lang.Appendable](http://docs.oracle.com/javase/6/docs/api/java/lang/Appendable.html)

1. **Prototype -** creates objects by cloning an existing object.



****

|  |
| --- |
| **class** Bike **implements** Cloneable {  **private** **int** gears;  **private** String bikeType;  **private** String model;  **public** Bike() {  bikeType = "Standard";  model = "Leopard";  gears = 4;  }  **public** Bike clone() {  **return** **new** Bike();  }  **public** **void** makeAdvanced() {  bikeType = "Advanced";  model = "Jaguar";  gears = 6;  }  **public** String getModel() {  **return** model;  }  @Override  **public** String toString() {  **return** "Bike [gears=" + gears + ", bikeType=" + bikeType + ", model="  + model + "]";  }  }  **public** **class** PrototypeExample {  /\*\*  \* Takes a basic bike object and makes an advanced bike object.  \*  \* **@param** basicBike  \* **@return**  \*/  **public** **static** Bike makeJaguar(Bike basicBike) {  basicBike.makeAdvanced();  **return** basicBike;  }  **public** **static** **void** main(String[] args) {  Bike bike = **new** Bike();  Bike basicBike = bike.clone();  System.*out*.println(basicBike);  Bike advancedBike = *makeJaguar*(basicBike);  System.*out*.println("Prototype Design Pattern: " + advancedBike);  }  } |

### [Prototype](http://en.wikipedia.org/wiki/Prototype_pattern) (recognizeable by creational methods returning a different instance of itself with the same properties)

* [java.lang.Object#clone()](http://docs.oracle.com/javase/6/docs/api/java/lang/Object.html#clone%28%29) (the class has to implement [java.lang.Cloneable](http://docs.oracle.com/javase/6/docs/api/java/lang/Cloneable.html))

1. **Singleton -** Ensure a class has only one instance, and provide a global point of access to it.



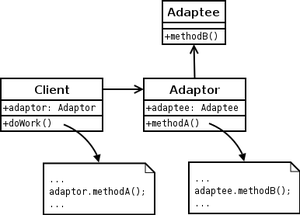
|  |
| --- |
| **package** patterns.creational;  /\*\*  \* Singleton instance created when class is loaded.  \*  \* **@author** Watsh  \*  \*/  **class** EagerSingleton {  **private** **static** EagerSingleton *INSTANCE* = **new** EagerSingleton();  **private** EagerSingleton() {  }  **public** **static** EagerSingleton getSingleInstance() {  **return** *INSTANCE*;  }  }  /\*\*  \* This is also lazy initialization. Only works for Java SE 5.0 or later.  \*  \*/  **class** DoubleCheckedLockingSingleton {  **private** **static** DoubleCheckedLockingSingleton *INSTANCE* = **null**;  **private** DoubleCheckedLockingSingleton() {  }  **public** **static** DoubleCheckedLockingSingleton getSingleInstance() {  **if** (*INSTANCE* == **null**) {  **synchronized** (DoubleCheckedLockingSingleton.**class**) {  **if** (*INSTANCE* == **null**) {  *INSTANCE* = **new** DoubleCheckedLockingSingleton();  }  }  }  **return** *INSTANCE*;  }  }  /\*\*  \* Initialization on-demand holder. The nested class is referenced no earlier  \* (and therefore loaded no earlier by the class loader) than the moment that  \* getInstance() is called. Thus, this solution is thread-safe without requiring  \* special language constructs (i.e. volatile or synchronized).  \*/  **class** Singleton {  // Private constructor prevents instantiation from other classes  **private** Singleton() {  }  /\*\*  \* SingletonHolder is loaded on the first execution of  \* Singleton.getInstance() or the first access to SingletonHolder.INSTANCE,  \* not before.  \*/  **private** **static** **class** SingletonHolder {  **private** **static** **final** Singleton *INSTANCE* = **new** Singleton();  }  **public** **static** Singleton getInstance() {  **return** SingletonHolder.*INSTANCE*;  }  }  /\*\*  \* Best approach - does not have the drawback of serializable objects. Any enum  \* value is instantiated only once in a Java program. Since Java enum values are  \* globally accessible, so is the singleton, initialized lazily by the  \* classloader.  \*  \* **@author** Watsh  \*  \*/  **enum** LazySingleton {  *INSTANCE*;  **public** **void** execute(String arg) {  System.*out*.println(arg);  }  }  **public** **class** SingletonExample {  **public** **static** **void** main(String[] args) {  LazySingleton in = LazySingleton.*INSTANCE*;  in.execute("test");  }  } |

### [Singleton](http://en.wikipedia.org/wiki/Singleton_pattern) (recognizeable by creational methods returning the same instance (usually of itself) everytime)

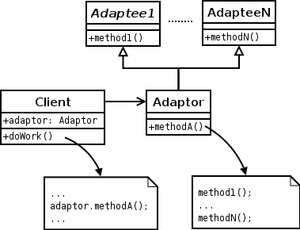
* [java.lang.Runtime#getRuntime()](http://docs.oracle.com/javase/6/docs/api/java/lang/Runtime.html#getRuntime%28%29)
* [java.awt.Desktop#getDesktop()](http://docs.oracle.com/javase/6/docs/api/java/awt/Desktop.html#getDesktop%28%29)

## Structural

1. **Adapter -** allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class. 2 types:
   1. **Object Adapter Pattern -** The adapter contains an instance of the class it wraps.



* 1. **Class Adapter Pattern -** The adapter is created by implementing or inheriting both the interface that is expected and the interface that is pre-existing.

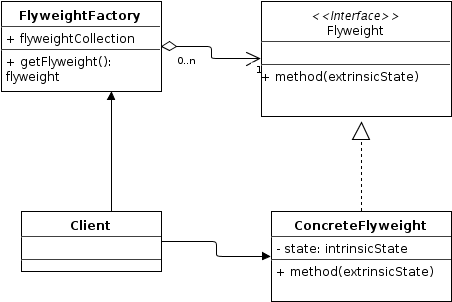


|  |
| --- |
| /\*\*  \* Adapter design pattern can be implemented in two ways. One using the  \* inheritance method and second using the composition method.  \*  \* **@author** Watsh  \*  \*/  **class** CylindricalSocket {  **public** String supply(String cylinStem1, String cylinStem2) {  String output = "Power power power...";  **return** output;  }  }  /\*\*  \* Adapter using inheritance. Class Adapter Pattern.  \*  \* **@author** Watsh  \*  \*/  **class** RectangularAdapter **extends** CylindricalSocket {  **public** String adapt(String rectaStem1, String rectaStem2) {  // some conversion logic  String cylinStem1 = rectaStem1;  String cylinStem2 = rectaStem2;  **return** supply(cylinStem1, cylinStem2);  }  }  /\*\*  \* Adapter using composition. Object Adapter pattern.  \*  \*/  **class** RectangularAdapter2 {  **private** CylindricalSocket socket;  **public** String adapt(String rectaStem1, String rectaStem2) {  // some conversion logic  socket = **new** CylindricalSocket();  String cylinStem1 = rectaStem1;  String cylinStem2 = rectaStem2;  **return** socket.supply(cylinStem1, cylinStem2);  }  }  **public** **class** AdapterExample {  **private** **static** String *rectaStem1* = "test";  **private** **static** String *rectaStem2* = "test";  **public** **static** **void** main(String[] args) {  RectangularAdapter adapter = **new** RectangularAdapter();  String power = adapter.adapt(*rectaStem1*, *rectaStem2*);  System.*out*.println(power);  }  } |

### [Adapter](http://en.wikipedia.org/wiki/Adapter_pattern) (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()](http://docs.oracle.com/javase/6/docs/api/java/util/Arrays.html#asList%28T...%29)
* [java.io.InputStreamReader(InputStream)](http://docs.oracle.com/javase/6/docs/api/java/io/InputStreamReader.html#InputStreamReader%28java.io.InputStream%29) (returns a Reader)
* [java.io.OutputStreamWriter(OutputStream)](http://docs.oracle.com/javase/6/docs/api/java/io/OutputStreamWriter.html#OutputStreamWriter%28java.io.OutputStream%29) (returns a Writer)
* [javax.xml.bind.annotation.adapters.XmlAdapter#marshal()](http://docs.oracle.com/javase/6/docs/api/javax/xml/bind/annotation/adapters/XmlAdapter.html#marshal%28BoundType%29) and [#unmarshal()](http://docs.oracle.com/javase/6/docs/api/javax/xml/bind/annotation/adapters/XmlAdapter.html#unmarshal%28ValueType%29)

1. **Flyweight -** reduces the cost of creating and manipulating a large number of similar objects.



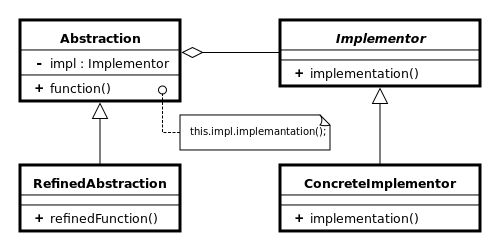
|  |
| --- |
| **import** java.util.ArrayList;  **import** java.util.HashMap;  **import** java.util.List;  **import** java.util.Map;  // Instances of CoffeeFlavour will be the Flyweights  **class** CoffeeFlavour {  **private** **final** String name;  CoffeeFlavour(String newFlavor) {  **this**.name = newFlavor;  }  @Override  **public** String toString() {  **return** name;  }  }  // Menu acts as a factory and cache for CoffeeFlavour flyweight objects  **class** Menu {  **private** Map<String, CoffeeFlavour> flavours = **new** HashMap<String, CoffeeFlavour>();  CoffeeFlavour lookup(String flavorName) {  **if** (!flavours.containsKey(flavorName))  flavours.put(flavorName, **new** CoffeeFlavour(flavorName));  **return** flavours.get(flavorName);  }  **int** totalCoffeeFlavoursMade() {  **return** flavours.size();  }  }  **class** Order {  **private** **final** **int** tableNumber;  **private** **final** CoffeeFlavour flavour;  Order(**int** tableNumber, CoffeeFlavour flavor) {  **this**.tableNumber = tableNumber;  **this**.flavour = flavor;  }  **void** serve() {  System.*out*.println("Serving " + flavour + " to table " + tableNumber);  }  }  **public** **class** FlyweightExample {  **private** **final** List<Order> orders = **new** ArrayList<Order>();  **private** **final** Menu menu = **new** Menu();  **void** takeOrder(String flavourName, **int** table) {  CoffeeFlavour flavour = menu.lookup(flavourName);  Order order = **new** Order(table, flavour);  orders.add(order);  }  **void** service() {  **for** (Order order : orders)  order.serve();  }  String report() {  **return** "\ntotal CoffeeFlavour objects made: "  + menu.totalCoffeeFlavoursMade();  }  **public** **static** **void** main(String[] args) {  FlyweightExample shop = **new** FlyweightExample();  shop.takeOrder("Cappuccino", 2);  shop.takeOrder("Frappe", 1);  shop.takeOrder("Espresso", 1);  shop.takeOrder("Frappe", 897);  shop.takeOrder("Cappuccino", 97);  shop.takeOrder("Frappe", 3);  shop.takeOrder("Espresso", 3);  shop.takeOrder("Cappuccino", 3);  shop.takeOrder("Espresso", 96);  shop.takeOrder("Frappe", 552);  shop.takeOrder("Cappuccino", 121);  shop.takeOrder("Espresso", 121);  shop.service();  System.*out*.println(shop.report());  }  } |

### [Flyweight](http://en.wikipedia.org/wiki/Flyweight_pattern) (recognizeable by creational methods returning a cached instance, a bit the "multiton" idea)

* [java.lang.Integer#valueOf(int)](http://docs.oracle.com/javase/6/docs/api/java/lang/Integer.html#valueOf%28int%29) (also on [Boolean](http://docs.oracle.com/javase/6/docs/api/java/lang/Boolean.html#valueOf%28boolean%29), [Byte](http://docs.oracle.com/javase/6/docs/api/java/lang/Byte.html#valueOf%28byte%29), [Character](http://docs.oracle.com/javase/6/docs/api/java/lang/Character.html#valueOf%28char%29), [Short](http://docs.oracle.com/javase/6/docs/api/java/lang/Short.html#valueOf%28short%29) and [Long](http://docs.oracle.com/javase/6/docs/api/java/lang/Long.html#valueOf%28long%29))

1. **Bridge -** decouples an abstraction from its implementation so that the two can vary independently.

 The bridge pattern is useful when both the class as well as what it does vary often. The class itself can be thought of as the *implementation* and what the class can do as the *abstraction*. The bridge pattern can also be thought of as two layers of abstraction.



As shown in the diagram below, we can add more password verification implementations without changing the Application code. Also we can easily unit test the verifiers on their own without affecting the existing test cases. This is very important advantage of this design pattern.

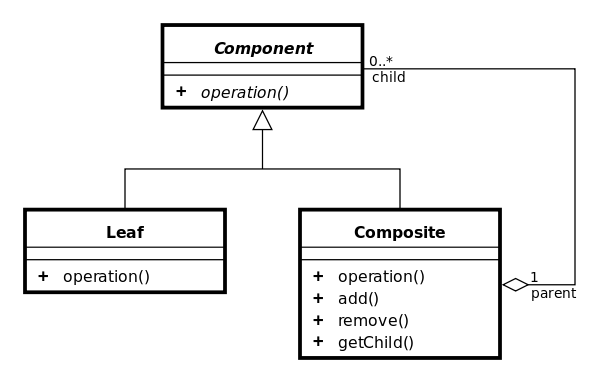
****

|  |
| --- |
| /\*\* "Implementor" \*/  **interface** Workshop {  **abstract** **public** **void** work();  }  /\*\* "ConcreteImplementor" 1/2 \*/  **class** Produce **implements** Workshop {  @Override  **public** **void** work() {  System.*out*.print("Produced");  }  }  /\*\* "ConcreteImplementor" 2/2 \*/  **class** Assemble **implements** Workshop {  @Override  **public** **void** work() {  System.*out*.println(" Assembled.");  }  }  /\*\*  \* abstraction in bridge pattern  \*  \*/  **abstract** **class** Vehicle {  **protected** Workshop workShop1;  **protected** Workshop workShop2;  **protected** Vehicle(Workshop workShop1, Workshop workShop2) {  **this**.workShop1 = workShop1;  **this**.workShop2 = workShop2;  }  **abstract** **public** **void** manufacture();  }  /\*\* "Refined Abstraction" \*/  **class** Car **extends** Vehicle {  **public** Car(Workshop workShop1, Workshop workShop2) {  **super**(workShop1, workShop2);  }  @Override  **public** **void** manufacture() {  System.*out*.print("Car ");  workShop1.work();  workShop2.work();  }  }  /\*\* "Refined Abstraction" \*/  **class** Bike **extends** Vehicle {  **public** Bike(Workshop workShop1, Workshop workShop2) {  **super**(workShop1, workShop2);  }  @Override  **public** **void** manufacture() {  System.*out*.print("Bike ");  workShop1.work();  workShop2.work();  }  }  /\*\* "Client" \*/  **public** **class** BridgeExample {  **public** **static** **void** main(String[] args) {  Vehicle vehicle1 = **new** Car(**new** Produce(), **new** Assemble());  vehicle1.manufacture();  Vehicle vehicle2 = **new** Bike(**new** Produce(), **new** Assemble());  vehicle2.manufacture();  }  } |

### [Bridge](http://en.wikipedia.org/wiki/Bridge_pattern) (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()](http://docs.oracle.com/javase/6/docs/api/java/util/Collections.html#newSetFromMap%28java.util.Map%29) and [singletonXXX()](http://docs.oracle.com/javase/6/docs/api/java/util/Collections.html#singleton%28T%29) methods however comes close.

1. **Composite -** composes zero-or-more similar objects so that they can be manipulated as one object.



File system can be modeled with composite pattern as shown below. Directory can be a composite of Files. Files could be either RegularFile (leafs) or Directories (composites).

****

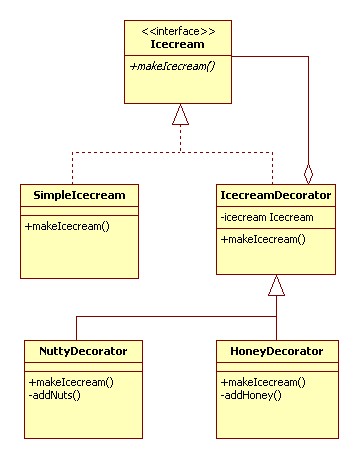
|  |
| --- |
| **import** java.util.ArrayList;  **import** java.util.List;  /\*\* Component \*/  **interface** Group {  **public** **void** assemble();  }  /\*\* Leaf \*/  **class** Block **implements** Group {  **public** **void** assemble() {  System.*out*.println("Block");  }  }  /\*\* Composite \*/  **class** Structure **implements** Group {  // Collection of child groups.  **private** List<Group> groups = **new** ArrayList<Group>();  **public** **void** assemble() {  **for** (Group group : groups) {  group.assemble();  }  }  // Adds the group to the structure.  **public** **void** add(Group group) {  groups.add(group);  }  // Removes the group from the structure.  **public** **void** remove(Group group) {  groups.remove(group);  }  }  **public** **class** CompositeExample {  **public** **static** **void** main(String[] args) {  //Initialize three blocks  Block block1 = **new** Block();  Block block2 = **new** Block();  Block block3 = **new** Block();  //Initialize three structure  Structure structure = **new** Structure();  Structure structure1 = **new** Structure();  Structure structure2 = **new** Structure();  //Composes the groups  structure1.add(block1);  structure1.add(block2);  structure2.add(block3);  structure.add(structure1);  structure.add(structure2);  structure.assemble();  }  } |

### [Composite](http://en.wikipedia.org/wiki/Composite_pattern) (recognizeable by behavioral methods taking an instance of same abstract/interface type into a tree structure)

* [java.awt.Container#add(Component)](http://docs.oracle.com/javase/6/docs/api/java/awt/Container.html#add%28java.awt.Component%29) (practically all over Swing thus)
* [javax.faces.component.UIComponent#getChildren()](http://docs.oracle.com/javaee/6/api/javax/faces/component/UIComponent.html#getChildren%28%29) (practically all over JSF UI thus)

1. **Decorator -** dynamically adds/overrides behavior in an existing method of an object.



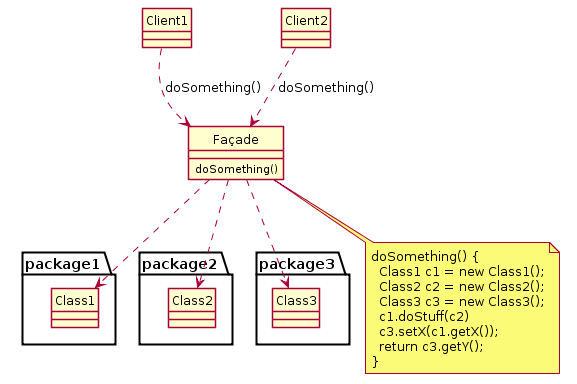


|  |
| --- |
| **interface** Icecream {  **public** String makeIcecream();  }  **class** SimpleIcecream **implements** Icecream {  @Override  **public** String makeIcecream() {  **return** "Base Icecream";  }  }  **abstract** **class** IcecreamDecorator **implements** Icecream {  **protected** Icecream specialIcecream;  **public** IcecreamDecorator(Icecream specialIcecream) {  **this**.specialIcecream = specialIcecream;  }  **public** String makeIcecream() {  **return** specialIcecream.makeIcecream();  }  }  **class** NuttyDecorator **extends** IcecreamDecorator {  **public** NuttyDecorator(Icecream specialIcecream) {  **super**(specialIcecream);  }  **public** String makeIcecream() {  **return** specialIcecream.makeIcecream() + addNuts();  }  **private** String addNuts() {  **return** " + cruncy nuts";  }  }  **class** HoneyDecorator **extends** IcecreamDecorator {    **public** HoneyDecorator(Icecream specialIcecream) {  **super**(specialIcecream);  }    **public** String makeIcecream() {  **return** specialIcecream.makeIcecream() + addHoney();  }    **private** String addHoney() {  **return** " + sweet honey";  }  }  **public** **class** DecoratorExample {  **public** **static** **void** main(String args[]) {  Icecream icecream = **new** HoneyDecorator(**new** NuttyDecorator(**new** SimpleIcecream()));  System.*out*.println(icecream.makeIcecream());  }  } |

### [Decorator](http://en.wikipedia.org/wiki/Decorator_pattern) (recognizeable by creational methods taking an instance of same abstract/interface type which adds additional behaviour)

* All subclasses of [java.io.InputStream](http://docs.oracle.com/javase/6/docs/api/java/io/InputStream.html), [OutputStream](http://docs.oracle.com/javase/6/docs/api/java/io/OutputStream.html), [Reader](http://docs.oracle.com/javase/6/docs/api/java/io/Reader.html) and [Writer](http://docs.oracle.com/javase/6/docs/api/java/io/Writer.html) have a constructor taking an instance of same type.
* [java.util.Collections](http://docs.oracle.com/javase/6/docs/api/java/util/Collections.html), the [checkedXXX()](http://docs.oracle.com/javase/6/docs/api/java/util/Collections.html#checkedCollection%28java.util.Collection,%20java.lang.Class%29), [synchronizedXXX()](http://docs.oracle.com/javase/6/docs/api/java/util/Collections.html#synchronizedCollection%28java.util.Collection%29) and [unmodifiableXXX()](http://docs.oracle.com/javase/6/docs/api/java/util/Collections.html#unmodifiableCollection%28java.util.Collection%29)methods.
* [javax.servlet.http.HttpServletRequestWrapper](http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequestWrapper.html) and [HttpServletResponseWrapper](http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletResponseWrapper.html)

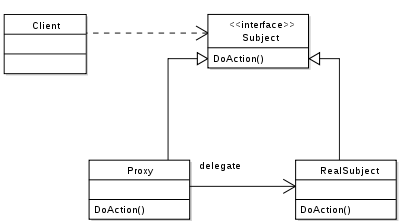
1. **Façade -** provides a simplified interface to a large body of code.



### [Facade](http://en.wikipedia.org/wiki/Facade_pattern) (recognizeable by behavioral methods which internally uses instances of different independent abstract/interface types)

* [javax.faces.context.FacesContext](http://docs.oracle.com/javaee/6/api/javax/faces/context/FacesContext.html), it internally uses among others the abstract/interface types [LifeCycle](http://docs.oracle.com/javaee/6/api/javax/faces/lifecycle/Lifecycle.html), [ViewHandler](http://docs.oracle.com/javaee/6/api/javax/faces/application/ViewHandler.html), [NavigationHandler](http://docs.oracle.com/javaee/6/api/javax/faces/application/NavigationHandler.html) and many more without that the enduser has to worry about it (which are however overrideable by injection).
* [javax.faces.context.ExternalContext](http://docs.oracle.com/javaee/6/api/javax/faces/context/ExternalContext.html), which internally uses [ServletContext](http://docs.oracle.com/javaee/6/api/javax/servlet/ServletContext.html),[HttpSession](http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpSession.html), [HttpServletRequest](http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html), [HttpServletResponse](http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletResponse.html), etc.

1. **Proxy -** provides a placeholder for another object to control access, reduce cost, and reduce complexity.



|  |
| --- |
| **import** java.lang.reflect.InvocationHandler;  **import** java.lang.reflect.Method;  **import** java.lang.reflect.Proxy;  **interface** Animal {  **public** **void** getSound();  }  **class** Lion **implements** Animal {  **public** **void** getSound() {  System.*out*.println("Roar");  }  }  /\*\*  \* InvocationHandler is the interface implemented by the invocation handler of a  \* proxy instance. Each proxy instance has an associated invocation handler.  \* When a method is invoked on a proxy instance, the method invocation is  \* encoded and dispatched to the invoke method of its invocation handler.  \*/  **class** AnimalInvocationHandler **implements** InvocationHandler {  **public** AnimalInvocationHandler(Object realSubject) {  **this**.realSubject = realSubject;  }  **public** Object invoke(Object proxy, Method m, Object[] args) {  Object result = **null**;  **try** {  /\*  \* Execute the getSound() method of realSubject.  \*/  result = m.invoke(realSubject, args);  } **catch** (Exception ex) {  ex.printStackTrace();  }  **return** result;  }  **private** Object realSubject = **null**;  }  /\*\*  \* Proxy provides static methods for creating dynamic proxy classes and  \* instances, and it is also the superclass of all dynamic proxy classes created  \* by those methods.  \*  \* **@author** Watsh  \*  \*/  **public** **class** ProxyExample {  **public** **static** **void** main(String[] args) {  Animal realSubject = **new** Lion();  ClassLoader classLoader = realSubject.getClass().getClassLoader();  /\*  \* interfaces below will be Animal  \*/  Class<?>[] interfaces = realSubject.getClass().getInterfaces();  AnimalInvocationHandler invocationHandler = **new** AnimalInvocationHandler(  realSubject);  /\*  \* Creates a dynamic proxy for Lion's realSubject instance for the  \* interface methods that Lion implements.  \*/  Animal proxy = (Animal) Proxy.*newProxyInstance*(classLoader, interfaces,  invocationHandler);  /\*  \* Calling an interface method on proxy instance will call invoke() of  \* realSubject instance's invocation Handler.  \*/  proxy.getSound();  }  } |

### [Proxy](http://en.wikipedia.org/wiki/Proxy_pattern) (recognizeable by creational methods which returns an implementation of given abstract/interface type which in turndelegates/uses a different implementation of given abstract/interface type)

* [java.lang.reflect.Proxy](http://docs.oracle.com/javase/6/docs/api/java/lang/reflect/Proxy.html)
* [java.rmi.\*](http://docs.oracle.com/javase/6/docs/api/java/rmi/package-summary.html), the whole API actually.

## Behavioral

1. **Chain of Responsibility -** delegates commands to a chain of processing objects.

|  |
| --- |
| **interface** Chain {  **public** **abstract** **void** setNext(Chain nextInChain);  **public** **abstract** **void** process(Number request);  }  **class** Number {  **private** **int** number;  **public** Number(**int** number) {  **this**.number = number;  }  **public** **int** getNumber() {  **return** number;  }  }  **class** NegativeProcessor **implements** Chain {  **private** Chain nextInChain;  **public** **void** setNext(Chain c) {  nextInChain = c;  }  **public** **void** process(Number request) {  **if** (request.getNumber() < 0) {  System.*out*.println("NegativeProcessor : " + request.getNumber());  } **else** {  nextInChain.process(request);  }  }  }  **class** ZeroProcessor **implements** Chain {  **private** Chain nextInChain;  **public** **void** setNext(Chain c) {  nextInChain = c;  }  **public** **void** process(Number request) {  **if** (request.getNumber() == 0) {  System.*out*.println("ZeroProcessor : " + request.getNumber());  } **else** {  nextInChain.process(request);  }  }  }  **class** PositiveProcessor **implements** Chain {  **private** Chain nextInChain;  **public** **void** setNext(Chain c) {  nextInChain = c;  }  **public** **void** process(Number request) {  **if** (request.getNumber() > 0) {  System.*out*.println("PositiveProcessor : " + request.getNumber());  } **else** {  nextInChain.process(request);  }  }  }  **public** **class** ChainOfResponsibilityExample {  **public** **static** **void** main(String[] args) {  // configure Chain of Responsibility  Chain c1 = **new** NegativeProcessor();  Chain c2 = **new** ZeroProcessor();  Chain c3 = **new** PositiveProcessor();  c1.setNext(c2);  c2.setNext(c3);  // calling chain of responsibility  c1.process(**new** Number(99));  c1.process(**new** Number(-30));  c1.process(**new** Number(0));  c1.process(**new** Number(100));  }  } |

### [Chain of responsibility](http://en.wikipedia.org/wiki/Chain_of_responsibility_pattern) (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()](http://docs.oracle.com/javase/6/docs/api/java/util/logging/Logger.html#log%28java.util.logging.Level,%20java.lang.String%29)
* [javax.servlet.Filter#doFilter()](http://docs.oracle.com/javaee/6/api/javax/servlet/Filter.html#doFilter%28javax.servlet.ServletRequest,%20javax.servlet.ServletResponse,%20javax.servlet.FilterChain%29)

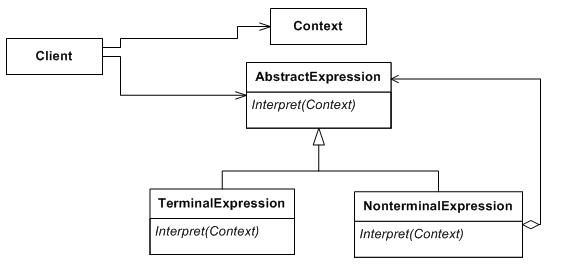
1. **Command -** creates objects which encapsulate actions and parameters.

|  |
| --- |
| **import** java.util.ArrayList;  **import** java.util.List;  **interface** ConsumerElectronics {  **public** **abstract** **void** on();  **public** **abstract** **void** mute();  }  **class** Television **implements** ConsumerElectronics {  **public** **void** on() {  System.*out*.println("Television is on!");  }  @Override  **public** **void** mute() {  System.*out*.println("Television is muted!");  }  }  **class** SoundSystem **implements** ConsumerElectronics {  **public** **void** on() {  System.*out*.println("Sound system is on!");  }  @Override  **public** **void** mute() {  System.*out*.println("Sound system is muted!");  }  }  **interface** Command {  **public** **abstract** **void** execute();  }  **class** OnCommand **implements** Command {  **private** ConsumerElectronics ce;  **public** OnCommand(ConsumerElectronics ce) {  **this**.ce = ce;  }  **public** **void** execute() {  ce.on();  }  }  **class** MuteAllCommand **implements** Command {  List<ConsumerElectronics> ceList;  **public** MuteAllCommand(List<ConsumerElectronics> ceList) {  **this**.ceList = ceList;  }  @Override  **public** **void** execute() {  **for** (ConsumerElectronics ce : ceList) {  ce.mute();  }  }  }  **class** Button {  Command c;  **public** Button(Command c) {  **this**.c = c;  }  **public** **void** click() {  c.execute();  }  }  **class** UniversalRemote {  **public** **static** ConsumerElectronics getActiveDevice() {  // here we will have a complex electronic circuit :-)  // that will maintain current device  Television tv = **new** Television();  **return** tv;  }  } |

### [Command](http://en.wikipedia.org/wiki/Command_pattern) (recognizeable by behavioral methods in an abstract/interface type which invokes a method in an implementation of adifferent abstract/interface type which has been encapsulated by the command implementation during its creation)

* All implementations of [java.lang.Runnable](http://docs.oracle.com/javase/6/docs/api/java/lang/Runnable.html)
* All implementations of [javax.swing.Action](http://docs.oracle.com/javase/6/docs/api/javax/swing/Action.html)

1. **Interpreter -** implements a specialized language.



|  |
| --- |
| **import** java.util.Stack;  **interface** IExpression {  **public** **int** interpret();  }  /\*\* Terminal Expression \*/  **class** NumberExpression **implements** IExpression {  **int** number;  **public** NumberExpression(**int** i) {  number = i;  }  **public** NumberExpression(String s) {  number = Integer.*parseInt*(s);  }  @Override  **public** **int** interpret() {  **return** number;  }  }  /\*\* Non-terminal expression \*/  **class** MultiplyExpression **implements** IExpression {  IExpression leftExpression;  IExpression rightExpresion;  **public** MultiplyExpression(IExpression leftExpression,  IExpression rightExpresion) {  **this**.leftExpression = leftExpression;  **this**.rightExpresion = rightExpresion;  }  @Override  **public** **int** interpret() {  **return** leftExpression.interpret() \* rightExpresion.interpret();  }  }  **class** PlusExpression **implements** IExpression {  IExpression leftExpression;  IExpression rightExpresion;  **public** PlusExpression(IExpression leftExpression, IExpression rightExpresion) {  **this**.leftExpression = leftExpression;  **this**.rightExpresion = rightExpresion;  }  @Override  **public** **int** interpret() {  **return** leftExpression.interpret() + rightExpresion.interpret();  }  }  **class** MinusExpression **implements** IExpression {  IExpression leftExpression;  IExpression rightExpresion;  **public** MinusExpression(IExpression leftExpression,  IExpression rightExpresion) {  **this**.leftExpression = leftExpression;  **this**.rightExpresion = rightExpresion;  }  @Override  **public** **int** interpret() {  **return** leftExpression.interpret() - rightExpresion.interpret();  }  }  /\*-  \* Postfix calculator.  \*  \* Algorithm for postfix expression parser implementation is simpler than for  \* infix.  \*  \* Read token one by one and loop till end of expression  \* Is read element a number  \* 1. true then, push it to a stack  \* 2. false then,  \* 1. pop two elements from stack  \* 2. apply the operator  \* 3. push the result to stack  \*  \*/  **public** **class** InterpreterExample {  **public** **static** **void** main(String args[]) {  String tokenString = "4 3 2 - 1 + \*";  Stack<IExpression> stack = **new** Stack<IExpression>();  String[] tokenList = tokenString.split(" ");  **for** (String s : tokenList) {  **if** (*isOperator*(s)) {  IExpression rightExpression = stack.pop();  IExpression leftExpression = stack.pop();  IExpression operator = *getOperatorInstance*(s, leftExpression,  rightExpression);  **int** result = operator.interpret();  stack.push(**new** NumberExpression(result));  } **else** {  IExpression i = **new** NumberExpression(s);  stack.push(i);  }  }  System.*out*.println("Result: " + stack.pop().interpret());  }  **public** **static** **boolean** isOperator(String s) {  **if** (s.equals("+") || s.equals("-") || s.equals("\*"))  **return** **true**;  **else**  **return** **false**;  }  **public** **static** IExpression getOperatorInstance(String s, IExpression left,  IExpression right) {  **switch** (s) {  **case** "+":  **return** **new** PlusExpression(left, right);  **case** "-":  **return** **new** MinusExpression(left, right);  **case** "\*":  **return** **new** MultiplyExpression(left, right);  }  **return** **null**;  }  } |

### [Interpreter](http://en.wikipedia.org/wiki/Interpreter_pattern) (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](http://docs.oracle.com/javase/6/docs/api/java/util/regex/Pattern.html)
* [java.text.Normalizer](http://docs.oracle.com/javase/6/docs/api/java/text/Normalizer.html)
* All subclasses of [java.text.Format](http://docs.oracle.com/javase/6/docs/api/java/text/Format.html)
* All subclasses of [javax.el.ELResolver](http://docs.oracle.com/javaee/6/api/javax/el/ELResolver.html)

1. **Iterator -** accesses the elements of an object sequentially without exposing its underlying representation.

|  |
| --- |
| **import** java.util.ArrayList;  **import** java.util.List;  **class** Animal {  **private** String animalName;  **private** String animalType;  **public** Animal(String animalName, String animalType) {  **this**.animalName = animalName;  **this**.animalType = animalType;  }  **public** String getAnimalName() {  **return** animalName;  }  **public** **void** setAnimalName(String animalName) {  **this**.animalName = animalName;  }  **public** String getAnimalType() {  **return** animalType;  }  **public** **void** setAnimalType(String animalType) {  **this**.animalType = animalType;  }  }  **interface** Iterator {  **public** Animal nextAnimal();  **public** **boolean** isLastAnimal();  **public** Animal currentAnimal();  }  **class** WildIterator **implements** Iterator {  **public** List<Animal> animalList;  **private** **int** position;  **public** WildIterator(List<Animal> animalList) {  **this**.animalList = animalList;  }  @Override  **public** Animal nextAnimal() {  Animal animal = **null**;  **for** (; position < animalList.size(); position++) {  **if** ("Wild".equals((animalList.get(position)).getAnimalType())) {  animal = animalList.get(position);  position++;  **break**;  }  }  **return** animal;  }  @Override  **public** **boolean** isLastAnimal() {  **for** (**int** i = position; i < animalList.size(); i++) {  **if** ("Wild".equals((animalList.get(i)).getAnimalType())) {  **return** **false**;  }  }  **return** **true**;  }  @Override  **public** Animal currentAnimal() {  **if** (position < animalList.size()) {  **return** animalList.get(position);  }  **return** **null**;  }  }  **class** DomesticIterator **implements** Iterator {  List<Animal> animalList;  **private** **int** position;  **public** DomesticIterator(List<Animal> animalList) {  **this**.animalList = animalList;  }  @Override  **public** Animal nextAnimal() {  Animal animal = **null**;  **for** (; position < animalList.size(); position++) {  **if** ("Domestic".equals((animalList.get(position)).getAnimalType())) {  animal = animalList.get(position);  position++;  **break**;  }  }  **return** animal;  }  @Override  **public** **boolean** isLastAnimal() {  **for** (**int** i = position; i < animalList.size(); i++) {  **if** ("Domestic".equals((animalList.get(i)).getAnimalType())) {  **return** **false**;  }  }  **return** **true**;  }  @Override  **public** Animal currentAnimal() {  **if** (position < animalList.size()) {  **return** animalList.get(position);  }  **return** **null**;  }  }  **interface** IZoo {  **public** List<Animal> getAnimals();  **public** **void** addAnimal(Animal animal);  **public** **void** removeAnimal(Animal animal);  **public** Iterator createIterator(String iteratorType);  }  **class** ZooImpl **implements** IZoo {  List<Animal> animalList;  **public** ZooImpl() {  animalList = **new** ArrayList<Animal>();  }  @Override  **public** List<Animal> getAnimals() {  **return** animalList;  }  @Override  **public** **void** addAnimal(Animal animal) {  animalList.add(animal);  }  @Override  **public** **void** removeAnimal(Animal animal) {  animalList.remove(animal);  }  @Override  **public** Iterator createIterator(String iteratorType) {  **if** ("Wild".equals(iteratorType)) {  **return** **new** WildIterator(animalList);  } **else** {  **return** **new** DomesticIterator(animalList);  }  }  }  **public** **class** IteratorExample {  **public** **static** **void** main(String args[]) {  ZooImpl zoo = **new** ZooImpl();  zoo.addAnimal(**new** Animal("Tiger", "Wild"));  zoo.addAnimal(**new** Animal("Lion", "Wild"));  zoo.addAnimal(**new** Animal("Tom Cat", "Domestic"));  zoo.addAnimal(**new** Animal("Raging Bull", "Wild"));  zoo.addAnimal(**new** Animal("Scooby Doo", "Domestic"));  Iterator wildIterator = zoo.createIterator("Wild");  **while** (!wildIterator.isLastAnimal()) {  System.*out*.println("Wild Animal: "  + wildIterator.nextAnimal().getAnimalName());  }  Iterator domesticIterator = zoo.createIterator("Domestic");  **while** (!domesticIterator.isLastAnimal()) {  System.*out*.println("Domestic Animal: "  + domesticIterator.nextAnimal().getAnimalName());  }  }  } |

### [Iterator](http://en.wikipedia.org/wiki/Iterator_pattern) (recognizeable by behavioral methods sequentially returning instances of a different type from a queue)

* All implementations of [java.util.Iterator](http://docs.oracle.com/javase/6/docs/api/java/util/Iterator.html) (thus among others also [java.util.Scanner](http://docs.oracle.com/javase/6/docs/api/java/util/Scanner.html)!).
* All implementations of [java.util.Enumeration](http://docs.oracle.com/javase/6/docs/api/java/util/Enumeration.html)

As of Java 5, objects implementing the [Iterable](http://docs.oracle.com/javase/7/docs/api/java/lang/Iterable.html) interface, which returns an Iterator from its only method, can be traversed using the[enhanced for loop syntax](http://en.wikipedia.org/wiki/Foreach_loop#Java).[[2]](http://en.wikipedia.org/wiki/Iterator_pattern#cite_note-2) The [Collection](http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html) interface from the [Java collections framework](http://en.wikipedia.org/wiki/Java_collections_framework) extends Iterable.

1. **Mediator -** allows [loose coupling](http://en.wikipedia.org/wiki/Loose_coupling) between classes by being the only class that has detailed knowledge of their methods.

|  |
| --- |
| **interface** IATCMediator {  **public** **void** registerRunway(Runway runway);  **public** **void** registerFlight(Flight flight);  **public** **boolean** isLandingOk();  **public** **void** setLandingStatus(**boolean** status);  }  **interface** ICommand {  **void** land();  }  **class** Flight **implements** ICommand {  **private** IATCMediator atcMediator;  **public** Flight(IATCMediator atcMediator) {  **this**.atcMediator = atcMediator;  }  **public** **void** land() {  **if** (atcMediator.isLandingOk()) {  System.*out*.println("Landing done....");  atcMediator.setLandingStatus(**true**);  } **else**  System.*out*.println("Will wait to land....");  }  **public** **void** getReady() {  System.*out*.println("Getting ready...");  }  }  **class** Runway **implements** ICommand {  **private** IATCMediator atcMediator;  **public** Runway(IATCMediator atcMediator) {  **this**.atcMediator = atcMediator;  atcMediator.setLandingStatus(**true**);  }  @Override  **public** **void** land() {  System.*out*.println("Landing permission granted...");  atcMediator.setLandingStatus(**true**);  }  }  **class** ATCMediator **implements** IATCMediator {  **private** Flight flight;  **private** Runway runway;  **public** **boolean** land;  **public** **void** registerRunway(Runway runway) {  **this**.runway = runway;  }  **public** **void** registerFlight(Flight flight) {  **this**.flight = flight;  }  **public** **boolean** isLandingOk() {  **return** land;  }  @Override  **public** **void** setLandingStatus(**boolean** status) {  land = status;  }  }  **public** **class** MediatorExample {  **public** **static** **void** main(String args[]) {  IATCMediator atcMediator = **new** ATCMediator();  Flight sparrow101 = **new** Flight(atcMediator);  Runway mainRunway = **new** Runway(atcMediator);  atcMediator.registerFlight(sparrow101);  atcMediator.registerRunway(mainRunway);  sparrow101.getReady();  mainRunway.land();  sparrow101.land();  }  } |

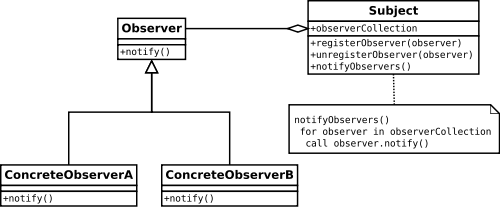
### [Mediator](http://en.wikipedia.org/wiki/Mediator_pattern) (recognizeable by behavioral methods taking an instance of different abstract/interface type (usually using the command pattern) which delegates/uses the given instance)

* [java.util.Timer](http://docs.oracle.com/javase/6/docs/api/java/util/Timer.html) (all scheduleXXX() methods)
* [java.util.concurrent.Executor#execute()](http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/Executor.html#execute%28java.lang.Runnable%29)
* [java.util.concurrent.ExecutorService](http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/ExecutorService.html) (the invokeXXX() and submit() methods)
* [java.util.concurrent.ScheduledExecutorService](http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/ScheduledExecutorService.html) (all scheduleXXX() methods)
* [java.lang.reflect.Method#invoke()](http://docs.oracle.com/javase/6/docs/api/java/lang/reflect/Method.html#invoke%28java.lang.Object,%20java.lang.Object...%29)

1. **Memento -** provides the ability to restore an object to its previous state (undo).

|  |
| --- |
| **import** java.util.ArrayList;  **import** java.util.List;  **class** Memento {  **private** String state;  **public** Memento(String state) {  **this**.state = state;  }  **public** String getState() {  **return** state;  }  }  **class** Originator {  // this String is just for example  // in real world application this  // will be a java class - the object  // for which the state to be stored  **private** String state;  **public** **void** setState(String state) {  **this**.state = state;  }  **public** String getState() {  **return** state;  }  **public** Memento createMemento() {  **return** **new** Memento(state);  }  **public** **void** setMemento(Memento memento) {  state = memento.getState();  }  }  **class** Caretaker {  **private** List<Memento> statesList = **new** ArrayList<Memento>();  **public** **void** addMemento(Memento m) {  statesList.add(m);  }  **public** Memento getMemento(**int** index) {  **return** statesList.get(index);  }  }  **public** **class** MementoExample {  **public** **static** **void** main(String[] args) {  Originator originator = **new** Originator();  originator.setState("Lion");  Memento memento = originator.createMemento();  Caretaker caretaker = **new** Caretaker();  caretaker.addMemento(memento);  originator.setState("Tiger");  originator.setState("Horse");  memento = originator.createMemento();  caretaker.addMemento(memento);  originator.setState("Elephant");  System.*out*  .println("Originator Current State: " + originator.getState());  System.*out*.println("Originator restoring to previous state...");  memento = caretaker.getMemento(1);  originator.setMemento(memento);  System.*out*  .println("Originator Current State: " + originator.getState());  System.*out*.println("Again restoring to previous state...");  memento = caretaker.getMemento(0);  originator.setMemento(memento);  System.*out*  .println("Originator Current State: " + originator.getState());  }  } |

1. **Observer -** is a publish/subscribe pattern which allows a number of observer objects to see an event.

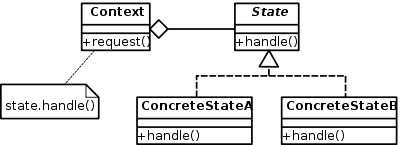


|  |
| --- |
| **import** java.util.ArrayList;  **import** java.util.List;  /\*  \* Same as Observable.  \*/  **interface** Subject {  **public** **void** registerObserver(Observer observer);  **public** **void** notifyObserver();  **public** **void** unRegisterObserver(Observer observer);  **public** Object getUpdate();  }  **class** Blog **implements** Subject {  List<Observer> observersList;  **private** **boolean** stateChange;  **public** Blog() {  **this**.observersList = **new** ArrayList<Observer>();  stateChange = **false**;  }  **public** **void** registerObserver(Observer observer) {  observersList.add(observer);  }  **public** **void** unRegisterObserver(Observer observer) {  observersList.remove(observer);  }  **public** **void** notifyObserver() {  **if** (stateChange) {  **for** (Observer observer : observersList) {  observer.update();  }  }  }  **public** Object getUpdate() {  Object changedState = **null**;  // should have logic to send the  // state change to querying observer  **if** (stateChange) {  changedState = "Observer Design Pattern";  }  **return** changedState;  }  **public** **void** postNewArticle() {  stateChange = **true**;  notifyObserver();  }  }  **interface** Observer {  **public** **void** update();  **public** **void** setSubject(Subject subject);  }  **class** User **implements** Observer {  **private** String article;  **private** Subject blog;  **public** **void** setSubject(Subject blog) {  **this**.blog = blog;  article = "No New Article!";  }  @Override  **public** **void** update() {  System.*out*.println("State change reported by Subject.");  article = (String) blog.getUpdate();  }  **public** String getArticle() {  **return** article;  }  }  **public** **class** ObserverExample {  **public** **static** **void** main(String args[]) {  Blog blog = **new** Blog();  User user1 = **new** User();  User user2 = **new** User();  blog.registerObserver(user1);  blog.registerObserver(user2);  user1.setSubject(blog);  user2.setSubject(blog);  System.*out*.println(user1.getArticle());  blog.postNewArticle();  System.*out*.println(user1.getArticle());  }  } |

### [Observer (or Publish/Subscribe)](http://en.wikipedia.org/wiki/Observer_pattern) (recognizeable by behavioral methods which invokes a method on an instance of anotherabstract/interface type, depending on own state)

* [java.util.Observer](http://docs.oracle.com/javase/6/docs/api/java/util/Observer.html)/[java.util.Observable](http://docs.oracle.com/javase/6/docs/api/java/util/Observable.html) (rarely used in real world though)
* All implementations of [java.util.EventListener](http://docs.oracle.com/javase/6/docs/api/java/util/EventListener.html) (practically all over Swing thus)
* [javax.servlet.http.HttpSessionBindingListener](http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpSessionBindingListener.html)
* [javax.servlet.http.HttpSessionAttributeListener](http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpSessionAttributeListener.html)
* [javax.faces.event.PhaseListener](http://docs.oracle.com/javaee/6/api/javax/faces/event/PhaseListener.html)

1. **State -** allows an object to alter its behavior when it’s internal state changes.

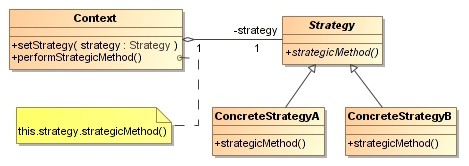


|  |
| --- |
| /\*\*  \* State interface.  \*/  **interface** MobileAlertState {  **public** **void** alert(AlertStateContext ctx);  }  **class** Vibration **implements** MobileAlertState {  @Override  **public** **void** alert(AlertStateContext ctx) {  System.*out*.println("vibration...");  }  }  **class** Silent **implements** MobileAlertState {  @Override  **public** **void** alert(AlertStateContext ctx) {  System.*out*.println("silent...");  }  }  /\*\*  \* This class maintains the current state and is the core of the state design  \* pattern. A client should access / run the whole setup through this class.  \*/  **class** AlertStateContext {  **private** MobileAlertState currentState;  **public** AlertStateContext() {  currentState = **new** Vibration();  }  **public** **void** setState(MobileAlertState state) {  currentState = state;  }  **public** **void** alert() {  currentState.alert(**this**);  }  }  **public** **class** StateExample {  **public** **static** **void** main(String[] args) {  AlertStateContext stateContext = **new** AlertStateContext();  stateContext.alert();  stateContext.alert();  stateContext.setState(**new** Silent());  stateContext.alert();  stateContext.alert();  stateContext.alert();  }  } |

### [State](http://en.wikipedia.org/wiki/State_pattern) (recognizeable by behavioral methods which changes its behaviour depending on the instance's state which can be controlled externally)

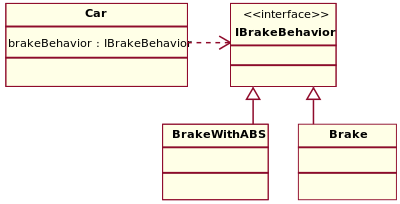
* [javax.faces.lifecycle.LifeCycle#execute()](http://docs.oracle.com/javaee/6/api/javax/faces/lifecycle/Lifecycle.html#execute%28javax.faces.context.FacesContext%29) (controlled by [FacesServlet](http://docs.oracle.com/javaee/6/api/javax/faces/webapp/FacesServlet.html), the behaviour is dependent on current phase (state) of JSF lifecycle)

1. **Strategy -** allows one of a family of algorithms to be selected on-the-fly at runtime.



|  |
| --- |
| /\*\*  \* The classes that implement a concrete strategy should implement this. The  \* Context class uses this to call the concrete strategy.  \*/  **interface** Strategy {  **int** execute(**int** a, **int** b);  };  /\*\* Implements the algorithm using the strategy interface \*/  **class** Add **implements** Strategy {  **public** **int** execute(**int** a, **int** b) {  System.*out*.println("Called Add's execute()");  **return** a + b; // Do an addition with a and b  }  };  **class** Subtract **implements** Strategy {  **public** **int** execute(**int** a, **int** b) {  System.*out*.println("Called Subtract's execute()");  **return** a - b; // Do a subtraction with a and b  }  };  **class** Multiply **implements** Strategy {  **public** **int** execute(**int** a, **int** b) {  System.*out*.println("Called Multiply's execute()");  **return** a \* b; // Do a multiplication with a and b  }  };  // Configured with a ConcreteStrategy object and maintains  // a reference to a Strategy object  **class** Context {  **private** Strategy strategy;  **public** Context(Strategy strategy) {  **this**.strategy = strategy;  }  **public** **int** executeStrategy(**int** a, **int** b) {  **return** **this**.strategy.execute(a, b);  }  };  /\*\* Tests the pattern \*/  **class** StrategyExample {  **public** **static** **void** main(String[] args) {  Context context;  // Three contexts following different strategies  context = **new** Context(**new** Add());  **int** resultA = context.executeStrategy(3, 4);  context = **new** Context(**new** Subtract());  **int** resultB = context.executeStrategy(3, 4);  context = **new** Context(**new** Multiply());  **int** resultC = context.executeStrategy(3, 4);  System.*out*.println("Result A : " + resultA);  System.*out*.println("Result B : " + resultB);  System.*out*.println("Result C : " + resultC);  }  } |

## **Strategy and open/closed principle[[edit](http://en.wikipedia.org/w/index.php?title=Strategy_pattern&action=edit&section=3" \o "Edit section: Strategy and open/closed principle)]**

[](http://en.wikipedia.org/wiki/File:StrategyPattern_IBrakeBehavior.svg)

[http://bits.wikimedia.org/static-1.23wmf18/skins/common/images/magnify-clip.png](http://en.wikipedia.org/wiki/File:StrategyPattern_IBrakeBehavior.svg)

Accelerate and [brake](http://en.wikipedia.org/wiki/Brake) behaviors must be declared in each new [car model](http://en.wikipedia.org/wiki/Car_model).

According to the strategy pattern, the behaviors of a class should not be inherited. Instead they should be encapsulated using interfaces. As an example, consider a car class. Two possible functionalities for car are *brake* and *accelerate*.

Since accelerate and brake behaviors change frequently between models, a common approach is to implement these behaviors in subclasses. This approach has significant drawbacks: accelerate and brake behaviors must be declared in each new Car model. The work of managing these behaviors increases greatly as the number of models increases, and requires code to be duplicated across models. Additionally, it is not easy to determine the exact nature of the behavior for each model without investigating the code in each.

The strategy pattern uses aggregation instead of inheritance. In the strategy pattern, behaviors are defined as separate interfaces and specific classes that implement these interfaces. Specific classes encapsulate these interfaces. This allows better decoupling between the behavior and the class that uses the behavior. The behavior can be changed without breaking the classes that use it, and the classes can switch between behaviors by changing the specific implementation used without requiring any significant code changes. Behaviors can also be changed at run-time as well as at design-time. For instance, a car object’s brake behavior can be changed from BrakeWithABS() to Brake() by changing the brakeBehavior member to:

brakeBehavior = new Brake();

This gives greater flexibility in design and is in harmony with the [Open/closed principle](http://en.wikipedia.org/wiki/Open/closed_principle) (OCP) that states that classes should be open for extension but closed for modification.

### [Strategy](http://en.wikipedia.org/wiki/Strategy_pattern) (recognizeable by behavioral methods in an abstract/interface type which invokes a method in an implementation of adifferent abstract/interface type which has been passed-in as method argument into the strategy implementation)

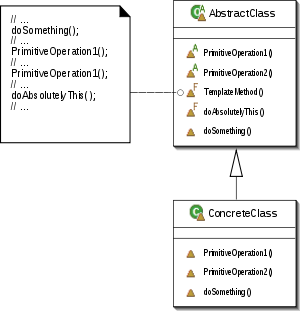
* [java.util.Comparator#compare()](http://docs.oracle.com/javase/6/docs/api/java/util/Comparator.html#compare%28T,%20T%29), executed by among others Collections#sort().
* [javax.servlet.http.HttpServlet](http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServlet.html), the service() and all doXXX() methods takeHttpServletRequest and HttpServletResponse and the implementor has to process them (and not to get hold of them as instance variables!).
* [javax.servlet.Filter#doFilter()](http://docs.oracle.com/javaee/6/api/javax/servlet/Filter.html#doFilter%28javax.servlet.ServletRequest,%20javax.servlet.ServletResponse,%20javax.servlet.FilterChain%29)

1. **Template Method -** defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior.

The template method is used in frameworks, where each implements the invariant parts of a domain's architecture, leaving "placeholders" for customisation options. This is an example for [inversion of control](http://en.wikipedia.org/wiki/Inversion_of_control), also called the [Hollywood principle](http://en.wikipedia.org/wiki/Hollywood_principle). Reasons to use the template method are to:[[4]](http://en.wikipedia.org/wiki/Template_method_pattern" \l "cite_note-4)

* Let subclasses implement (through [method overriding](http://en.wikipedia.org/wiki/Method_overriding_(programming))) behavior that can vary.
* Avoid duplication in the code: the general workflow structure is implemented once in the abstract class's [algorithm](http://en.wikipedia.org/wiki/Algorithm), and necessary variations are implemented in each of the subclasses.
* Control at what point(s) [subclassing](http://en.wikipedia.org/wiki/Inheritance_(computer_science)) is allowed. As opposed to a simple polymorphic override, where the base method would be entirely rewritten allowing radical change to the workflow, only the specific details of the workflow are allowed to change.

The control structure ([inversion of control](http://en.wikipedia.org/wiki/Inversion_of_control)) that is the result of the application of a template pattern is often referred to as the Hollywood Principle: "Don't call us, we'll call you." Using this principle, the template method in a parent class controls the overall process by calling subclass methods as required.

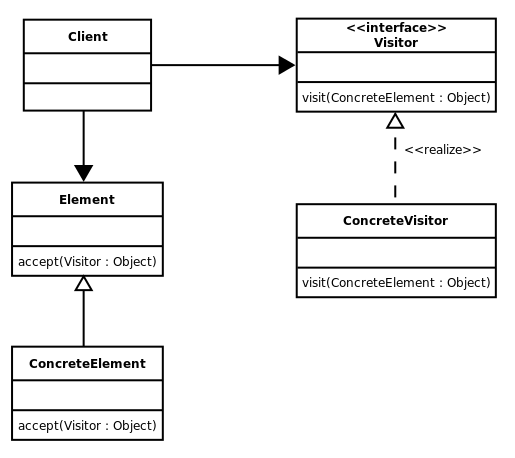


|  |
| --- |
| **import** java.util.Random;  /\*\*  \* An abstract class that is common to several games in which players play  \* against the others, but only one is playing at a given time.  \*/  **abstract** **class** Game {  **protected** **int** playersCount;  **abstract** **void** initializeGame();  **abstract** **void** makePlay(**int** player);  **abstract** **boolean** endOfGame();  **abstract** **void** printWinner();  /\* A template method : \*/  **public** **final** **void** playOneGame(**int** playersCount) {  **this**.playersCount = playersCount;  initializeGame();  **int** j = 0;  **while** (!endOfGame()) {  makePlay(j);  j = (j + 1) % playersCount;  }  printWinner();  }  }  // Now we can extend this class in order  // to implement actual games:  **class** Monopoly **extends** Game {  **private** **int** currentPlayer;  /\* Implementation of necessary concrete methods \*/  **void** initializeGame() {  // Initialize players  // Initialize money  }  **void** makePlay(**int** player) {  // Process one turn of player  System.*out*.println("Playing player: " + player);  **this**.currentPlayer = player;  }  **boolean** endOfGame() {  // Return true if game is over  // according to Monopoly rules  Random rand = **new** Random();  **return** rand.nextInt(100) < 5 ? **true** : **false**;  }  **void** printWinner() {  // Display who won  System.*out*.println("Winner of Monopoly game is player: "  + currentPlayer);  }  /\* Specific declarations for the Monopoly game. \*/  // ...  }  **class** Chess **extends** Game {  **private** **int** currentPlayer;  /\* Implementation of necessary concrete methods \*/  **void** initializeGame() {  // Initialize players  // Put the pieces on the board  }  **void** makePlay(**int** player) {  // Process a turn for the player  System.*out*.println("Playing player: " + player);  **this**.currentPlayer = player;  }  **boolean** endOfGame() {  // Return true if in Checkmate or  // Stalemate has been reached  Random rand = **new** Random();  **return** rand.nextInt(100) < 10 ? **true** : **false**;  }  **void** printWinner() {  System.*out*.println("Winner of Chess game is player: " + currentPlayer);  }  /\* Specific declarations for the chess game. \*/  // ...  }  **public** **class** TemplateMethodExample {  **public** **static** **void** main(String[] args) {  Game g1 = **new** Chess();  g1.playOneGame(2);    Monopoly m1 = **new** Monopoly();  m1.playOneGame(4);  }  } |

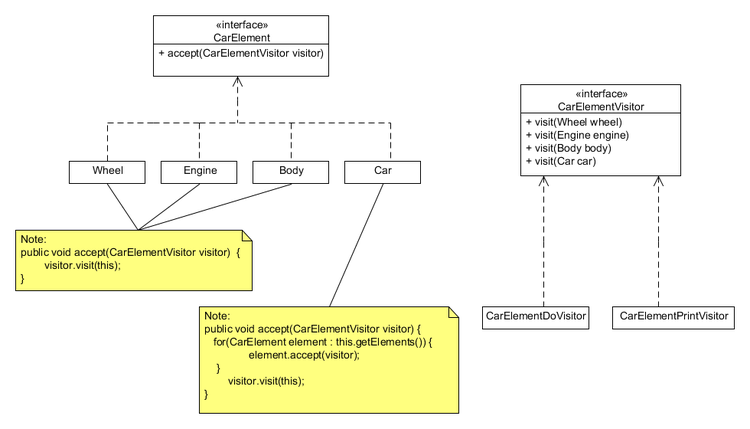
### [Template method](http://en.wikipedia.org/wiki/Template_method_pattern) (recognizeable by behavioral methods which already have a "default" behaviour definied by an abstract type)

* All non-abstract methods of [java.io.InputStream](http://docs.oracle.com/javase/6/docs/api/java/io/InputStream.html), [java.io.OutputStream](http://docs.oracle.com/javase/6/docs/api/java/io/OutputStream.html), [java.io.Reader](http://docs.oracle.com/javase/6/docs/api/java/io/Reader.html)and [java.io.Writer](http://docs.oracle.com/javase/6/docs/api/java/io/Writer.html).
* All non-abstract methods of [java.util.AbstractList](http://docs.oracle.com/javase/6/docs/api/java/util/AbstractList.html), [java.util.AbstractSet](http://docs.oracle.com/javase/6/docs/api/java/util/AbstractSet.html) and[java.util.AbstractMap](http://docs.oracle.com/javase/6/docs/api/java/util/AbstractMap.html).
* [javax.servlet.http.HttpServlet](http://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServlet.html), 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.

1. **Visitor -** separates an algorithm from an object structure by moving the hierarchy of methods into one object.



In the example below, every car element has to do perform same algorithm (print and do, but in their own way) so we can take out the algorithms into their own classes and invoke them when the corresponding visit method is called. In future more algorithms may be added to all car elements as a new class extending CarElementVisitor.



|  |
| --- |
| **interface** ICarElementVisitor {  **void** visit(Wheel wheel);  **void** visit(Engine engine);  **void** visit(Body body);  **void** visit(Car car);  }  **interface** ICarElement {  **void** accept(ICarElementVisitor visitor); // CarElements have to provide  // accept().  }  **class** Wheel **implements** ICarElement {  **private** String name;  **public** Wheel(String name) {  **this**.name = name;  }  **public** String getName() {  **return** **this**.name;  }  **public** **void** accept(ICarElementVisitor visitor) {  /\*  \* accept(ICarElementVisitor) in Wheel implements  \* accept(ICarElementVisitor) in ICarElement, so the call to accept is  \* bound at run time. This can be considered the first dispatch.  \* However, the decision to call visit(Wheel) (as opposed to  \* visit(Engine) etc.) can be made during compile time since 'this' is  \* known at compile time to be a Wheel. Moreover, each implementation of  \* ICarElementVisitor implements the visit(Wheel), which is another  \* decision that is made at run time. This can be considered the second  \* dispatch.  \*/  visitor.visit(**this**);  }  }  **class** Engine **implements** ICarElement {  **public** **void** accept(ICarElementVisitor visitor) {  visitor.visit(**this**);  }  }  **class** Body **implements** ICarElement {  **public** **void** accept(ICarElementVisitor visitor) {  visitor.visit(**this**);  }  }  **class** Car **implements** ICarElement {  ICarElement[] elements;  **public** Car() {  // create new Array of elements  **this**.elements = **new** ICarElement[] { **new** Wheel("front left"),  **new** Wheel("front right"), **new** Wheel("back left"),  **new** Wheel("back right"), **new** Body(), **new** Engine() };  }  **public** **void** accept(ICarElementVisitor visitor) {  **for** (ICarElement elem : elements) {  elem.accept(visitor);  }  visitor.visit(**this**);  }  }  **class** CarElementPrintVisitor **implements** ICarElementVisitor {  **public** **void** visit(Wheel wheel) {  System.*out*.println("Visiting " + wheel.getName() + " wheel");  }  **public** **void** visit(Engine engine) {  System.*out*.println("Visiting engine");  }  **public** **void** visit(Body body) {  System.*out*.println("Visiting body");  }  **public** **void** visit(Car car) {  System.*out*.println("Visiting car");  }  }  **class** CarElementDoVisitor **implements** ICarElementVisitor {  **public** **void** visit(Wheel wheel) {  System.*out*.println("Kicking my " + wheel.getName() + " wheel");  }  **public** **void** visit(Engine engine) {  System.*out*.println("Starting my engine");  }  **public** **void** visit(Body body) {  System.*out*.println("Moving my body");  }  **public** **void** visit(Car car) {  System.*out*.println("Starting my car");  }  }  **public** **class** VisitorExample {  **public** **static** **void** main(String[] args) {  ICarElement car = **new** Car();  car.accept(**new** CarElementPrintVisitor());  car.accept(**new** CarElementDoVisitor());  }  } |

### [Visitor](http://en.wikipedia.org/wiki/Visitor_pattern) (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](http://docs.oracle.com/javase/6/docs/api/javax/lang/model/element/AnnotationValue.html) and [AnnotationValueVisitor](http://docs.oracle.com/javase/6/docs/api/javax/lang/model/element/AnnotationValueVisitor.html)
* [javax.lang.model.element.Element](http://docs.oracle.com/javase/6/docs/api/javax/lang/model/element/Element.html) and [ElementVisitor](http://docs.oracle.com/javase/6/docs/api/javax/lang/model/element/ElementVisitor.html)
* [javax.lang.model.type.TypeMirror](http://docs.oracle.com/javase/6/docs/api/javax/lang/model/type/TypeMirror.html) and [TypeVisitor](http://docs.oracle.com/javase/6/docs/api/javax/lang/model/type/TypeVisitor.html)

# Object Oriented Programming Principles

## SOLID

1. **S**ingle Responsibility Principle – there should not be more than one reason for a class to change, or a class should always handle single functionality. If you put more than one functionality in one [Class in Java](http://javarevisited.blogspot.com/2011/10/class-in-java-programming-general.html)  it introduce **coupling** between two functionality and even if you change one functionality there is chance you broke coupled functionality,  which require another round of testing to avoid any surprise on production environment.
2. **O**pen/Closed principle – Classes, methods or functions should be Open for extension (new functionality) and Closed for modification.  The idea was that once completed, the implementation of a class could only be modified to correct errors; new or changed features would require that a different class be created. That class could reuse coding from the original class through [inheritance](http://en.wikipedia.org/wiki/Inheritance_(computer_science)). The derived subclass might or might not have the same [interface](http://en.wikipedia.org/wiki/Protocol_(object-oriented_programming)) as the original class.

Meyer's definition advocates [implementation inheritance](http://en.wikipedia.org/wiki/Implementation_inheritance). Implementation can be reused through inheritance but interface specifications need not be. The existing implementation is closed to modifications, and new implementations need not implement the existing interface.

1. **L**iskov substitution principle - “objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.”
2. **I**nterface segregation principle - “many client-specific interfaces are better than one general-purpose interface.”
3. **D**ependency inversion principle - one should “Depend upon Abstractions. Do not depend upon concretions.

The principle states:

*A. High-level modules should not depend on low-level modules. Both should depend on*[*abstractions*](http://en.wikipedia.org/wiki/Abstraction_(computer_science))*.*

*B. Abstractions should not depend on details. Details should depend on abstractions.*

This is facilitated by the separation of high-level components and low-level components into separate packages/libraries, where [interfaces](http://en.wikipedia.org/wiki/Interface_(computer_science)) defining the behavior/services required by the high-level component are owned by, and exist within the high-level component's package. The implementation of the high-level component's interface by the low level component requires that the low-level component package depend upon the high-level component for compilation, thus inverting the conventional dependency relationship.

Another way to think of it is,

Low level components = server side

High level components = client side

Both server and client sides depend on the same interface (contract). The server side needs to implement the interface required by client side. The interface source files are required for server side compilation as well. This brings in the dependency of server side on client side (which is inverse of the dependency of client on server).

 Various patterns such as [Plugin](http://en.wikipedia.org/wiki/Plug-in_(computing)), [Service Locator](http://en.wikipedia.org/wiki/Service_locator_pattern), or [Dependency Injection](http://en.wikipedia.org/wiki/Dependency_Injection) are then employed to facilitate the run-time provisioning of the chosen low-level component implementation to the high-level component.

## DRY

Don’t repeat yourself – no duplicate code.

# CAP Theorem

Proposed by Eric Brewer. States that it is impossible for a [distributed computer system](http://en.wikipedia.org/wiki/Distributed_computing) to simultaneously provide all three of the following guarantees:

1. **Consistency** – all nodes see the same data at the same time
2. **Availability** – a guarantee that every request receives a response about whether it was successful or failed
3. **Partition tolerance** – the system continues to operate despite arbitrary message loss when network is partitioned.

CA = If network is partitioned, the nodes continue to be available by themselves and data continues to be consistent on the same node even when there is no partition tolerance.

AP = Nodes may still be available even when we ensure partition tolerance by disallowing further writes but data will not remain consistent between the nodes anymore as nodes cannot exchange message due to network partitioning.

PC = Nodes may be consistent but in order to ensure partition tolerance they may not be available to write.

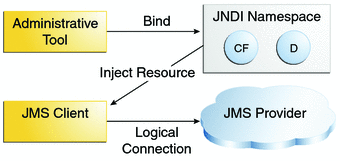
<http://www.julianbrowne.com/article/viewer/brewers-cap-theorem>

<http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed>

# Enterprise Integration Patterns

# JMS 1.1

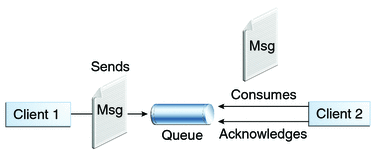
 Administrative tools allow you to bind destinations and connection factories into a JNDI namespace. A JMS client can then use resource injection to access the administered objects in the namespace and then establish a logical connection to the same objects through the JMS provider.



## Point to point messaging

* Each message has only one consumer.
* A sender and a receiver of a message have no timing dependencies. The receiver can fetch the message whether or not it was running when the client sent the message.
* The receiver acknowledges the successful processing of a message.

Use PTP messaging when every message you send must be processed successfully by one consumer.



## Publish/subscribe messaging

* Each message can have multiple consumers.
* Publishers and subscribers have a timing dependency. A client that subscribes to a topic can consume only messages published after the client has created a subscription, and the subscriber must continue to be active in order for it to consume messages.



The JMS API relaxes this timing dependency to some extent by allowing subscribers to create **durable subscriptions**, which receive messages sent while the subscribers are not active.

Use pub/sub messaging when each message can be processed by any number of consumers (or none).

Messages could be consumed – synchronously (blocking w/ timeout) or asynchronously (provider calls onMessage() of listener).

The basic building blocks of a JMS application are:

* Administered objects: connection factories and destinations
* Connections
* Sessions
* Message producers
* Message consumers
* Messages

An EJB Client – message producer.

|  |
| --- |
| import helloworld.beans.SingletonHelloWorldBeanRemote; import helloworld.vo.GreetingRequest; import javax.jms.Connection; import javax.jms.ConnectionFactory; import javax.jms.Destination; import javax.jms.JMSException; import javax.jms.Message; import javax.jms.MessageProducer; import javax.jms.Session; import javax.jms.TextMessage; import javax.naming.Context; import javax.naming.InitialContext; import javax.naming.NamingException;  /\*\*  \* The message producer.  \* @author rwatsh  \*/ public class MessageProducerApp {      private static Message createJMSMessageForjmsHelloWorldQueue(Session session, Object messageData) throws JMSException {         // TODO create and populate message to send         TextMessage tm = session.createTextMessage();         tm.setText(messageData.toString());         return tm;     }      private static void sendJMSMessageToHelloWorldQueue(Object messageData) throws NamingException, JMSException {         Context c = new InitialContext();         ConnectionFactory cf = (ConnectionFactory) c.lookup("jms/HelloWorldQueueFactory");         Connection conn = null;         Session s = null;         try {             conn = cf.createConnection();             s = conn.createSession(false, s.AUTO\_ACKNOWLEDGE);             Destination destination = (Destination) c.lookup("jms/HelloWorldQueue");             MessageProducer mp = s.createProducer(destination);             mp.send(createJMSMessageForjmsHelloWorldQueue(s, messageData));         } finally {             if (s != null) {                 try {                     s.close();                 } catch (JMSException e) {                     System.err.println("Cannot close session - " + e);                 }             }             if (conn != null) {                 conn.close();             }         }     }      private static SingletonHelloWorldBeanRemote lookupSingletonHelloWorldBeanRemote() {         try {             Context c = new InitialContext();             return (SingletonHelloWorldBeanRemote) c.lookup("java:global/EJB31/EJB31-ejb/SingletonHelloWorldBean");         } catch (NamingException ne) {             ne.printStackTrace();             throw new RuntimeException(ne);         }     }      public static void main(String[] args) {         SingletonHelloWorldBeanRemote hwr = lookupSingletonHelloWorldBeanRemote();         GreetingRequest result = hwr.sayHello();         String resultStr = result.toString();         try {             sendJMSMessageToHelloWorldQueue(resultStr);             System.out.println("MessageProducerApp.main: sent message - " + resultStr);         } catch (Exception ex) {             System.err.println("MessageProducerApp.main: failed to send message - " + ex);         }      } } |

MDB message receiver:

|  |
| --- |
| import helloworld.vo.GreetingRequest; import java.util.logging.Level; import java.util.logging.Logger; import javax.annotation.PostConstruct; import javax.annotation.PreDestroy; import javax.ejb.ActivationConfigProperty; import javax.ejb.MessageDriven; import javax.jms.JMSException; import javax.jms.Message; import javax.jms.MessageListener; import javax.jms.ObjectMessage; import javax.jms.TextMessage;  /\*\*  \* MDB that consumes the message.  \* @author rwatsh  \*/ @MessageDriven(mappedName = "jms/HelloWorldQueue", activationConfig = {     @ActivationConfigProperty(propertyName = "acknowledgeMode", propertyValue = "Auto-acknowledge"),     @ActivationConfigProperty(propertyName = "destinationType", propertyValue = "javax.jms.Queue") }) public class HelloWorldMDB implements MessageListener {      public HelloWorldMDB() {     }      @PostConstruct     private void postConstruct() {         System.out.println("HelloWorldMDB: @PostConstruct");     }      @Override     public void onMessage(Message message) {         if (message instanceof TextMessage) {             try {                 TextMessage textMessage = (TextMessage) message;                 String messageValue = textMessage.getText();                 System.out.println("HelloWorldMDB.onMessage: received text message - " + messageValue);             } catch (JMSException ex) {                 Logger.getLogger(HelloWorldMDB.class.getName()).log(Level.SEVERE, null, ex);             }          } else if (message instanceof ObjectMessage) {             try {                 ObjectMessage objMessage = (ObjectMessage) message;                 Object contents = objMessage.getObject();                 if (contents instanceof GreetingRequest) {                     String messageValue = contents.toString();                     System.out.println("HelloWorldMDB.onMessage: received object message - " + messageValue);                 }             } catch (JMSException ex) {             }          } else { //do nothing         }     }      @PreDestroy     private void destroy() {         System.out.println("HelloWorldMDB: @PreDestroy");     } } |

# AMQP

# Data Distribution Service

The **Data Distribution Service** for Real-Time Systems (**DDS**) is an [Object Management Group](http://en.wikipedia.org/wiki/Object_Management_Group) (OMG) [machine to machine](http://en.wikipedia.org/wiki/Machine_to_machine)[middleware](http://en.wikipedia.org/wiki/Middleware) "m2m" standard that aims to enable [scalable](http://en.wikipedia.org/wiki/Scalability), [real-time](http://en.wikipedia.org/wiki/Real-time_computing), [dependable](http://en.wikipedia.org/wiki/Safety_critical), [high performance](http://en.wikipedia.org/wiki/Many-task_computing) and [interoperable](http://en.wikipedia.org/wiki/Interoperable) [data exchanges](http://en.wikipedia.org/wiki/Data_exchange) between publishers and subscribers.

The DDS specification describes two levels of interfaces:

* A lower DCPS (Data-centric publish-subscribe) level that is targeted towards the efficient delivery of the proper information to the proper recipients.
* An optional higher DLRL (data local reconstruction layer) level, which allows for a simple integration of DDS into the [application layer](http://en.wikipedia.org/wiki/Application_layer).

DDS is networking [middleware](http://en.wikipedia.org/wiki/Middleware) that simplifies complex [network programming](http://en.wikipedia.org/wiki/Computer_network_programming). It implements a [publish/subscribe](http://en.wikipedia.org/wiki/Publish/subscribe) model for sending and receiving data, events, and commands among the [nodes](http://en.wikipedia.org/wiki/Node_(networking)). Nodes that are producing information (publishers) create "topics" (e.g., temperature, location, pressure) and publish "samples." DDS takes care of delivering the sample to all subscribers that declare an interest in that topic.

DDS handles all the transfer chores: message addressing, [data marshalling and demarshalling](http://en.wikipedia.org/wiki/Serialization) (so subscribers can be on different platforms than the publisher), delivery, flow control, retries, etc. Any node can be a publisher, subscriber, or both simultaneously.

The DDS publish-subscribe model virtually eliminates complex network programming for distributed applications.

DDS supports mechanisms that go beyond the basic publish-subscribe model. The key benefit is that applications that use DDS for their communications are entirely decoupled. Very little design time has to be spent on how to handle their mutual interactions. In particular, the applications never need information about the other participating applications, including their existence or locations. DDS automatically handles all aspects of message delivery, without requiring any intervention from the user applications, including:

* determining who should receive the messages
* where recipients are located
* what happens if messages cannot be delivered

This is made possible by the fact that DDS allows the user to specify [Quality of Service](http://en.wikipedia.org/wiki/Quality_of_Service) (QoS) parameters as a way to configure automatic-discovery mechanisms and specify the behavior used when sending and receiving messages. The mechanisms are configured up-front and require no further effort on the user's part. By exchanging messages in a completely anonymous manner, DDS greatly simplifies distributed application design and encourages modular, well-structured programs.

DDS also automatically handles **hot-swapping redundant publishers** if the primary fails. Subscribers always get the sample with the highest priority whose data is still valid (that is, whose publisher-specified validity period has not expired). It automatically switches back to the primary when it recovers, too.

Salient points about DDS:

* 1. DDS is in charge of transferring information: Subscribers and publishers employing DDS can use different platforms or operating systems and still communicate with each other. Exchanges can take place through tens of thousands of devices at the same time, each one of which can be publishers, subscribers, or both simultaneously.
  2. Systems can independently communicate with each other: A publisher can still publish information even if there is no subscriber seeking the information. A subscriber can receive information from other publishers if the original publisher it was getting information from fails.
  3. **Location agnostic:** Publishers don’t need to know the location or address of subscribers. DDS knows about the subscribers and what to do with the message if the subscriber is unavailable.
  4. Real-time: With very low overhead and efficient processing, messages are sent with minimal latencies (generally measured in the microseconds). It has a flexible architecture that is also scalable: it can adapt to processing both large and small amounts of data.
  5. **No single point of failure** – no message broker service or directory service. A publisher can still publish information even if there is no subscriber seeking the information, or if a subscriber becomes “lost” for any reason. A subscriber can search for other publishers if the publisher it is getting information from fails or is lost.
  6. **DDS filters data for unique users** – say only authenticated subscribers can get the data only pertinent to them.
  7. **Reliable and always available** - Data is cached by the publisher until all subscribers have received the information, so even if the network is unavailable the information is not lost. The publisher and subscriber merely try again.
  8. QoS policies – allow user to configure different communication behavior (reliability, persistent delivery, data presentation, filtering, redundancy and fail-over requirements).

### Features and Benefits

* Interoperable across languages, operating systems, CPU families and network types
* Well-defined component interfaces for rapid integration
* Low latency and high throughput
* Deterministic even under load and at scale
* **Reliable multicast for efficient data distribution**
* **Peer-to-peer architecture** with **no message brokers** and servers
* Non-stop availability with live upgrades and no single point of failure
* **Multiple enterprise integration patterns** for differing developmental approaches
* Fully compliant with the [OMG DDS standard](http://www.rti.com/products/dds/omg-dds-standard.html), including API and RTPS protocol

# REST

|  |  |
| --- | --- |
| **Annotation** | **Description** |
| @PATH(your\_path) | Sets the path to base URL + /your\_path. The base URL is based on your application name, the servlet and the URL pattern from the *web.xml*configuration file. |
| @POST, @GET, @PUT, @DELETE | Indicates that the following method will answer to an HTTP POST/GET/PUT/DELETE request, respectively. |
| @Produces(MediaType.TEXT\_PLAIN[, more-types]) | @Produces defines which MIME type is delivered by a method annotated with @GET. |
| @Consumes(type[, more-types]) | @Consumes defines which MIME type is consumed by this method. |
| @PathParam | Used to inject values from the URL into a method parameter. This way you inject, for example, the ID of a resource into the method to get the correct object. |

|  |
| --- |
| **import** javax.ws.rs.GET;  **import** javax.ws.rs.Path;  **import** javax.ws.rs.Produces;  **import** javax.ws.rs.core.MediaType;  //Plain old Java Object it does not extend as class or implements  //an interface  //The class registers its methods for the HTTP GET request using the @GET annotation.  //Using the @Produces annotation, it defines that it can deliver several MIME types,  //text, XML and HTML.  //The browser requests per default the HTML MIME type.  //Sets the path to base URL + /hello  @Path("/hello")  **public** **class** Hello {  // This method is called if TEXT\_PLAIN is request  @GET  @Produces(MediaType.*TEXT\_PLAIN*)  **public** String sayPlainTextHello() {  **return** "Hello Jersey";  }  // This method is called if XML is request  @GET  @Produces(MediaType.*TEXT\_XML*)  **public** String sayXMLHello() {  **return** "<?xml version=\"1.0\"?>" + "<hello> Hello Jersey" + "</hello>";  }  // This method is called if HTML is request  @GET  @Produces(MediaType.*TEXT\_HTML*)  **public** String sayHtmlHello() {  **return** "<html> " + "<title>" + "Hello Jersey" + "</title>"  + "<body><h1>" + "Hello Jersey" + "</body></h1>" + "</html> ";  }  } |

|  |
| --- |
| **import** **static** org.junit.Assert.*assertEquals*;  **import** java.net.URI;  **import** javax.ws.rs.core.MediaType;  **import** javax.ws.rs.core.UriBuilder;  **import** org.junit.BeforeClass;  **import** org.junit.Test;  **import** com.sun.jersey.api.client.Client;  **import** com.sun.jersey.api.client.ClientResponse;  **import** com.sun.jersey.api.client.WebResource;  **import** com.sun.jersey.api.client.config.ClientConfig;  **import** com.sun.jersey.api.client.config.DefaultClientConfig;  **public** **class** HelloTest {  **private** **static** WebResource *service*;  @BeforeClass  **public** **static** **void** setUpBeforeClass() **throws** Exception {  ClientConfig config = **new** DefaultClientConfig();  Client client = Client.*create*(config);  *service* = client.resource(*getBaseURI*());  }  @Test  **public** **void** testSayPlainTextHello() {  String msg = *service*.path("rest").path("hello")  .accept(MediaType.*TEXT\_PLAIN*).get(ClientResponse.**class**)  .toString();  System.*out*.println(msg);  *assertEquals*(**true**, msg.contains("200"));  }  **private** **static** URI getBaseURI() {  **return** UriBuilder.*fromUri*("http://localhost:8080/rest").build();  }  } |

# Maven

Maven supports project scaffolding, i.e. template generation, for different project types, so called*archtype*. The goal of this scaffolding is to allow a fast start into the Maven world and supports a "standardized" folder structure of software projects.

You can create a project by executing the generate *goal* on the archetype *plugin* : $mvn archetype:generate . This starts the generate process in the interactive mode and asks you for several settings. **Artifact properties**

| **Name** | **Description** |
| --- | --- |
| groupId | defines a unique base name of the organization or group that created the project. This is normally a domain name. For the generation the groupId also defines the package of the main class. |
| artifactId | defines the unique name of the artifact. In our case it's the same like the groupId, but it could also be a simple name like "SuperCalculator". The generation uses this value as name of the root folder for out project. |
| packaging | defines the packaging method. This could be e.g. jar, war or ear. This setting has influence on the whole life cycle. |
| version | This defines the version of the artifacts generated from our project. |
| url | defines the project site. |

The configuration of a Maven project is done via a *Project Object Model*, which is represented by the*pom.xml* file.

By default, this is the only configuration file required for the build process. Every build follows a specified *life cycle*. Maven comes with a default *life cycle* that includes the most common build *phases*like compiling, testing and packaging. All *phases* are made up of *plugin goals*. *Plugin goals* are tasks which are more specific than *phases*. So a *phase* can be defined to run more than one *plugin goal*.

The result of a build is called *artifact*. An artifact, for example, can be a executable or an archive of documents.

With this single line, the maven lifecycle will be started. So the configuration from the *pom.xml* will be loaded and the dependencies will be resolved. After this, Maven tries to load the lastest version of the depended artifacts from the central repository into a local repository, which is placed as*.m2/repository* in the users home directory. As part of the lifecycle, the build tool compiles the sources and tests, runs the tests and packs the compiled files in, e.g., JAR archives. As last step the resulting artifact will be saved in the local repository, so it can be used by other projects.

Maven creates the build result in the *target* folder. If you run the mvn install command, Maven compiles the source code, builds the result, e.g., the *JAR* file.

# default to run Maven

# compiles, build and install the build result

maven install

By default, Maven checks online if the dependencies have been changed. If you want to use your local repository, you can use the *-o* to tell Maven to work offline.

# work offline , i.e. use the local maven repository

maven -o clean install

C:\Java\maven >mvn archetype:generate -DgroupId=mavenexample -DartifactId=mavenexample -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false

CD mavenexample

C:\Java\maven\mavenexample >mvn archetype:generate -DgroupId=mavenexample -DartifactId=mavenexample -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false

C:\Java\maven\mavenexample>java -cp target/mavenexample-1.0-SNAPSHOT.jar mavenexample.App

To run specific phases:

$ mvn package

$ mvn test

$ mvn compile

$ mvn clean

# OSGi

[OSGi technology](http://www.osgi.org/About/Technology) is the dynamic module system for [Java](http://www.oracle.com/technetwork/java/index.html).  Java provides the portability that is required to support products on many different platforms. The OSGi technology provides the standardized primitives that allow applications to be constructed from small, reusable and collaborative components. These components can be composed into an application and deployed.

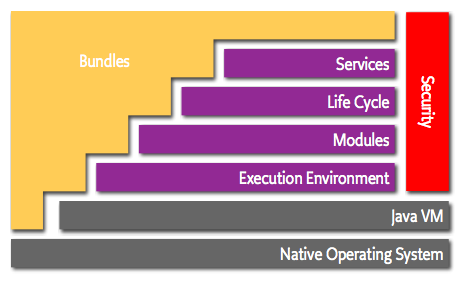
The OSGi Service Platform provides the functions to change the composition dynamically on the device of a variety of networks, without requiring restarts. To minimize the coupling, as well as make these couplings managed, the OSGi technology provides a service-oriented architecture that enables these components to dynamically discover each other for collaboration. The OSGi Alliance has developed many standard component interfaces for common functions like HTTP servers, configuration, logging, security, user administration, XML and many more. Plug-compatible implementations of these components can be obtained from different vendors with different optimizations and costs. However, service interfaces can also be developed on a proprietary basis.

Apache Karaf is a small [OSGi-based](http://www.osgi.org/Main/HomePage) runtime which provides a lightweight container onto which various components and applications can be deployed.

Features:

* 1. Hot deployment: Karaf supports hot deployment of OSGi bundles by monitoring jar files inside the $home/deploy directory.
  2. Dynamic configuration: Services are usually configured through the ConfigurationAdmin OSGi service. Such configuration can be defined in Karaf using property files inside the $home/etc directory. These configurations are monitored and changes on the properties files will be propagated to the services.
  3. Logging System: using a centralized logging back end supported by Log4J
  4. Provisioning: Provisioning of libraries or applications can be done through a number of different ways, by which they will be downloaded locally, installed and started.
  5. Native OS integration: Karaf can be integrated into your own Operating System as a service
  6. Extensible Shell console: Karaf features a nice text console where you can manage the services, install new applications or libraries and manage their state. This shell is easily extensible by deploying new commands dynamically along with new features or applications.
  7. Remote access: use any SSH client to connect to Karaf and issue commands in the console
  8. Security framework based on JAAS
  9. Managing instances: Karaf provides simple commands for managing multiple instances. You can easily create, delete, start and stop instances of Karaf through the console.

## Architecture

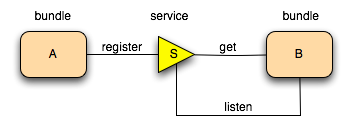


* Bundles - Bundles are the OSGi components made by the developers.
* Services - The services layer connects bundles in a dynamic way by offering a publish-find-bind model for plain old Java objects.
* Life-Cycle - The API to install, start, stop, update, and uninstall bundles.
* Modules - The layer that defines how a bundle can import and export code.
* Security - The layer that handles the security aspects.
* Execution Environment - Defines what methods and classes are available in a specific platform.

<http://www.osgi.org/Technology/WhatIsOSGi>

In Java terms, a bundle is a plain old JAR file. However, where in standard Java everything in a JAR is completely visible to all other JARs, OSGi hides everything in that JAR unless explicitly exported. A bundle that wants to use another JAR must explicitly import the parts it needs. By default, there is no sharing.

Though the code hiding and explicit sharing provides many benefits (for example, allowing multiple versions of the same library being used in a single VM), the code sharing was only there to support OSGi services model. The services model is about bundles that collaborate.



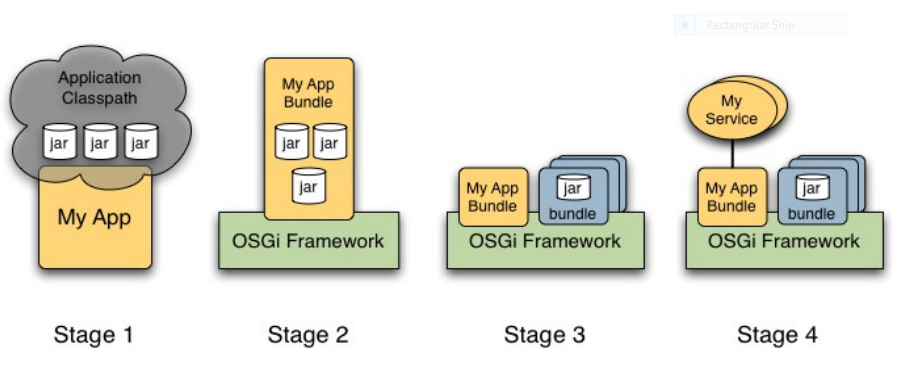
Bundles are deployed on an OSGi framework, the bundle runtime environment. This is not a container like Java Application Servers. It is a collaborative environment. Bundles run in the same VM and can actually share code. The framework uses the explicit imports and exports to wire up the bundles so they do not have to concern themselves with class loading. Another contrast with the application servers is that the management of the framework is standardized. A simple API allows bundles to install, start, stop, and update other bundles, as well as enumerating the bundles and their service usage. This API has been used by manymanagement agents to control OSGi frameworks.

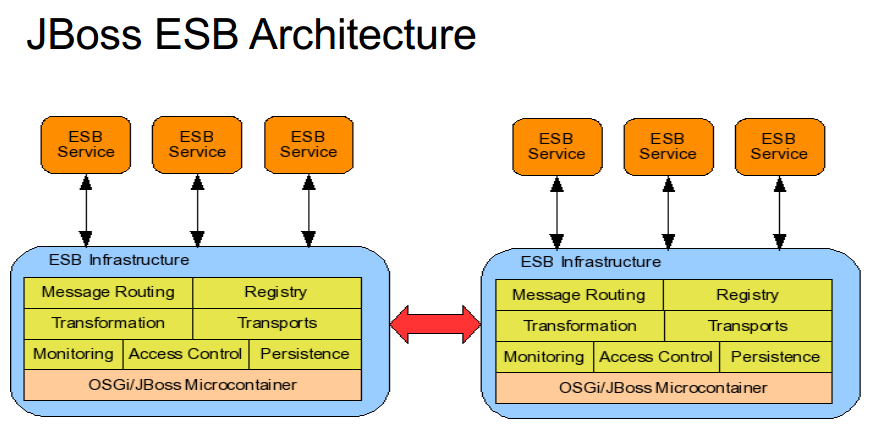
## Normalized Message Router

The Normalized Message Router (NMR) is a general-purpose message bus used for communication between bundles in the OSGi container.  
It's modeled after the Normalized Message Router (NMR) defined in the Java Business Integration (JBI) specification.

## Role of OSGi in a SOA Runtime

1. Pluggability – extend runtime with additional functionality – adding services, containers etc.
2. Isolation – control packages exposed and consumed
3. Dynamism – bundles have lifecycle independent of VM
4. Dependency Management – dependency between services





# Scale out Architectures

## Map Reduce

MapReduce is a parallel and distributed solution approach developed by Google for processing large datasets. Described in this paper - <http://research.google.com/archive/mapreduce.html>

Map transforms a set of data into key value pairs and Reduce aggregates this data into a scalar. A reducer receives all the data for an individual "key" from all the mappers.

The approach assumes that there are no dependencies between the input data. This make it easy to parallelize the problem. The number of parallel reduce task is limited by the number of distinct "key" values which are emitted by the map function.

MapReduce incorporates usually also a framework which supports MapReduce operations. A master controls the whole MapReduce process. The MapReduce framework is responsible for load balancing, re-issuing task if a worker as failed or is to slow, etc. The master divides the input data into separate units, send individual chunks of data to the mapper machines and collects the information once a mapper is finished. If the mapper are finished then the reducer machines will be assigned work. All key/value pairs with the same key will be send to the same reducer.

The classical example for using MapReduce is logfile analysis.

Big logfiles are split and a mapper search for different webpages which are accessed. Every time a webpage is found in the log a key / value pair is emitted to the reducer where the key is the webpage and the value is "1". The reducers aggregate the number of for certain webpages. As a end result you have the total number of hits for each webpage.

## Big Table