**DESIGN PATTERNS**

Design patterns are recurring solutions to software design problems you find again and again in real-world application development. Patterns are about design and interaction of objects, as well as providing a communication platform concerning elegant, reusable solutions to commonly encountered programming challenges.

The Gang of Four (GoF) patterns are generally considered the foundation for all other patterns. They are categorized in three groups: Creational, Structural, and Behavioral. Here you will find information on these important patterns.

To give you a head start, the C# source code is provided in 2 forms: 'structural' and 'real-world'. Structural code uses type names as defined in the pattern definition and UML diagrams. Real-world code provides real-world programming situations where you may use these patterns.

|  |
| --- |
| **Creational Patterns** |

**1. Abstract Factory**

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

**2. Builder**

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

**3. Factory Method**

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

**4. Prototype**

Specify the kind of objects to create using a prototypical instance, and create new objects by copying this prototype.

**5. Singleton**

Ensure a class has only one instance and provide a global point of access to it.

|  |
| --- |
| **Structural Patterns** |

**1. Adapter**

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

**2. Bridge**

Decouple an abstraction from its implementation so that the two can vary independently.

**3. Composite**

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

**4. Decorator**

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

**5. Facade**

Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.

**6. Flyweight**

Use sharing to support large numbers of fine-grained objects efficiently.

**7. Proxy**

Provide a surrogate or placeholder for another object to control access to it.

|  |
| --- |
| **Behavioral Patterns** |

**1. Chain of Responsibility**

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

**2. Command**

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

**3. Interpreter**

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

**4. Iterator**

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

**5. Mediator**

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.

**6. Memento**

Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

**7. Observer**

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

**8. State**

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

**9. Strategy**

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

**10. Template Method**

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 changing the algorithm's structure.

**11. Visitor**

Represent 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.