**Design Patterns**

In software engineering, a **design pattern** is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

### **Uses of Design Patterns**

Design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for coders and architects familiar with the patterns.

Often, people only understand how to apply certain software design techniques to certain problems. These techniques are difficult to apply to a broader range of problems. Design patterns provide general solutions, documented in a format that doesn't require specifics tied to a particular problem.

In addition, patterns allow developers to communicate using well-known, well understood names for software interactions.

### **6** [**Creational design patterns**](https://sourcemaking.com/design_patterns/creational_patterns)

These design patterns are all about class instantiation. This pattern can be further divided into class-creation patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.

* [**Factory Method**](https://sourcemaking.com/design_patterns/factory_method)  
  Creates an instance of several derived classes
* [**Abstract Factory**](https://sourcemaking.com/design_patterns/abstract_factory)  
  Creates an instance of several families of classes
* [**Singleton**](https://sourcemaking.com/design_patterns/singleton)  
  A class of which only a single instance can exist
* [**Builder**](https://sourcemaking.com/design_patterns/builder)  
  Separates object construction from its representation
* [**Prototype**](https://sourcemaking.com/design_patterns/prototype)  
  A fully initialized instance to be copied or cloned
* [**Object Pool**](https://sourcemaking.com/design_patterns/object_pool)  
  Avoid expensive acquisition and release of resources by recycling objects that are no longer in use

### **7** [**Structural design patterns**](https://sourcemaking.com/design_patterns/structural_patterns)

These design patterns are all about Class and Object composition. Structural class-creation patterns use inheritance to compose interfaces. Structural object-patterns define ways to compose objects to obtain new functionality.

* [**Adapter**](https://sourcemaking.com/design_patterns/adapter)  
  Match interfaces of different classes
* [**Bridge**](https://sourcemaking.com/design_patterns/bridge)  
  Separates an object’s interface from its implementation
* [**Composite**](https://sourcemaking.com/design_patterns/composite)  
  A tree structure of simple and composite objects
* [**Decorator**](https://sourcemaking.com/design_patterns/decorator)  
  Add responsibilities to objects dynamically
* [**Facade**](https://sourcemaking.com/design_patterns/facade)  
  A single class that represents an entire subsystem
* [**Flyweight**](https://sourcemaking.com/design_patterns/flyweight)  
  A fine-grained instance used for efficient sharing
* [**Proxy**](https://sourcemaking.com/design_patterns/proxy)  
  An object representing another object
* [**Private Class Data**](https://sourcemaking.com/design_patterns/private_class_data)  
  Restricts accessor/mutator access

### **10** [**Behavioral design patterns**](https://sourcemaking.com/design_patterns/behavioral_patterns)

These design patterns are all about Class's objects communication. Behavioral patterns are those patterns that are most specifically concerned with communication between objects.

* [**Chain of responsibility**](https://sourcemaking.com/design_patterns/chain_of_responsibility)  
  A way of passing a request between a chain of objects
* [**Command**](https://sourcemaking.com/design_patterns/command)  
  Encapsulate a command request as an object
* [**Interpreter**](https://sourcemaking.com/design_patterns/interpreter)  
  A way to include language elements in a program
* [**Iterator**](https://sourcemaking.com/design_patterns/iterator)  
  Sequentially access the elements of a collection
* [**Mediator**](https://sourcemaking.com/design_patterns/mediator)  
  Defines simplified communication between classes
* [**Memento**](https://sourcemaking.com/design_patterns/memento)  
  Capture and restore an object's internal state
* [**Null Object**](https://sourcemaking.com/design_patterns/null_object)  
  Designed to act as a default value of an object
* [**Observer**](https://sourcemaking.com/design_patterns/observer)  
  A way of notifying change to a number of classes
* [**State**](https://sourcemaking.com/design_patterns/state)  
  Alter an object's behavior when its state changes
* [**Strategy**](https://sourcemaking.com/design_patterns/strategy)  
  Encapsulates an algorithm inside a class
* [**Template method**](https://sourcemaking.com/design_patterns/template_method)  
  Defer the exact steps of an algorithm to a subclass
* [**Visitor**](https://sourcemaking.com/design_patterns/visitor)  
  Defines a new operation to a class without change

### [**Creational design patterns**](https://sourcemaking.com/design_patterns/creational_patterns)

# Factory Pattern

**define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate.** In other words, subclasses are responsible to create the instance of the class.

# In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

The Factory Method Pattern is also known as **Virtual Constructor.**

#### **Advantage of Factory Design Pattern**

* Factory Method Pattern allows the sub-classes to choose the type of objects to create.
* It promotes the **loose-coupling** by eliminating the need to bind application-specific classes into the code. That means the code interacts solely with the resultant interface or abstract class, so that it will work with any classes that implement that interface or that extends that abstract class.

#### **Usage of Factory Design Pattern**

* When a class doesn't know what sub-classes will be required to create
* When a class wants that its sub-classes specify the objects to be created.
* When the parent classes choose the creation of objects to its sub-classes.



# Abstract Factory Pattern

### Abstract Factory patterns work around a super-factory which creates other factories. This factory is also called as factory of factories.

**define an interface or abstract class for creating families of related (or dependent) objects but without specifying their concrete sub-classes.**

That means Abstract Factory lets a class returns a factory of classes. So, this is the reason that Abstract Factory Pattern is one level higher than the Factory Pattern. An Abstract Factory Pattern is also known as **Kit.**

#### **Advantage of Factory Design Pattern**

* Factory Method Pattern allows the sub-classes to choose the type of objects to create.
* It promotes the **loose-coupling** by eliminating the need to bind application-specific classes into the code. That means the code interacts solely with the resultant interface or abstract class, so that it will work with any classes that implement that interface or that extends that abstract class.

#### **Usage of Factory Design Pattern**

* When a class doesn't know what sub-classes will be required to create
* When a class wants that its sub-classes specify the objects to be created.
* When the parent classes choose the creation of objects to its sub-classes.



# Singleton design pattern in Java

Singleton Pattern says that just **"define a class that has only one instance and provides a global point of access to it".**

In other words, a class must ensure that only single instance should be created and single object can be used by all other classes.

There are two forms of singleton design pattern

* **Early Instantiation:** creation of instance at load time.
* **Lazy Instantiation:** creation of instance when required.

#### **Advantage of Singleton design pattern**

* Saves memory because object is not created at each request. Only single instance is reused again and again.

#### **Usage of Singleton design pattern**

* Singleton pattern is mostly used in multi-threaded and database applications. It is used in logging, caching, thread pools, configuration settings etc.



# Prototype Design Pattern

Prototype pattern refers to creating duplicate object while keeping performance in mind. Prototype Pattern says that **cloning of an existing object instead of creating new one and can also be customized as per the requirement**.

This pattern should be followed, if the cost of creating a new object is expensive and resource intensive

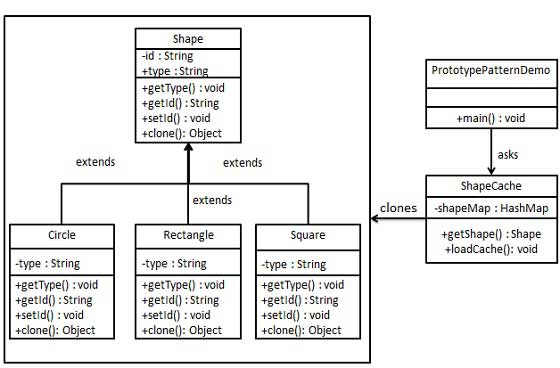
#### **Advantage of Prototype Pattern**

The main advantages of prototype pattern are as follows:

* It reduces the need of sub-classing.
* It hides complexities of creating objects.
* The clients can get new objects without knowing which type of object it will be.
* It lets you add or remove objects at runtime.

#### **Usage of Prototype Pattern**

* When the classes are instantiated at runtime.
* When the cost of creating an object is expensive or complicated.
* When you want to keep the number of classes in an application minimum.
* When the client application needs to be unaware of object creation and representation.



# Builder Design Pattern

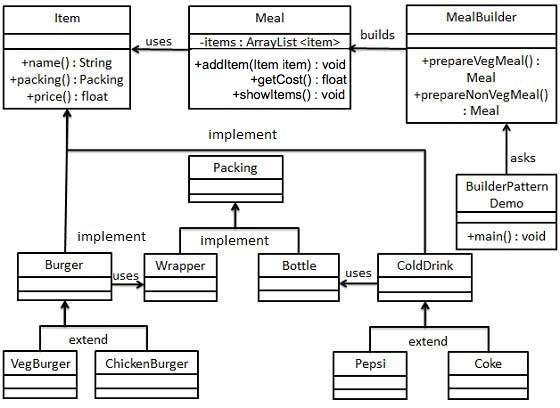
Builder Pattern says that **"construct a complex object from simple objects using step-by-step approach"**

It is mostly used when object can't be created in single step like in the de-serialization of a complex object.

#### **Advantage of Builder Design Pattern**

The main advantages of Builder Pattern are as follows:

* It provides clear separation between the construction and representation of an object.
* It provides better control over construction process.
* It supports to change the internal representation of objects.



# Object Pool Pattern

Mostly, performance is the key issue during the software development and the object creation, which may be a costly step. Object Pool Pattern says that **" to reuse the object that are expensive to create".**

Basically, an Object pool is a container which contains a specified amount of objects. When an object is taken from the pool, it is not available in the pool until it is put back.**Objects in the pool have a lifecycle: creation, validation and destroy.**

#### **Advantage of Object Pool design pattern**

* It boosts the performance of the application significantly.
* It is most effective in a situation where the rate of initializing a class instance is high.
* It manages the connections and provides a way to reuse and share them.
* It can also provide the limit for the maximum number of objects that can be created.

#### **Usage:**

* When an application requires objects which are expensive to create. Eg: there is a need of opening too many connections for the database then it takes too longer to create a new one and the database server will be overloaded.
* When there are several clients who need the same resource at different times.

### [**Structural design patterns**](https://sourcemaking.com/design_patterns/structural_patterns)

# Adapter Pattern

Adapter pattern works as a bridge between two incompatible interfaces This pattern involves a single class which is responsible to join functionalities of independent or incompatible interfaces.

An Adapter Pattern says that just **"converts the interface of a class into another interface that a client wants".**

In other words, to provide the interface according to client requirement while using the services of a class with a different interface.

The Adapter Pattern is also known as **Wrapper.**

#### **Example**

* A real life example could be a case of card reader which acts as an adapter between memory card and a laptop.
* Mobile Charger

#### **Advantage of Adapter Pattern**

* It allows two or more previously incompatible objects to interact.
* It allows reusability of existing functionality.

**Usage of Adapter pattern:**

It is used:

* When an object needs to utilize an existing class with an incompatible interface.
* When you want to create a reusable class that cooperates with classes which don't have compatible interfaces.
* When you want to create a reusable class that cooperates with classes which don't have compatible interfaces

# Bridge Pattern

A Bridge Pattern says that just **"decouple the functional abstraction from the implementation so that the two can vary independently".** This pattern involves an interface which acts as a bridge which makes the functionality of concrete classes independent from interface implementer classes. Both types of classes can be altered structurally without affecting each other

The Bridge Pattern is also known as **Handle or Body.**

#### **Advantage of Bridge Pattern**

* It enables the separation of implementation from the interface.
* It improves the extensibility.
* It allows the hiding of implementation details from the client.

**Usage of Bridge Pattern**

* When you don't want a permanent binding between the functional abstraction and its implementation.
* When both the functional abstraction and its implementation need to extended using sub-classes.
* It is mostly used in those places where changes are made in the implementation does not affect the clients.

# Composite Pattern

A Composite Pattern says that just **"allow clients to operate in generic manner on objects that may or may not represent a hierarchy of objects".**

Composite pattern is used where we need to treat a group of objects in similar way as a single object. Composite pattern composes objects in term of a tree structure to represent part as well as whole hierarchy. This pattern creates a class that contains group of its own objects. This class provides ways to modify its group of same objects.

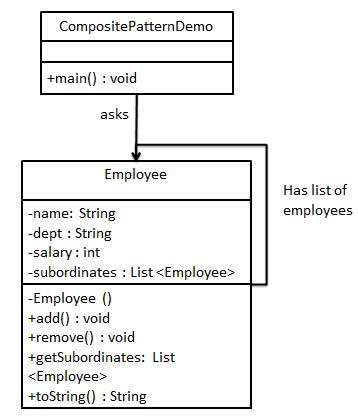
#### **Advantage of Composite Design Pattern**

* It defines class hierarchies that contain primitive and complex objects.
* It makes easier to you to add new kinds of components.
* It provides flexibility of structure with manageable class or interface.

**Usage of Composite Pattern**

It is used:

* When you want to represent a full or partial hierarchy of objects.
* When the responsibilities are needed to be added dynamically to the individual objects without affecting other objects. Where the responsibility of object may vary from time to time.



# Decorator Pattern

A Decorator Pattern says that just **"attach a flexible additional responsibilities to an object dynamically".** Decorator pattern allows a user to add new functionality to an existing object without altering its structure. This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.

In other words, The Decorator Pattern uses composition instead of inheritance to extend the functionality of an object at runtime.

The Decorator Pattern is also known as **Wrapper.**

#### **Advantage of Decorator Pattern**

* It provides greater flexibility than static inheritance.
* It enhances the extensibility of the object, because changes are made by coding new classes.
* It simplifies the coding by allowing you to develop a series of functionality from targeted classes instead of coding all of the behavior into the object.

#### **Usage of Decorator Pattern**

It is used:

* When you want to transparently and dynamically add responsibilities to objects without affecting other objects.
* When you want to add responsibilities to an object that you may want to change in future.
* Extending functionality by sub-classing is no longer practical.

# Facade Pattern

A Facade Pattern says that just **"just provide a unified and simplified interface to a set of interfaces in a subsystem, therefore it hides the complexities of the subsystem from the client".**

Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system. This pattern involves a single class which provides simplified methods required by client and delegates calls to methods of existing system classes.

In other words, Facade Pattern describes a higher-level interface that makes the sub-system easier to use.

Practically, **every Abstract Factory** is a type of **Facade.**

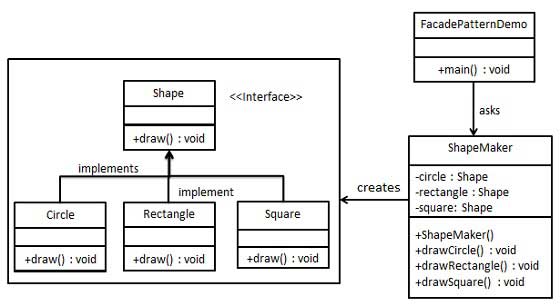
#### **Advantage of Facade Pattern**

* It shields the clients from the complexities of the sub-system components.
* It promotes loose coupling between subsystems and its clients.

**Usage of Facade Pattern:**

It is used:

* When you want to provide simple interface to a complex sub-system.
* When several dependencies exist between clients and the implementation classes of an abstraction.



# Flyweight Pattern

A Flyweight Pattern says that just **"to reuse already existing similar kind of objects by storing them and create new object when no matching object is found"**.

Flyweight pattern is primarily used to reduce the number of objects created and to decrease memory footprint and increase performance. We will demonstrate this pattern by drawing 20 circles of different locations but we will create only 5 objects. Only 5 colors are available so color property is used to check already existing *Circle* objects.

#### **Advantage of Flyweight Pattern**

* It reduces the number of objects.
* It reduces the amount of memory and storage devices required if the objects are persisted

**Usage of Flyweight Pattern**

* When an application uses number of objects
* When the storage cost is high because of the quantity of objects.
* When the application does not depend on object identity.

# Proxy Pattern

Simply, proxy means an object representing another object.

According to GoF, a Proxy Pattern **"provides the control for accessing the original object".**

In proxy pattern, a class represents functionality of another class. In proxy pattern, we create object having original object to interface its functionality to outer world. So, we can perform many operations like hiding the information of original object, on demand loading etc. Proxy pattern is also known as **Surrogate or Placeholder.**

#### **Advantage of Proxy Pattern**

* It provides the protection to the original object from the outside world.

**Usage of Proxy Pattern:**

It is used:

* It can be used in **Virtual Proxy** scenario---Consider a situation where there is multiple database call to extract huge size image. Since this is an expensive operation so here we can use the proxy pattern which would create multiple proxies and point to the huge size memory consuming object for further processing. The real object gets created only when a client first requests/accesses the object and after that we can just refer to the proxy to reuse the object. This avoids duplication of the object and hence saving memory.
* It can be used in **Protective Proxy** scenario---It acts as an authorization layer to verify that whether the actual user has access the appropriate content or not. For example, a proxy server which provides restriction on internet access in office. Only the websites and contents which are valid will be allowed and the remaining ones will be blocked.
* It can be used in **Remote Proxy** scenario---A remote proxy can be thought about the stub in the RPC call. The remote proxy provides a local representation of the object which is present in the different address location. Another example can be providing interface for remote resources such as web service or REST resources.
* It can be used in **Smart Proxy** scenario---A smart proxy provides additional layer of security by interposing specific actions when the object is accessed. For example, to check whether the real object is locked or not before accessing it so that no other objects can change it.

### [**Behavioral design patterns**](https://sourcemaking.com/design_patterns/behavioral_patterns)

# Chain of Responsibility Pattern

In chain of responsibility, sender sends a request to a chain of objects. The request can be handled by any object in the chain.

A Chain of Responsibility Pattern says that just **"avoid coupling the sender of a request to its receiver by giving multiple objects a chance to handle the request".** For example, an ATM uses the Chain of Responsibility design pattern in money giving process. In other words, we can say that normally each receiver contains reference of another receiver. If one object cannot handle the request then it passes the same to the next receiver and so on.

#### **Advantage of Chain of Responsibility Pattern**

* It reduces the coupling.
* It adds flexibility while assigning the responsibilities to objects.
* It allows a set of classes to act as one; events produced in one class can be sent to other handler classes with the help of composition.

**Usage of Chain of Responsibility Pattern:**

It is used:

* When more than one object can handle a request and the handler is unknown.
* When the group of objects that can handle the request must be specified in dynamic way

# Command Pattern

A Command Pattern says that "***encapsulate a request under an object as a command and pass it to invoker object. Invoker object looks for the appropriate object which can handle this command and pass the command to the corresponding object and that object executes the command*"**.

It is also known as **Action or Transaction.**

#### **Advantage of command pattern**

* It separates the object that invokes the operation from the object that actually performs the operation.
* It makes easy to add new commands, because existing classes remain unchanged.

#### **Usage of command pattern:**

It is used:

* When you need parameterize objects according to an action perform.
* When you need to create and execute requests at different times.
* When you need to support rollback, logging or transaction functionality.

# Interpreter Pattern

An Interpreter Pattern says that **"to define a representation of grammar of a given language, along with an interpreter that uses this representation to interpret sentences in the language".**

#### **Advantage of Interpreter Pattern**

* It is easier to change and extend the grammar.
* Implementing the grammar is straightforward.

**Usage of Interpreter pattern:**

It is used:

* When the grammar of the language is not complicated.
* When the efficiency is not a priority.

# Iterator Pattern

According to GoF, Iterator Pattern is used **"to access the elements of an aggregate object sequentially without exposing its underlying implementation".**

The Iterator pattern is also known as **Cursor.**

In collection framework, we are now using Iterator that is preferred over Enumeration.

#### **Advantage of Iterator Pattern**

* It supports variations in the traversal of a collection.
* It simplifies the interface to the collection.

#### **Usage of Iterator Pattern:**

It is used:

* When you want to access a collection of objects without exposing its internal representation.
* When there are multiple traversals of objects need to be supported in the collection.

# Mediator Pattern

A Mediator Pattern says that "**to define an object that encapsulates how a set of objects interact".** Mediator pattern is used to reduce communication complexity between multiple objects or classes. This pattern provides a mediator class which normally handles all the communications between different classes and supports easy maintainability of the code by loose coupling.

## **Benefits:**

* It decouples the number of classes.
* It simplifies object protocols.
* It centralizes the control.
* The individual components become simpler and much easier to deal with because they don't need to pass messages to one another.

The components don't need to contain logic to deal with their intercommunication and therefore, they are more generic.

## **Usage:**

* It is commonly used in message-based systems likewise chat applications.
* When the set of objects communicate in complex but in well-defined ways.

# Memento Pattern

A Memento Pattern says that **"to restore the state of an object to its previous state"**. But it must do this without violating Encapsulation. Such case is useful in case of error or failure.

The Memento pattern is also known as **Token**.

Undo or backspace or ctrl+z is one of the most used operation in an editor. Memento design pattern is used to implement the undo operation. This is done by saving the current state of the object as it changes state.

## **Benefits:**

* It preserves encapsulation boundaries.
* It simplifies the originator.

**Usage:**

* It is used in Undo and Redo operations in most software.
* It is also used in database transactions.

# Observer Pattern

An Observer Pattern says that **"just define a one-to-one dependency so that when one object changes state, all its dependents are notified and updated automatically".**

The Memento pattern is also known as Dependents or Publish-Subscribe.

## **Benefits:**

* It describes the coupling between the objects and the observer.
* It provides the support for broadcast-type communication.

## **Usage:**

* When the change of a state in one object must be reflected in another object without keeping the objects tight coupled.
* When the framework we writes and needs to be enhanced in future with new observers with minimal changes.

# State Pattern

A State Pattern says that **"the class behavior changes based on its state"**. In State Pattern, we create objects which represent various states and a context object whose behavior varies as its state object changes.

The State Pattern is also known as **Objects for States**.

## **Benefits:**

* It keeps the state-specific behavior.
* It makes any state transitions explicit.

## **Usage:**

* When the behavior of object depends on its state and it must be able to change its behavior at runtime according to the new state.
* It is used when the operations have large, multipart conditional statements that depend on the state of an object.

# Strategy Pattern

A Strategy Pattern says that **"defines a family of functionality, encapsulate each one, and make them interchangeable"**.

The Strategy Pattern is also known as **Policy**.

## **Benefits:**

* It provides a substitute to subclassing.
* It defines each behavior within its own class, eliminating the need for conditional statements.
* It makes it easier to extend and incorporate new behavior without changing the application.

## **Usage:**

* When the multiple classes differ only in their behaviors.e.g. Servlet API.
* It is used when you need different variations of an algorithm.

# Template Pattern

A Template Pattern says that **"just define the skeleton of a function in an operation, deferring some steps to its subclasses"**.

## **Benefits:**

* It is very common technique for reusing the code. This is only the main benefit of it.

## **Usage:**

* It is used when the common behavior among sub-classes should be moved to a single common class by avoiding the duplication.

## **Source:**

<https://sourcemaking.com/design_patterns>

<https://www.javatpoint.com/design-patterns-in-java>

<https://www.tutorialspoint.com/design_pattern/facade_pattern.htm>

#### The three categories of design patterns are creational patterns, which allow freedom to choose how to create objects while hiding the logic; structural patterns, which identify and simplify relationships between objects and use inheritance to give the objects additional functions; and behavioral patterns, which identify how the objects communicate with one another.

#### What is the difference between a factory and abstract factory design pattern?

**Answer:**  
Both factory and abstract factories are creational design patterns. The major difference between these two is, a factory pattern creates an object through inheritance and produces only one Product. On the other hand, an abstract factory pattern creates the object through composition and produce families of products.

### How is the bridge pattern different from the adapter pattern?

**Example:** "The adapter and bridge patterns are very similar. They're both considered a form of encapsulation and facilitate more control over code changes without impacting the class structure. However, the main difference between the two is that using a bridge pattern separates the essential data from its implementation, while an adapter pattern allows incompatible classes to interface with each other without changing the source code.

For example, the memory card slot on a computer is the adapter that allows the memory card to interface with the computer. A bridge pattern can be represented through the example of a household light switch. The switch for the device can be in the form of a chain, a 2-position switch or a dimmer, but regardless of the form, the bridge pattern makes sure that the switch performs the task of turning the light on and off."

## **What are the difference between a static class and a singleton class?**

Following are the differences between a static class and a singleton class.

* A static class can not be a top level class and can not implement interfaces where a singleton class can.
* All members of a static class are static but for a Singleton class it is not a requirement.
* A static class get initialized when it is loaded so it can not be lazily loaded where a singleton class can be lazily loaded.
* A static class object is stored in stack whereas singlton class object is stored in heap memory space.

# SOLID Design Principles Explained

SOLID is one of the most popular sets of design principles in object-oriented software development. It’s a mnemonic acronym for the following five design principles:

* Single Responsibility Principle
* [Open/Closed Principle](https://stackify.com/solid-design-open-closed-principle/)
* [Liskov Substitution Principle](https://stackify.com/solid-design-liskov-substitution-principle/)
* [Interface Segregation Principle](https://stackify.com/interface-segregation-principle/)
* [Dependency Inversion](https://stackify.com/dependency-inversion-principle/)

All of them are broadly used and worth knowing. But in this first post of my series about the SOLID principles, I will focus on the first one: the Single Responsibility Principle.

## **Open–closed principle**

In object-oriented programming, the open–closed principle states "software entities should be open for extension, but closed for modification"; that is, such an entity can allow its behaviour to be extended without modifying its source code.

## What is Agile Methodology?

AGILE methodology is a practice that promotes **continuous iteration** of development and testing throughout the software development lifecycle of the project. In the Agile model, both development and testing activities are concurrent, unlike the Waterfall model.