Contents

[Design Pattern Detailed Overview 1](#_Toc499899039)

[What is Design Pattern? 1](#_Toc499899040)

[2.What all are the advantages of Design Patterns? 2](#_Toc499899041)

[3. What all are the types of Design Patterns? 2](#_Toc499899042)

[4. What is Creational Design Pattern? 2](#_Toc499899043)

[5. What is Structural Design Pattern? 3](#_Toc499899044)

[7.What is Factory Design Pattern? 3](#_Toc499899045)

[8. What is Abstract Factory Design Pattern? 3](#_Toc499899046)

[9. What is Prototype Design Pattern? 4](#_Toc499899047)

[10. What is Builder Design Pattern? 4](#_Toc499899048)

[11. What is Adapter Design Pattern? 4](#_Toc499899049)

[12. What is Bridge Design Pattern? 4](#_Toc499899050)

[13. What is Composite Design Pattern? 4](#_Toc499899051)

[14. What is Decorator Design Pattern? 5](#_Toc499899052)

[15. What is Facade Design Pattern? 5](#_Toc499899053)

[16. What is Flyweight Design Pattern? 5](#_Toc499899054)

[17.What is Proxy Design Pattern? 6](#_Toc499899055)

[Design Pattern Brief Overview 6](#_Toc499899056)

[**Creational Design Patterns:** 6](#_Toc499899057)

[**Behavioural Design Patterns:** 6](#_Toc499899058)

[**Structural Design Patterns:** 7](#_Toc499899059)

# Design Pattern Detailed Overview

### **What is Design Pattern?**

A design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. A design pattern is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations. Patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system

### 2. What all are the advantages of Design Patterns?

Patterns solve software structural and non functional problems.

#### **Structural problems like:**

* *Abstraction*
* *Encapsulation*
* *Data Hiding*
* *Separation of Concerns*
* *Separation of Interface and Implementation*
* *Single point of reference*
* *Coupling and Cohesion etc..*

#### **Non Functional problems like:**

* Efficiency
* Reliability
* Interoperability
* Testability
* Reusability etc.

### 3. What all are the types of Design Patterns?

There are 3 types of Design Pattern.

* Creational Patterns
  + This type of pattern address problems of creating an object and separating it from operations
* Structural Patterns
  + This type of pattern address problems of using object oriented constructs to organize classes and objects
* Behavioural Patterns
  + This type of pattern address problems of assigning responsibilities to classes

### 4. What is Creational Design Pattern?

In software engineering, **creational design patterns**are **design patterns** that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.

In C#, we have 5 types of Design Patterns in Creational Category.

* Singleton
* Factory
* Abstract Factory
* Prototype
* Builder

### 5. What is Structural Design Pattern?

**Structural patterns** are concerned with how classes and objects are composed to form larger **structures**; the class form of the Adapter **design pattern** is an example.

**Structural** class **patterns** use inheritance to compose interface or implementations.

Structural Design Patterns are Design Patterns that ease the design by identifying a simple way to realize relationships between entities.

In C#, We have 7 types of design patterns in Structural Category.

* Adapter
* Bridge
* Composite
* Decorator
* Facade
* Flyweight
* Proxy

**6. What is Singleton Design Pattern?**

* Singleton ensures a class only has one instance.
* Singleton Provides a global point of access to it.

You can see more details on Singleton Design patterns [**here**](http://www.csharpstar.com/creational-design-patterns/).

### 7.What is Factory Design Pattern?

Factory Design patterns:

* Define an Interface for creating an object but let subclasses decide which class to instantiate
* Lets a class defer instantiation to subclasses

You can see more details on Factory Design patterns [**here**](http://www.csharpstar.com/creational-design-patterns/).

### 8. What is Abstract Factory Design Pattern?

Abstract Factory Design patterns:

* Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
* Abstract Factory patterns acts a super-factory which creates other factories. This pattern is also called as Factory of factories

You can see more details on Abstract Factory Design patterns [**here**](http://www.csharpstar.com/creational-design-patterns/).

### 9. What is Prototype Design Pattern?

Prototype Design patterns:

* Prototype pattern specifies the kind of objects to create using a prototypical instance, and create new objects by copying this prototype.
* It is used to create a duplicate object or clone of the current object to enhance performance.

You can see more details on Prototype Design patterns [**here**](http://www.csharpstar.com/creational-design-patterns/).

### 10. What is Builder Design Pattern?

Builder Design patterns:

* Separate the construction of a complex object from its representation so that the same construction process can create different representations.
* In other words, you will have to design the system in such a way that the client application will simply specify the parameters that should be used to create the complex object and the builder will take care of building the complex object.

You can see more details on Builder Factory Design patterns [**here**](http://www.csharpstar.com/creational-design-patterns/).

### 11. What is Adapter Design Pattern?

Adapter Design patterns:

* The adapter pattern is adapting between classes and objects
* This pattern involves a single class called adapter which is responsible for communication between two independent or incompatible interfaces
* This works like a bridge between two incompatible interfaces

You can see more details on Abstract Design patterns [**here**](http://www.csharpstar.com/structural-design-patterns/).

### 12. What is Bridge Design Pattern?

Bridge Design patterns:

* Bridge Pattern separates abstraction from its implementation, so that both can be modified Independently
* Bridge Pattern behaves like a bridge between abstraction class and Implementer class.

You can see more details on Bridge Design patterns [**here**](http://www.csharpstar.com/structural-design-patterns/).

### 13. What is Composite Design Pattern?

Composite Design patterns:

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

You can see more details on Composite Design patterns [**here**](http://www.csharpstar.com/structural-design-patterns/).

### 14. What is Decorator Design Pattern?

Decorator Design patterns:

* Decorator pattern is used to add new functionality to an existing object without changing its structure.
* Decorators provide a flexible alternative to subclass for extending functionality.
* This pattern creates a decorator class which wraps the original class and add new behaviors/operations to an object at run-time.

You can see more details on Decorator Design patterns [**here**](http://www.csharpstar.com/structural-design-patterns/).

### 15. What is Facade Design Pattern?

Facade Design patterns:

* Facade Design Pattern makes a software library easier to use, understand and test
* Facade Design Pattern make the library more readable
* Facade Design Pattern reduce dependencies of outside code on the inner workings of a library
* Facade Design Pattern wrap a poorly designed collection of APIs with a single well-designed API.

You can see more details on Facade Design patterns [**here**](http://www.csharpstar.com/structural-design-patterns/).

### 16. What is Flyweight Design Pattern?

Flyweight Design patterns:

* Flyweight design pattern is an object that minimizes memory use by sharing as much data as possible with other similar objects
* Flyweight pattern is used to reduce the number of objects created, to decrease memory and resource usage. As a result it increase performance
* Flyweight design pattern provides  a way to use objects in large numbers when a simple repeated representation would use an unacceptable amount of memory.
* The flyweight pattern uses the concepts of intrinsic and extrinsic data. **Intrinsic data** is held in the properties of the shared flyweight objects. This information is stateless and generally remains unchanged, if any change occurs it would be reflected among all of the objects that reference the flyweight. **Extrinsic data** is computed on the fly means at runtime and it is held outside of a flyweight object. Hence it can be stateful.

### 17. What is Proxy Design Pattern?

Proxy Design patterns:

* Proxy Design pattern involves a class, called proxy class, which represents functionality of another class.
* Proxy is a wrapper or agent object that is being called by the client to access the real serving object behind the scenes.

# Design Pattern Brief Overview

**Creational Design Patterns:**

[**Singleton**](http://www.oodesign.com/singleton-pattern.html) - Ensure that only one instance of a class is created and Provide a global access point to the object.

[When to Use](javascript:void(0)), [Common Usage](javascript:void(0)), and Example: [Lazy Singleton in Java](http://www.oodesign.com/singleton-pattern.html#lazy-singleton)

[**Factory**](http://www.oodesign.com/factory-pattern.html)(Simplified version of Factory Method) - Creates objects without exposing the instantiation logic to the client and Refers to the newly created object through a common interface.

[**Factory Method**](http://www.oodesign.com/factory-method-pattern.html) - Defines an interface for creating objects, but let subclasses to decide which class to instantiate and Refers to the newly created object through a common interface.

[**Abstract Factory**](http://www.oodesign.com/abstract-factory-pattern.html) - Offers the interface for creating a family of related objects, without explicitly specifying their classes.

[**Builder**](http://www.oodesign.com/builder-pattern.html) - Defines an instance for creating an object but letting subclasses decide which class to instantiate and Allows a finer control over the construction process.

Example: [Text Converter in Java](http://www.oodesign.com/builder-pattern.html#text-converting-example)

[**Prototype**](http://www.oodesign.com/prototype-pattern.html) - Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

[**Object Pool**](http://www.oodesign.com/object-pool-pattern.html) - reuses and shares objects that are expensive to create..

Sourcecode: [Database Connection Pool in Java](http://www.oodesign.com/)

**Behavioural Design Patterns:**

[**Chain of Responsibiliy**](http://www.oodesign.com/chain-of-responsibility-pattern.html) - It avoids attaching the sender of a request to its receiver, giving this way other objects the possibility of handling the request too.  
- The objects become parts of a chain and the request is sent from one object to another across the chain until one of the objects will handle it.

Restaurant Example

[**Command**](http://www.oodesign.com/command-pattern.html) - Encapsulate a request in an object, Allows the parameterization of clients with different requests and Allows saving the requests in a queue.

[**Interpreter**](http://www.oodesign.com/interpreter-pattern.html) - Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language / Map a domain to a language, the language to a grammar, and the grammar to a hierarchical object-oriented design

[**Iterator**](http://www.oodesign.com/iterator-pattern.html) - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

[**Mediator**](http://www.oodesign.com/mediator-pattern.html) - Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

[**Observer**](http://www.oodesign.com/observer-pattern.html) - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

[**Strategy**](http://www.oodesign.com/strategy-pattern.html) - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

[**Template Method**](http://www.oodesign.com/template-method-pattern.html) - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses / Template Method lets subclasses redefine certain steps of an algorithm without letting them to change the algorithm's structure.

[**Visitor**](http://www.oodesign.com/visitor-pattern.html) - Represents an operation to be performed on the elements of an object structure / Visitor lets you define a new operation without changing the classes of the elements on which it operates.

**Structural Design Patterns:**

[**Adapter**](http://www.oodesign.com/adapter-pattern.html) - Convert the interface of a class into another interface clients expect. / Adapter lets classes work together, that could not otherwise because of incompatible interfaces.

[**Bridge**](http://www.oodesign.com/bridge-pattern.html) - Compose objects into tree structures to represent part-whole hierarchies. / Composite lets clients treat individual objects and compositions of objects uniformly.

[**Composite**](http://www.oodesign.com/composite-pattern.html) - Compose objects into tree structures to represent part-whole hierarchies. / Composite lets clients treat individual objects and compositions of objects uniformly.

[**Decorator**](http://www.oodesign.com/decorator-pattern.html) - add additional responsibilities dynamically to an object.

[**Flyweight**](http://www.oodesign.com/flyweight-pattern.html) - use sharing to support a large number of objects that have part of their internal state in common where the other part of state can vary.

[**Memento**](http://www.oodesign.com/memento-pattern.html) - capture the internal state of an object without violating encapsulation and thus providing a mean for restoring the object into initial state when needed.

[**Proxy**](http://www.oodesign.com/proxy-pattern.html) - provide a “Placeholder” for an object to control references to it.