# Design Patterns

# What is Design Patterns?

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.

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.

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.

# Common themes in design patterns

Design patterns are used by experts to make their designs more flexible and reusable, so in studying them you tend to see:

* *decoupling* of classes or objects, so something can be changed without changing something else. "Each design pattern lets some aspect of the system vary independently of other aspects, thereby making a system more robust to a particular kind of change."
* elements in a pattern which know how to do one general thing very well and don't know much about anything else.
* efforts to simplify code, to make it more readable, more understandable.
* a major emphasis on interfaces ("program to an interface, not to an implementation"), and on the distinction between types and classes.
* a favoring of object composition over class inheritance: too much inheritance leads to an explosion of the number of classes in a system.

# The 23 patterns from the book

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Creational** | **Structural** | **Behavioral** |
| **Class** | Factory Method | Adapter (class) | Interpreter Template Method |
| **Object** | Abstract Factory Builder Prototype Singleton | Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy | Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor |

# List of the Original 23 Patterns

|  |  |  |
| --- | --- | --- |
| **Purpose** | **Design Pattern** | **Aspect(s) that can vary** |
| **Creational** | Abstract Factory | families of product objects |
| Builder | how a composite object gets created |
| Factory Method | subclass of object that is instantiated |
| Prototype | class of object that is instantiated |
| Singleton | the sole instance of a class |
| **Structural** | Adapter | interface to an object |
| Bridge | implementation of an object |
| Composite | structure and composition of an object |
| Decorator | responsibilities of an object without subclassing |
| Facade | interface to a subsystem |
| Flyweight | storage costs of objects |
| Proxy | how an object is accessed; its location |
| **Behavioral** | Chain of Responsibility | object that can fulfill a request |
| Command | when and how a request is fulfilled |
| Interpreter | grammar and interpretation of a language |
| Iterator | how an aggregate's elements are accessed, traversed |
| Mediator | how and which objects interact with each other |
| Memento | what private information is stored outside an object, and when |
| Observer | number of objects that depend on another object; how the dependent objects stay up to date |
| State | states of an object |
| Strategy | an algorithm |
| Template Method | steps of an algorithm |
| Visitor | operations that can be applied to object(s) without changing their class(es) |

# Classification of Design Patterns

Design Patterns are categorized mainly into three categories:

1. **Creational Design Pattern**
2. **Structural Design Pattern**
3. **Behavioural Design Pattern.**

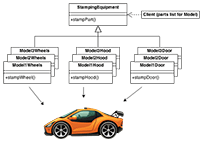
These are differed from each other on the basis of their level of detail, complexity, and scale of applicability to the entire system being design.

# 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.

Class-creation patterns use inheritance effectively in the instantiation process

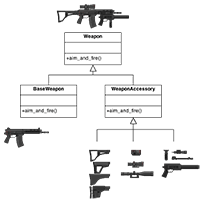
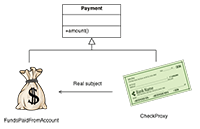
Object-creation patterns use delegation effectively to get the job done.

[](https://sourcemaking.com/design_patterns/abstract_factory)

* [**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

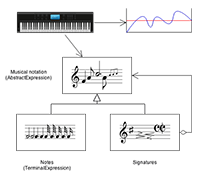
These design patterns are all about Class and Object composition. Structural class-creation patterns use inheritance to compose interfaces. Structural object-patterns define ways to compose objects to obtain new functionality.

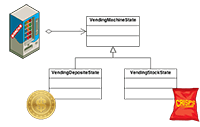
**[](https://sourcemaking.com/design_patterns/proxy)**

* [**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

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

[](https://sourcemaking.com/design_patterns/interpreter)

* [**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
* **[](https://sourcemaking.com/design_patterns/state)**

[**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