### 1. Creational Design Patterns

A creational design pattern deals with object creation and initialization, providing guidance about which objects are created for a given situation. These design patterns are used to increase flexibility and to reuse existing code.

* **Factory Method**: Creates objects with a common interface and lets a class defer instantiation to subclasses.
* **Abstract Factory**: Creates a family of related objects.
* **Builder**: A step-by-step pattern for creating complex objects, separating construction and representation.
* **Prototype**: Supports the copying of existing objects without code becoming dependent on classes.
* **Singleton**: Restricts object creation for a class to only one instance.

### 2. Structural Design Patterns

A structural design pattern deals with class and object composition, or how to assemble objects and classes into larger structures.

* **Adapter**: How to change or adapt an interface to that of another existing class to allow incompatible interfaces to work together.
* **Bridge**: A method to decouple an interface from its implementation.
* **Composite**: Leverages a tree structure to support manipulation as one object.
* **Decorator**: Dynamically extends (adds or overrides) functionality.
* **Façade**: Defines a high-level interface to simplify the use of a large body of code.
* **Flyweight**: Minimize memory use by sharing data with similar objects.
* **Proxy**: How to represent an object with another object to enable access control, reduce cost and reduce complexity.

### 3. Behavioral Design Patterns

A behavioral design pattern is concerned with communication between objects and how responsibilities are assigned between objects.

* **Chain of Responsibility**: A method for commands to be delegated to a chain of processing objects.
* **Command**: Encapsulates a command request in an object.
* **Interpreter**: Supports the use of language elements within an application.
* **Iterator**: Supports iterative (sequential) access to collection elements.
* **Mediator**: Articulates simple communication between classes.
* **Memento**: A process to save and restore the internal/original state of an object.
* **Observer**: Defines how to notify objects of changes to other object(s).
* **State**: How to alter the behavior of an object when its stage changes.
* **Strategy**: Encapsulates an algorithm inside a class.
* **Visitor**: Defines a new operation on a class without making changes to the class.
* **Template Method**: Defines the skeleton of an operation while allowing subclasses to refine certain steps.

## **Why Do We Need Design Patterns?**

Design patterns offer a best practice approach to support object-oriented software design, which is easier to design, implement, change, test and reuse. These design patterns provide best practices and structures.

### 1. Proven Solution

Design patterns provide a proven, reliable solution to a common problem, meaning the [software developer](https://www.netsolutions.com/hire-developers) does not have to “reinvent the wheel” when that problem occurs.

### 2. Reusable

Design patterns can be modified to solve many kinds of problems – they are not just tied to a single problem.

### 3. Expressive

Design patterns are an elegant solution.

### 4. Prevent the Need for Refactoring Code

Since the design pattern is already the optimal solution for the problem, this can avoid refactoring.

### 5. Lower the Size of the Codebase

Each pattern helps software developers change how the system works without a full redesign. Further, as the “optimal” solution, the design pattern often requires less code.

## **Benefits of Design Patterns in Software Engineering**

As noted above under “Why do we need design patterns?”, the best software design patterns will leverage a common language, making it easier for developers to communicate about problems and to improve code readability and architecture in early stages of the planning. When used well, design patterns can both speed up the development process and generally reduce the chance of errors.

Design patterns are also language agnostic (for object-oriented languages), although some patterns are more useful with certain languages vs others.

## **Criticism of Software Design Patterns**

Software design patterns can come under criticism for being overused, a crutch for programmers to lean back on when a more simple solution can do. Further, there is not always a straightforward solution to implementing each pattern, with the risk of creating an **anti-pattern** (an ineffective or counterproductive solution) if the wrong method is chosen.

Additionally, a design pattern can be used as a bridge for weaknesses or missing features in the programming language, often creating more bloat than is necessary to get the program to function as desired. In the important process of [choosing the right tech stack](https://www.netsolutions.com/insights/tech-stack-for-web-mobile-app/), ensure that the language has the features needed to avoid an over-reliance on design patterns. Alternatively, the choice in tech stack can lead you to a framework that already has these design patterns implemented directly in the framework in the most ideal way possible.

## **7 Best Software Design Patterns**

Although there are 23 design patterns listed in Design Patterns: Elements of Reusable Object-Oriented Software, of those there are 7 that are considered the most influential or important. This section outlines the 7 best software design patterns, why they are important, and when to use them.

### 1. Singleton Design Pattern

The singleton design pattern falls under the “creational” type, restricting object creation for a class to only one instance and providing global access to a global variable. For example, many web developers lock up the “sitemap” to a single version that has global scope. Further, other patterns such as factory method, builder, prototype can use singletons. Facade and state objects are also often singletons.

While you may only have or need one instance of a class, this does not necessarily mean that is the time to use a singleton pattern to lock that object up or to force it into a global state. Singletons are a controversial design pattern, with some even arguing that singletons are an [antipattern](https://www.contentful.com/blog/singleton-pattern-evil-or-misused/) to be avoided because locking up an object restricts future flexibility.

### 2. Factory Method Design Pattern

In the factory method, a “creation” design pattern, developers create objects with a common interface but allow a class defer instantiation to subclasses. The factory method promotes loose coupling and code reuse, a “virtual constructor” that works with any class implementing the interface and allowing greater freedom for the sub-classes to choose the objects being created. When new classes are needed, they can be added to the factory.

The factory method is not appropriate for simple scenarios, an instance where developers risk over-complicating processes in order to use a design pattern.

### 3. Facade Design Pattern

The facade design pattern is a “structural” design pattern that helps provide one interface (class) for access to a large body of code / various objects. A facade hides complexities of various sub-systems (often organized into a class) with a simple interface. For example, an eCommerce customer only wants one point of interacting with a brand, rather than individually communicating (interfacing) with each system to support the sale such as product inventory, authentication, security, payment processing, order fulfillment, etc. In this case, the Facade has encapsulated all the “order” activities and systems to provide a single interface – the customer remains completely unaware of what’s going on behind the scenes. Facade is an important concept to support the loosely coupled [microservices architecture](https://www.netsolutions.com/insights/monolithic-vs-microservices/).

### 4. Strategy Design Pattern

The strategy design pattern is a “behavioral” software design pattern that is sometimes known as a policy pattern. In the strategy pattern, interchangeable algorithms are encapsulated together into a “family” with one of the algorithms being selected at runtime as needed. For example, a family of algorithms may be related to “sorting” products in an [eCommerce](https://www.netsolutions.com/hub/digital-commerce) website – by size, colour, prize, etc. The strategy is implemented based upon the actions of the customer.

The strategy design pattern is incredibly powerful in personalization marketing strategies, responding to client location, inputs, or actions in order to deliver a different experience to each user.

### 5. Observer Design Pattern

The observer design pattern is “behavioral,” linking an object (subject) to dependents (observers) in a one-to-many pattern. When any of the observers change, the subject is notified. The observer design pattern is useful in any kind of event-driven programming such as notifying a user of a new comment on Facebook, sending an email when an item ships, etc.

### 6. Builder Design Pattern

The builder design pattern is “creational,” separating the object construction from the representation. This design pattern allows greater control over the [design process](https://www.netsolutions.com/insights/user-experience-design-process/) (more a step-by-step), but it also decouples the representation to support multiple representations of an object using the same base construction code (the ConcreteBuilder step).

The builder pattern executes in sequential steps as the object is being built, calling only those steps that are necessary for each iteration of the object.

### 7. Adapter Design Pattern

The adapter design pattern is a “wrapper” that converts one kind of interface into another existing kind of interface. The adapter design pattern helps classes work together when they are incompatible, allowing code to work together. Adapter patterns are useful to take mixed interfaces and make them a consistent API.

## **Popular Software Architectural Patterns**

Although somewhat outside the scope of this guide, it is important to mention that there are also architectural patterns that can be applied to the design of the software as a whole. What is an architectural pattern? A general, reusable solution to common problems in architecture (see how the definition is almost the same for software design?). These three design patterns are similar, with different sets of dependencies and levels of binding.

### 1. MVC Design Pattern

The model-view-controller (MVC) design pattern is the earliest architectural pattern that is made up of three parts:

* **Model** – the backend business logic and data
* **View** – the interface components to display the data. Leverages the Observer pattern to update with Model and display the updated model when necessary.
* **Controller** – Input is directed here first, processing the request through the model and passing it back to view

The MVC design pattern is important because it provides separation of concern (SoC), separating the front and backend code into distinct parts to make it easier to update and scale the application without interference or interruption. The MVC model also allows multiple developers to work on different parts of the application at the same time. The risk, however, is that exposing the model to view can introduce security and performance concerns.

MVC is common for web apps, libraries, and user interfaces.

### 2. MVP Design Pattern

The model-view-presenter (MVP) design pattern is derived from MVC but replaces the controller with the presenter and really focuses only on modeling the presentation layer.

* **Model** – the backend business logic and data
* **View** – input begins here and the requested action is presented here
* **Presenter** – One-to-one listening to the views and models, processing the request through the model and passing it back to view

In this model, the presenter acts as a mediator between the view and the model, supporting a more loosely coupled model. MVP is ideally suited to make views reusable and to support unit testing.

MVP is commonly used for websites, web apps, and mobile apps (particularly Android).

### 3. MVVM Design Pattern

In the model view view-model (MVVM) design pattern, there is two-way data binding between view and view-model (replacing presenter in the MVP design pattern), more cleanly separating the user interface and application logic:

* **Model** – the backend business logic and data
* **View** – input begins here and the requested action is presented here
* **View-Model** – has no reference to view, its only purpose is to maintain the state of view and manipulate the model as the actions of view change

MVVM allows for view-specific subsets of a model to be created with the state and logic bound to the view, requiring less logic in the code to run the view. MVVM is ideally suited to help improve performance and allow for greater customization and personalization of the view.

MVVM is also commonly used for mobile apps (growing use in Android) where bi-directional data binding is important.

## **Conclusion**

In software engineering, there are advantages and disadvantages to using software design patterns. Knowing when to use software design patterns – and when not – and how best to implement each pattern comes down to having an experienced team.