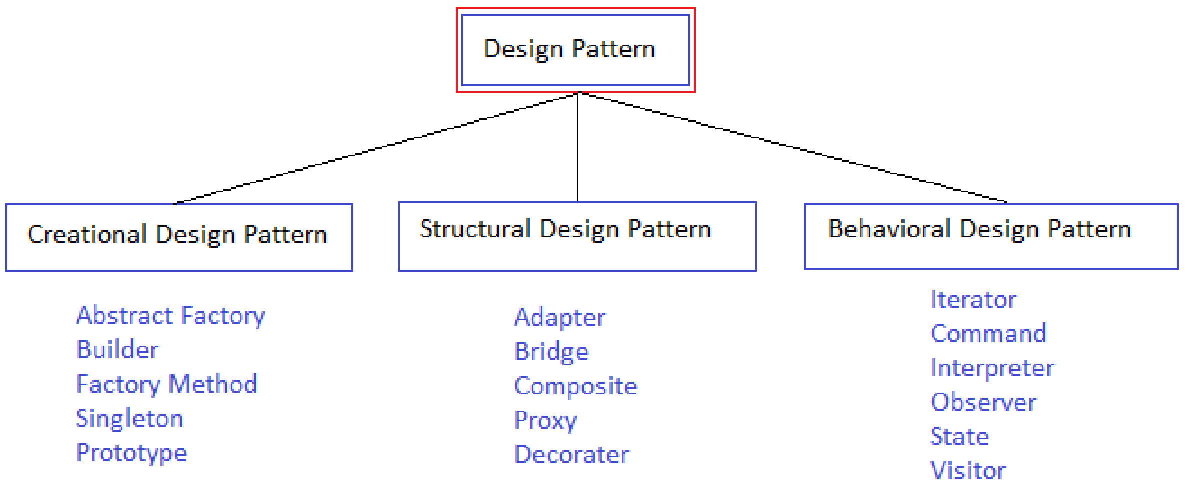
C++ Design Patterns

Software design patterns are general reusable solutions to problems which occur over and over again in object-oriented design environment.

It is not a finished design that can be transformed into source code directly, but it is template how to solve the problem.

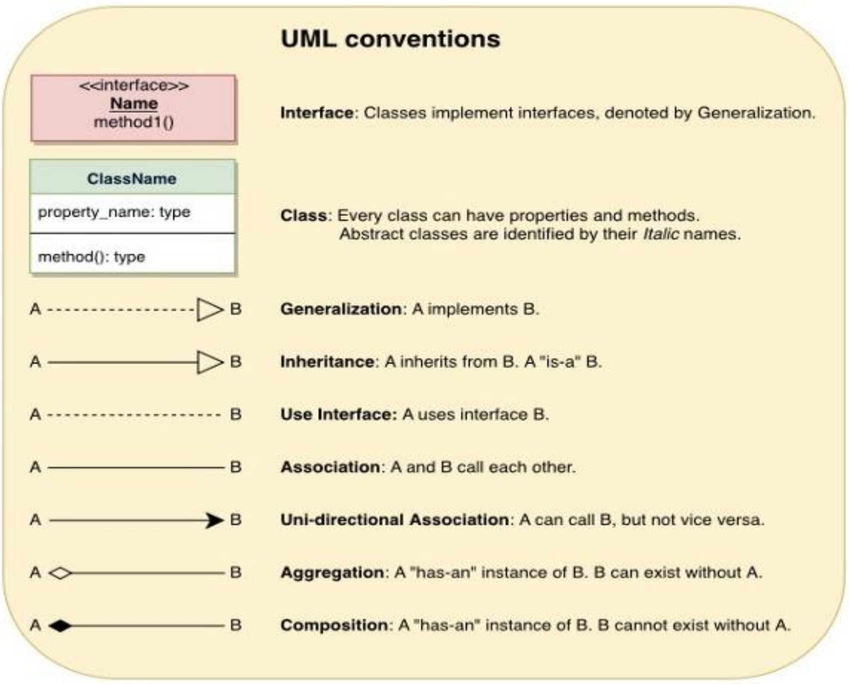
The patterns typically show relationships and interactions between classes or objects. The idea is to speed up the development process by providing well tested, proven development/design paradigm. Design patterns are programming language independent strategies for solving a common problem. That means a design pattern represents an idea, not a particular implementation. By using the design pattern, you can make your code more flexible, reusable and maintainable.

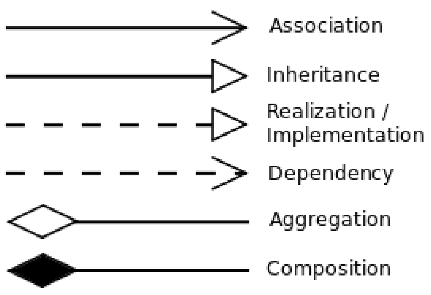


Visibility:

1. ***+ Public***
2. ***- Private***
3. ***# Protected***
4. ***~ Package***

Relationship:


UML conventions
<<interface>>
Name
method1()
ClassName
property_name: type
method(): type
A
B
Interface: Classes implement interfaces, denoted by Generalization.
Class: Every class can have properties and methods. Abstract classes are identified by their Italic names.
Generalization: A implements B.
Inheritance: A inherits from B. A "is-a" B.
Use Interface: A uses interface B.
Association: A and B call each other.
Uni-directional Association: A can call B, but not vice versa.
Aggregation: A "has-an" instance of B. B can exist without A.
Composition: A "has-an" instance of B. B cannot exist without A.



Creational Design 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.

Use case:

Suppose a developer wants to create a simple DBConnection class to connect to a database and wants to access the database at multiple locations from code, generally what developer will do is create an instance of DBConnection class and use it for doing database operations wherever required. Which results in creating multiple connections from the database as each instance of DBConnection class will have a separate connection to the database. In order to deal with it, we create DBConnection class as a singleton class, so that only one instance of DBConnection is created and a single connection is established. Because we can manage DB Connection via one instance so we can control load balance and the unnecessary connections.

1) **Abstract Factory -** families of product objects

Creates an instance of several families of classes

2) **Builder -** how a composite object gets created

Separates object construction from its representation

3) **Factory Method -** subclass of object that is instantiated

Creates an instance of several derived classes

4) **Prototype -** class of object that is instantiated

A fully initialized instance to be copied or cloned

5) **Singleton -** the sole instance of a class

A class of which only a single instance can exist

6) **Object Pool -**

Avoid expensive acquisition and release of resources by recycling objects that are no longer in use

Structural Design 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.

Use case:

When 2 interfaces are not compatible with each other and want to make establish a relationship between them through an adapter it is called adapter design pattern. Adapter pattern converts the interface of a class into another interface or classes the client expects that is adapter lets classes works together that could not otherwise because of incompatibility. So, in these types of incompatible scenarios, we can go for a adapter pattern.

1) **Adapter -** interface to an object

Match interfaces of different classes

2) **Bridge -** implementation of an object

Separates an object's interface from its implementation

3) **Composite -** structure and composition of an object

A tree structure of simple and composite objects

4) **Decorator -** responsibilities of an object without subclassing

Add responsibilities to objects dynamically

5) **Facade -** interface to a subsystem

A single class that represents an entire subsystem

6) **Flyweight -** storage costs of objects

A fine-grained instance used for efficient sharing

7) **Proxy -** how an object is accessed (its location)

An object representing another object

8) **Private Class Data -**

Restricts accessor/mutator access

Behavioral Design Patterns:

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

Use case:

Template pattern defines the skeleton of an algorithm in an operation deferring some steps to sub-classes, Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm structure. say for an example in your project you want the behaviour of the module can be extended, such that we can make the module behave in new and different ways as the requirements of the application change, or to meet the needs of new applications. However, No one is allowed to make source code changes to it. it means you can add but can't modify the structure in those scenarios a developer can approach template design patterns.

1) **Chain of Responsibility -** object that can full fill a request

A way of passing a request between a chain of objects

2) **Command -** when and how a request is fulfilled

Encapsulate a command request as an object

3) **Interpreter -** grammar and interpretation of a language

Way to include language elements in a program

4) **Iterator -** how an aggregate's elements are accessed

Sequentially access the elements of a collection

5) **Mediator -** how and which objects interact with each other

Defines simplified communication between classes

6) **Memento -** what private information is stored outside an object, and when

Capture and restore an object's internal state

7) **Observer -** how the dependent objects stay up to date

A way of notifying change to a number of classes

8) **State -** states of an object

Alter an object's behavior when its state changes

9) **Strategy -** algorithm

Encapsulates an algorithm inside a class

10) **Template Method -** steps of an algorithm

Defer the exact steps of an algorithm to a sub-class

11) **Visitor -** operations that can be applied to objects without changing their classes

Defines a new operation to a class without change

12) **Null Object -**

Designed to act as a default value of an object