**Design Patterns**

****

In software design, a software design pattern is a generic, reusable solution to a common problem in a specific environment.

It's not a finished design that can be turned into source or machine code right away. Rather, it is a description or template for solving a problem that may be applied to a variety of scenarios. When developing an application or system, design patterns are formalized best practices that a programmer can employ to overcome common challenges.

Object-oriented design patterns generally depict relationships and interactions between classes or objects without naming the actual application classes or objects. Functional programming languages may not be suitable to patterns that entail mutable state. Some patterns may be rendered obsolete in languages that provide built-in support for tackling the problem at hand, while object-oriented patterns may not be appropriate for non-object-oriented languages.

Design patterns are a structured approach to computer programming that falls in between a programming paradigm and a concrete method.

**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. Common design patterns can be improved over time, making them more robust than ad-hoc designs.

**Creational Design Patterns**

It's all about class instantiation in these design patterns. This pattern can be further subdivided into patterns for creating classes and patterns for creating objects. While class-creation patterns make good use of inheritance in the instantiation process, object-creation patterns make good use of delegation.

* [**Abstract Factory**](https://sourcemaking.com/design_patterns/abstract_factory)  
  Creates an instance of several families of classes
* [**Builder**](https://sourcemaking.com/design_patterns/builder)  
  Separates object construction from its representation
* [**Factory Method**](https://sourcemaking.com/design_patterns/factory_method)  
  Creates an instance of several derived classes
* [**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
* [**Prototype**](https://sourcemaking.com/design_patterns/prototype)  
  A fully initialized instance to be copied or cloned
* [**Singleton**](https://sourcemaking.com/design_patterns/singleton)  
  A class of which only a single instance can exist

**Structural Design Patterns**

It's all about Class and Object composition in these design patterns. Inheritance is used to construct interfaces in structural class-creation patterns. Structural object-patterns define how to put objects together to get 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
* [**Private Class Data**](https://sourcemaking.com/design_patterns/private_class_data)  
  Restricts accessor/mutator access
* [**Proxy**](https://sourcemaking.com/design_patterns/proxy)  
  An object representing another object

**Behavioral Design Patterns**

The communication of Class's objects is the focus of these design patterns. Behavioral patterns are those that are primarily concerned with the communication between things.

* [**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

Some in the field of computer science have challenged the concept of design patterns.

**Targets the wrong problem**

The need for patterns results from using computer languages or techniques with insufficient abstraction ability. Under ideal factoring, a concept should not be copied, but merely referenced. But if something is referenced instead of copied, then there is no "pattern" to label and catalog. Paul Graham writes in the essay Revenge of the Nerds**.**

**Lacks formal foundations**

The study of design patterns has been far too haphazard, according to others, and the subject urgently needs to be formalized. The Gang of Four were subjected to a show trial at OOPSLA 1999, in which they were "charged" with several crimes against computer science (with their full assistance). Two-thirds of the "jurors" who attended the trial were "convicted."

**Leads to inefficient solutions**

A design pattern is an attempt to standardize what are already well recognized best practices. While this may appear to be useful in theory, it frequently leads in excessive code duplication in practice. Using a well-factored implementation rather than a "just barely good enough" design style is nearly always a more efficient approach.

**Does not differ significantly from other abstractions**

The concept of a design pattern is to codify what are already widely accepted best practices. In theory, this appears to be advantageous, but in fact, it frequently leads in unneeded code duplication. Using a well-factored implementation rather to a "just barely good enough" design style is nearly always the more efficient solution.