

# DESIGN PATTERNS

Version 0.1

**GRASSFIELD.org**  
MARCH 2008

**VERSION HISTORY**

<b>DATE</b>	<b>VERSION</b>	<b>DESCRIPTION</b>	<b>AUTHOR</b>
03/29/08	0.1	Initial Draft	Pandian R

## 1 About this document

This document can be used as a nutshell guide to a few design patterns. This may not be suitable for detailed view or detailed study.

The audiences are intended to be Senior developers, or application designers. Knowledge of OOPS is mandatory for better understanding.

I may be poor in documenting. Please keep me informed about the grammatical or typo errors.

## 2 About Design patterns

We face many problems in software designing. We spend considerable amount of effort to rectify them. But the solutions proposed for a particular problem need to be documented for future usage; so that it can be applied where ever applicable. So, this kind of good practices are known as design patterns. They are not solutions, but they are like templates to solve the problems.

## Table of Contents

1	About this document.....	3
2	About Design patterns.....	3
1	Types of Design pattern.....	5
1.1	Creational Pattern.....	5
1.1.1	Factory.....	5
1.1.2	Abstract Factory.....	5
1.1.3	Lazy initialization.....	5
1.1.4	Object pool.....	5
1.1.5	Singleton.....	5
1.2	structural patterns.....	5
1.2.1	Adapter.....	5
1.2.2	Bridge.....	5
1.2.3	Composite.....	5
1.2.4	Decorator.....	6
1.2.5	Façade.....	6
1.2.6	Proxy.....	6
1.3	Behavioral patterns.....	6
1.3.1	Command.....	6
1.3.2	Iterator.....	6
1.3.3	Observer.....	6
1.3.4	State.....	6

# 1 Types of Design pattern

We can classify the patterns into three categories:

- *Creational patterns* : This deal with object creation mechanism. Many times, default method of creating the objects will lead to design problems. Creational patterns will come in to picture to solve those problems, by controlling the object creation.
- *Structural patterns*: These patterns will ease the design by identifying a simple way to realize relationship between entities.
- *Behavioral patterns*: These patterns deal with communication between entities, and how the relation between the entities help to pass the communication.

## 1.1 Creational Pattern

### 1.1.1 Factory

Define an interface, let the subclasses implement it. Have a mechanism to decide which subclass to be invoked at runtime.

### 1.1.2 Abstract Factory

Simply, factory of factories. Provide an interface to create group of related classes without specifying the corresponding concrete class

### 1.1.3 Lazy initialization

It is delaying a task purposefully; it may be a costlier process. Or It may not be a reusable process. So we will invoke this on call.

### 1.1.4 Object pool

Avoid resource being wasted in acquisition. Reuse the existing instances that are no longer in use.

### 1.1.5 Singleton

Ensure the class has only one instance at any point of time; and provide global access to it

## 1.2 structural patterns

### 1.2.1 Adapter

Simply, convert one type of interface into another type of interface client expects. It helps classes to work together that couldn't otherwise because of incompatible interfaces.

### 1.2.2 Bridge

Extract the abstract/interface from its implementation, so that both can vary independently.

### 1.2.3 Composite

Composite patterns creates a tree like structure of part-whole hierarchies. Composite helps the client to treat

the individual object and group of objects as same.

#### 1.2.4 Decorator

Decorator attach additional capabilities to an object dynamically. It provides an alternative way to subclassing.

#### 1.2.5 Façade

Façade provides access to unified interface to a set of interfaces in a system. It makes the access easier to implement and provide better understanding.

#### 1.2.6 Proxy

Control the access of one object by another object. The access is subject to realization with respect to proxy object.

### 1.3 Behavioral patterns

#### 1.3.1 Command

This is a data driven pattern. The request is considered as a command string, based on which, the corresponding modules will be invoked at runtime.

#### 1.3.2 Iterator

It helps to iterate through the collection of the implementation classes without going in detail

#### 1.3.3 Observer

Representing one to many relationship between objects. So, when the state of one object changes, it will change the other objects too.

#### 1.3.4 State

Allow an object to alter its behavior when when its internal state changes

## 2 References

[http://www.allapplabs.com/java\\_design\\_patterns/command\\_pattern.htm](http://www.allapplabs.com/java_design_patterns/command_pattern.htm)

[http://en.wikipedia.org/wiki/Design\\_pattern\\_\(computer\\_science\)](http://en.wikipedia.org/wiki/Design_pattern_(computer_science))