ABOUT ROHIT JOSHI

Rohit Joshi is a Senior Software Engineer from India. He is a Sun Certified Java Programmer and has worked on projects related to different domains. His expertise is in Core Java and J2EE technologies, but he also has good experience with front-end technologies like Javascript, JQuery, HTML5, and JQWidgets.

Introduction to Design Patterns

Posted by: Rohit Joshi in Core Java September 30th, 2015

This article is part of our Academy Course titled Java Design Patterns.

In this course you will delve into a vast number of Design Patterns and see how those are implemented and utilized in Java. You will understand the reasons why patterns are so important and learn when and how to apply each one of them. Check it out here!

Want to be a Java Master ?

Subscribe to our newsletter and download Java Design Patterns right now!

In order to help you master the Java programming language, we have compiled a kick-ass guide with all the must-know Design Patterns for Java! Besides studying them online you may download the eBook in PDF format!

Email address:

Your email address

Sign up

Table Of Contents

1. Introduction

In the late 70's, an architect named Christopher Alexander started the concept of patterns. Alexander's work focused on finding patterns of solutions to particular sets of forces within particular contexts.

Christopher Alexander was a civil engineer and an architect, his patterns were related to architects of buildings, but the work done by him inspired an interest in the object-oriented (OO) community, and a number of innovators started developing patterns for software design. Kent Beck and Ward Cunningham were among the few who presented the set of design patterns for Smalltalk in an OOPSLA conference. James Coplien was another who actively promoted the tenets of patterns.

NEWSLETTER

172,305 insiders are already er
weekly updates and complimentary
whitepapers!
Join them now to gain ex
access to the latest news in the J
as well as insights about Android, Sc
Groovy and other related technologi

Email address:

Your email address

Sign up

RECENT JOBS

No job listings found.

JOIN US

With **1,240,6**
unique visitors
500 authors
placed among
related sites a
Constantly bei
lookout for pa
encourage you
So If you have
unique and interesting content then yo
check out our **JCG** partners program.
be a **guest writer** for Java Code Geek
your writing skills!

Soon, the patterns community started growing at OOPSLA, as it placed an environment for the members to share their innovations and ideas about the patterns. Another important forum for the evolution of the patterns movement was the Hillside Group, established by Kent Beck and Grady Booch.

This is what design patterns are — the distillation of expertise by an exuberant and robust community. This is crowd sourcing at its best. The patterns community that has grown over the decade-plus since the original GoF work is large and energetic. Grady Booch and Celso Gonzalez have been collecting every pattern they can find in the industry. So far, they have over 2,000 of them.

Later, we will also see how patterns are organized, and categorized into different groups according to their behavior and structure.

In the next several lessons, we will discuss about the different design patterns one by one. We will go into depth and analyze each and every design pattern, and will also see how to implement them in Java.

2. What are Design Patterns

As an Object Oriented developer, we may think that our code contains all the benefits provided by the Object Oriented language. The code we have written is flexible enough that we can make any changes to it with less or any pain. Our code is re-usable so that we can re-use it anywhere without any trouble. We can maintain our code easily and any changes to a part of the code will not affect any other part of the code.

Unfortunately, these advantages do not come by its own. As a developer, it is our responsibility to design the code in such a way which allow our code to be flexible, maintainable, and re-usable.

Designing is an art and it comes with the experience. But there are some set of solutions already written by some of the advanced and experienced developers while facing and solving similar designing problems. These solutions are known as Design Patterns.

The Design Patterns is the experience in designing the object oriented code.

Design Patterns are general reusable solution to commonly occurring problems. These are the best practices, used by the experienced developers. Patterns are not complete code, but it can use as a template which can be applied to a problem. Patterns are re-usable; they can be applied to similar kind of design problem regardless to any domain. In other words, we can think of patterns as a formal document which contains recurring design problems and its solutions. A pattern used in one practical context can be re-usable in other contexts also.

Christopher had said that "Each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice".

In general, a pattern has four essential elements:

1. **Pattern name**, is used to provide a single and meaningful name to the pattern which defines a design problem and a solution for it. Naming a design pattern helps itself to be referred to others easily. It also becomes easy to provide documentation for and the right vocabulary word makes it easier to think about the design.
2. **The problem describes when to apply the pattern.** It explains the problem and its context. It might describe specific design problems such as how to represent algorithms as objects. It might describe a class or object structures that are symptomatic of an inflexible design. Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern.
3. The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution is not the complete code, but it works as a template which can be fulfilled with code. Instead, the pattern provides an abstract description of a design problem and how a general arrangement of elements (classes and objects in our case) solves it.
4. The **results and consequences** of applying the pattern. The consequences for software often concern space and time trade-offs. They may address language and implementation issues as well. Since reuse is often a factor in object-oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility, or portability. Listing these consequences explicitly helps you understand and evaluate them.

3. Why use them

Flexibility: Using design patterns your code becomes flexible. It helps to provide the correct level of abstraction due to which objects become loosely coupled to each other which makes your code easy to change.

Reusability: Loosely coupled and cohesive objects and classes can make your code more reusable. This kind of code becomes easy to be tested as compared to the highly coupled code.

Shared Vocabulary: Shared vocabulary makes it easy to share your code and thought with other team members. It creates more understanding between the team members related to the code.

Capture best practices: Design patterns capture solutions which have been successfully applied to problems. By learning these patterns and the related problem, an inexperienced developer learns a lot about software design.

Design patterns make it easier to reuse successful designs and architectures.

Expressing proven techniques as design patterns makes them more accessible to developers of new systems. Design patterns help you choose design alternatives that make a system reusable and avoid alternatives that compromise reusability. Design patterns can even improve the documentation and maintenance of existing systems by furnishing an explicit specification of class and object interactions and their underlying intent. Put simply, design patterns help a designer get a design "right" faster.

4. How to select and use one

There are a number of design patterns to choose from; to choose one, you must have good knowledge of each one of them. There are many design patterns which look very similar to one another. They solve almost a similar type of design problem and also have similar implementation. One must have a very deep understanding of them in order to implement the correct design pattern for the specific design problem.

First, you need to identify the kind of design problem you are facing. A design problem can be categorized into creational, structural, or behavioral. Based to this category you can filter the patterns and selects the appropriate one. For example:

instance for the entire application. It also helps to decrease the memory size.

2. **Classes are too much dependent on each other. A Change in one class affects all other dependent classes:** you can use Bridge, Mediator, or Command to solve this design problem.
3. **There are two different incompatible interfaces in two different parts of the code, and your need is to convert one interface into another which is used by the client code to make the entire code work:** the Adapter pattern fits into this problem.

A design pattern can be used to solve more than one design problem, and one design problem can be solved by more than one design patterns. There could be plenty of design problems and solutions for them, but, to choose the pattern which fits exactly is depends on your knowledge and understanding about the design patterns. It also depends on the code you already have in place.

5. Categorization of patterns

Design patterns can be categorized in the following categories:

1. Creational patterns
2. Structural patterns
3. Behavior patterns

5.1. Creational patterns

Creational design patterns are used to design the instantiation process of objects. The creational pattern uses the inheritance to vary the object creation.

There are two recurring themes in these patterns. First, they all encapsulate knowledge about which concrete classes the system uses. Second, they hide how instances of these classes are created and put together. All the system at large knows about the objects is their interfaces as defined by abstract classes. Consequently, the creational patterns give you a lot of flexibility in what gets created, who creates it, how it gets created, and when.

There can be some cases when two or more patterns looks fit as a solution to a problem. At other times, the two patterns complement each other for example; Builder can be used with other pattern to implements which components to get built.

5.2. Structural patterns

Structural patterns are concerned with how classes and objects are composed to form larger structures. Structural class patterns use inheritance to compose interfaces or implementations. As a simple example, consider how multiple inheritance mixes two or more classes into one. The result is a class that combines the properties of its parent classes. This pattern is particularly useful for making independently developed class libraries work together.

Rather than composing interfaces or implementations, structural object patterns describe ways to compose objects to realize new functionality. The added flexibility of object composition comes from the ability to change the composition at run-time, which is impossible with static class composition.

5.3. Behavior patterns

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them. These patterns characterize complex control flow that's difficult to follow at run-time. They shift your focus away from flow of control to let you concentrate just on the way objects are interconnected.

Behavioral object patterns use object composition rather than inheritance. Some describe how a group of peer objects cooperate to perform a task that no single object can carry out by itself. An important issue here is how peer objects know about each other. Peers could maintain explicit references to each other, but that would increase their coupling. In the extreme, every object would know about every other. The Mediator pattern avoids this by introducing a mediator object between peers. The mediator provides the indirection needed for loose coupling.

The below tables shows the list of patterns under their respective categories:

Creational Patterns	Structural Patterns	Behavioral Patterns
Abstract Factory	Adapter	Chain of Responsibility
Builder	Bridge	Command
Factory Method	Composite	Interpreter
Prototype	Decorator	Iterator
Singleton	Façade	Mediator