Revision history

|  |  |  |
| --- | --- | --- |
| Revision # | Date of Revision | Changes Made |
| 1.0 | NA | First Release |
|  |  |  |

Reference documents

|  |  |  |
| --- | --- | --- |
| S.No | Document Title | Date |
| 1 |  | 05 Jun 2017 |

Glossary

| **Terms** | **Description** |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Table of contents

[1 Installation Guide 6](#_Toc487030814)

[1.1 Download 6](#_Toc487030815)

[1.2 Install 6](#_Toc487030816)

[1.3 Start MongoDB 6](#_Toc487030817)

[1.4 Connect mongoDB from Client 7](#_Toc487030818)

THIS PAGE IS INTENTIONALLY LEFT BLANK

# JAVA Enterprise Design Patterns

## Gang Of Four (GOF)

The GoF Design Patterns are broken into three categories: Creational Patterns for the creation of objects, Structural Patterns provide for a relationship between objects and finally, Behavioral Patterns help define how objects interact.

### Creational Design Patterns

#### Factory Method - Creates objects without specifying the exact class to create

#### Abstract Factory - Allows for the creation of objects without specifying their concrete type

#### Builder - Used to create complex objects

The builder pattern allows you to enforce a step-by-step process to construct a complex object as a finished product. In this pattern, the step-by-step construction process remains same but the finished products can have different representations.

Both the abstract factory and builder patterns are similar, as both can be used to abstract object creation. But there are distinct differences between the two. While abstract factory emphasizes on creating a family of related objects in one go, builder is about creating an object through a step-by-step construction process and returning the object as the final step. In short abstract factory is concerned with **what**is made, while the builder with **how**it is made

#### Prototype - Creates a new object from an existing object

Using the prototype pattern, you do not create a new object for each client requesting the object. Instead, you start by creating a single object, called a prototype and make copies of it for each client requesting the object. In Java, this is achieved through object cloning, a way to make a copy of an object with the same state as the original object.

The prototype pattern is a simple pattern but a common drawback to using it is with the complexities involved in making object copies. When an object refers other objects, some of which in turn refer other objects. While this is true to some extent, there are circumstances where the benefits that prototype provide far outweigh the inherent complexities of object copying. Especially in enterprise application development, you will encounter situations where copying an object can be significantly efficient than creating a new object. In such situations, use the prototype pattern

#### Singleton - Ensures only one instance of an object is created

The singleton pattern implies a class which can **only be instantiated once,**and a global point of access to that instance is provided.

Ref: <https://en.wikipedia.org/wiki/Double-checked_locking>

##### Type-1

Lazy initialization – make private constructor, declare private static class reference and create object in public static method.

public class SingletonClass {

private static SingletonClass instance = null;

private SingletonClass() {

}

public static SingletonClass getInstance() {

if (instance == null) {

instance = new SingletonClass();

}

return instance;

}

}

##### Type-2

Thread safe singleton object creation

public class SingletonClass {

private static SingletonClass instance = null;

private SingletonClass() {

}

public static synchronized SingletonClass getInstance() {

if (instance == null) {

instance = new SingletonClass();

}

return instance;

}

}

##### Type-3

**nitialization-on-demand holder** idiom

**public** **class** **Something** {

**private** Something() {}

**private** **static** **class** **LazyHolder** {

**static** **final** Something INSTANCE = **new** Something();

}

**public** **static** Something getInstance() {

**return** LazyHolder.INSTANCE;

}

}

The implementation of the idiom relies on the initialization phase of execution within the [Java Virtual Machine](https://en.wikipedia.org/wiki/Java_Virtual_Machine) (JVM) as specified by the Java Language Specification (JLS).[[3]](https://en.wikipedia.org/wiki/Initialization-on-demand_holder_idiom#cite_note-3) When the class Something is loaded by the JVM, the class goes through initialization. Since the class does not have any static variables to initialize, the initialization completes trivially. The static class definition LazyHolder within it is not initialized until the JVM determines that LazyHolder must be executed. The static class LazyHolder is only executed when the static method getInstance is invoked on the class Something, and the first time this happens the JVM will load and initialize the LazyHolder class. The initialization of the LazyHolder class results in static variable INSTANCE being initialized by executing the (private) constructor for the outer class Something. Since the class initialization phase is guaranteed by the JLS to be sequential, i.e., non-concurrent, no further synchronization is required in the static getInstance method during loading and initialization. And since the initialization phase writes the static variable INSTANCE in a sequential operation, all subsequent concurrent invocations of the getInstance will return the same correctly initialized INSTANCE without incurring any additional synchronization overhead.

##### Type-4

Double-checked locking idiom – instead of synchronizing complete method, making only object creation thread-safe.

*// Broken multithreaded version*

*// "Double-Checked Locking" idiom*

**class** **Foo** {

**private** Helper helper;

**public** Helper getHelper() {

**if** (helper == **null**) {

**synchronized**(**this**) {

**if** (helper == **null**) {

helper = **new** Helper();

}

}

}

**return** helper;

}

*// other functions and members...*

}

##### Type-5

Using volatile keyword

*// Works with acquire/release semantics for volatile in Java 1.5 and later*

*// Broken under Java 1.4 and earlier semantics for volatile*

**class** **Foo** {

**private** **volatile** Helper helper;

**public** Helper getHelper() {

Helper result = helper;

**if** (result == **null**) {

**synchronized**(**this**) {

result = helper;

**if** (result == **null**) {

helper = result = **new** Helper();

}

}

}

**return** result;

}

*// other functions and members...*

}

### Structural Design Patterns

#### Adaptor Pattern - Allows for two incompatible classes to work together by wrapping an interface around one of the existing classes.

This pattern is typically used when an incompatible module needs to be integrated with an existing module without making any source code modifications.

Some programmers argue that the Adapter pattern is a workaround for a system, which was not well designed keeping into considerations new possibilities. While this is true to some extent, we cannot expect an enterprise application, which will often have a pluggable architecture, to be designed considering all components that might get added in future.  
In enterprise application development, it is likely that you might need to hook in other libraries, APIs, and “off the shelf” components, and if they are not aligned with the existing system, put the adapter pattern to use. After all, being a core GOF pattern, it is a tested and proven solution used over a long period of time.

With the Spring Framework, you will be using adapters built into the framework.

[Spring Integration](https://docs.spring.io/spring-integration/reference/html/overview.html) uses JMS adapters to send and receive JMS messages and JDBC adapters to convert messages to database queries and result sets back to messages.

Spring also uses the adapter design pattern to handle load-time-weaving used in [Aspect-Oriented Programming (AOP)](https://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop.html). An adapter is used to inject AspectJ’s aspects to bytecode during class loading done by the servlet container.

#### Bridge - Decouples an abstraction so two classes can vary independently

#### Composite - Takes a group of objects into a single object

#### Decorator - Allows for an object’s behavior to be extended dynamically at run time

#### Façade - Provides a simple interface to a more complex underlying object

#### Flyweight - Reduces the cost of complex object models

#### Proxy - Provides a placeholder interface to an underlying object to control access, reduce cost, or reduce complexity

### Behavior Design Pattern

#### Chain Of Responsibility - Delegates commands to a chain of processing objects

#### Command - Creates objects which encapsulate actions and parameters

#### [Interpreter](https://springframework.guru/gang-of-four-design-patterns/interpreter-pattern/) - Implements a specialized language.

#### [Iterator](https://springframework.guru/gang-of-four-design-patterns/iterator-pattern/) - Accesses the elements of an object sequentially without exposing its underlying representation.

#### [Mediator](https://springframework.guru/gang-of-four-design-patterns/mediator-pattern/) - Allows loose coupling between classes by being the only class that has detailed knowledge of their methods.

#### [Memento](https://springframework.guru/gang-of-four-design-patterns/memento-pattern/) - Provides the ability to restore an object to its previous state.

#### [Observer](https://springframework.guru/gang-of-four-design-patterns/observer-pattern/) - Is a publish/subscribe pattern which allows a number of observer objects to see an event.

#### [State](https://springframework.guru/gang-of-four-design-patterns/state-pattern/) - Allows an object to alter its behavior when its internal state changes.

#### [Strategy](https://springframework.guru/gang-of-four-design-patterns/strategy-pattern/) - Allows one of a family of algorithms to be selected on-the-fly at run-time.

#### [Template Method](https://springframework.guru/gang-of-four-design-patterns/template-method-pattern/) - Defines the skeleton of an algorithm as an abstract class, allowing its sub-classes to provide concrete behavior.

#### [Vistor](https://springframework.guru/gang-of-four-design-patterns/visitor-pattern/) - Separates an algorithm from an object structure by moving the hierarchy of methods into one object

# SOLID Programming Principles

To achieve algorithm robustness and high cohesion programmers follow SOLID principles.

[Cohesion](https://en.wikipedia.org/wiki/Cohesion_%28computer_science%29) is a way to measure how much the code segments within one module (methods of a class, classes inside a package…) belong together.

Robustness, could be defined as the ability of a computer system or algorithm to handle mistakes and malfunctions (which could be caused by various factors such as programmer’s mistake or incorrectly formatted user input).

## Single Responsibility

A class (or some other module) should only have one responsibility i.e. one reason to change.

As you become more comfortable using Spring [components](https://springframework.guru/creating-spring-beans/) and coding to support Inversion of Control and [Dependency Injection in Spring](https://springframework.guru/dependency-injection-example-using-spring/), you will find your classes will naturally adhere to the single responsibility principle

## Open / Closed

Software entities should be open for extension, but closed for modification.

## Liskov Substitution

Objects in a program should be replaceable with instances of their sub types without altering the correctness of that program.

## Interface Segregation Principle

Many client-specific interfaces are better than one general-purpose interface

## Dependency Inversion Principle