22. Summary of the problems design patterns are intended to solve

This chapter summarizes which design patterns can be used to solve any specific type of software development problem. It's intended to be a reference guide to help you find the right design pattern quickly.

All problem categories that we have discussed in this section are listed here. For each one of them, all suitable design patterns are provided. If there are multiple design patterns that can solve a problem of a particular type, a one-sentence summary will be provided next to each to help you decide which of them is more suitable for your specific situation.

Not knowing what object implementations you'll need ahead of time

- **Factory Method** - if an object needs to be instantiated in one go
- **Abstract Factory** - if multiple objects needs to be instantiated in one go
- **Builder** - if an object needs to be built step-by-step

Making several exact copies of a complex object

- **Prototype** - was specifically invented for easy cloning of objects

Using many instances of an object while keeping code running smoothly

- **Object pool** - facilitates the reuse of pre-instantiated objects
- **Flyweight** - allows you to have a very large number of similar objects without much performance penalty, but makes the code complicated
- **Prototype** - can be combined with Object Pool to make the initial creation of the objects easier

Using the same single instance of an object throughout the application

- **Singleton** - forces a single instance of an object of a particular type to be used throughout the application

Third-party components aren't directly compatible with your code

- **Adapter** - uses an intermediate translation object to convert between two different API formats

Adding new functionality to existing objects that cannot be modified

- **Decorator** - adds new capabilities to an existing object by overriding its methods

Accessing complex back-end logic from the presentation layer

- **Facade** - simplifies the access interface to the complex logic
- **Proxy** - caches the result of expensive operations

User interface and business logic are developed separately

- **Bridge** - suitable when front-end and back-end can be designed together up-front
- **Facade** - suitable when the back-end is hosted by a third party or cannot be designed alongside the user interface up-front
- **Proxy** - prevents service outages during back-end redeployment

Building a complex object hierarchy

- **Composite** - invented for creating complex tree-like structures of objects that can have any arbitrary capabilities

Implementing complex conditional logic

- **Strategy** - facilitates a conditional one-off action
- **Factory Method** - facilitates a conditional creation of a long-lived object
- **Abstract Factory** - facilitates a conditional creation of multiple long-lived objects

Multiple object instances of different types need to be able to communicate with each other

- **Mediator** - easier to implement when communication logic between different objects doesn't expect to be changed
- **Observer** - easier to implement when communication between different objects is expected to change at runtime or during configuration

Multiple stages of processing are needed

- **Chain of Responsibility** - suitable in scenarios where processing steps are pre-defined and a one-off logical flow is executed
- **Builder** - suitable in scenarios where the order of processing stages can be arbitrary and a reusable object is being built

The system is controlled by complex combinations of inputs

- **Command** - keeps the logic for a complex operation in a single place

Ability to undo an action that has been applied

- **Memento** - allows you to store the exact snapshots of the state
- **Command** - allows you to revert by performing an opposite action

Ability to traverse a collection without knowing its underlying structure

- **Iterator** - performs a traversal of any complex data structure internally while exposing a simple interface externally

Creating a family of related algorithms

- **Template Method** - easy to implement, but might violate the Liskov substitution principle
- **Visitor** - allows you to separate an object from its behavior and add many differential types of behavior to the objects
- **State** - allows you to change the behavior of an entire object in one go by changing the mode (state) that the object is in
- **Strategy** - suitable when algorithms are selected by conditional logic