Top 7 Software Design Patterns You Should Know

Swimm Team

What are software design patterns in software engineering?

Software design patterns are reusable solutions to common problems that arise during the design of software applications. These patterns provide a standardized approach, best practices, and templates to tackle specific problems, allowing developers to improve the efficiency, maintainability, and scalability of their code.

Design patterns are not complete solutions, but rather guidelines or blueprints that can be adapted and applied to various situations in software development.

This is part of a series of articles about system design.

In this article:

Why do you need software architecture design patterns?

We need software architecture design patterns for several reasons:

- Enhance code maintainability: Design patterns promote modular and well-structured code, making it easier to maintain, modify, and extend as requirements evolve.
- Improve code reusability: Since design patterns provide reusable solutions to common problems, developers can save time and effort by reusing proven solutions instead of reinventing the wheel.
- Knowing design patterns is important even if you don’t use them: Familiarity with design patterns can inform architectural decisions and help developers understand and review existing code.
- Promote best practices: Design patterns embody best practices and principles, which can lead to more robust, efficient, and scalable software systems.
- Facilitate collaboration: Design patterns facilitate collaboration among team members, as

they provide a common language to discuss, evaluate, and agree on software design decisions.

7 commonly used software development design patterns

1. Singleton Design Pattern

The singleton pattern ensures that a class has only one instance and provides a global point of access to that instance. This pattern is useful when a single instance of a class needs to coordinate actions across the entire system. It helps in maintaining a consistent state and controlling access to shared resources.

Consider a print spooler system in an office environment that needs to manage print jobs from multiple users. Using the Singleton pattern, we can ensure that there is only one print spooler instance managing the print queue, avoiding conflicts and maintaining a consistent state across the system.

Implementation typically involves:

- Making the constructor private to prevent external instantiation.
- Declaring a static variable of the same class type.
- Providing a public static method to access the instance.

2. Factory Method Design Pattern

The factory method pattern defines an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created. This pattern promotes loose coupling by separating the object creation process from the actual usage of the objects.

In plain terms, consider an application that generates reports in various formats, like PDF, Excel, or Word. The Factory Method pattern could be used to create report generator objects specific to the required format, without the client knowing the details of the report generator implementations

Implementation typically involves:

- Defining a factory interface or abstract class with a method to create objects.
- Implementing the factory method in concrete factory classes to return specific object types.
- Clients use the factory method to instantiate objects, allowing for flexibility and extensibility.

3. Facade Design Pattern

The facade pattern provides a simplified interface to a complex subsystem, hiding its complexity and making it easier for clients to interact with the subsystem. It doesn't encapsulate the subsystem but rather composes the subsystem's components to provide a unified interface.

For example, consider a home automation system with subsystems for lighting, heating, and security. A facade pattern can be used to create a simplified interface that allows users to control all subsystems with simple commands, like "Morning routine" or "Night mode," hiding the complexity of interacting with each subsystem individually.

Implementation typically involves:

- Creating a facade class that composes the complex subsystem components.

- Implementing simplified methods in the facade class that delegate calls to the appropriate subsystem components.
- Clients interact with the facade class, reducing their dependency on the subsystem's complexity.

4. Strategy Design Pattern

The strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. This allows for selecting an algorithm at runtime and promotes loose coupling by separating the algorithm's implementation from its usage.

For example, in a navigation app, the user can choose different routing algorithms, like the shortest path, fastest route, or avoiding tolls. The Strategy pattern allows the app to switch between these algorithms at runtime, without the need for modifying the code.

Implementation typically involves:

- Defining a strategy interface or abstract class with a common method for all algorithms.
- Implementing concrete strategy classes for each algorithm, adhering to the strategy interface.
- Context class composes a strategy object and uses it to execute the desired algorithm.

5. Observer Design Pattern

The observer pattern defines a one-to-many dependency between objects so that when one object (the subject) changes its state, all its dependents (observers) are notified and updated automatically. This promotes loose coupling between the subject and its observers.

For example, a weather station collects data from sensors and updates multiple displays (like temperature, humidity, and air pressure) whenever new data is available. The Observer pattern can be used to notify all displays when there is new data so that they can update their respective information accordingly.

Implementation typically involves:

- Defining an observer interface with a method for updating observers.
- Creating concrete observer classes that implement the observer interface.
- Implementing a subject class that maintains a list of observers and provides methods to add, remove, and notify them.
- When the subject's state changes, it notifies all registered observers.

6. Builder Design Pattern

The builder pattern separates the construction of a complex object from its representation, allowing for the same construction process to create different representations. It's particularly useful when constructing objects with many optional or varying parts.

For example, an online pizza ordering system allows customers to create custom pizzas with various combinations of crust, sauce, cheese, and toppings. The Builder pattern can be employed to construct pizza objects with different compositions, while keeping the construction process consistent.

Implementation typically involves:

- Defining a builder interface or abstract class with methods for constructing the object's parts.
- Implementing concrete builder classes for each object representation.
- Creating a director class that takes a builder object and constructs the object using the builder's methods.
- Clients use the director with a specific builder to create the desired object representation.

7. Adapter Design Pattern

The adapter pattern allows two incompatible interfaces to work together by converting the interface of one class into another interface that clients expect. This promotes reusability and flexibility by enabling the integration of existing components with new systems.

For example, consider a media player application that supports playing audio files in different formats, such as MP3, WAV, or AAC. To add support for a new format, like OGG, without modifying the existing code, the Adapter pattern can be used to create an adapter that converts the OGG format to an interface the media player can understand.

Implementation typically involves:

- Defining a target interface that the client expects.
- Creating an adapter class that implements the target interface and composes an instance of the existing class (adaptee).
- Implementing the target interface methods in the adapter class by delegating calls to the adaptee's methods.
- Clients use the adapter class, interacting with the target interface while the adapter translates calls to the adaptee, enabling seamless integration of the two incompatible interfaces.

Pros and cons of software design patterns

Pros of software design patterns:

- Accelerated development: Design patterns offer proven solutions that can speed up the development process, reducing the time spent on solving complex problems from scratch.
- Enhanced code quality: By following design patterns, developers can create more robust and reliable code that adheres to best practices and proven principles.
- Easier debugging: Design patterns result in more structured and organized code, making it easier to identify and fix issues during debugging.
- Design consistency: Design patterns provide a unified approach to solving specific problems, leading to greater consistency across the application and making it easier for new team members to get up to speed.

Cons of software design patterns:

- Overuse or misuse: Inappropriately applying design patterns or using them unnecessarily can lead to overly complex code and decrease maintainability.
- Learning curve: Some design patterns can be challenging to understand and implement, requiring an initial investment of time and effort to master.
- Premature optimization: Relying too heavily on design patterns early in the development process can result in over-optimization, leading to an unnecessarily complex codebase that may not address actual requirements.
- Limited flexibility: Some design patterns can impose constraints on software design, limiting

flexibility in specific situations. Developers must carefully consider whether the benefits of a pattern outweigh any potential drawbacks for their particular use case.

If your team already uses design patterns, you are already enjoying their benefits (given that you have used them correctly). However, they may still pose a challenge for engineers in your team – who may not be familiar with them and get confused when using them or stumbling upon them while debugging.

Swimm is an automated documentation solution that can help engineers get familiar with design patterns. The best way to explain a design pattern is by demonstrating it, and it's especially great if the example is on your own codebase.

With Swimm, you can create this kind of document automatically, to help developers learn design patterns exactly when they need to. Thanks to the discoverability of Swimm documents within the IDE, the docs are found whenever someone stumbles upon code that includes the specific design pattern. If something ever changes in the system and the example changes, your document will be automatically updated.

In case some design patterns are ubiquitous in your codebases, you can describe them in a Swimm Playlist that every developer should go through. This is also a wonderful way to encourage engineers to use the patterns in the first place.

Learn more about Swimm, the knowledge management tool for code.