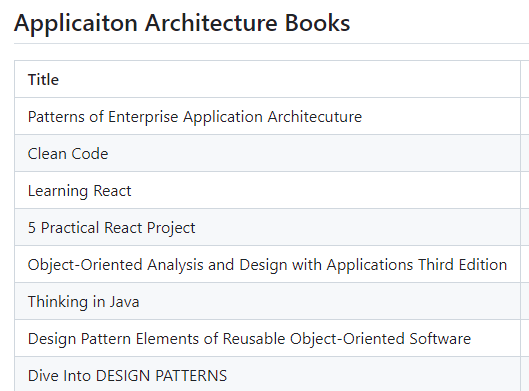
Design Patterns By WIKI



Patterns by type[[edit](https://en.wikipedia.org/w/index.php?title=Design_Patterns&action=edit&section=3)]

**Creational**[[edit](https://en.wikipedia.org/w/index.php?title=Design_Patterns&action=edit&section=4)]

*Main article:*[*Creational pattern*](https://en.wikipedia.org/wiki/Creational_pattern)

[Creational patterns](https://en.wikipedia.org/wiki/Creational_pattern) are ones that create objects, rather than having to instantiate objects directly. This gives the program more flexibility in deciding which objects need to be created for a given case.

* [Abstract factory](https://en.wikipedia.org/wiki/Abstract_factory_pattern) groups object factories that have a common theme.[SuperFactory to create Family Of Objects]
* [Builder](https://en.wikipedia.org/wiki/Builder_pattern) constructs complex objects by separating construction and representation. [Object Construction in MultiSteps ]
* [Factory method](https://en.wikipedia.org/wiki/Factory_method_pattern) creates objects without specifying the exact class to create. [ Return specific instance of class from multiple SubClasses family ]
* [Prototype](https://en.wikipedia.org/wiki/Prototype_pattern) creates objects by cloning an existing object. [Clone the Object]
* [Singleton](https://en.wikipedia.org/wiki/Singleton_pattern) restricts object creation for a class to only one instance.[Only One instance of class]

**Structural**[[edit](https://en.wikipedia.org/w/index.php?title=Design_Patterns&action=edit&section=5)]

[Structural patterns](https://en.wikipedia.org/wiki/Structural_pattern) concern class and object composition. They use inheritance to compose interfaces and define ways to compose objects to obtain new functionality.

* [Adapter](https://en.wikipedia.org/wiki/Adapter_pattern) allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class.

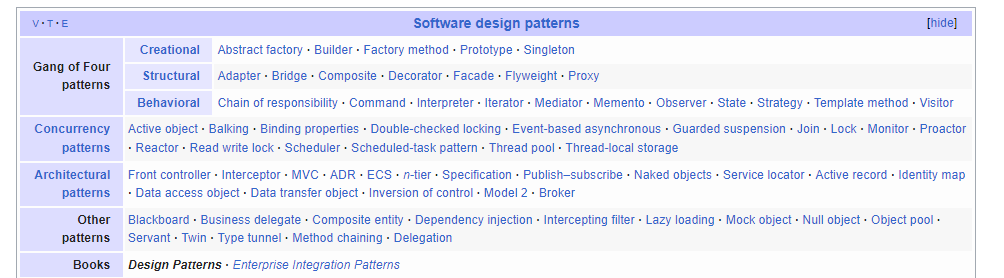
[incompatible interfaces to work together]

* [Bridge](https://en.wikipedia.org/wiki/Bridge_pattern) decouples an abstraction from its implementation so that the two can vary independently.[ decouples an abstraction from its implementation , to vary them independently]
* [Composite](https://en.wikipedia.org/wiki/Composite_pattern) composes zero-or-more similar objects so that they can be manipulated as one object. [ Uniform traversal of objects tree ]
* [Decorator](https://en.wikipedia.org/wiki/Decorator_pattern) dynamically adds/overrides behaviour in an existing method of an object. [ Dynamically Add Responsibility to existing Object]
* [Facade](https://en.wikipedia.org/wiki/Facade_pattern) provides a simplified interface to a large body of code.[Expose simplified interface for complex subsytems]
* [Flyweight](https://en.wikipedia.org/wiki/Flyweight_pattern) reduces the cost of creating and manipulating a large number of similar objects. [Reduce cost of creating lots of similar objects]
* [Proxy](https://en.wikipedia.org/wiki/Proxy_pattern) provides a placeholder for another object to control access, reduce cost, and reduce complexity.

**Behavioral**[[edit](https://en.wikipedia.org/w/index.php?title=Design_Patterns&action=edit&section=6)]

Most [behavioral design patterns](https://en.wikipedia.org/wiki/Behavioral_pattern) are specifically concerned with communication between objects.

* [Chain of responsibility](https://en.wikipedia.org/wiki/Chain-of-responsibility_pattern) delegates commands to a chain of processing objects.
* [Command](https://en.wikipedia.org/wiki/Command_pattern) creates objects that encapsulate actions and parameters.
* [Interpreter](https://en.wikipedia.org/wiki/Interpreter_pattern) implements a specialized language.
* [Iterator](https://en.wikipedia.org/wiki/Iterator_pattern) accesses the elements of an object sequentially without exposing its underlying representation. [Iterate Objects uniformly without exposing]
* [Mediator](https://en.wikipedia.org/wiki/Mediator_pattern) allows [loose coupling](https://en.wikipedia.org/wiki/Loose_coupling) between classes by being the only class that has detailed knowledge of their methods.
* [Memento](https://en.wikipedia.org/wiki/Memento_pattern) provides the ability to restore an object to its previous state (undo).
* [Observer](https://en.wikipedia.org/wiki/Observer_pattern) is a publish/subscribe pattern, which allows a number of observer objects to see an event.
* [State](https://en.wikipedia.org/wiki/State_pattern) allows an object to alter its behavior when its internal state changes.
* [Strategy](https://en.wikipedia.org/wiki/Strategy_pattern) allows one of a family of algorithms to be selected on-the-fly at runtime.
* [Template method](https://en.wikipedia.org/wiki/Template_method_pattern) defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior.
* [Visitor](https://en.wikipedia.org/wiki/Visitor_pattern) separates an algorithm from an object structure by moving the hierarchy of methods into one object.s



**Gangs of Four (GoF) Design Patterns**

Published on August 3, 2022

* [Design Patterns](https://www.digitalocean.com/community/tags/design-patterns)
* [Java](https://www.digitalocean.com/community/tags/java)



By Pankaj



While we believe that this content benefits our community, we have not yet thoroughly reviewed it. If you have any suggestions for improvements, please let us know by clicking the “report an issue“ button at the bottom of the tutorial.

Gangs of Four Design Patterns is the collection of 23 design patterns from the book “Design Patterns: Elements of Reusable Object-Oriented Software”.

Gangs Of Four Design Patterns Book

This book was first published in 1994 and it’s one of the most popular books to learn design patterns. The book was authored by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. It got nicknamed as Gangs of Four design patterns because of four authors. Furthermore, it got a shorter name as “GoF Design Patterns”.

[**GoF Design Pattern Types**](https://www.digitalocean.com/community/tutorials/gangs-of-four-gof-design-patterns#gof-design-pattern-types)

GoF Design Patterns are divided into three categories:

1. **Creational**: The design patterns that deal with the creation of an object.
2. **Structural**: The design patterns in this category deals with the class structure such as Inheritance and Composition.
3. **Behavioral**: This type of design patterns provide solution for the better interaction between objects, how to provide lose coupling, and flexibility to extend easily in future.

[**Creational Design Patterns**](https://www.digitalocean.com/community/tutorials/gangs-of-four-gof-design-patterns#creational-design-patterns)

There are 5 design patterns in the creational design patterns category.

| **Pattern Name** | **Description** |
| --- | --- |
| [Singleton](https://www.digitalocean.com/community/tutorials/java-singleton-design-pattern-best-practices-examples) | The singleton pattern restricts the initialization of a class to ensure that only one instance of the class can be created. |
| [Factory](https://www.digitalocean.com/community/tutorials/factory-design-pattern-in-java) | The factory pattern takes out the responsibility of instantiating a object from the class to a Factory class. |
| [Abstract Factory](https://www.digitalocean.com/community/tutorials/abstract-factory-design-pattern-in-java) | Allows us to create a Factory for factory classes. |
| [Builder](https://www.digitalocean.com/community/tutorials/builder-design-pattern-in-java) | Creating an object step by step and a method to finally get the object instance. |
| [Prototype](https://www.digitalocean.com/community/tutorials/prototype-design-pattern-in-java) | Creating a new object instance from another similar instance and then modify according to our requirements. |

[**Structural Design Patterns**](https://www.digitalocean.com/community/tutorials/gangs-of-four-gof-design-patterns#structural-design-patterns)

There are 7 structural design patterns defined in the Gangs of Four design patterns book.

| **Pattern Name** | **Description** |
| --- | --- |
| [Adapter](https://www.digitalocean.com/community/tutorials/adapter-design-pattern-java) | Provides an interface between two unrelated entities so that they can work together. |
| [Composite](https://www.digitalocean.com/community/tutorials/composite-design-pattern-in-java) | Used when we have to implement a part-whole hierarchy. For example, a diagram made of other pieces such as circle, square, triangle, etc. |
| [Proxy](https://www.digitalocean.com/community/tutorials/proxy-design-pattern) | Provide a surrogate or placeholder for another object to control access to it. |
| [Flyweight](https://www.digitalocean.com/community/tutorials/flyweight-design-pattern-java) | Caching and reusing object instances, used with immutable objects. For example, string pool. |
| [Facade](https://www.digitalocean.com/community/tutorials/facade-design-pattern-in-java) | Creating a wrapper interfaces on top of existing interfaces to help client applications. |
| [Bridge](https://www.digitalocean.com/community/tutorials/bridge-design-pattern-java) | The bridge design pattern is used to decouple the interfaces from implementation and hiding the implementation details from the client program. |
| [Decorator](https://www.digitalocean.com/community/tutorials/decorator-design-pattern-in-java-example) | The decorator design pattern is used to modify the functionality of an object at runtime. |

[**Behavioral Design Patterns**](https://www.digitalocean.com/community/tutorials/gangs-of-four-gof-design-patterns#behavioral-design-patterns)

There are 11 behavioral design patterns defined in the GoF design patterns.

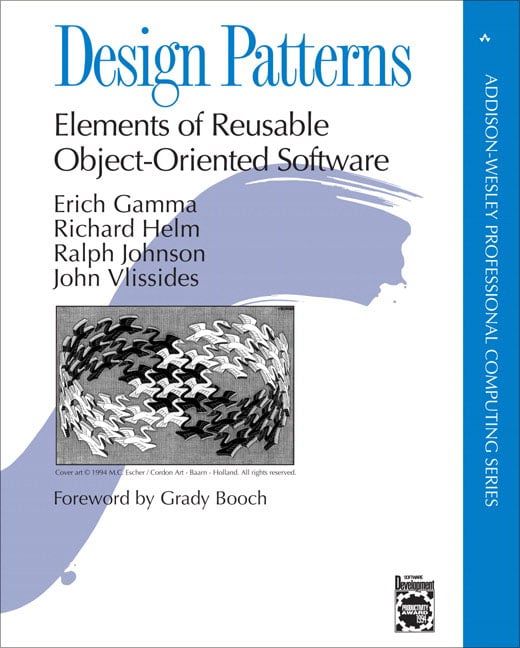
| **Pattern Name** | **Description** |
| --- | --- |
| [Template Method](https://www.digitalocean.com/community/tutorials/template-method-design-pattern-in-java) | used to create a template method stub and defer some of the steps of implementation to the subclasses. |
| [Mediator](https://www.digitalocean.com/community/tutorials/mediator-design-pattern-java) | used to provide a centralized communication medium between different objects in a system. |
| [Chain of Responsibility](https://www.digitalocean.com/community/tutorials/chain-of-responsibility-design-pattern-in-java) | used to achieve loose coupling in software design where a request from the client is passed to a chain of objects to process them. |
| [Observer](https://www.digitalocean.com/community/tutorials/observer-design-pattern-in-java) | useful when you are interested in the state of an object and want to get notified whenever there is any change. |
| [Strategy](https://www.digitalocean.com/community/tutorials/strategy-design-pattern-in-java-example-tutorial) | Strategy pattern is used when we have multiple algorithm for a specific task and client decides the actual implementation to be used at runtime. |
| [Command](https://www.digitalocean.com/community/tutorials/command-design-pattern) | Command Pattern is used to implement lose coupling in a request-response model. |
| [State](https://www.digitalocean.com/community/tutorials/state-design-pattern-java) | State design pattern is used when an Object change it’s behavior based on it’s internal state. |
| [Visitor](https://www.digitalocean.com/community/tutorials/visitor-design-pattern-java) | Visitor pattern is used when we have to perform an operation on a group of similar kind of Objects. |
| [Interpreter](https://www.digitalocean.com/community/tutorials/interpreter-design-pattern-java) | defines a grammatical representation for a language and provides an interpreter to deal with this grammar. |
| [Iterator](https://www.digitalocean.com/community/tutorials/iterator-design-pattern-java) | used to provide a standard way to traverse through a group of Objects. |
| [Memento](https://www.digitalocean.com/community/tutorials/memento-design-pattern-java) | The memento design pattern is used when we want to save the state of an object so that we can restore later on. |

[**Conclusion**](https://www.digitalocean.com/community/tutorials/gangs-of-four-gof-design-patterns#conclusion)

Gangs of Four design patterns lay the foundation of core design patterns in programming. There are many other design patterns built on top of these patterns for specific requirements.

Gang of Four Design Patterns

[Home](https://springframework.guru/) › Gang of Four Design Patterns

[](https://springframework.guru/wp-content/uploads/2015/04/9780201633610.jpg)

Over 20 years ago the iconic computer science book “[Design Patterns: Elements of Reusable Object-Oriented Software](https://www.amazon.com/gp/product/0201633612/ref=as_li_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN=0201633612&linkCode=as2&tag=triatcraft-20&linkId=XRGUDJCGWC6AJNZM)” was first published. The four authors of the book: Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, have since been dubbed “The Gang of Four”. In technology circles, you’ll often see this nicknamed shorted to GoF. Even though the GoF [Design Patterns](https://www.amazon.com/gp/product/0201633612/ref=as_li_tl?ie=UTF8&camp=1789&creative=390957&creativeASIN=0201633612&linkCode=as2&tag=triatcraft-20&linkId=XRGUDJCGWC6AJNZM) book was published over 20 years ago, it still continues to be an Amazon best seller.

The GoF wrote the book in a C++ context but it still remains very relevant to Java programming. C++ and Java are both object-oriented languages. The GoF authors, through their experience in coding large scale enterprise systems using C++, saw common patterns emerge. These design patterns are not unique to C++. The design patterns can be applied in any object oriented language.

As a Java developer using the Spring Framework to develop enterprise class applications, you will encounter the GoF Design Patterns on a daily basis.

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

Gang of Four Design Patterns

Creational Design Patterns

* [Abstract Factory](https://springframework.guru/gang-of-four-design-patterns/abstract-factory-design-pattern/). Allows the creation of objects without specifying their concrete type.
* [Builder](https://springframework.guru/gang-of-four-design-patterns/builder-pattern/). Uses to create complex objects.
* [Factory Method](https://springframework.guru/gang-of-four-design-patterns/factory-method-design-pattern/). Creates objects without specifying the exact class to create.
* [Prototype](https://springframework.guru/gang-of-four-design-patterns/prototype-pattern/). Creates a new object from an existing object.
* [Singleton](https://springframework.guru/gang-of-four-design-patterns/singleton-design-pattern/). Ensures only one instance of an object is created.

Structural Design Patterns

* [Adapter](https://springframework.guru/gang-of-four-design-patterns/adapter-pattern/). Allows for two incompatible classes to work together by wrapping an interface around one of the existing classes.
* [Bridge](https://springframework.guru/gang-of-four-design-patterns/bridge-pattern/). Decouples an abstraction so two classes can vary independently.
* [Composite](https://springframework.guru/gang-of-four-design-patterns/composite-pattern/). Takes a group of objects into a single object.
* [Decorator](https://springframework.guru/gang-of-four-design-patterns/decorator-pattern/). Allows for an object’s behavior to be extended dynamically at run time.
* [Facade](https://springframework.guru/gang-of-four-design-patterns/facade-pattern/). Provides a simple interface to a more complex underlying object.
* [Flyweight](https://springframework.guru/gang-of-four-design-patterns/flyweight-pattern/). Reduces the cost of complex object models.
* [Proxy](https://springframework.guru/gang-of-four-design-patterns/proxy-pattern/). Provides a placeholder interface to an underlying object to control access, reduce cost, or reduce complexity.

Behavior Design Patterns

* [Chain of Responsibility](https://springframework.guru/gang-of-four-design-patterns/chain-of-responsibility-pattern/). Delegates commands to a chain of processing objects.
* [Command](https://springframework.guru/gang-of-four-design-patterns/command-pattern/). 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.
* [Visitor](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.

s